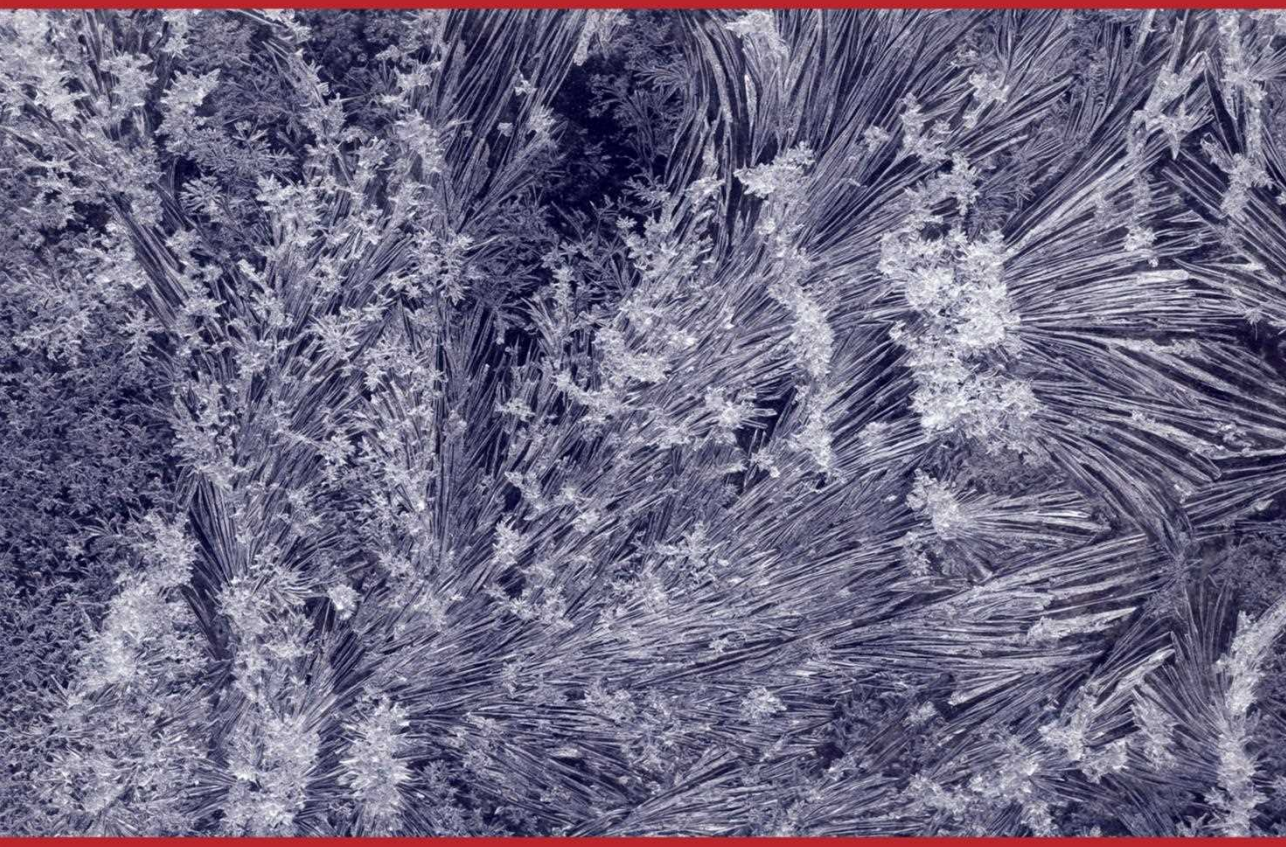
Saša Malkov

Razvoj softvera



Matematički fakultet

Razvoj softvera

Saša Malkov

Razvoj softvera

Saša Malkov

Univerzitet u Beogradu – Matematički fakultet
Beograd, 2024.

Razvoj softvera

Univerzitetski udžbenik

Prvo izdanje

Autor:	dr Saša Malkov
Izdavač:	Univerzitet u Beogradu – Matematički fakultet Studentski trg 16, 11000 Beograd +381 (11) 2027 801 matf@matf.bg.ac.rs
Za izdavača:	dr Zoran Rakić, dekan
Izdavački odbor:	dr Nebojša Ikodinović, predsednik dr Miroslava Antić dr Bojan Arbutina dr Filip Marić dr Petar Melentijević
Recenzenti:	dr Ivan Čukić dr Nenad Korolija
Ilustracije:	*Autor*
Grafička priprema:	*Autor*
Korice:	*Autor*
Štampa i povez:	Donat Graf d.o.o., Grocka
Tiraž:	500 primeraka

ISBN: 978-86-7589-196-3

Porodici

Poglavlja

Sadržaj

Spisak slika

Predgovor

Ako ste naučili neki programski jezik i umete da napišete manje programe, verovatno ste se našli pred dilemom: *Da li mogu da napišem neki veći i složeniji program? Kako to da uradim?*

Ako ste već napisali neki veći program ili bar pokušali da ga napišete, onda ste imali priliku da se suočite sa nekim problemima na koje vas osnovni kursevi programiranja nisu pripremili. Sa nekima ste izašli na kraj lakše, sa nekima teže, a neki vas možda još uvek muče. Sigurno ste se susreli sa pitanjima poput: *Ako imam dva moguća rešenja nekog problema, kako da znam koje je bolje? Zašto često moram da pravim velike izmene? Da li to može nekako da se spreči? Kako da pronađem i popravim greške? Kako da sprečim nastajanje bagova? Kako da napišem efikasan softver?*

Ako ste pročitali neke knjige ili članke o programiranju, možda niste sigurni oko toga šta označavaju neki termini i imate pitanja poput: *Šta je posetilac? U čemu je razlika između kohezije i spregnutosti? Šta je stabilnost softvera? U čemu je smisao programiranja u paru? Šta znače te silne linije i brojevi na dijagramima klasa?*

Ova knjiga bi trebalo da vam pomogne da pronađete odgovore na takva i slična pitanja. Namenjena je prvenstveno čitaocima, koji su već naučili da programiraju i osposobljeni su da pišu manje programe i oblikuju i implementiraju relativno složene algoritme. Njen zadatak je da ih upozna sa nekim od najvažnijih koncepata, tehnika i metoda savremenog razvoja softvera. Obuhvaćene su najvažnije metodološke teme, izabrane teme iz oblasti projektovanja softvera, kao i izabrane razvojne tehnike.

Sadržaj i forma knjige su uređeni tako da ona može da služi kao udžbenik za predmet Razvoj softvera, koji pohađaju studenti Matematičkog fakulteta Univerziteta u Beogradu na studijskim programima Informatika i Matematika. Sadržaj knjige čine teme koje su tokom više godina obrađivane u okviru predmeta Razvoj softvera.

Izloženim sadržajem se čitaocima pruža dovoljno materijala za sticanje solidnog uvida u ovu veoma široku oblast. Predstavljanje izabranih razvojnih tehnika i isticanje njihovog značaja ima za cilj njihovo upoznavanje i postepeno uključivanje u svakodnevnu praksu. Insistiranje na njihovom temeljnom upoznavanju bi trebalo da ukaže na njihovu složenost, koja možda nije uvek u dovoljnoj meri očigledna, a nikako ne bi smela da se zanemari. Zato se na kraju svakog poglavlja predlaže literatura za dalje izučavanje.

U slučaju nekih od predstavljenih tehnika, njihove karakteristike su lako sagledive, a koristi od njihove primene su relativno očigledne i lako razumljive. Međutim, u nekim drugim slučajevima, zbog složenosti tehnika ili zbog relativne složenosti skupa okolnosti u kojima se one primenjuju, rezultati njihove primene mogu da budu teže uočljivi ili razumljivi. Zbog toga je pri obrađivanju takvih tehnika posebna pažnja posvećena predstavljanju i objašnjavanju motivacije za njihovo oblikovanje i primenu. Obrađene teme i predstavljene tehnike su propraćene odgovarajućim primerima. U nekim slučajevima primeri su morali da budu malo većeg obima.

Poželjno je da su čitaoci prethodno upoznati sa osnovama programiranja i objektno-orijentisanim programiranjem, da dobro poznaju programski jezik *C* i bar jedan objektno-orijentisan programski jezik, kao i da imaju osnovna znanja iz programskog jezika *C++*. Takođe, pretpostavlja se da imaju dovoljno iskustva u radu sa osnovnim alatima za prevođenje programa. Za ilustrovanje izabranih tehnika i metoda se upotrebljava programski jezik *C++*, ali većina primera može relativno lako da se razume i uz poznavanje drugih objektno-orijentisanih jezika.

U knjizi se pokazivači upotrebljavaju pretežno neposredno, na način koji je u programski jezik C++ uveden od prve verzije jezika. U nekim primerima se koriste tzv. pametni pokazivači, koji su uvedeni od verzije C++11 i kasnije dodatno unapređivani. Takav pristup je odabran zato što je važno da studenti u potpunosti razumeju rad sa pokazivačima. Odsustvo dobrog poznavanja rada sa neposrednim pokazivačima ima za posledicu nerazumevanje funkconisanja pametnih pokazivača, što je najčešći uzrok grešaka pri njihovoj upotrebi. Naravno, u praksi se preporučuje upotreba pametnih pokazivača.

Knjiga obuhvata 13 poglavlja. Svako poglavlje se bavi jednom od važnih tema iz oblasti razvoja softvera, na način i u obimu koji su prilagođeni pretpostavljenom prethodnom znanju i omogućavaju čitaocu da razume i konkretnu temu i njeno mesto u okviru oblasti razvoja softvera.

U prvom poglavlju „*1 - Uvod*“ se uvode osnovni pojmovi. Naglašava se razlika između razvoja softvera i programiranja, kao i sličnosti i razlike između pojmova razvoj softvera i softversko inženjerstvo. Uvodi se i pojam razvojne metodologije.

Poglavlje „*2 - Problemi pri razvoju softvera*“ upoznaje čitaoca sa problemima do kojih dolazi pri razvoju softvera, kao i sa posledicama koje ti problemi mogu da

izazovu. Razumevanje ozbiljnosti i razmera problema i mogućih posledica je neophodno da bi se u punoj meri razumela odgovornost koju na svojim leđima nose razvijaoci softvera. Kada se vide razmere mogućih posledica, onda postaje mnogo lakše da se podnese teret primene brojnih složenih tehnika, zato što se bolje razume njihova opravdanost.

U poglavlju „*3 - Objektna orijentacija*“ je pažnja posvećena objektno-orijentisanoj paradigmi programiranja i razvoja softvera, koja je danas najzastupljenija. Ukazuje se na motivaciju za njeno uvođenje ali i na neka od njenih neugodnih ograničenja.

U poglavlju „*4 - Uvod u projektovanje softvera*“ započinje upoznavanje osnovnih tema iz oblasti projektovanja softvera. Ova veoma važna oblast se dodatno obrađuje i u drugim poglavljima.

Razvoj softvera danas ne može da se zamisli bez *Objedinjenog jezika za modeliranje*, kome je posvećeno poglavlje „*5 - UML*“. Izlažu se okolnosti u kojima je *UML* nastao, motivacija za njegovo definisanje, kao i manji deo samog jezika, u meri u kojoj je on neophodan za dalje praćenje sadržaja ove knjige.

Poglavlje „*6 - Principi projektovanja softvera*“ je posvećeno upoznavanju principa projektovanja, koji imaju ulogu putokaza, čija primena bi trebalo da nam olakša snalaženje u prilično gustom i složenom prostoru mogućih rešenja, kao i da nam pruži oslonac pri donošenju odluka.

Savremeno objektno-orijentisano programiranje i projektovanje se u velikoj meri oslanjaju na primenu obrazaca za projektovanje. Njima se bavi poglavlje „*7 - Obrasci za projektovanje*“. Umesto izlaganja većeg broja obrazaca, što je na raspolaganju u drugim knjigama, ovde je više pažnje posvećeno upoznavanju i razumevanju koncepta obrasca i motivacije za njihovu primenu. Izloženo je i nekoliko primera.

U poglavlju „*8 - Agilni razvoj softvera*“ se predstavljaju agilne metodologije razvoja softvera. Ova tema se sa pravom može smatrati i metodološkom i tehničkom, zato što su savremene agilne metodologije donele sa sobom i obilje razvojnih tehnika, koje su iz osnova promenile oblast razvoja softvera. Neke od tehnika su predstavljene malo detaljnije, a nekima su posvećena i posebna poglavlja.

Poglavlje „*9 - Razvoj vođen testovima*“ se bavi istoimenom metodom razvoja softvera, ali i tehnikom testiranja jedinica koda, na kojoj taj metod počiva. Osim uloge u implementiranju softvera i staranju o kvalitetu softvera, ovaj metod ima i druge uloge i sa pravom se može nazvati jednim od glavnih činilaca agilnog razvoja.

U poglavlju „*10 - Refaktorisanje*“ se obrađuje agilna tehnika čiji je cilj održavanje kvalitetnog dizajna programa u okolnostima njegovog čestog i čak neprestanog menjanja, što je uobičajeno za agilni razvoj. Ova tehnika se odlično nadopunjuje sa obrascima za projektovanje i razvojem vođenim testovima.

Poglavlje „*11 - Polimorfizam*“ se bavi jednim od najvažnijih aspekata savremenih programskih jezika. Polimorfizam predstavlja jedan od osnovnih alata za pisanje

apstraktnog programskog koda. Većina čitalaca se kroz upoznavanje sa objektno-orijentisanim programiranjem već susrela i dobro upoznala sa *hijerarhijskim polimorfizmom*, pa se u ovom poglavlju pažnja uglavnom posvećuje drugim oblicima polimorfizma, a pre svega *parametarskom polimorfizmu*.

Poglavlje „*12 - Debagovanje*“ se bavi tehnikama za pronalaženje i ispravljanje grešaka u programima. Debagovanje je jedna od najneugodnijih oblasti razvoja softvera, ali je ujedno i jedna od oblasti u koje svako od razvijalaca softvera mora da uđe, želeo to ili ne. Deo ovog poglavlja je posvećen prevenciji grešaka, što je jedan od najvažnijih aspekata rada sa greškama.

Za sam kraj je ostavljena tema koja spada u najizazovnije, ali istovremeno i „najklizavije“ teme u razvoju softvera. Poglavlje „*13 - Optimizacija softvera*“ je posvećeno staranju o performansama softvera i posebno nivoima, strategijama i tehnikama optimizacije softvera. Ova zahtevna oblast skriva mnoge zamke, za čije uočavanje i zaobilaženje je potrebno temeljno poznavanje razvojnog procesa i odgovarajućih alata, ali i platforme i okolnosti u kojima se softver razvija.

Izložene teme bi mogle da se podele u tri veće celine, uz određena preklapanja među njima. Prvu celinu čine opšte i metodološke teme. One se bave nekim od najvažnijih opštih pitanja i koncepata u razvoju softvera. Njihovo poznavanje je neophodno za ispravno razumevanje preostalih tema i dobro snalaženje u savremenom razvojnom procesu. Opšte i metodološke teme obuhvataju poglavlja:

1 - Uvod
2 - Problemi pri razvoju softvera
3 - Objektna orijentacija
5 - UML
8 - Agilni razvoj softvera

Drugu celinu čine teme iz oblasti projektovanja softvera. U odgovarajućim poglavljima je pažnja usmerena na strukturno projektovanje i na teme koje su važne svakom razvijaocu softvera:

4 - Uvod u projektovanje softvera
5 - UML
6 - Principi projektovanja softvera
7 - Obrasci za projektovanje

Treću celinu čine izabrane tehnike razvoja softvera. Predstavljene su tehnike koje su nezaobilazne u savremenom agilnom razvoju:

7 - Obrasci za projektovanje

9 - Razvoj vođen testovima

10 - Refaktorisanje

11 - Polimorfizam

12 - Debagovanje

13 - Optimizacija softvera

Redosled tema je izabran tako da razumevanje izloženih sadržaja bude što lakše. Ipak, neke teme su toliko jako međusobno povezane da nije moglo da se izbegne referisanje na delove teksta koji tek slede. Na primer, u poglavlju *6 - Principi projektovanja softvera* se pri objašnjavanju nekih principa radi ilustracije pominju pojedini obrasci za projektovanje, koji se obrađuju tek u narednom poglavlju. Poznavanje i bar delimično razumevanje principa projektovanja je neophodno da bi se uvideo značaj i smisao obrazaca za projektovanje, pa zato izlaganje ide tim redom, ali navedeni principi projektovanja i načini njihove primene u praksi mogu u pravoj meri da se razumeju tek nakon upoznavanja obrazaca za projektovanje.

Preporučen redosled čitanja knjige odgovara redosledu izlaganja. Radi boljeg razumevanja izloženih sadržaja i njihovog jasnijeg i temeljnijeg povezivanja, preporučuje se da se teme koje se odnose na projektovanje softvera (pre svega poglavlja *4 - Uvod u projektovanje softvera* i *6 - Principi projektovanja softvera*) pročitaju još jedanput, nakon što se pročita cela knjiga.

Čitaoci koji više vole da prvo osete stvari „pod prstima“, pa da se tek onda bave „teorijom“, mogu da odmah posle uvodnog poglavlja pređu na poglavlje *5 - UML*, pa na poglavlje *7 - Obrasci za projektovanje* i da zatim nastave do kraja knjige, pa da se tek posle toga vrate na preskočena poglavlja 2, 3, 4 i 6.

Oblast razvoja softvera je veoma široka i zbog toga ima mnogo tema za koje se u ovoj knjizi nije našlo dovoljno mesta. Na primer, nisu izložene specifične teme u vezi sa staranjem o kvalitetu softvera (mada većina izloženih tema ima dodira sa staranjem o kvalitetu), teme koje se odnose na bezbednost softvera ili etiku u razvoju softvera. Izostale su i neke veoma važne tehnike implementiranja, kao što su konkurentno programiranje i tehnike izgradnje programa. U planu je da se neki od tih sadržaja prirede i objave u bliskoj budućnosti, mada možda u nekoj drugoj formi.

Pri pripremi i držanju predavanja iz predmeta Razvoj softvera, pa tako i pri pripremi materijala i pisanju ove knjige, od velike pomoći su bili asistenti na ovom i povezanim predmetima. Najveću zahvalnost dugujem svojim bivšim asistentima na predmetu Razvoj softvera, a današnjim kolegama, docentima dr Ivanu Čukiću i dr Mirjani Maljković Ružičić, kao i bivšim i sadašnjim asistentima Nemanji Mićoviću, Nikoli Ajzenhameru, Momiru Adžemoviću, Andrijani Aleksić, Petru Đorđeviću i

Aleksandru Stefanoviću. Zahvaljujem se svojim kolegama sa Katedre za računarstvo i informatiku Matematičkog fakulteta, sa kojima sam često imao veoma korisne razmene mišljenja u vezi sa materijalima za ovu knjigu, a posebno dr Nenadu Mitiću. Veliku zahvalnost dugujem recenzentima dr Ivanu Čukiću i dr Nenadu Koroliji, kao i kolegi dr Jovanu Popoviću, koji su svojim korisnim sugestijama u završnim fazama pripreme ove knjige doprineli da tekst postane još jasniji i potpuniji.

Posebno sam zahvalan svojim studentima, čija su brojna pitanja doprinela da bolje razumem probleme sa kojima se suočavaju pri upoznavanju oblasti razvoja softvera i da prema njima prilagodim izbor tema i način izlaganja.

Autor

Beograd, 2024.

Razvoj softvera

1 - Uvod

Ne pretvaram se da znamo sve odgovore.
Ali pitanja svakako zaslužuju da se o njima razmisli.
Artur Klark

Vežite se...

Nakon što nauče osnove programiranja i upoznaju veći broj opštih i specifičnih algoritama, mladi programeri često imaju utisak da su naučili da prave softver. Ali, stvari stoje drugačije. Time što je naučeno programiranje, samo je obezbeđena ulaznica u svet razvoja softvera. Put do dobrih rezultata je dugačak i ispunjen brojnim drugim aktivnostima, a ne samo programiranjem.

Mnogo softvera se piše ad-hok – bez velikih planova i namera; za jednokratnu upotrebu, da bi se rešio neki tekući problem. U takvim slučajevima pojmovi „softver“ i „program“ se često ne razlikuju – sve što pišemo je jedan jednostavan program, čije pisanje često ne zahteva ni posebne tehnike ni disciplinu u radu. Zato ponekad može da se učini da nema mnogo razlike između pravljenja softvera i programiranja, ali to je veoma daleko od istine. Već u slučajevima kada je potrebno da se napiše ne samo jedan jednostavan program nego više njih, i to tako da njihova upotreba bude jasna, logična i međusobno usklađena, stvari postaju složenije; a kada poraste i složenost programa koje pišemo, onda prelazimo u sasvim drugi prostor problema. Što je softver složeniji to njegovo pravljenje ima više različitih činilaca, bez kojih softver ne može da se efikasno i kvalitetno pravi, upotrebljava i održava.

Obično se najpre uočava da je neophodno da se pravi i održava neki vid dokumentacije. Dokumentacija je, pre svega, sredstvo komunikacije, kojim se prenose informacije o upotrebi softvera od razvijalaca prema korisnicima, ali i razmenjuju informacije o strukturi i implementaciji softvera među njegovim

razvijaocima. Iz ugla programera, dokumentaciju čine i formalne specifikacije delova softvera, različiti projekti, planovi, definicije interfejsa i drugo.

Ispostavlja se da je programiranje samo jedna od brojnih aktivnosti koje se preduzimaju tokom razvoja softvera. Štaviše, programiranje u okviru kompleksnijeg procesa razvoja softvera postaje mnogo drugačije i složenije od programiranja koje smo prethodno upoznali – nije više dovoljno da se napiše program, nego pri tome moraju da se prate određena pravila, a delovi programa koje pišemo moraju da budu u skladu sa nekim ustanovljenim principima.

Pored programiranja, pravljenje softvera obuhvata i sve one pripremne korake, koji prethode pisanju programa, kao i sve one postupke koji tokom ili nakon pisanja programa pomažu da se programiranje izvede što kvalitetnije i koji obezbeđuju da napisani programi predstavljaju što dugotrajniju vrednost. U nekom jednostavnijem slučaju, ti dodatni poslovi bi možda mogli da se ne uoče, zato što ih je malo ili su sasvim jednostavni, ali u složenijim slučajevima razvoja softvera, ne samo da je tih dodatnih poslova mnogo, nego oni čak postaju veoma složeni i obimni, tako da se relativno često događa da se njima posvećuje više pažnje i vremena nego programiranju.

Različite aktivnosti se tokom rada povezuju, kombinuju i prepliću, pa ih nije uvek lako ni prepoznati ni nabrojati. Skup tih aktivnosti zavisi od ugla posmatranja i primenjene razvojne metodologije, ali i specifičnosti problema koje rešavamo, načina grupisanja različitih poslova u celine, složenosti projekta na kome radimo i mnogih drugih faktora. Radi ilustracije, mogli bismo, na primer, da izdvojimo sledeće aktivnosti:

- prikupljanje informacija o ciljevima i potrebama;
- analiziranje problema;
- prepoznavanje i oblikovanje zahteva;
- projektovanje softvera;
- oblikovanje korisničkog interfejsa;
- implementiranje softvera;
- testiranje softvera;
- održavanje softvera;
- planiranje resursa;
- planiranje i praćenje toka razvoja;
- staranje o kvalitetu softvera;
- upravljanje razvojnim procesom;
- uređivanje tehničke i korisničke dokumentacije;

- pravljenje i uređivanje fotografija, ilustracija, video i zvučnih zapisa za potrebe softverskog projekta;
- pripremanje i pakovanje proizvedenog softvera za distribuiranje i instaliranje
- i drugo.

Ovaj spisak bi mogao da bude kraći ali i duži. Na primer, testiranje je navedeno kao posebna aktivnost iako bi moglo da se svrsta u staranje o kvalitetu softvera, a delimično i u implementiranje softvera. Nasuprot tome, testiranje bi moglo da se podeli na različite vrste testiranja i da se zameni sa više različitih aktivnosti.

Na ovom spisku naizgled nema programiranja – ono namerno nije eksplicitno navedeno, već se pretpostavlja da je sadržano u okviru implementiranja softvera. To je urađeno da bi se dodatno istakla činjenica da je programiranje samo jedan od činilaca razvoja softvera. Štaviše, u nekim posebnim slučajevima imamo primere razvoja softvera u kojima uopšte nema programiranja. Na primer, ako upakujemo gotove programe, određena uputstva za konfigurisanje i neke dodatne sadržaje (slike, tekstove, video ili audio zapise), onda možemo da dobijemo nov softverski proizvod, a da pri tome nismo napisali nijedan nov red programa. Važno je da razumemo da postoje i takvi projekti, iako se u daljem tekstu bavimo isključivo razvojem softvera u kome je najvažniji deo pravljenje programskih elemenata.

Softversko inženjerstvo

Preduzimanje velikog broja različitih aktivnosti pri pravljenju softvera ima za posledicu da u tom poduhvatu moraju da učestvuju razvijaoci softvera koji imaju veliki broj različitih specijalnosti. Brojne aktivnosti i rad većeg broja ljudi nije lako organizovati, pa je neophodno da se u tom radu primenjuje sistematičan pristup, koji počiva na primeni naučnih rezultata, ali i dobrih prethodnih iskustava – inženjerstvo.

Inženjerstvo je praktična primena naučnih saznanja i matematike pri rešavanju problema i pravljenju proizvoda. Pri tome se aktivno koriste prethodna iskustva, oličena u pravilima ili standardima, a pomoću kojih se obezbeđuje odgovarajući nivo kvaliteta napravljenog rezultata. U skladu s tim se definiše i termin *softversko inženjerstvo* [*IEEE 2017*]:

> *Softversko inženjerstvo je*
> *primena sistematskog, disciplinovanog i merljivog pristupa*
> *razvoju, funkcionisanju i održavanju softvera;*
> *odnosno primena inženjerstva na softver*
> *IEEE*

Iz navedene definicije se vidi da se softversko inženjerstvo odlikuje sistematičnim i disciplinovanim inženjerskim pristupom problemu proizvodnje softvera. Pri tome je i sveobuhvatno, u smislu da se odnosi na *sve* aktivnosti koje se preduzimaju pri proizvodnji softvera.

Da bismo se lakše nosili sa složenošću proizvodnje softvera i softverskog inženjerstva, obično se problemi posmatraju iz različitih aspekata. Tri najvažnija pristupa su iz aspekta procesa, metoda i kvaliteta[1]. Svaki od ovih aspekata nam pruža odgovarajući deo odgovora na pitanja razvoja softvera, ali nam tek svi zajedno pružaju kompletne odgovore na probleme sa kojima se suočavamo.

Jedan od najvažnijih aspekata softverskog inženjerstva predstavlja upravljanje razvojnim procesom i njegovo redovno praćenje i nadziranje. Možemo da kažemo da nam posmatranje problema iz ugla procesa pruža odgovore na pitanja „Šta radimo?“ i „Kada to radimo?“.

Odvijanje različitih aktivnosti tokom proizvodnje softvera se naziva *proces razvoja softvera*, ili *razvojni proces*. Termin *proces* se upotrebljava kako da označi celokupan razvojni proces, tako i da označi odvijanje manjeg broja aktivnosti (ili čak pojedinačnih aktivnosti) u nekom segmentu razvojnog procesa. Pri planiranju i upravljanju razvojnim procesom mora da se ima u vidu šta su preduslovi za odvijanje određenih aktivnosti i šta su rezultati tih aktivnosti. Sagledavanjem svih potrebnih aktivnosti stiču se uslovi za planiranje rasporeda njihovog odvijanja.

Drugi važan aspekt softverskog inženjerstva čini planiranje i odabiranje skupa metoda, tehnika i alata koji će se upotrebljavati na nekom projektu. Ako nam je upravljanje procesom odredilo šta se i kada radi, onda nam planiranje metoda, tehnika i alata odgovara na pitanje „Kako to radimo?“.

Pri obavljanju svake od aktivnosti, koja čini razvojni proces, mogu da se upotrebljavaju različite *tehnike* i odgovarajući *alati*. Da bi rezultati jedne aktivnosti bili upotrebljivi u nekoj drugoj, neophodno je da se usklade alati i tehnike koji se koriste u tim aktivnostima. Za obavljanje istog posla često mogu da se upotrebe različite tehnike, kao što i neke tehnike mogu da se upotrebljavaju u većem broju različitih aktivnosti. U razvoju softvera se često upotrebljava i termin *metod*. U nekim slučajevima se ne pravi značajna razlika između termina metod i tehnika. Ipak, uglavnom je uobičajeno da se terminu metod dodeljuje nešto šire značenje, pa možemo da kažemo da metod predstavlja specifičan način primene jedne ili više tehnika u okviru nekog dela razvojnog procesa ili pojedinačne aktivnosti. Na primer, specifičan način primene tehnike testiranja jedinica koda predstavlja osnovu metoda razvoja vođenog testovima.

[1] U literaturi se ovi aspekti nazivaju i slojevima softverskog inženjerstva. Takođe, metodi se negde zamenjuju alatima i tehnikama. U svakom slučaju, suština je ista.

Treći aspekt softverskog inženjerstva je staranje o kvalitetu. Ne bismo mnogo pogrešili ako bismo rekli da je staranje o kvalitetu najvažniji aspekt softverskog inženjerstva, a da ostali aspekti postoje samo da bi staranje o kvalitetu moglo da se ostvari u punoj meri. Staranje o kvalitetu je često najprepoznatljivija karakteristika sistematičnog i disciplinovanog razvoja softvera. Dok je u manje ozbiljnim poduhvatima često dovoljno da program „proradi" i isporuči odgovarajuće rezultate, u komercijalnoj proizvodnji softvera je od presudnog značaja da rezultat zadovolji definisane kriterijume kvaliteta. Bez staranja o kvalitetu, svi drugi činioci softverskog inženjerstva postaju bezvredni.

Da bi razvojni proces mogao da se lakše razume i planira, a zbog velikog broja različitih aktivnosti o kojima mora da se vodi računa, obično pokušavamo da ovaj problem malo pojednostavimo tako što grupišemo aktivnosti u nekoliko većih celina. Ima više načina da se to uradi. Jedna od mogućih podela je prema osnovnom poslu kome pripadaju aktivnosti, pa tako razlikujemo sledeće celine:

- **planiranje softvera** – često se naziva i specifikacija softvera; obuhvata sve aktivnosti koje se odnose na prikupljanje informacija o ciljevima, sagledavanje i definisanje zahteva i potrebnih karakteristika softvera, procenu potrebnih resursa i druge poslove neophodne za odgovaranje na pitanje *šta se pravi*;
- **projektovanje i implementiranje softvera** – obuhvata sve aktivnosti u vezi projektovanja i implementiranja, uključujući i programiranje; ponekad se naziva i *razvoj softvera*;
- **staranje o kvalitetu** – sve aktivnosti koje imaju za cilj obezbeđivanje potrebnog nivoa kvaliteta, ali i proveravanje da li je taj nivo kvaliteta ostvaren; u velikoj meri se prepliće sa drugim celinama;
- **održavanje softvera** – aktivnosti koje imaju za cilj da produže vrednost softvera, kroz prilagođavanje izmenjenim zahtevima korisnika ili tržišta; često se tu ubrajaju i aktivnosti u vezi sa pripremanjem softvera za rad, konfigurisanjem, obučavanjem korisnika i slično.

Nešto drugačiji pristup predstavlja grupisanje poslova u tzv. „okvirne aktivnosti", uz konstatovanje da se u većim razvojnim projektima one odvijaju iterativno, u razvojnim ciklusima [*Pressman 2020*]:

- **komunikacija** – razmena informacija sa klijentom i prikupljanje potrebnih informacija da bi moglo da se započne planiranje; cilj je razumevanje ciljeva i potreba klijenta;

- **planiranje** – pravljenje okvirnog plana realizacije projekta, sa analizom rizika i okvirnom procenom potrebnih resursa; pravljenje okvirnog rasporeda aktivnosti;
- **modeliranje** – projektovanje novog proizvoda; preciznost i obim projekta zavise od specifičnosti konkretnog proizvoda;
- **konstrukcija** – izgradnja softvera obuhvata detaljno projektovanje i implementiranje; savremene metodologije podrazumevaju da se tokom konstrukcije izvodi i značajan deo testova kvaliteta;
- **pripremanje za puštanje u rad** – proizveden softver mora da se pripremi za rad; preduzimaju se različite aktivnosti u zavisnosti od vrste i složenosti softvera.

Možemo da uočimo neke sličnosti i razlike između ovih podela, koje mogu da nam pomognu da razumemo složenost i obim razvojnog procesa. Na primer, u prvoj podeli nije navedena „komunikacija“, već se pretpostavlja da je ona sastavni deo „planiranja“, ali se danas uglavnom očekuje da se komunikacija odvija tokom čitavog razvojnog procesa; u prvoj podeli su „projektovanje i implementiranje“ objedinjeni, a u drugoj su podeljeni na „modeliranje“ i „konstrukciju“; prva podela ima izdvojenu grupu „staranje o kvalitetu“, a u drugoj se ona ne pojavljuje kao samostalna grupa, već se podrazumeva da se staranje o kvalitetu prepliće sa svim ostalim aktivnostima; prva podela prepoznaje „održavanje“, a u drugoj se pretpostavlja da je održavanje softvera samo još jedan razvojni ciklus, koji čine sve navedene okvirne aktivnosti.

Razvoj softvera

Termin *razvoj softvera* se koristi još češće od softverskog inženjerstva[2]. Razvoj softvera i softversko inženjerstvo imaju mnogo toga zajedničkog i relativno često možemo da upotrebimo bilo koji od ova dva termina, bez bojazni da će doći do nesporazuma. Ipak, za razliku od termina softversko inženjerstvo, čija je upotreba uglavnom ujednačena, termin razvoj softvera se u literaturi i praksi upotrebljava u različitim kontekstima i sa različitim značenjem.

Kada govorimo o konkretnom softverskom projektu, razvoj softvera, u najširem smislu, obuhvata doslovno sve aktivnosti koje se odnose na pravljenje softvera, pa i sam razvojni proces. Upotrebljava se i kao sinonim za pravljenje softvera ili proizvodnju softvera. U skladu s tim, ako se govori o disciplini razvoja softvera,

[2] Na primer, *Google* pronalazi oko 117.000.000 rezultata ako tražimo „*software engineering*“ i oko 202.000.000 rezultata ako tražimo „*software development*“.

onda možemo da kažemo da ona, u najširem smislu, predstavlja sinonim za softversko inženjerstvo.

Nasuprot tome, u najužem smislu, razvoj softvera se odnosi samo na poslove projektovanja i implementiranja softvera, pa i disciplina razvoj softvera obuhvata samo poslove koji čine projektovanje i implementiranje softvera. Takvo određenje ovog termina je prilagođeno klasičnim razvojnim metodologijama, u kojima su različiti poslovi bili jasno razdvojeni po fazama. Tada su i poslovi projektovanja i implementacije bili jasno odvojeni i međusobno i od ostalih poslova, kako vremenski tako i po specijalnostima subjekata koji su obavljali ove poslove. Međutim, takvo organizovanje razvojnog procesa nam danas donosi više problema nego koristi, pa se zato i termin razvoj softvera relativno retko koristi sa ovako uskim značenjem.

U daljem tekstu ćemo da podrazumevamo da u kontekstu razvojnog projekta termin razvoj softvera ima najšire značenje, a da se disciplina razvoj softvera odnosi prvenstveno na aktivnosti koje su neposredno ili relativno blisko povezane sa projektovanjem i implementiranjem softvera. U odnosu na ranije prepoznate grupe poslova, jasno je da je fokus discipline razvoja softvera prvenstveno na grupi „projektovanje i implementiranje softvera“ (ili na grupama „modeliranje“ i „konstruisanje“, zavisno od toga koju podelu razmatramo). Međutim, važno je da primetimo da i u ostalim grupama poslova postoje brojne aktivnosti koje su relativno blisko povezane sa projektovanjem i implementiranjem i koje se takođe razmatraju kao činioci razvoja softvera. Na primer, definisanje interfejsa komponenti se svrstava i u planiranje i u projektovanje; testiranje jedinica koda se jednako svrstava u programiranje i u staranje o kvalitetu; sve aktivnosti u vezi sa projektovanjem i implementiranjem su zastupljene i u održavanju softvera i drugo.

Takvo tumačenje discipline razvoja softvera, koje je negde između dva ekstrema, danas preovladava i u literaturi i u praksi. Posebno je zgodno što je ono prilagođeno savremenim agilnim razvojnim metodologijama. Danas se različite razvojne aktivnosti prepliću i vremenski i sadržajem, a od razvijalaca se očekuje da imaju široka znanja i da su osposobljeni za mnoge aktivnosti, koje nisu neposredno povezane sa projektovanjem i programiranjem, ali sa njima imaju dodirnih tačaka.

Softverski inženjeri i razvijaoci softvera

Posledica razlikovanja softverskog inženjerstva i razvoja softvera bi trebalo da bude i odgovarajuće razlikovanje termina *softverski inženjer* i *razvijalac softvera*, ali savremena praksa tu nije dosledna. Ovi pojmovi se često izjednačavaju, ali se u drugim prilikama nailazi i na raznovrsna tumačenja njihovih razlika.

Savremene metodologije u velikoj meri počivaju na visokom nivou stručnosti i samostalnosti svih članova razvojnog tima, pa je kvalitetno i široko obrazovanje jedan od osnovnih kriterijuma za angažovanje razvijalaca. U savremenoj praksi se zvanje softverskog inženjera često dodeljuje praktično svim razvijaocima koji rade na razvoju programskih elemenata softvera, po čemu se oni razlikuju od razvijalaca koji

se bave drugim aspektima projekta (na primer oblikovanjem korisničkog interfejsa, tehničkom pripremom dokumentacije, pravljenjem i uređivanjem ilustracija, video ili zvučnih zapisa i slično).

U skladu sa ranije usvojenim značenjem termina razvoj softvera, u daljem tekstu ćemo podrazumevati da je razvijalac softvera svako ko se bavi poslovima projektovanja i implementiranja softvera, kao i poslovima koji sa njima imaju dodira.

Metodologije

U razvoju softvera se koriste mnoge tehnike i aktivnosti, ali ispada da one nisu ni idealne ni dovoljne, pa se zato neprekidno usavršavaju i razvijaju nove. Veliki broj alata, tehnika i aktivnosti otežava snalaženje među njima i odabiranje odgovarajućih metoda za rešavanje nekog konkretnog problema. Dodatni problem je što pri izboru načina rešavanja jednog problema moramo da vodimo računa i o tome kako ćemo rešavati neke druge probleme u sklopu istog projekta, zato što će njihovi ulazni sadržaji i rezultati morati da se usklađuju.

Da bi se olakšalo donošenje pojedinačnih odluka tokom rada na projektu, uobičajeno je da se na samom početku razvojnog procesa, najpre ustanove neki skup principa rada i odgovarajući skupovi metoda i procesa, koje ćemo kasnije dosledno primenjivati tokom čitavog projekta. Takvo lokalizovanje donošenja nekih značajnih odluka na samom početku projekta ima i dobre i loše strane. Naravno, dobro je što ćemo kasnije lakše i brže da donosimo odluke; dobro je i što će biti lakše da se usklađuju metodi i procesi između različitih delova projekta; ali loše je što onda moramo da napravimo izbor već na samom početku projekta, kada možda još uvek nismo svesni svih pojedinačnih karakteristika problema sa kojim se suočavamo. Štaviše, dok pri rešavanju pojedinačnih problema možemo da biramo metode i procese iz nekog manjeg skupa onih koji se odnose na tu vrstu problema, pravljenje opšteg odabira na počeku projekta predstavlja daleko teži i obimniji posao.

Prirodno se postavlja pitanje da li bismo mogli da ponovimo sličnu stvar – kao što smo pokušali da olakšamo izbor metoda za pojedinačne probleme, prebacujući ih na nivo projekta, možda bismo mogli da olakšamo i izbor na nivou projekta tako što bismo ga napravili objedinjeno za veći skup projekata? Iz takvog pristupa izrasta koncept metodologije.

Metodologija razvoja softvera (ili *razvojna metodologija*) je preporučeni skup metoda i procesa, koji su međusobno usklađeni i omogućavaju uspešnu realizaciju razvojnog projekta.

Metodologija predstavlja okvir u kome se odvijaju sve razvojne aktivnosti, od planiranja razvojnog procesa i odabiranja načina rešavanja pojedinačnih problema, pa sve do puštanja softvera u rad. Osnovna uloga metodologije je da olakša sistematičan pristup razvoju softvera, tako što na određeni način sužava izbor i preporučuje puteve i alate kojima se valja služiti. Metodologija može da bude veoma stroga i da precizno određuje i redosled koraka i način njihovog odvijanja i svaki

pojedinačan alat. Nasuprot tome, postoje i manje stroge metodologije, koje propisuju samo neke okvirne principe kojih bi trebalo da se pridržavamo tokom razvoja, a ostavljaju relativno veliku slobodu pri izboru konkretnih alata. Takve metodologije se uobičajeno nazivaju *okvirnim metodologijama*, pa se čak kaže i da nisu *prave* metodologije ili da su *nekompletne*.

Razvoj i oblikovanje metodologija je vid spoja računarskih nauka i softverksog inženjerstva. Tokom istorije računarstva nastalo je mnogo različitih metodologija. Svaka od njih je osmišljena da bi u datim okolnostima neka vrsta problema mogla da se reši bolje nego do tada postojećim metodologijama. Vremenom su se menjale okolnosti u kojima se razvija softver, pa su se menjale i metodologije. Neke od najvažnijih vrsta metodologija su procesne, strukturne, objektno-orijentisane i agilne.

Procesne metodologije su u centar pažnje stavljale životni ciklus softvera i posvećivanje procesima u domenu. Modeliranje procesa je na kraju imalo za rezultat algoritme, na osnovu kojih su se pravili programi. Životni ciklus je imao različite oblike, od kojih je najpoznatiji tzv. *model vodopada*[3], po kome su se ključni poslovi (na primer: istraživanje, analiziranje, planiranje, projektovanje, implementiranje, testiranje, pripremanje za rad) odvijali u definisanom redosledu i u jasno razdvojenim fazama, tako da je projekat prolazio kroz te faze u jednom smeru, kao što voda teče niz vodopad.

Strukturne metodologije su prepoznavale da se softverski sistemi grade nad nekoliko vrsta strukturnih elemenata (subjekti, procesi, podaci i interfejsi) i usmeravale su razvoj tako da u centru pažnje bude ostvarivanje dobre unutrašnje strukture sistema. Imale su dosta sličnosti, ali i razlika, u odnosu na procesne metodologije, pa su vremenom nastale i tzv. kombinovane metodologije, koje su predstavljale kombinaciju procesnih i strukturnih.

Objektno-orijentisane metodologije (OOM) su nastale kao vid nadgradnje objektno-orijentisanog programiranja (OOP). Kao što je kod OOP akcenat na opisivanju objekata i klasa u domenu programa, tako je kod OOM akcenat na opisivanju objekata i klasa u domenu problema. Teži se da se svi elementi posmatranog problema opišu u terminima objekata, klasa, metoda i interfejsa, sa idejom da se tako, uz primenu enkapsulacije, objedinjeno opisuju i struktura i procesi. Na sličan način se modeliraju posmatrani problemi u prostoru domena i implementacija u prostoru rešenja. Ovakve metodologije su danas veoma zastupljene u praksi. Veliki broj novih projekata se zasniva na primeni neke od objektno-orijentisanih metodologija.

---

[3] Čak je toliko poznat da se često koristi i pogrešan termin *metodologija vodopada*. Ovaj termin nije dobar zato što je životni ciklus vodopada zastupljen u više različitih metodologija, t.j. ne podrazumeva neki konkretan skup metoda.

Agilne metodologije su nastale usled potrebe da se dinamika razvoja softvera prilagodi dinamici promena u domenu za koji se softver pravi. Osnovni problem sa prethodnim metodologijama je bio što razvoj softvera može da traje veoma dugo, a u međuvremenu mogu da se promene okolnosti, pa usled toga i ciljevi i zahtevi koji se postavljaju pred razvojni projekat. Agilne metodologije počivaju na pretpostavci da će se zahtevi menjati i sve ostale aspekte u značajnoj meri podređuju toj pretpostavci. Neke od agilnih metodologija predstavljaju samo okvirne metodologije, pa se u praksi primenjuju uz kombinovanje sa drugim metodologijama (najčešće OOM), iz kojih se preuzimaju metodi i procesi.

Jedna od najvažnijih karakteristika agilnih metodologija je da su dobro prilagođene održavanju softvera. Štaviše, one kao jedan od važnijih kriterijuma kvaliteta prepoznaju lakoću održavanja softvera, odnosno njegovu trajnost. Za razliku od fizički opipljivih vrsta proizvoda, softver nije podložan starenju. Građevine, mašine i računarski hardver mogu da se oštete pod uticajem spoljašnjih faktora i mogu da se troše dok obavljaju svoju funkciju, pa se sa protokom vremena lakše i češće kvare. Softver ima sasvim drugačiju prirodu, pa na njega protok vremena nema neposrednog uticaja. Ali ako softver ne stari, to ne znači da je večan. Softver se ne troši, ali mu kvalitet ipak postepeno opada [*Pressman 2020*]. Vremenom se menjaju uslovi u kojima se softver upotrebljava i njegova vrednost postepeno opada – menjaju se računarski sistemi na kojima se koristi, ili se menja oblik ulaznih ili izlaznih podataka, ili se menjaju spoljašnje okolonosti tako da korisnici od njega očekuju više nego što on može da ponudi. Zbog toga softver povremeno mora da se usklađuje sa potrebama i okolnostima – tj. da se *održava*. Svaki pojedinačan ciklus održavanja softvera je veoma sličan razvojnom procesu – potrebno je prvo da se isplanira šta će da se menja, pa da se isprojektuje i implementira i na kraju da se proveri da li je sve urađeno u skladu sa planovima i potrebama.

Često se upotrebljava i termin *klasične metodologije*, ali sa dva različita značenja. Kada se govori o agilnim metodologijama, termin klasične metodologije se obično odnosi na sve metodologije koje nisu agilne. Međutim, ovaj termin se često upotrebljava i da predstavi sve metodologije koje su prethodile objektno-orijentisanim metodologijama. Razlika između ova dva značenja je samo u tome da li se OO metodologije koje nisu agilne smatraju za klasične ili ne. Uporedo sa ovim terminom koristi se i termin *tradicionalne metodologije*, sa istim značenjima.

Metodologije razvoja softvera ne bi trebalo da mešamo sa metodologijama i tehnikama programiranja. Ipak, to se ponekad čini, pa se u razvojne metodologije ponekad pogrešno svrstavaju i različite paradigme programiranja ili modeli projektovanja (npr. funkcionalno programiranje, razvoj vođen testiranjem, razvoj vođen događajima i drugo). Često se pogrešno izjednačavaju i pojmovi „objektno-orijentisana metodologija“ i „objektno-orijentisano programiranje“. Kao što smo već opisali, razvojna metodologija određuje principe i tehnike koje se koriste kroz čitav razvojni proces. Metodologija ili paradigma programiranja se, sa druge strane,

odnose samo na principe i tehnike koje se koriste pri programiranju, tj. pri implementiranju softvera. Na primer, OO metodologije počivaju na upotrebi OO koncepata u celom razvojnom procesu i na svi nivoima rada: prave se objektni modeli pri definisanju zahteva, opisivanju domena, definisanju arhitekture, specificiranju poslova i zadataka, pa sve do isporučivanja softvera. Sve celine posla se posmatraju po objektnom modelu uz poštovanje principa enkapsulacije i jasno definisanog interfejsa. Nasuprot tome, OO programiranje se odnosi striktno na programiranje, tj. na implementaciju određenih delova softvera primenom OO programskih jezika i tehnika. Štaviše, primena OO metodologije ne znači da programski kod mora da se implementira isključivo primenom OO programiranja. U savremenom razvoju softvera se veoma često primena OO metodologija kombinuje sa upotrebom različitih paradigmi za implementaciju različitih delova softvera, pa se često koriste ne samo OOP već i funkcionalno programiranje, razvoj vođen testovima i drugo.

Polećemo!

Ova knjiga se bavi izabranim temama iz oblasti razvoja softvera. Najviše pažnje je posvećeno problemima projektovanja i implementiranja softvera, ali je značajan deo prostora posvećen i nekim opštijim metodološkim pitanjima.

Danas se smatra da je neophodno da praktično svi razvijaoci softvera raspolažu solidnim poznavanjem osnovnih tema iz oblasti strukturnog projektovanja. Zato je oblast projektovanja softvera zastupljena u obimu koji omogućava čitaocima da naprave prve korake u toj oblasti i da zahvaljujući stečenim znanjima budu u stanju da oblikuju i pišu kvalitetnije programe. To svakako nije dovoljno da bi neko mogao da se nazove projektantom softvera, ali bi trebalo da posluži kao odskočna daska za dalje usavršavanje.

Preplitanje različitih metoda i procesa u savremenom razvoju suočava razvijaoce sa obavezom da dobro razumeju sve različite aspekte posla koji obavljaju, a da bi se dobro razumelo kombinovanje aspekata razvoja, kao i da bi se razumeo širi kontekst razvoja softvera, neophodno je poznavanje osnovnih metodoloških tema.

Predstavljeni su izabrani metodi i tehnike implementiranja softvera. Jedan od važnijih kriterijuma izbora je bio da su svi ti metodi i pripadajuće tehnike relativno bliski sa staranjem o kvalitetu softvera. Sa druge strane, izloženi sadržaji su povezani i međusobno i sa izloženim temama iz oblasti projektovanja softvera, tako da predstavljaju jednu zaokruženu i relativno čvrsto povezanu celinu.

Razvoj softvera zahteva disciplinu i sistematičnost u radu. Zbog toga neke od tehnika mogu da izgledaju kao dosadne liste pravila ili koraka koje je potrebno naučiti i zatim preduzimati. Međutim, svaki razvojni projekat je priča za sebe, sa svojim specifičnim problemima i prioritetima, zbog čega mnoge tehnike i principi ne mogu da se na isti način primenjuju u svakom kontekstu. Zato cilj izučavanja predstavljenih tehika i principa nije da se odgovarajuća pravila ili postupci nauče

napamet i zatim dosledno i nekritički primenjuju, nego da se razume zašto i kako su ta pravila i postupci oblikovani i kako možemo da ih prilagođavamo specifičnostima konkretnih projekata.

Pored aktivnosti koje čine razvoj softvera, softversko inženjerstvo obuhvata i brojne specifične poslove, kao što su poslovi planiranja i kontrole kvaliteta celovitog proizvoda, poslovi oko konfigurisanja i puštanja softvera u rad, brojni tehnički poslovi na usklađivanju aktivnosti, poslovi staranja o resursima i vođenja projekata i drugo. U ovoj knjizi se uglavnom nećemo baviti takvim specifičnim činiocima softverskog inženjerstva. Za temeljnije izučavanje ovih tema, ali i softverskog inženjerstva kao celovite i zaokružene oblasti, upućujemo čitaoce, pre svega, na [*Pressman 2020*], a zatim i na [*Sommerville 2016*, *Pfleeger 2006*, *Popović 2019*].

2 - Problemi pri razvoju softvera

Ne paničite!
Daglas Adams

2.1 Problemi i neuspesi pri razvoju softvera

U razvoju softvera je problem sve ono što negativno utiče na razvojni proces i krajnji rezultat. Problemi mogu da otežavaju posao, da ga produžavaju, da mu povećavaju cenu, da nepotrebno zamaraju razvijaoce, da negativno utiču na odnose u razvojnom timu, da prouzrokuju niži kvalitet rezultata, pa čak i da u potpunosti obesmisle razvoj konkretnog proizvoda. Problemi vrlo retko mogu da se u potpunosti eliminišu, ali uz ulaganje dovoljno kvalitetnog rada na sprečavanju i prevazilaženju problema, njihove posledice mogu da se značajno umanje ili čak učine zanemarljivim.

Sa druge strane, ako se potencijalnim problemima ne posveti dovoljno pažnje, onda oni mogu da dovedu i do značajnih negativnih posledica po razvojni proces ili rezultat razvoja. Negativne posledice problema nazivamo neuspehom. Grubo posmatrano, neuspeh je svako nepovoljno odstupanje od plana. Uobičajeno je da se kao osnovno merilo neuspeha posmatra izgubljena materijalna vrednost. Ona se najčešće prepoznaje kroz prekoračenje predviđenog obima ulaganja sredstava ili uloženog vremena, ali ponekad i kroz posledice koje se odnose na čitav poslovni sistem. U skladu sa tim, obično se prepoznaju sledeće osnovne vrste neuspeha:

- prekoračenje troškova;
- prekoračenje vremenskih rokova;
- neupotrebljivost rezultata i
- odustajanje od projekta.

Osnovni oblik izgubljene materijalne vrednosti predstavlja neplanirano povećanje obima uloženih sredstava, tj. prekoračenje predviđenih troškova. Skoro svaki problem koji nastupi tokom razvoja, ima za posledicu potrebu da se deo nekog posla popravlja, ponavlja ili proširuje, a svako povećavanje obima posla podrazumeva i više radnih sati i više uloženih sredstava.

Povećanje obima posla nekada može da se kompenzuje povećanjem broja angažovanih učesnika u razvoju, tako da pored izgubljenih sredstava ne mora da dođe i do izgubljenog vremena i kašnjenja realizacije projekta. Ipak, vrlo često nije moguće da se dodatni radni sati uklope u planirani raspored poslova, pa se kao dodatna posledica pojavljuje i kašnjenje.

Pored već prepoznatog povećanja troškova usled angažovanja dodatne radne snage, izgubljeno vreme može da prouzrokuje dodatne troškove i iz raznih drugih poslovnih razloga, od produženog angažovanja prostora i sredstava za rad, pa do neugodnih posledica koje korisnik softvera može da ima zbog njegovog kašnjenja. Zakasnelo puštanje softvera u rad i nepredviđeno povećanje troškova ili redukovana upotrebljivost proizvoda mogu ne samo da naprave velike neprijatnosti, već i da u posebno ozbiljnim slučajevima proizvedu kritične posledice po ceo poslovni sistem za koji se softver razvija. Te posledice mogu da idu toliko daleko da obesmisle dalji rad na softveru, ili čak da dovedu u pitanje opstanak planiranog poslovnog poduhvata ili čak čitavog poslovnog sistema za koji se softver razvija.

Čak i kada je softver završen u skladu sa planovima (ili uz eventualno prihvatljivo probijanje rokova ili budžeta), može da se dogodi da ne može da se upotrebi i da je čitav razvoj bio uzaludan. Neupotrebljivost je obično posledica neispunjenosti nekog od ključnih funkcionalnih ili nefunkcionalnih zahteva ili prisustva veoma ozbiljnih bagova. Uobičajen uzrok takvih problema je loše praćenje projekta, tj. izostanak kvalitetnog nadzora i kontrole.

Najčešći oblici neupotrebljivosti razvijenog softvera obuhvataju:

- nesaglasnost zahteva po kojima je softver implementiran sa stvarnim potrebama;
- neispunjenost funkcionalnih zahteva;
- neispunjenost nefunkcionalnih zahteva;
- katastrofalni bagovi;
- nemogućnost da se sistem pusti u rad;
- neupotrebljivost korisničkog interfejsa i
- neostvarivost procedure korišćenja u realnim uslovima.

Možda izgleda čudno, ali relativno često se dešava da je softver razvijen u potpunosti u skladu sa planovima i funkcionalnim zahtevima, da nema značajnih

bagova, a da ipak ne može da se primeni. Da problem bude još neugodniji, do toga najčešće dolazi kod veoma velikih projekata. Razlog za to je obično u promenljivosti poslovnih procesa i dugotrajnosti razvoja složenih softverskih sistema – ako se softver razvija nekoliko godina, prema unapred definisanoj specifikaciji, sasvim je moguće da u međuvremenu te specifikacije prestanu da odgovaraju realnim potrebama poslovnog sistema i da softver samim tim više ne može da se primeni u planiranom obliku.

Loš korisnički interfejs i procedure korišćenja koje nisu prilagođene realnim praktičnim uslovima upotrebe takođe mogu da dovedu u pitanje upotrebljivost softvera. Loš korisnički interfejs po pravilu usporava upotrebu softvera i prouzrokuje veći zamor korisnika, a time i veći broj grešaka koje nastaju pri upotrebi softvera. Neprilagođene procedure korišćenja obično su posledica neispravnih pretpostavki o uslovima upotrebe softvera ili stručnosti i mogućnostima korisnika.

Jedan od značajnih uzroka neupotrebljivosti gotovih projekata predstavljaju propusti u vezi sa puštanjem sistema u rad. Složeni softverski sistemi se danas najčešće razvijaju sa ciljem da zamene neka postojeća rešenja. U takvim okolnostima puštanje novog sistema u rad nije sasvim jednostavno i može da zahteva prilično složene poslove prebacivanja podataka iz starog sistema u nov, isključivanje starog i uključivanje novog sistema po delovima i slične postupke. Međutim, u osetljivim radnim okruženjima dodatnu otežavajuću okolnost može da predstavlja potreba da poslovanje ne sme da bude prekinuto zbog puštanja novog sistema u rad. Ako pri planiranju novog sistema nije dovoljno dobro i detaljno isplaniran postupak puštanja u rad, onda u toj fazi može da dođe do značajnih problema, koji često zahtevaju veće modifikacije softvera, nekada i toliko obuhvatne da dovedu u pitanje smisao celog projekta.

2.2 Primeri neuspeha

Neuspesi nisu lepa tema, a oni koji ih dožive ili čak prouzrokuju, ne ponose se mnogo njima, pa se zato o neuspesima ne govori često. Ipak, s vremena na vreme neki od neuspešnih slučajeva razvoja softvera dospe u naučnu ili stručnu literaturu, ili bude predstavljen javnosti putem medija. Ovde ćemo istaći neke od zanimljivih nalaza istraživanja, koje je sprovela Stendiš grupa 1994. godine [*Standish Group 1994*], a čitaocima sugerišemo da to istraživanje pročitaju u celosti:

- Prosečno godišnje ulaganje u razvoj primena IT u SAD je iznad $250.000.000.000 i to u prosečno 175.000 IT projekata.
- Oko 31% projekata je obustavljeno pre završetka.
- Oko 52.7% „uspešnih“ projekata je proizvelo prekoračenje početnog budžeta za oko 89%.

- Dodatni troškovi, koji ne čine neposredna ulaganja u razvoj, već su posledica kašnjenja ili neuspeha projekta, procenjuju se na bilione dolara (hiljade milijardi).
- Procenjuje se da je utrošeno oko $81.000.000.000 na obustavljene projekte.
- Procenjuje se da je utrošeno oko $59.000.000.000 na dodatne neplanirane troškove projekata koji su dovršeni.
- Samo 16.2% projekata je završeno na vreme i bez probijanja planiranog budžeta.
- Samo 9% projekata u velikim kompanijama je završeno na vreme i bez probijanja planiranog budžeta.

Jedna novija analiza [*Standish Group 2015*] nam sugeriše da je uspeh projekata (završetak uz poštovanje rokova i plana troškova) obrnuto proporcionalan njihovoj veličini. U studiji se ispostavilo da je od posmatranih projekata bilo uspešno tek oko 58% malih agilnih i 44% malih klasičnih projekata, kao i tek oko 18% velikih agilnih i 3% velikih klasičnih projekata. Sličan je i odnos složenosti i uspešnosti.

U skladu s tim, prepoznaju se tzv. *pobednički uslovi*, kao mali projekti sa iskusnim razvojnim timovima i savremenim razvojnim metodima, ali i *gubitnički uslovi*, kao veliki projekti sa neiskusnim timovima i prevaziđenim razvojnim metodima.

Istraživanje o razlozima neuspešnosti, sprovedeno na 52 IT projekta u oblasti zdravstva u Saudijskoj Arabiji [*Abouzahra 2011*] pokazuje da su najčešći uzroci neuspeha:

- nedefinisani okviri projekta (44%);
- neprepoznati rizici (30%);
- neprepoznati ulagači[4] (14%);
- loša komunikacija (7%) i
- drugi razlozi (5%).

[4] Pod ulagačima (engl. *stakeholder*) se podrazumevaju sva fizička i pravna lica koja su na neki način zainteresovana za rezultat razvojnog projekta. To su najčešće konkretni klijenti, koji su oblikovali ciljeve ili uložili određena sredstva u projekat, ali mogu da budu i svi drugi potencijalni korisnici softvera, pa čak i lica na koja softver ima samo posrednog uticaja.

Ekstremni primeri slučajeva neuspeha se navode u radu [*Charette 2005*]:

- Kompanija *FoxMeyer Drug Co.* iz Teksasa je 1996. godine bankrotirala zbog loše implementacije sistema za planiranje resursa. Vrednost kompanije je neposredno pre toga bila procenjena na $5.000.000.000.
- Federalna uprava za vazduhoplovstvo SAD je radila na razvoju sistema za kontrolu letova od 1981. do 1994. godine, kada se odustalo od projekta posle ulaganja od $2.600.000.000, što je bilo trostruko više od planiranih troškova. Ukupni gubici zbog neostvarenog unapređenja poslovanja se procenjuju na više od $50.000.000.000.

U istom radu je napravljena i dobra analiza razloga ovih (i drugih) neuspeha, pa ga toplo preporučujemo čitaocima.

2.3 Uzroci problema

Uzroci problema se obično dele na uzroke na strani klijenta (tj. naručioca i budućeg korisnika softvera), uzroke na strani razvijalaca softvera i uzroke koji se odnose na obe strane. Ovakvu podelu pravimo samo radi lakšeg sagledavanja uzroka i njihovih međusobnih odnosa, a ne i da bismo sugerisali ko je odgovoran za rešavanje problema.

Iako sami uzroci problema mogu da se nalaze na strani klijenta, pa čak može da bude sasvim očigledno da je klijent u potpunosti odgovoran za njihovo nastajanje, ipak moramo da imamo u vidu da je odgovornost za uočavanje i prevazilaženje problema nastalih tokom razvoja softvera praktično uvek na strani razvijalaca i posebno na strani rukovodilaca razvojnog tima. Naime, ako je neke probleme klijent sam uočio i rešio, onda razvojni tim obično nije ni imao priliku da prepozna takve probleme i njihove posledice. Nasuprot tome, ako neke probleme klijent nije primetio, onda će upravo razvojni tim biti u prilici da se sa njima suoči. Zbog toga razvojni tim mora uvek da bude na oprezu i da se trudi da blagovremeno uoči probleme na strani klijenta, pre nego što njihove posledice počnu da se u značajnijem obimu odražavaju na razvojni proces.

Uzroci na strani klijenta

Najčešći uzroci problema koji se prepoznaju na strani klijenta su:

- nerealni ili nejasni ciljevi projekta;
- neusklađenost ciljeva i strategije;
- politika ulagača;

- komercijalni pritisak i
- otpor korisnika prema primeni novog softvera.

Nerealni i nejasni ciljevi projekta se odnose na sve one slučajeve kada ne postoji prepoznat cilj koji je ostvariv u rokovima i sa troškovima koje klijent može i želi da podnese. Savremene informacione tehnologije su svojom sveprisutnošću stvorile u delu šire javnosti utisak da sve može da se napravi i da svaki cilj može da se ostvari. To može da podstakne potencijalne investitore da pokrenu razvojne projekte bez dovoljno dobro sprovedene prethodne razrade i bez jasno definisanih ciljeva.

Osnovni problem sa nepreciznim i nerealnim ciljevima je to što oni imaju tendenciju da se tokom vremena menjaju. Njihovim postepenim menjanjem na kraju može da se dođe do jasnijih i realnijih ciljeva, ali je problem što promene ciljeva obično povlače za sobom velike promene funkcionalnih zahteva. Svaka promena zahteva u toku razvoja dovodi do postavljanja pitanja da li su time poslovi koji su već obavljeni i rezultati rada koji su do tada ostvareni možda postali pogrešni ili nepotrebni, pa tako preti da praktično otpiše deo već uloženih sredstava i proizvede značajne gubitke.

Sličan problem nastaje i ako na strani klijenta ne postoji jedinstven stav o strategiji razvoja, pa se tako dešava da jedan sektor kompanije postavi vrlo jasne i utemeljene ciljeve razvojnog IT projekta (bar na lokalnom nivou, iz ugla tog sektora) i uloži sredstva da bi se on ostvario, a da zatim na višem nivou odlučivanja dođe do nekih strateških odluka koje potpuno obesmisle čitav projekat. Takvi problemi nisu retkost u poslovnim okruženjima u kojima nisu dovoljno jasno definisane nadležnosti i ne postoji dobra komunikacija između delova kompanije. Tu možemo da primetimo vrlo neprijatan paradoks – takvo stanje obično nastupa u brzo rastućim kompanijama, a one čine možda najvažnije investitore u oblasti informacionih tehnologija.

Neujednačen stav o projektu među različitim subjektima na strani klijenta može da proizvede različite vrste problema, od nepreciznih specifikacija i čestih promena zahteva, pa sve do neupotrebljivosti implementiranog softvera. Neujednačen stav može da bude posledica, između ostalog, neposvećenosti rukovodilaca, nedovoljne komunikacije sa budućim korisnicima ili veštačke integracije više potencijalnih projekata u jedan (obično radi zamišljene uštede), kao i nekih drugih razloga. Sve ove probleme prepoznajemo kao probleme koji potiču iz politike ulagača.

U vezi sa prethodnim je i potencijalni otpor dela budućih korisnika prema upotrebi razvijenog softvera. Takav otpor je često posledica predrasuda, ali u nekim slučajevima predstavlja i racionalan ili iracionalan otpor prema promenama koje na neki način narušavaju položaj zaposlenih ili uslove rada. Uopšteno posmatrano, svako uvođenje informacionih tehnologija u poslovne procese dovodi do bar dve stvari koje zaposleni ne vole: do veće kontrole (najpre do boljeg uvida u odvijanje aktivnosti, a time i do veće kontrole zaposlenih) i do promena u načinu obavljanja

poslova (a time i neophodnosti učenja novih postupaka). U dobro organizovanim sredinama taj otpor može da se usmeri u pozitivnom smeru uključivanjem šireg skupa zaposlenih u proces planiranja, tako da se i bolja kontrola i promene u načinu rada prepoznaju kao unapređenje uslova rada, a ne kao njihovo narušavanje.

Svaki komercijalni razvojni projekat, pa i ulaganje u razvoj softvera, pokreće se da bi se ostvario profit. Iz različitih razloga (na primer, zbog finansijskog neuspeha nekog drugog projekta, zbog nepoverenja u razvojni tim i mnogo drugih razloga) u toku razvoja može da se pojavi pritisak da se planirani profit, ili bar njegov deo, ostvari što pre. U takvim okolnostima vrlo često se donose pogrešne odluke, poput menjanja redosleda implementiranja delova softvera, nerealnog smanjivanja rokova, smanjivanja ili potpunog preskakanja nekih vidova kontrole kvaliteta i drugo. Naravno, najbolje je da se projektni tim odupire takvim pritiscima, ali u realnim uslovima to često nije moguće, pa se reakcija razvojnog tima često svodi na pokušaje kontrole štete, tj. pokušaje da se insistira na doslednoj implementaciji najvažnijih delova softvera, a da se uštede (a time i eventualni problemi) ograniče na manje osetljive delove softvera.

Uzroci na strani razvojnog tima

Kada se posmatraju uzroci problema na strani razvijalaca softvera, njihova zajednička karakteristika je da su praktično u potpunosti u domenu rukovođenja razvojnim procesom. U najčešće uzroke se ubrajaju:

- slabo vođenje projekta;
- neprecizna procena potrebnih resursa;
- slabo izveštavanje o stanju projekta;
- neupravljani rizici;
- upotreba „nezrelih“ tehnologija;
- nesposobnost da se iznese složenost projekta i
- tok razvoja bez čvrstih principa i pravila.

Svaki od ovih uzroka nastaje kao posledica nestručnosti ili lošeg rada rukovodilaca razvojnog tima. Doduše, u nekim slučajevima (na primer, u slučaju upotrebe nezrele tehnologije koja je nametnuta od strane klijenta) prostor za reagovanje može objektivno da bude sužen, pa je onda i lična odgovornost nešto niža.

Najčešći uzroci problema koji se nalaze i na strani klijenta i na strani razvijalaca softvera su:

- slaba komunikacija između klijenta, razvijaoca i korisnika;

- nepoverenje između klijenta i razvijaoca i
- loše definisani zahtevi.

Kao što smo videli, svaki od uzroka na strani razvojnog tima ima korene među rukovodiocima razvoja, ali i ostali problemi često mogu da se na vreme prepoznaju i otklone od strane rukovodioca razvoja, pa zato svaka priča o problemima obično počinje i završava se u okviru rukovodećeg dela razvojnog tima. U praksi, iza svakog neuspeha obično stoji više uzroka. Pojedinačni uzroci problema obično nemaju kapacitet da upropaste projekat, ali ako se nezgodno složi više uzroka, onda ishodi mogu da budu veoma nepovoljni. Tim pre je važno da se praćenju projekta i sagledavanju mogućih rizika posvećuje puna pažnja tokom čitavog trajanja projekta. Svako kašnjenje u uočavanju problema ili reagovanju na njih može da ima loše posledice.

Loše planiranje

Jedan od najčešćih uzroka problema i neuspeha je loše planiranje. Loše planiranje može da nastupi u bilo kojoj fazi razvoja, pa se tako loše planiranje u početnim fazama razvoja obično ispoljava u obliku nerealnih ili nejasnih ciljeva projekta ili loše definisanih zahteva, a u kasnijim fazama kao slabo vođenje projekta ili neprecizna procena potrebnih resursa. Loše planiranje se obično izjednačava sa nedovoljnim planiranjem, ali to je ozbiljan previd, zato što je nedovoljno planiranje samo jedan od oblika lošeg planiranja – drugi oblik, a koji može da ima jednako neugodne posledice, jeste preterano planiranje.

Nedovoljno planiranje je obično posledica nedovoljno dobro sprovedene analize zadatka ili loše ili neprecizno definisanih zahteva. Da bi mogao da se napravi dobar plan razvoja, neophodno je da se prethodno dobro analizira problem koji pokušava da se reši razvijanim softverom. Analiza problema mora da bude dovoljno široka i duboka da pruži odgovore na sva pitanja iz domena problema, koji su potrebni da bi se napravili projekat softvera i plan razvoja. Posledice nedovoljno dobre analize su neuočavanje nekih značajnih elemenata domena koji utiču na rešavanje problema, previđanje nekih poslova koje je neophodno uraditi ili nekih koraka kojima se obezbeđuju neophodne ulazne informacije, neprepoznavanje specifičnih vrsta subjekata koji će imati različita očekivanja od softvera i drugo. Kada projekat softvera nastane na osnovu nedovoljno dobre analize, obično iz njega izostanu neki neophodni koncepti ili komponente, ili neki zastupljeni koncepti ili komponente počivaju na nižem nivou apstrakcije nego što je potrebno, što kasnije otežava dogradnju izostavljenih funkcija.

Neprecizno definisani zahtevi su obično posledica „podrazumevanja trivijalnih stvari“ od strane projektanta. Moramo da priznamo da navođenje trivijalnih stvari zaista može da nepotrebno optereti projekat, ali i da primetimo da je procenjivanje „trivijalnosti“ veoma subjektivno i da može da odvede projekat u lošem smeru. Čak i

ako je projektant možda već pravio mnogo sličnih projekata, svejedno mora da ima u vidu da neki od razvijalaca možda nemaju takvo iskustvo i da zbog toga ne mogu da razumeju (odnosno da podrazumevaju) ono što nije napisano u okviru specifikacije zahteva. Projektna dokumentacija je sredstvo za komunikaciju članova razvojnog tima i svaka nepotpunost te dokumentacije preti da dovede do nerazumevanja između članova tima, a samim tim i do različitih problema koji iz tog nerazumevanja mogu da proisteknu.

U najvažnije posledice nedovoljnog planiranja spadaju nesrazmerno veliki broj naknadnih izmena zahteva, slaba upotrebljivost rešenja i probijanje rokova i troškova. Ako se slabosti u zahtevima ne uoče na vreme, onda može da se dobije softver koji nije funkcionalan i koji u praksi ne može da se upotrebi u punoj meri i u planiranom obliku. Sa druge strane, ako se uoče, onda će biti neophodne naknadne korekcije. Naknadne promene ili dopune zahteva obično proizvode dodatne poslove, zato što novonastalim izmenama mora da se prilagođava sve što je počivalo na prethodnim verzijama zahteva, uključujući projekte, planove, a često i implementacije nekih delova softvera. To dalje utiče na povećavanje troškova i trajanja razvoja.

Možda izgleda paradoksalno, ali relativno čest uzrok problema u razvoju softvera je neki vid preteranog planiranja. Preterano planiranje se obično ogleda kroz:

- preširoko i nekoncentrisano ulaženje u projekat;
- suviše duboku i obimnu analizu sa ranim detaljnim projektovanjem;
- preširoko ulaženje u implementaciju i
- preambicioznost, nesrazmernu realnim mogućnostima (tzv. pozlaćivanje).

Preterano planiranje može da proizvede projektnu dokumentaciju velikog obima, na primer i do nekoliko hiljada stranica. Veliki obim dokumentacije otežava uočavanje važnih informacija i lako može da se dogodi da se zbog toga naprave neki suštinski propusti. Neke od najčešćih posledica preteranog planiranja su kašnjenje u uočavanju propusta i otežana tranzicija. Što je neka projektna dokumentacija veća, to je teže sagledati sve njene aspekte. Zbog velikog obima dokumentacije razvijaocima je veoma teško da pri razvoju nekih elemenata uvek uzimaju u razmatranje i sve ostale povezane elemente, pa je im je zbog toga teško i da ispravno implementiraju i da ispravno testiraju implementirano. Zbog toga se dešava da se čak i neki konceptualni propusti uočavaju tek pri testiranju gotovog proizvoda, što je obično suviše kasno da bi popravke mogle da se uklope u rokove i budžet.

U vezi sa tim je i činjenica da je složenije softverske sisteme teže pustiti u rad. Tranzicija je uvek složena, pa je posebno važno da je ne činimo još složenijom ako to nije neophodno. Pozlaćivanjem projekta, njegovim nepotrebnim širenjem i imple-

mentacijom elemenata koji nisu posebno važni za korisnika ne gubi se samo na vremenu i sredstvima za razvoj, nego često i još mnogo više na vremenu i sredstvima za puštanje sistema u rad.

2.4 Prevencija problema

Analize najčešćih problema i uzroka njihovog nastajanja pokazuju da u najvažnije uzroke problema spadaju loša komunikacija razvijalaca i klijenata, loše planiranje i izmene koje nastaju nakon projektovanja, a pre dovršavanja softvera. Savremene razvojne metodologije pokušavaju da se suoče upravo sa tim problemima. One počivaju na konceptima i tehnikama koje omogućavaju pouzdaniji razvoj i obezbeđuju manji rizik nastajanja problema. U nastavku ovog poglavlja sagledaćemo neke od važnijih koncepata, dok ćemo izabrane tehnike razvoja softvera predstaviti u narednim poglavljima.

Pojačana komunikacija među subjektima

Procenjeno je da je jedan od osnovnih problema klasičnih metodologija relativno nizak nivo komunikacije između svih subjekata uključenih u projekat razvoja softvera, a prvenstveno između klijenta i razvojnog tima. Veoma često se dešavalo da je ta komunikacija bila intenzivna na početku projekta, u vreme planiranja i projektovanja, a zatim je praktično skoro potpuno izostajala tokom dugačkog perioda razvoja. Česta posledica je bilo značajno odstupanje izvedenog projekta od onoga što je klijent zaista želeo. To je moglo da se desi iz više razloga, a najčešće zato što bi se u međuvremenu promenila realnost poslovanja klijenta, a time i njegove potrebe u odnosu na projekat, ili bi neki segment projekta bio pogrešno protumačen od strane projektanata ili implementatora, a da klijent to nije shvatio u fazama planiranja ili ugovaranja posla.

Zato jedan od najvažnijih koncepata savremenog razvoja softvera predstavlja insistiranje na intenzivnoj i redovnoj komunikaciji među subjektima koji na bilo koji način učestvuju u razvoju softvera, a pre svega između klijenta i razvojnog tima. Cilj pojačavanja komunikacije je prevencija problema koji nastaju usled izostanka komunikacije, ali i stvaranje uslova za kvalitetniju saradnju i bolje rezultate rada.

Osnovna dobit od redovne komunikacije između klijenta i razvojnog tima je u blagovremenom uočavanju eventualnih odstupanja od klijentskih potreba i reagovanju na njih. Zaista, ako klijent u regularnim intervalima razmatra stanje razvoja softvera, onda može da na vreme doprinese ne samo uočavanju grešaka nego i eventualnom oblikovanju još boljih rešenja za neke segmente posla. Stalna komunikacija dodatno doprinosi porastu međusobnog poverenja između klijenta i razvojnog tima. Sa jedne strane, ako klijent vidi da se razvojni proces odvija po planu, onda će biti manje zabrinut za svoju investiciju, imaće više poverenja u razvijaoce i vršiće manji pritisak na njih. Sa druge strane, ako razvijaoci vide da je

klijent zadovoljan, onda će moći da rade sa manje rizika, u rasterećenijem okruženju. Na osnovu informacija koje dobiju od različitih subjekata sa klijentske strane, članovi razvojnog tima mogu da steknu tačniju sliku o potrebnim ciljevima, čime se smanjuje rizik razvoja neupotrebljivog rešenja. Slično, ako neka strana nije zadovoljna, bolje je da se to ustanovi i reši na vreme, nego da se nezadovoljstvo prikuplja do tačke pucanja projekta po šavovima.

Pojačana komunikacija podrazumeva i intenzivniju razmenu informacija između svih članova razvojnog tima, kao i između različitih timova koji učestvuju u razvoju. Jedan od ciljeva te komunikacije je distribuiranje informacija među članovima tima, tako da više članova tima može da se uključi u rešavanje eventualnih problema ali i da se lakše nadoknadi nečije eventualno odsustvo.

Stalna i intenzivna komunikacija nosi i neke rizike. Velika količina dostupnih informacija može da ima upravo suprotne efekte i da klijentu oteža sagledavanje stanja projekta, umesto da ga učini lakšim i jasnijim.

Takođe, pružanjem prilike da neprekidno utiče na tok i način rada, možemo nepripremljenog klijenta da dovedemo u situaciju da neracionalno velikim brojem sugestija za menjanje i unapređivanje planova i rezultata rada napravi suprotne efekte. Neodmereno i nestručno učešće klijenta može da dovede do usporavanja projekta i putem eventualno preteranog planirana ili *naduvavanja* čitavog projekta ili pojedinačnih koraka projekta. Poseban problem mogu da predstavljaju tzv. neprekidni nizovi izmena, kada klijent stalno iznosi nove zahteve za izmene, ne čekajući ni da prethodno izneseni zahtevi budu implementirani. Nasuprot tome, posvećivanjem prevelike pažnje stavovima konzervativnijih subjekata, koji su navikli na postojeće procese i ne sagledavaju dobro planirane izmene, može da se smanji obim suštinskih funkcionalnih izmena u okviru pojedinačnih razvojnog koraka, ali tako da se time nepotrebno uspori razvoj i poveća potreban broj razvojnih koraka na putu do finalnog proizvoda.

Uključivanjem budućih korisnika u razvojni proces omogućava se blagovremeno uočavanje i popravljanje eventualnih slabosti korisničkog interfejsa ili zamišljenih procedura upotrebe softvera. Budući korisnici se upoznaju sa novim softverom u njegovim ranim fazama i tako se postupno obučavaju za njegovu upotrebu. Na taj način klijent praktično od prvog dana razvoja može da započne na pripremama za upotrebu novog sistema.

Jedan vid pojačavanja komunikacije sa klijentom predstavlja pravljenje *prototipova*. Prototip obično predstavlja kostur rešenja iz koga, kroz relativno primitivno izveden korisnički interfejs i simulirano izvođenje poslova, može da se sagleda kako će softver ili deo softvera da funkcioniše kada bude završen. Iz ugla razvojnog tima prototip nekada može da izgleda kao suvišan posao, ali je njegova uloga u razvojnom procesu često nezamenljiva. On obično pruža samo *iluziju* upotrebe, ali u dovoljnoj meri da budući korisnici i klijenti mogu da steknu utisak o

načinu funkcionisanja budućeg softvera, neposrednije i tačnije nego što bi to mogli samo na osnovu projektne dokumentacije, koju obično i ne razumeju u potpunosti zato što nemaju dovoljno tehničkih znanja. Kada vide prototip i „osete pod prstima" kako će softver da radi, klijenti će moći da uspešnije uoče eventualne nedostatke i da konkretnije iznesu eventualne primedbe. Na taj način se smanjuje rizik od donošenja pogrešnih odluka u ranim fazama razvoja.

Ipak, i prototipovi nose neke rizike. Oni obično predstavljaju samo funkcionalne aspekte korisničkog interfejsa, a ne i unutrašnju strukturu sistema koji se razvija. To može da ima za posledicu da se klijent fokusira na korisnički interfejs i možda zapostavi neke važnije aspekte planiranog softvera. Istovremeno, prototipovi koji nisu dovoljno široki, mogu da prikriju nedostatke u delovima softvera koji njima nije predstavljen. Takođe, preterano posvećivanje prototipovima, na primer sa težnjom da izgledaju i vizualno i sadržajno kao gotovo rešenje, može da uzme nepotrebno mnogo vremena..

Savremene razvojne metodologije zahtevaju stalnu i temeljnu kontrolu tokom čitavog razvojnog procesa. Što je neki projekat veći i što više različitih ljudi učestvuje u razvoju, to je važnije da se svaki korak temeljno proverava. Pored proveravanja ispravnosti urađenih delova posla, u kontrolne procedure spada i dobro praćenje i izveštavanje o stanju projekta. Blagovremeno i dovoljno iscrpno izveštavanje o stanju projekta može da bude presudno za blagovremeno uočavanje propusta i da značajno smanji troškove zbog pravljenja odgovarajućih korekcija. I to je vid komunikacije među učesnicima u razvoju softvera.

Iterativni razvoj

Jedan od važnih koncepata savremenih razvojnih metodologija je *iterativni* ili *inkrementalni razvoj*[5]. Klasične razvojne metodologije su se odlikovale posmatranjem razvoja softvera kao celovitog proizvoda. Neke od njih su prepoznavale potrebu da se projekat razvija po delovima, ali je svejedno bilo uobičajeno da se u početnim fazama posla do detalja razmatra i projektuje čitav proizvod. Iterativni razvoj, umesto toga, promoviše podelu razvojnog procesa na više faza i podelu posla na više manjih celina, koje se i razmatraju i implementiraju tek u odgovarajućim fazama.

Iterativni pristup razvoju softvera se razlikuje od, na primer, građenja zgrade, gde se najpre sve detaljno isprojektuje, a onda se sve to i implementira. Umesto toga, posao se prvo deli na manje celine i onda se jedan po jedan deo softvera najpre projektuje pa onda implementira. Na početku projekta se, naravno, i dalje sagleda-

[5] Uobičajen je termin *inkrementalni razvoj*, ali on implicira inkrementalnu prirodu faza razvoja, tj. jedan specifičan pristup planiranju i ostvarivanju iteracija. U daljem tekstu ćemo videti da to nije jedini pristup, pa je bolje da se koristi opštiji termin *iterativni razvoj*.

vaju potrebni delovi softvera, kao i poslovi koje je potrebno uraditi. Razlika je u tome što se na početku posla prave samo relativno grube procene, dok se detaljno analiziranje i projektovanje svakog dela radi tek u odgovarajućoj fazi. U tom smislu, razvoj softvera više liči na izgradnju jednog velikog naselja nego na izgradnju jedne zgrade – iako na početku imamo nekakvu veliku sliku o tome šta će i kako biti napravljeno, celine se rade jedna po jedna, pa zato pojedinačni delovi lakše trpe veće izmene. Na taj način se skraćuje vreme koje protekne od planiranja do završetka svakog pojedinačnog dela, a time se smanjuje i rizik da u toku implementacije jednog dela dođe do većih izmena u specifikaciji. Naravno, manje celine je mnogo lakše planirati i praviti, pa se dobija i na smanjenju troškova.

Svaka iteracija može da bude inkrementalna, evolutivna ili kombinovana. *Inkrementalna iteracija* dodaje softveru novi deo (komponentu, inkrement). Nizom inkrementalnih iteracija softver postepeno raste od jedne komponente do celog proizvoda, nalik na pravljenje i slaganje kockica.

Nasuprot tome, *evolutivna iteracija* ne dodaje novi deo, već unapređuje neki već postojeći deo (komponentu) softvera. Nizom evolutivnih iteracija softver raste tako što se postepeno povećavaju funkcionalnost i složenost postojećih komponenti.

Kombinovana iteracija dodaje nove delove i istovremeno unapređuje neke već postojeće delove softvera. Inkrementalne iteracije takođe često moraju da obuhvate neka manja prilagođavanja postojećih delova programa, pa se zato za iteraciju kaže da je kombinovana samo ako pravi relativno značajne izmene u postojećem kodu.

Primarna očekivana korist od iterativnog pristupa je u smanjivanju trajanja razvojnog ciklusa i intenziviranju komunikacije između klijenta i razvojnog tima. Sekundarna dobit je povećana preciznost napravljenih procena, zato što se detaljne procene prave u manjem obimu i na kraće rokove, pa su zato i merodavnije. Uspostavljanje ritma redovnog isporučivanja novih verzija ili delova sistema omogućava blisku saradnju sa klijentom i brže uspostavljanje visokog stepena poverenja između klijenta i razvijalaca. Omogućavaju se i redovnija kontrola kvaliteta i postepeno privikavanje korisnika na nove elemente sistema.

Iterativni razvoj ima i nekih slabosti. Najpre, ako se u prvim iteracijama potpuno ili delimično zanemare predstojeće iteracije, onda postoji rizik da će u narednim iteracijama biti neophodno da se preduzmu nešto veće izmene već implementiranih delova softvera. Sa druge strane, ako se u prvim iteracijama uzmu u obzir i sve predstojeće iteracije, onda postoji rizik da se složenost tih prvih iteracija približi složenosti čitavog sistema („naduvavanje koraka“), čime ovakav pristup gubi smisao. Nezgodna karakteristika iterativnog razvoja je i to što često nije moguće da se unapred tačno sagledaju broj, cena i ukupno trajanje svih iteracija. Takođe, ako se vrši preterano učestalo isporučivanje novih iteracija, onda to može i da podigne cenu projekta, zato što svaka isporuka zahteva dodatno vreme i dodatni rad.

Specifičan oblik iterativnog razvoja je da se iteracije ne planiraju prema poslovima već prema rokovima. Određivanje koraka prema rokovima počinje od ustanovljavanja koliko dugo će neka iteracija da traje i koliko ljudi će na njoj da radi. Zatim se odabiru poslovi za tu iteraciju, prema značaju, mogućnosti povezivanja sa već implementiranim delovima, ali pre svega prema kapacitetu te iteracije. Više savremenih agilnih metodologija propagira ovakav razvoj. Motivacija počiva na pretpostavci da ovakav pristup planiranju iteracija dodatno uvećava kvalitet iteracija, kako kroz preciznije pridržavanje planova rokova i sredstava tako i kroz promovisanje još intenzivnije komunikacije između razvijalaca i klijenata.

Određivanje koraka prema rokovima ima i neke potencijalno nezgodne karakteristike, koje moramo da imamo u vidu. Uspostavljanje tesnih okvira inkrementalnog koraka može da oteža pojedine korake i da podigne ukupnu cenu i trajanje razvoja. Neki elementi sistema ne mogu da se prirodno podele na različite korake, pa je zato nekada isplativije da se relaksiraju (produže) neke iteracije nego da se nešto veći poslovi veštački dele na više iteracija.

U prvim iteracijama se obično biraju poslovi koji donose veću dobit klijentu, pa kako se razvoj bliži kraju, nastupa rizik da se razvoj prekine pre nego što se implementiraju i poslovi „čija je cena veća od dobiti“ iako su oni možda presudno značajni za uspešno funkcionisanje softverskog sistema kao celine.

Objektna orijentacija

Savremene razvojne metodologije su po pravilu objektno-orijentisane (OO), što znači da se u prvi plan stavljaju objekti, a ne procesi, koji su uobičajeno zauzimali centralno mesto kod klasičnih metodologija. Osnovna dobit od fokusiranja na objekte je u povećanoj stabilnosti modela i proizvoda. Suština je u tome da promene u načinu poslovanja mnogo češće imaju za posledicu promene u postojećim procesima (postupcima), pa čak i nestajanje nekih i pojavljivanje potpuno novih procesa, nego što je to slučaj sa objektima.

Značajan faktor stabilnosti OO modela predstavlja enkapsulacija. Sakrivanjem implementacije iza interfejsa se omogućava da većina promena ima sasvim lokalnu prirodu. Naravno, ako se procesi implementiraju kao metodi nekih klasa objekata, onda će te klase objekata morati da se prilagođavaju promenama, ali se ispostavlja da pri tome postojeće klase nestaju i nove nastaju relativno retko (bar u odnosu na procese), a čak i pri njihovom modifikovanju, u ključnim odnosima među klasama dolazi do sasvim umerenog broja promena. Zbog toga, ako projekat i model softvera ili dela softvera zasnujemo na objektima i odnosima među objektima, sva je prilika da će najvažniji i najosetljiviji elementi projekta morati da se menjaju ređe nego što bi to bio slučaj da smo ih zasnovali na procesima.

Ako bismo, na primer, razvijali softver za praćenje i podršku upisa na fakultet, onda bi nam primena klasičnih metodologija nalagala da posebnu pažnju posvetimo prepoznavanju ključnih procesa i opisivanju njihovog odvijanja i njihovih među-

sobnih odnosa. Kompletan proces upisa bismo postepeno dekomponovali na procese pripreme upisnog roka, raspisivanja konkursa, prijavljivanja na konkurs, planiranja prijemenog ispita, polaganja prijemnog ispita, ocenjivanja prijemnog ispita, rangiranja kandidata, proveravanja dokumenata, upisa kandidata, odabira izbornih predmeta i možda još neke pomoćne procese. Pri tome bismo opisivali i potrebne strukture podataka i povezali bismo ih sa procesima koji ih upotrebljavaju.

Nasuprot tome, primena OO metodologija bi nam nalagala da umesto procesa najpre prepoznamo ključne objekte i klase objekata, kao i njihovu strukturu, ponašanje i međusobne odnose. Tako bismo, na primer, prepoznali da imamo studijske programe, upisni rok, kandidate, prijave, prijemne ispite, upisane studente, upisane predmete i drugo. Za svaku klasu objekata bismo najpre prepoznali odgovornosti, tj. koja klasa je zadužena za koje poslove, a onda i unutrašnju strukturu, tj. sadržane podatke, kao i odnose među klasama.

U oba slučaja moramo da opišemo i postupke i podatke. Ključna razlika je u tome što nas fokusiranje na procese ne tera da podatke lokalizujemo unutar pojedinačnih procesa, već se oni često dele između više procesa. Pored toga, i procesi i odgovarajuće strukture podataka često imaju krupniju granulaciju nego u slučaju OO pristupa. Posledica takvog pristupa je da se među procesima uvodi veći broj različitih vrsta neposrednih i posrednih zavisnosti[6]. Sa druge strane, OO pristup nam nalaže da podatke lokalizujemo unutar objekata, tako da svaki objekat obuhvata samo one podatke koje on neposredno koristi u svojim postupcima. To dovodi i do enkapsulacije i do finije granulacije struktura podataka. Najvažnija posledica toga je manja međusobna zavisnost elemenata softvera.

Na primer, u procesnom pristupu bi moglo da bude u redu da podaci o prijavljenom kandidatu i upisanom studentu predstavljaju jednu celinu, dok bismo u OO pristupu za kandidata i studenta imali različite objekte, sa razdvojenim podacima (među kojima se, naravno, nekako ostvaruje posredna veza). Zbog takve razdvojenosti bismo u OO slučaju mogli da poslove sa kandidatima i poslove sa studentima razvijamo i održavamo lakše i nezavisnije nego u slučaju kada bi oni počivali na deljenim podacima. Ako bi, na primer, u nekom trenutku bio ukinut

[6] U poglavlju *4 - Uvod u projektovanje softvera* ćemo videti da takve zavisnosti predstavljaju *spregnutost* i da nam one značajno otežavaju sve faze razvoja softvera. Ovde ćemo samo da istaknemo da nam spregnutost najviše smeta kada se prave izmene u softveru. Svaki put kada menjamo neki deo softvera, moramo da se zapitamo da li zbog toga moramo da menjamo i neke druge povezane delove softvera? Što su različiti elementi softvera (procesi, podaci i drugo) jače međusobno spregnuti, to će naše izmene imati za posledicu više dodatnih izmena. U posebno neugodnim slučajevima može da dođe do prave lavine izmena, zbog kojih čak može da bude lakše da se softver ponovo razvija nego da se menja.

prijemni ispit, onda bi posledica bila da bi nestali neki od procesa. Uticaj na povezane procese bi bio relativno visok zbog neposrednog uticaja na odvijanje dugih procesa i zbog posrednog uticaja na druge procese usled niske izolovanosti podataka. U takvim okolnostima ne bi bilo lako da se unapred proceni i sagleda obim potrebnih poslova. Sa druge strane, potrebne izmene u OO slučaju bi bile i manjeg obima i relativno predvidive, zato što lokalizacija, enkapsulacija i niža granularnost podataka imaju za posledicu lakše lokalizovanje čak i tako značajnih izmena.

Dodatna dobit od upotrebe OO koncepata je da se takav pristup lakše prilagođava iterativnom razvoju. Sa povećavanjem broja iteracija neminovno dolazi i do povećanja broja potrebnih izmena u softveru, pa je poželjno da se primenjuje metodologija koja proizvodi manje zavisnosti između različitih elemenata softvera. Kada se u ranim fazama planiranja više pažnje posveti objektima, a tek u kasnijim fazama se posvetimo procesima, onda se smanjuje i rizik od donošenja pogrešnih odluka u ranim fazama razvoja, čije bi naknadne izmene bile veoma skupe.

Osnovni rizik pri fokusiranju na objekte je eventualno potpuno zanemarivanje razmatranja procesa u ranim fazama razvoja, što može da vodi pogrešnoj arhitekturi sistema, koja se kasnije teško (tj. skupo) menja. Sa druge strane, detaljno razmatranje procesa u početnim koracima modeliranja sistema preti da dovede do „naduvavanja“ tih koraka. Zbog toga je neophodno da se u početnim fazama modeliranja sistema procesi pažljivo i odmereno razmatraju.

2.5 Umesto zaključka

Savremeni metodi razvoja softvera su donekle umanjile učestalost neuspeha, ali još uvek se veliki broj softverskih projekata ne završava sa planiranim pozitivnim rezultatima. Zbog toga je veoma važno da se sprovode istraživanja koja su usmerena na što bolje prepoznavanje problema i pronalaženje boljih, efikasnijih i pouzdanijih metoda za njihovu prevenciju.

Zainteresovanima za ovu oblast preporučujemo radove o razlozima neuspeha u razvoju softvera, kao i knjige koje se bave metodološkim pitanjima razvoja informacionih sistema, u kojima se često posvećuje pažnja i uzrocima neuspeha i načinima njihovog sprečavanja. Pored izvora na koje je već referisano u tekstu [*Stangish Group 1994, 2015; Abouzahra 2011; Charette 2005*] čitaoce možemo da uputimo i na [*Avison 2003; Al-Ahmad 2009; Boehm 1991; Fortune 2005*].

Praktično svaka od savremenih tehnika razvoja softvera je oblikovana radi prevazilaženja ili smanjivanja rizika od nastajanja neke vrste problema sa kojima se svakodnevno suočavamo pri razvoju softvera. Preporučujemo čitaocima da pri upoznavanju razvojnih tehnika i metodologija uvek to imaju u vidu i da pokušaju da u tom kontekstu što bolje sagledaju doprinose svake od tehnika predstavljenih u ovoj knjizi.

3 - Objektna orijentacija

Objektno-orijentisano programiranje je izuzetno loša ideja,
koja je mogla nastati samo u Kaliforniji.
Edsher Daikstra

3.1 Objektno-orijentisano programiranje

Objektno-orijentisano programiranje je nastalo na talasu inovacija koje su sredinom 20. veka težile da sistematizuju programiranje i programske jezike. Uzrok je predstavljala najpre intuitivno a kasnije i sistemski prepoznata *softverska kriza*. Sredinom 60-ih godina je praktično završen period inicijalnog razvoja računarstva. Do tada je bavljenje razvojem računarstva i programiranjem bilo praktično ista stvar, zato što je pisanje praktično svakog novog programa predstavljalo manji ili veći istraživački poduhvat. Sa porastom rasprostranjenosti računara došlo je do povećane potrebe za pisanjem programa i za programerima, koja nije mogla da bude zadovoljena na odgovarajući način bez unapređivanja metoda rada. Pisanje sve više i sve većih programa uz upotrebu zastarelih tehnologija imalo je za rezultat sve više neuspešnih projekata.

Prvi veliki iskorak prema unapređenju tehnologije programiranja bila je pojava *strukturnog programiranja*. Iako je strukturno programiranje u naučnim krugovima nastajalo postepeno još od kasnih 50-ih godina, tek je prepoznavanje softverske krize dovelo do značajnijeg pomaka u zastupljenosti strukturnog programiranja u široj praksi. Strukturno programiranje počiva na prepoznavanju osnovnih struktura algoritama i programskih jezika, koje su dovoljno izražajne da mogu da opišu svaki algoritam i program, a sa druge strane dovoljno konceptualno čiste i jednostavne da mogu da se lako razumeju i da može da se formalno dokaže njihova korektnost.

Vrlo brzo je strukturiranje koda dopunjeno i strukturiranjem podataka, a kada je pokušano spajanje koda i podataka u jedinstvenu celinu praktično je stvorena osnova za razvoj objektno-orijentisanog programiranja (OOP). U ranim fazama je terminologija OOP bila drugačija od one koju danas poznajemo, pa su, na primer, klase nazivane „glavnim objektima", „modulima", „definicijama"... Kasnije, sa porastom broja novih programskih jezika, došlo je i do delimične standardizacije terminologije[7].

Paralelno sa porastom zastupljenosti OOP, težilo se i prenošenju objektno-orijentisane paradigme na druge oblasti računarstva, što je dovelo do rada na objektno-orijentisanim bazama podataka i na oblikovanju čitavih objektno-orijentisanih razvojnih metodologija.

3.2 Osnovni koncepti OOP

Objekti i klase

U objektno-orijentisanom programiranju u centru pažnje su *objekti*. Objekti nam služe da pomoću njih u programima modeliramo entitete iz domena problema. Svaki objekat bi trebalo da ima prepoznatljiv identitet, stanje i ponašanje, kao i životni vek, koji čine nastajanje, postojanje i nestajanje.

Radi ilustracije, navešćemo nekoliko definicija pojma objekta iz literature iz devedesetih godina XX veka:

- Objekat je apstrakcija nečega u domenu problema, koja opisuje sposobnost sistema da o tome čuva informaciju, interaguje sa time ili oboje [*Coad 1991*].
- Objekat je koncept, apstrakcija ili nešto sa jasnim granicama i smislom u odnosu na konkretan problem. Objekat ima dve svrhe: da pomogne razumevanju stvarnog sveta i pruži praktičnu osnovu za računarsku implementaciju [*Rumbaugh 1991*].
- Objekti se opisuju odgovorima koje mogu da daju na pitanja:
  - Ko sam ja?
  - Šta mogu da uradim?
  - Šta znam? [*Wirf-Brock 1990*]
- Objekti imaju stanje, ponašanje i identitet [*Booch 1990*].

[7] I dalje neki jezici imaju specifičnu terminologiju. Na primer, u programskim jeziku C++ se ne koristi termin „metod" već „funkcija-članica" i slično.

Jedan objekat predstavlja apstrakciju jednog entiteta iz domena problema. Međutim, programsko opisivanje svakog pojedinačnog entiteta bi bilo nezahvalan posao. Zato se kao dodatni nivo apstrakcije uvodi *klasa*. Klasa predstavlja apstrakciju nekog skupa objekata koji su međusobno *dovoljno slični*. Pri ustanovljavanju sličnosti, prevashodno se razmatra *ponašanje* objekata, a ne njihova struktura. Kada se ustanovi koji su to objekti koji su dovoljno slični, onda se prepoznaje *klasa* tih objekata i na programskom jeziku se opisuje njihovo ponašanje, ali i njihova struktura. Objekti se zatim prave kao primerci (instance) opisane klase.

Standardizacijom objektno-orijentisanih tehnika i tehnologija se bavi Grupa za upravljanje objektima (engl. *Object Management Group – OMG*[8]). Njihova zvanična definicija pojmova klase i objekta glasi[9]:

- *Klasa* je opis skupa objekata koji dele iste atribute, operacije, metode, odnose i semantiku. Svrha klase je da deklariše kolekciju metoda, operacija i atributa koja u potpunosti opisuje strukturu i ponašanje tih objekata.
- *Objekat* je primerak koji potiče iz klase, strukturiran je i ponaša se u skladu sa svojom klasom.

Struktura objekata se opisuje pomoću *atributa*. Atributi predstavljaju opisne karakteristike objekata. Definicijom klase se ustanovljava koje atribute moraju da imaju objekti te klase, a pri radu sa objektima se određuje koji objekat ima koje vrednosti tih atributa. Atributi su sredstvo za čuvanje stanja i znanja objekata.

Ponašanje objekata se opisuje pomoću *metoda*. Metodi predstavljaju algoritamske opise ponašanja objekata jedne klase. Pri opisivanju klase opisuju se i svi metodi koji modeliraju ponašanje objekata te klase.

U većini OO programskih jezika su koncepti klase i pripadanja objekta klasi veoma strogo definisani. Klasa objekta se određuje u trenutku njegovog pravljenja i kasnije ne može da se promeni, tako da svaki objekat pripada tačno jednoj klasi. U takvim jezicima postoji vrlo jasna korelacija između koncepta *klase* i koncepta *tipa* – klase predstavljaju jednu veoma važnu vrstu tipova i smatra se da objekat pripada klasi ako ima tip te klase.

---

[8] *OMG* je konzorcijum otvorenog tipa, u koji su učlanjene brojne značajne softverske kompanije. Osnovan je 1989. godine i bavi se razvojem i održavanjem standarda u oblasti objektno-orijentisanih tehnologija [*OMG*].

[9] Moramo da primetimo da je ova definicija cirkularna – klasa se definiše preko objekta a objekat preko klase. To je dobra ilustracija OO sveta, u kome je malo šta do kraja strogo i precizno definisano. O tome će biti više reči pri kraju ovog poglavlja.

U nekim programskim jezicima atributi i metodi mogu da se definišu i na nivou pojedinačnih objekata, a ne samo na nivou klasa. Ako bismo pokušali da pravimo razliku između takvih i ostalih OO jezika, možda bismo mogli da kažemo da su samo takvi jezici pravi „objektno-orijentisani" programski jezici, dok su ostali jezici pre „klasno-orijentisani". Tipičan primer „pravog OO programskog jezika" u tom smislu je programski jezik *ECMAScript*, šire poznat kao *JavaScript*, u kome se sve vrti oko objekata[10].

Postoje i programski jezici kod kojih klase postoje, ali se pripadnost objekta klasi ustanovljava dinamički, proveravanjem ispunjenosti nekih definisanih semantičkih pravila. U takvim jezicima objekat može da promeni klasu tokom svog života. Na primer, ako bi se kvadratu promenila veličina stranica, onda bi on mogao da prestane da bude kvadrat i postao bi pravougaonik. Primer takvog jezika je *D*, objektno-orijentisan jezik za rad sa podacima[11], koji su definisali Dejt i Darvin [*Date 2006*].

Enkapsulacija i interfejs

Jedan od ozbiljnih problema sa klasičnim tehnikama programiranja, uključujući i strukturno proceduralno programiranje, jeste puna raspoloživost svih podataka i algoritama svim programerima. To je imalo za posledicu da svaki programer može da pristupa bilo kom podatku i bilo kojem aspektu ponašanja, bez ikakvog ograničenja. Takav način rada može da potpuno zaobiđe određena pravila upotrebe nekih struktura podataka ili pomoćnih aspekata ponašanja i da dovede do neispravnog stanja softvera.

Da bi se prevazišao taj problem, u OO tehnologijama su oblikovani koncepti enkapsulacije i interfejsa. *Enkapsulacija* predstavlja princip skrivanja detalja implementacije klase od korisnika te klase, tj. od programera koji će raditi sa objektima te klase. Sve ono što zahteva dublje poznavanje interne implementacije klase se sakriva. Korisnicima se dozvoljava da koriste samo pažljivo definisan skup metoda, koji omogućavaju upotrebu objekata klase, a ne zahtevaju dublje poznavanje pojedinosti njene implementacije. Taj skup metoda predstavlja *interfejs* klase.

---

[10] U programskom jeziku *JavaScript* dugo uopšte nije postojao koncept klase. Sintaksa i semantika klase su dodati standardom *ECMAScript* 6 iz 2015. godine (naziva se i *ECMAScript 2015*; skraćeno *ES6*), pa od tada *JavaScript* mnogo više liči na druge „klasne" programske jezike. Ipak, *JavaScript* je zadržao svoje karakteristične osobine, pa i mogućnost da se „živom" objektu promene struktura i metodi, a time i klasa i tip, zbog čega je to i dalje pravi „objektni" jezik.

[11] Primetimo da taj jezik, osim imena, nema skoro ništa drugo zajedničko sa sve popularnijim programskim jezikom *D*, koga su razvili Volter Brajt i Andrej Aleksandresku [*Alexandrescu 2010*].

Interfejs klase bi trebalo da što bolje odgovara javnim aspektima ponašanja objekata te klase. On mora da omogući korisnicima klase da urade sa objektima sve što predstavlja odlike ponašanja te klase, a da istovremeno od njih bude skriveno što više informacija o internim elementima implementacije.

Jedno od pravila enkapsuliranja nalaže da korisnici ne smeju neposredno da koriste atribute klase, već da sve upotrebe atributa moraju da budu enkapsulirane u metode interfejsa. Dosledna primena ovog principa nalaže da enkapsulacija atributa mora da se preduzima čak i kada bi atributi trebalo da predstavljaju interfejs klase. U tim slučajevima se prave tzv. pristupni metodi za čitanje i menjanje vrednosti atributa[12].

Enkapsulacijom se sprečava nekontrolisano menjanje stanja objekata, mimo pravila koja su interno ustanovljena implementacijom klase. Istovremeno, u sve metode koji čine interfejs je ugrađeno rukovanje stanjem i skrivenim metodima uz puno poštovanje svih tih internih pravila. Na taj način se korisnik klase oslobađa obaveze staranja o tim pravilima. Smanjivanje količine potrebnog znanja o internim elementima implementacije omogućava programerima da uz manje učenja, lakše i pouzdanije koriste klase koje su napravili drugi programeri. Sa druge strane, autorima klase omogućava da bez većih ograničenja mogu da menjaju elemente interne strukture i implementaciju klase, a da to nema uticaja na njene korisnike – jedino što može neposredno da utiče na upotrebu jesu eventualne promene interfejsa klase.

Osim vida zaštite od neispravne upotrebe, enkapsulacija i interfejs pružaju korisnicima klase dodatni nivo apstrakcije. Dobar interfejs ne pruža korisniku nikakve naznake o načinu implementacije klase, već na zaokružen način predstavlja celinu klase isključivo u kontekstu njene funkcionalnosti, tj. mogućih načina upotrebe. Mogli bismo da uporedimo interfejs klase sa daljinskim upravljačem televizora – korisniku je važno da može da promeni program ili jačinu zvuka, a ni najmanje ga ne zanima kako televizor radi.

Enkapsulacija i interfejs se ne primenjuju samo na nivou klase, već i na nivou komponenti, pa i servisa. Svaka celina programa koju pišemo bi trebalo da enkapsulira elemente interne implementacije i da pruža korisnicima samo zaokružene interfejse. Na taj način se podiže nivo apstrakcije svake celine koda i omogućava se

[12] Pristupni metodi se na engleskom jeziku obično nazivaju *getter* i *setter*. U programskim jezicima kao što su *Java* ili *C#* uobičajeno je da se za svaki atribut prave pristupni metodi, bilo javni ili privatni. Sa druge strane, kada se koristi programski jezik *C++*, onda se teži da se pravi što manje pristupnih metoda i to samo ako su baš neophodni, tj. ako atribut predstavlja deo interfejsa.

niži nivo međusobne spregnutosti, a time i viši nivo međusobne nezavisnosti delova programa, što je veoma važno za lakše pisanje i održavanje programa.

Enkapsulacija i interfejs mogu da se koriste i kao sredstvo za analizu kvaliteta arhitekture programa. Ako je interfejs sačinjen tako da ne predstavlja jednu nedeljivu celinu, već ga čine elementi različitih aspekata ponašanja klase, onda to vrlo često može da nam ukaže na potencijalan problem neispravne dekompozicije – možda bi ta klasa trebalo da se podeli na dve ili više manjih? Slično tome, ako je za obavljanje nekog posla potrebno da se koristi interfejs većeg broja klasa i objekata, možda bi trebalo da se napravi nova klasa koja bi prikrila složene aspekte implementacije i upotrebe tih klasa?

Specijalizacija i generalizacija

Kao što klasa predstavlja poseban oblik tipa, tako se i među klasama uspostavljaju odnosi *potklasa* i *natklasa*, slično odnosima *podtip* i *nadtip*. Nasleđivanje klasa je osnova za uspostavljanje hijerarhijskog polimorfizma, koji je osnovni vid polimorfizma u većini OO programskih jezika[13]. Hijerarhijski polimorfizam počiva na osobini objektno-orijentisanih programskih jezika da programski kod, koji je napisan da radi sa objektima jedne klase, može da radi i sa objektima svih njenih potklasa.

Nasleđivanje klasa se uspostavlja na osnovu jednosmerne parcijalno uređene binarne relacije *jeste*. Kažemo da klasa A *jeste* klasa B ako i samo ako svaki objekat klase A ima sve osobine koje imaju i svi objekti klase B. Ako klasa A *jeste* klasa B, onda se kaže i da je klasa A *izvedena* iz B ili da klasa A *nasleđuje* klasu B, a da je klasa B *osnovna* ili *bazna* klasa za A. Slično, kažemo i da je klasa A *potklasa* ili *potomak* klase B, a da je klasa B *natklasa* ili *predak* klase A.

Na primer, neka imamo klase `Student` i `StudentOsnovnihStudija`. Ako svi objekti klase `StudentOsnovnihStudija` imaju sve osobine kao i svi objekti klase `Student` (i uz njih možda i još neke nove osobine) onda možemo da zaključimo da `StudentOsnovnihStudija` jeste `Student`, tj. da je klasa `StudentOsnovnihStudija` izvedena iz klase `Student`. Slično bismo zaključili da iz klase `Student` može da se izvede i klasa `StudentMasterStudija`.

Relacija *jeste* se često referiše terminima *specijalizacija* i *generalizacija*, gde je *specijalizacija* praktično isto što i relacija *jeste*, dok je *generalizacija* ista relacija ali u suprotnom smeru. Znači, ako A *jeste* B, onda je A *specijalizacija* (ili *poseban slučaj*) klase B, a B je *generalizacija* (ili *uopštenje*) klase A.

[13] Polimorfizmu, sa posebnim akcentom na parametarskom polimorfizmu, je posvećeno poglavlje *11 - Polimorfizam*.

Klasa `StudentOsnovnihStudija` je specijalizacija klase `Student`, a klasa `Student` je generalizacija klasa `StudentOsnovnihStudija` i `StudentMasterStudija`.

Osnovni kriterijum pri ustanovljavanju ovih odnosa među klasama predstavlja ponašanje klase, odnosno njenih objekata. Tako, na primer, u odnosu na ponašanje objekata ustanovljavamo da je ispravno da klasa `Kvadrat` nasleđuje klasu `Pravougaonik`, zato što svaki kvadrat *jeste* istovremeno i pravougaonik, tj. sve što znamo da važi za pravougaonike, važi i za sve kvadrate. Bila bi veoma ozbilljna greška da se umesto ponašanja razmatra struktura objekata, pa da se pretpostavi da pravougaonik nasleđuje kvadrat zato što kvadrat ima samo *širinu*, a pravougaonik i *širinu* i *dužinu*, što je *„specijalan“* slučaj. Pravila o ponašanju kvadrata ne važe za pravougaonik – na primer, obim pravougaonika ne može da se računa po formuli $O = 4a$. Nasleđivanje i relacija „jeste“ bi trebalo da se odnose striktno na ponašanje, a ne na strukturu objekata klase.

3.3 Objektno-orijentisane metodologije

Prve objektno-orijentisane metodologije (OOM) su počele da se razvijaju ubrzo po nastajanju prvih OO programskih jezika. Da bi došle do toga da budu sveprisutne u razvoju softvera, OOM su morale da pređu dugačak put. Istoriju OOM bismo mogli da posmatramo kroz tri osnovna perioda, koji se delimično preklapaju.

Prvi period se prostire od početka razvoja pa do oko 1997. godine i karakterišu ga *početni koraci* u definisanju metodologija. U tom periodu se pojavljuje veliki broj različitih metodologija, kao i veliki broj potpuno nezavisnih notacija za zapisivanje projekata i planova, pa i različitih terminologija. Ispostavilo se da većina od tih metodologija nije bila ni kompletna ni dovoljno široka i obuhvatna da zaživi mnogo dalje od ograničenog domena primene ili nekog užeg razvojnog okruženja u kome je nastala. Najznačajniji rezultat ovog perioda razvoja je nastajanje velikog broja metoda i tehnika razvoja. Skoro svaka tehnika koju danas koristimo se u nekom osnovnom obliku po prvi put pojavila u nekoj od metodologija iz tog perioda. Neke od najvažnijih metodologija iz tog perioda su predstavljene u knjigama Grejdija Buča [*Booch 1990*], Kouda i Jurdona [*Coad 1991*] i Martina i Oudela [*Martin 2003*]

Drugi period je trajao od 1995. godine do oko 2005. godine. Za njega je karakterističan rad na objedinjavanju metodologija i tehnika. U prvoj polovini ovog perioda je uočljivo usmeravanje velike energije na ujednačavanje notacije i definisanje (a kasnije i unapređivanje i standardizovanje) *UML-a* – objedinjenog jezika za zapisivanje modela [*UML*]. Ovaj period možemo da nazovemo *Oblikovanje UML-a*. Ovaj značajan poduhvat je pokrenut u okviru kompanije *Rational*, u kojoj su najpre Grejdi Buč i Džim Rambou radili na objedinjavanju svojih metodologija, da bi im se 1995. pridružio i Ajvar Jakobson. Njihov cilj je bio da razviju kombinovanu metodologiju, koja je najpre bila poznata pod imenom *Object Oriented Software Engineering* (*OOSE*), a koja je danas poznata pod imenom *Rational Unified Process*

(*RUP*). U okviru razvoja metodologije je paralelno tekao i razvoj objedinjene notacije za zapisivanje svih elemenata projekta, koja je dobila ime *Unified Modeling Language* (*UML*). Radom na *UML*-u je upravljao konzorcijum partnera, među kojima su bile vodeće svetske softverske kompanije.

Nezadrživo širenje *UML-a* je počelo 1997. godine, objavljivanjem prve zvanične verzije. Taj trenutak se može smatrati i završetkom prvog perioda razvoja OOM. Ubrzani razvoj *UML*-a je usporen najpre objavljivanjem stabilnih verzija 1.1, 1.2 i 1.3 do 1999. godine, a zatim posebno 2005. godine objavljivanjem verzije 2.0. Tokom ovog perioda se drastično smanjio broj novonastalih metodologija.

Treći period traje od 2000. godine i karakteriše ga sveprisutnost *UML*-a, kao jedne od najrasprostranjenijih razvojnih softverskih tehnologija. Mogli bismo ga nazvati *Post-UML period*. Praktično sve što se od 2000. godine radi u oblasti OOM, radi se uz pretpostavku da se kao notacija koristi *UML*. Više se ne posvećuje vreme izmišljanju novih notacija, već se uz pretpostavljanje ujednačene notacije radi na pravljenju novih metodologija. Razvija se veliki broj *agilnih* metodologija, pri čemu praktično sve one koriste brojne elemente *RUP-a* i drugih metodologija koje su iz njega potekle.

Danas se u praksi primenjuje mnogo različitih metodologija, pri čemu su one većinom objektno-orijentisane. Važna posledica razvoja *UML*-a je da se rezultati rada svih tih metodologija, pa i komunikacija među razvijaocima koji ih primenjuju, odvijaju sa punim međusobnim razumevanjem, bez straha da može doći do nesporazuma usled različite notacije. Štaviše, danas i neke metodologije koje po svojoj prirodi nisu objektno-orijentisane teže da u što većoj meri koriste *UML*.

Savremene OOM se odlikuju velikim brojem zajedničkih karakteristika. Pre svega, tu je opšta težnja da se sve modelira objektima i klasama. Obično se posmatraju četiri osnovna skupa objekata u sistemu:

- entiteti – sve vrste podataka koji postoje u softverskom sistemu;
- subjekti – različite vrste korisnika softverskog sistema;
- servisi – usluge koje softverski sistem pruža korisnicima i
- interfejsi – podsistemi putem kojih subjekti koriste servise.

Praktično sve OOM teže da na neki način skrate trajanje razvojnog ciklusa. I agilne i ne-agilne metodologije podstiču upotrebu iterativnog razvoja. Agilne metodologije propisuju i određivanje koraka prema rokovima i raspoloživim sredstvima. Takođe, insistira se i na pojačanoj komunikaciji među subjektima u svim fazama razvoja.

U poglavlju *5 - UML* ćemo predstaviti neke od osnovnih dijagramskih tehnika *UML-a*. U poglavljima *4 - Uvod u projektovanje softvera*, *6 - Principi projektovanja* i *7 - Obrasci za projektovanje* ćemo posvetiti pažnju projektovanju softvera i posebno

modeliranju strukture softvera. U poglavlju *8 - Agilni razvoj softvera* predstavićemo koncepte agilnih metodologija.

3.4 Slabosti objektno-orijentisanih koncepata

Objektno-orijentisano programiranje i uopšte objektno-orijentisani koncepti imaju i neke relativno značajne slabosti. Tradicionalno su se isticali nedostaci poput veličine izvršnog programa, brzine izvršnog programa, pa i nesrazmerno velikog ulaganja programerskog rada, ali prva dva od ovih problema su važna samo u ograničenom domenu, dok je treći prevaziđen rapidnim razvojem različitih tehnika i tehnologija razvoja softvera, od kojih neke predstavljamo i u ovoj knjizi.

Veličina i sporost izvršnog koda su nekada imali veliki značaj, pa su bili među najvažnijim razlozima za ograničen uspeh projekta *Smalltalk*, jednog od prvih pravih objektno-orijentisanih programskih jezika sa pratećim vizualnim razvojnim i korisničkim okruženjem. Ipak, ispostaviće se da je 1995. godine programski jezik *Java* sa svojim izvršnim okruženjem (Java virtualna mašina – *JVM*) uspeo da se pozicionira na tržištu tamo gde *Smalltalk* nije uspeo 1980. godine, a da je osnovna arhitektura sistema praktično iskopirana. Razlika je, pre svega, u tome što je između ova dva projekta prošlo 15 godina, tokom kojih su razvijani sve brži računari. I danas postoje primene u kojima savremeni objektno-orijentisani programski jezici, kao što su *Java* i *C#*, imaju teškoća da se izbore za svoje mesto, ali su sve veća brzina računara i unapređivanje tehnika implementiranja (tj. interpretiranja internog koda) ovih jezika značajno smanjili opseg takvih primena. Sa druge strane, razvoj tehnologija prevođenja je doveo do toga da je programski jezik *C++* u nekim stvarima čak i efikasniji nego *C*. Neefikasnost OOP je danas prisutna više kao posledica neadekvatne upotrebe nego kao karakteristika koncepta ili jezika. Na primer, nije retkost da dosledno modeliranje OO alatima dovede do programskog koda koji sadrži previše nepotrebnih apstrakcija, što zaista može da uspori rad programa.

Jedna od oblasti gde mora veoma pažljivo da se radi da bi se dobio dobar rezultat jesu aplikacije koje intenzivno koriste relacione baze podataka. Relacione baze podataka rade sa skupovima podataka (relacijama), dok OO programi rade sa pojedinačnim objektima. Zato obično mora da se pravi dodatni sloj aplikacije, tzv. sloj objektno-relacionog preslikavanja (engl. *object-relational mapping – ORM*), koji služi da se od programera sakrije relaciona priroda podataka. Posledica skrivanja relacionog modela, koji je nimalo slučajno potvrđen kao najbolji za većinu vrsta baza podataka, može da bude suviše striktno korišćenje OO koncepata i mehanizama, tako da svaki pristup podacima ide od pojedinačnih objekata, preko *ORM*-a pa do sistema za upravljanje bazom podataka. Takav rad može veoma značajno da uspori rad sa podacima, u odnosu na neposrednu upotrebu i izvođenje operacija neposredno na relacijama, ali to je pre svega posledica drugačijeg nivoa i čak različite vrste apstrakcije, a ne nekih konkretnih slabosti OO programskih jezika.

Ako danas razmatramo slabosti OOP i OOM, onda se one najviše prepoznaju na konceptualnom nivou i najčešće imaju pretežno teorijski značaj. Sve konceptualne slabosti OOP potiču iz činjenice da ne postoji matematički stroga i precizna definicija koncepata na kojima počivaju objektno-orijentisane tehnologije. Za razliku od OOP, neke druge danas sveprisutne softverske paradigme, kao strukturno programiranje, funkcionalno programiranje ili relacione baze podataka, počivaju na vrlo čvrstim matematičkim osnovama. Posledica takvog stanja je da OOP i OOM imaju određene nedostatke koji u specifičnim slučajevima mogu da dovedu do protivrečnosti i nekih neprijatnih ograničenja.

Rečenica Edshera Daikstre, čijim citiranjem je započeto ovo poglavlje, nastala je kao posledica činjenice da je Daikstra bio veoma posvećen problemima dokazivanja korektnosti programa i promovisanju strukturnog i disciplinovanog programiranja [*Dijkstra 1976*]. Zbog toga je relativno rano primetio da OOP, iako se intuitivno čini da nas ono vodi prema savremenijem, produktivnijem pa i kvalitetnijem načinu programiranja, zbog nedostatka odgovarajuće matematičke formalizacije praktično onemogućava dokazivanje korektnosti programa, a na duže staze i otežava pouzdano razumevanje programskog koda. Nakon višegodišnjeg fokusiranja značajnog broja istraživača u oblasti računarskih nauka na matematičko formalizovanje i strukturiranje algoritamskih programskih jezika, OOP je u tom smislu izvesno predstavljalo značajan korak unazad. Ovde ćemo predstaviti neke od značajnijih aspekata problema do kojih dovodi nedovoljno strogo definisana semantika OO programskih jezika.

Jedna od konceptualnih slabosti OO tehnologija je nedovoljno precizno definisan pojam *identiteta objekta*, pa zato na neka pitanja ne postoji jednoznačan odgovor. Na primer, da li mogu da postoje dva objekta koji imaju *isti sadržaj* ali ne predstavljaju isti objekat, ili to nije dopušteno? Ili, drugačije formulisano, da li objekti koji imaju isti sadržaj (vrednosti svih atributa) automatski predstavljaju jednu istu instancu klase? Problem identiteta objekata je posebno važan u oblasti OO baza podataka i povezivanja OO programa sa bazama podataka.

I koncept hijerarhijskog polimorfizma ima slabosti. Na primer, već smo naglasili da se kao osnovni kriterijum za uspostavljanje odnosa nasleđivanja koristi relacija „*jeste*“ i to prevashodno u pogledu ponašanja objekata. Naveli smo i naizgled jednostavan primer kvadrata i pravougaonika i naglasili da bi trebalo da kvadrat predstavlja specijalizaciju pravougaonika, a ne obrnutno. Međutim, ispostavlja se da nije baš sve tako jednostavno, kao što bi na prvi pogled moglo da izgleda.

Ako bi, na primer, pravougaonik imao metode za menjanje veličine stranica `postaviSirinu` i `postaviDuzinu`, onda ti metodi ne bi mogli da rade na odgovarajući način u slučaju kvadrata. Ako razmotrimo šta bi trebalo da bude rezultat izvršavanja metoda:

```
kvadrat.postaviSirinu( 20 );
```

onda možemo da primetimo da postoje dve osnovne varijante:

1. Ako bi se na kvadratu izvršavao *nasleđeni* metod, onda bi se promenila širina kvadrata, dok bi istovremeno dužina zadržala staru vrednost:
   - Naš „kvadrat“ sada ne bi više bio kvadrat, već samo pravougaonik!?
   - Da li tip objekta uopšte može da se promeni na taj način ili bi to i dalje ostao objekat klase „kvadrat“ iako mu stranice nisu jednake?
2. Ako bi kvadrat imao *sopstvenu implementaciju* ovog metoda, onda bi njegovom primenom mogle odjednom da se promene i širina i dužina:
   - Onda naš „kvadrat“ ostaje kvadrat.
   - Ali, primetimo da za svaki pravougaonik važi pravilo: „ako se dvostruko uveća širina pravougaonika, onda će se dvostruko uvećati i površina“. Takvo pravilo, međutim, ne bi važilo za naš kvadrat, zato što bi se njegova površina uvećala četvorostruko, a to onda znači da se naš kvadrat ponaša drugačije od pravougaonika, pa onda on više *nije* pravougaonik!?

Očigledno je da oba pristupa dovode do problematičnih rezultata, što narušava konzistentnost odnosa nasleđivanja među klasama. Opisan problem je u literaturi poznat kao problem „kvadrat-pravougaonik“ ili „krug-elipsa“. Iako ovaj problem na prvi pogled može da izgleda elementarno i intuicija može da nam govori da postoji neko relativno jednostavno rešenje, stvari zapravo stoje sasvim drugačije i ne bismo mogli reći da postoji dobro rešenje ovog problema za uobičajene OO programske jezike. Vratićemo se na ovaj problem kada budemo razmatrali principe projektovanja (*6 - Principi projektovanja*) i videćemo da samo delimično možemo da ga rešimo.

Drugi često spominjan problem se odnosi na to što objekti grade implicitno globalno stanje programa. Jedan od osnovnih principa strukturnog programiranja nalaže da se ne koriste globalne promenljive i globalno stanje programa. OOP je u osnovi oblikovano uz pretpostavku poštovanja tog principa. Programiranje sa objektima je zamišljeno tako da nijedan objekat ne može da pristupi strukturi drugog objekta, te da objekti međusobno razmenjuju poruke kojima zahtevaju jedni od drugih da se nešto izračuna ili da se promeni stanje nekog objekta. Međutim, da bi objekti mogli da šalju poruke jedni drugima, oni moraju da „znaju“ jedni za druge, ili da bar imaju na raspolaganju neki vid kataloga u kome mogu jedni druge da pronađu. Tako dolazimo do povezanog grafa koji čine svi međusobno povezani objekti (bilo da su povezani neposredno ili posredstvom jednog ili više kataloga), a taj graf nije ništa drugo nego velika i prilično složena povezana struktura koja predstavlja *globalno stanje programa*. Znači, ispada da je neposredna posledica enkapsulacije to što svaki objekat ima stanje, a posredna to što graf svih povezanih objekata čini globalno stanje našeg programa? Neki autori ističu da to baš i nije ono

što želimo da imamo, mada možemo da primetimo da ova povezanost podataka suštinski ne može da se izbegne ni ako koristimo klasično strukturno programiranje – čak je tada vidljivija i očiglednija, zato što nije enkapsulirana, a kod OOP bar može da bude u priličnoj meri prikrivena (što, doduše, može da bude i dobro i loše).

Kada se nešto temeljnije razmotre uzroci koji dovode do predstavljenih (i nekih drugih) konceptualnih problema, onda može da se ustanovi da koncepti tipova, sistema tipova i hijerarhija tipova, sa svim odlikama OOP, mogu da se zasnuju formalno i matematički precizno tako da budu neprotivrečni, ali da to onda ima prilično visoku cenu, koja iz ugla današnjih tehnologija razvoja softvera i savremenih OO programskih jezika uglavnom nije prihvatljiva – uvođenje neophodne konstantnosti objekata. Kada se objektima oduzme promenljivost stanja, a identitet se identifikuje sa jednakošću vrednosti svih atributa, onda može da se izgradi jedan zaokružen sistem koji može da se koristi i za programiranje i za objektne baze podataka, ali takav sistem se značajno razlikuje od ustaljenih normi u svetu OOP. Nasuprot tome, takav pristup je uobičajen i široko se primenjuje u svetu funkcionalnog programiranja, gde umesto neformalnih koncepata na scenu stupa formalna i konceptualno zaokružena teorija tipova.

U brojnim člancima može da se pronađe više informacija o problemima OO koncepata, kao i predlozi za njihovo rešavanje. Zainteresovanim čitaocima preporučujemo radove Dejta i Darvina, u kojima se razmatraju neki od problema OOP, na primer rad „*Treći manifest*“ [*Date 1995*] ili knjigu „*Baze podataka, tipovi i relacioni model*“ [*Date 2006*]. Tu može da se vidi da možemo da formalno razmišljamo o OO programima i da definišemo formalnu semantiku OOPJ, u slučaju kada se formalni koncepti OOP uvedu kao nadgradnja funkcionalnog programiranja i strogo formalno definisanog sistema tipova.

3.5 Umesto zaključka

Praktično sve razvojne metodologije koje su danas u upotrebi su ili potpuno objektno-orijentisane ili bar počivaju na objektno-orijentisanim konceptima. Pokazalo se da je objektno-orijentisani pristup primenjiv u svim fazama razvoja softvera, od početnog razmatranja projekta, pa do njegove finalne implementacije i održavanja.

I pored određenih poznatih nedostataka, kroz praktične primene se ispostavilo da je objektno-orijentisani model programiranja i modeliranja softvera najbolji i najpouzdaniji poznati pristup razvoju softvera. Zbog toga se citat Edshera Daikstre, naveden na početku ovog poglavlja, obično navodi i tumači kao šala, iako je jedan od najvećih teoretičara računarskih nauka njime izrazio zapravo veoma ozbiljnu i objektivnu kritiku nepotpune i nedovoljno formalne teorijske zasnovanosti koncepata OOP.

Poslednjih godina u savremenom razvoju softvera sve više pronalaze svoje mesto koncepti i tehnike funkcionalnog programiranja. Sve više objektno-orijentisanih

programskih jezika usvaja neke karakteristike koje su tradicionalno bile rezervisane za funkcionalne programske jezike. Čitaocima preporučujemo odličnu knjigu Ivana Čukića o funkcionalnom programiranju na programskom jeziku C++ [*Čukić 2018*], s tim da bi bilo dobro da pre nje najpre savladaju osnovne tehnike parametarskog polimorfizma (*11 - Polimorfizam*).

4 - Uvod u projektovanje softvera

Čitava istorija softverskog inženjerstva
je istorija podizanja nivoa apstrakcije
Grejdi Buč

4.1 Projekat softvera

Razumevanje problema projektovanja softvera, kao i pojam projekta i proces projektovanja, prošli su kroz brojne transformacije tokom razvoja računarstva i posebno tokom razvoja oblasti softverskog inženjerstva i razvoja softvera. Ono što je kroz sve periode razvoja računarstva bilo i ostalo zajedničko je da *projekat softvera* predstavlja plan prema kome će da se implementira softver. Međutim, sadržaj tog plana i način njegovog pravljenja su se značajno menjali. U početnim fazama računarstva razvoj softvera je bio ograničen na pisanje pojedinačnih programa, pa se projektovanje softvera svodilo na oblikovanje algoritama. Sa izgradnjom složenijih računarskih i softverskih sistema postepeno se dolazilo do razvijanja temeljnijih pristupa projektovanju softvera i do oblikovanja različitih razvojnih metodologija.

Projekat softvera, u opštem slučaju, čine različite vrste dokumenata i nacrta, iz kojih može da se razume *šta* se pravi, *zašto* se pravi i *kako* se pravi u nekom konkretnom razvojnom poduhvatu. Pored toga, obično je potrebno i mnogo dodatnih materijala koji omogućavaju temeljno razumevanje domena na koji se projekat i projektovani softver odnose. Vrste konkretnih dokumenata i nacrta u velikoj meri zavise od primenjene metodologije i značajno su se menjale tokom vremena; mogu da se odnose na različite elemente projekta i mogu da budu na različitim nivoima apstrakcije. Između ostalog, u okviru savremenih projekata mogu da se nađu:

- *vizija* – dokument koji bez mnogo ulaženja u detalje opisuje cilj i osnovnu ideju sa kojom se prilazi razvoju softvera;
- *modeli domena* ili *izveštaji o stanju domena* – različite informacije o domenu na koji se softverski projekat odnosi, uključujući prikupljene podatke i nacrte koji opisuju strukturu i ponašanje domena (tj. „stvarnog“ sveta) ili ciljnog stanja domena:
  - *strukturni model domena*;
  - *funkcionalni model domena*;
  - *model procesa domena*
  - i drugo;
- *specifikacija zahteva* – dokument (kome mogu da budu pridruženi odgovarajući nacrti) koji nabraja i opisuje konkretne zahteve koje bi planirani softver trebalo da ispuni;
- *analiza rizika* – dokument koji razmatra moguće rizike i planove reagovanja u slučaju njihovog nastupanja; može da obuhvati i obračun vrednosti rizika;
- *plan životnog ciklusa* – opisuje životni vek i faze kroz koje se planira da tokom svog života prolazi softver koji se razvija, od trenutka započinjanja njegovog planiranja do trenutka izbacivanja iz upotrebe;
- *kadrovski plan* – procenjene potrebe za različitim vrstama kadrova u toku rada na projektu, obično po fazama i celinama;
- *plan infrastrukture* – opisuje šta je sve od alata i tehnologija potrebno da se obezbedi i stavi na raspolaganje razvojnom timu tokom razvoja;
- *implementacioni model* –skup nacrta koji opisuju strukturu i ponašanje implementacije, tj. komponenti, klasa i drugih strukturnih elemenata planiranog softvera;
- *plan dokumentovanja* – dokument koji opisuje koje vrste dokumenata je kada i koliko iscrpno potrebno pripremati;
- *plan testiranja* – opisuje koje vrste testova će se kada i kako praviti i izvršavati radi praćenja i obezbeđivanja kvaliteta;
- i još mnogo toga.

Prethodnim spiskom smo tek zagrebali površinu i izdvojili samo neke od elemenata projekta koji se pojavljuju u savremenom razvoju. Način posmatranja problema projektovanja, vrste modela kojima se pridaje veća važnost, faze razvoja u kojima se prave određeni modeli, specijalnosti razvijalaca koji prave određene modele, kao i druge specifičnosti, u velikoj meri zavise od domena, obima i složenosti softverskog projekta i od izabrane razvojne metodologije.

Različiti projekti softvera mogu da imaju veoma različit obim ili nivo apstraktnosti i preciznosti. Što je veći obim projekta, to je veći i broj različitih vrsta planova i modela koji moraju da se pripreme pre implementacije. U najobuhvatnije slučajeve spada, na primer, projektovanje informacionih sistema velikih poslovnih organizacija. Takvi sistemi obuhvataju veliki broj ciljnih proizvoda, koji moraju da budu međusobno usklađeni i da predstavljaju zaokruženu celinu. Sa druge strane, u nekim jednostavnijim slučajevima, kada se pravi manji softver sa užom i jasno prepoznatom funkcijom, planiranje se često drastično skraćuje i pojednostavljuje, zato što sve što je potrebno za započinjanje implementacije može da se opiše u relativno sažetom obliku.

4.2 Elementi projektovanja softvera

Projektovanje softvera obuhvata sve aktivnosti tokom razvoja softvera, koje za cilj imaju izradu projekta softvera ili pojedinih elemenata projekta softvera.

Već smo istakli da projekti imaju potencijalno visoku složenost, pa odatle sledi da je i projektovanje softvera potencijalno veoma sadržajan i kompleksan proces, koji počiva na prikupljanju i analiziranju velike količine informacija i pravljenju različitih modela, procena i planova. Projektovanje softvera je posebna disciplina i njeno detaljno i iscrpno predstavljanje bi zahtevalo mnogo veći prostor nego što nam je ovde na raspolaganju. U savremenim OO metodologijama i posebno u slučaju primene agilnih razvojnih metodologija, uobičajeno je da svi članovi razvojnog tima učestvuju u pravljenju (ili bar održavanju i modifikovanju) elemenata projekta koji se odnose na definisanje i opisivanje strukturnih elemenata softvera i njihovog ponašanja. Zbog toga ćemo u nastavku ovog poglavlja, ali i u drugim poglavljima koja se bave projektovanjem, pažnju usmeriti pre svega na *projektovanje implementacije* i posebno na njegov deo koji se naziva *strukturno projektovanje*.

Posmatranje softverskog sistema i njegovih pojedinačnih delova se menja tokom procesa projektovanja. U svakoj fazi projektovanja je pažnja fokusirana na neke specifične aspekte i prikupljanje ili analiziranje nekog skupa informacija i pravljenje određene vrste dokumenata, planova ili modela. Da bi se bolje razumeli proces projektovanja softvera i mesto i uloga projektovanja implementacije i strukturnog projektovanja u tom procesu, u ovom odeljku ćemo pokušati da pružimo uvid u osnovne postupke koji čine projektovanje softvera.

Proces projektovanja i odnos prema pojedinačnim postupcima u tom procesu su se menjali tokom vremena i značajno se razlikuju od metodologije do metodologije. Neke od metodoloških razlika ćemo da sagledamo u narednom odeljku (*Pristupi projektovanju softvera,* na strani 51).

Istraživanje domena

Kada govorimo o *domenu*, misli se na širi kontekst na koji se odnosi poduhvat projektovanja i implementiranja softvera. To može da bude neki poslovni sistem za koji se razvija nov informacioni sistem, ili populacija biciklista za koje se pravi mobilna aplikacija za komunikaciju sa odgovarajućim servisima i slično.

Na početku procesa projektovanja se posmatra prostor domena i prikupljaju se i razmatraju različite informacije, kako o celom domenu u kome se planira da funkcioniše novi softver, tako i o njegovom užem delu na koji se taj softver neposredno odnosi. Prikupljanje informacija se obično naziva *istraživanjem domena*, a razmatranje tih prikupljenih informacija o domenu se naziva *analiziranjem domena*. Istraživanje i analiziranje domena obično predstavljaju prve korake u okviru procesa projektovanja softvera. U nekim razvojnim metodologijama se ova dva koraka jasno razdvajaju, dok se u nekim drugim u većoj meri prepliću, pa mogu čak i da se integrišu.

Osnovni ciljevi istraživanja i analiziranja domena su ostvarivanje što boljeg sagledavanja i razumevanja (1) postojeće strukture i načina funkcionisanja domena, (2) motivacije za planiranje i projektovanje novog (ili menjanje postojećeg) softvera i (3) celokupne izmenjene slike domena, koja bi trebalo da se dobije nakon puštanja novog softvera u rad.

Istraživanje domena podrazumeva veći broj različitih metoda prikupljanja informacija o domenu. Prikupljaju se i analiziraju dokumenti koji opisuju strukturu i način funkcionisanja domena, posmatra se kako službenici na različitim radnim mestima i nivoima obavaljaju svoj posao, evidentiraju se pravila odlučivanja, prave se ankete, organizuju se intervjui i drugo. Rezultat istraživanja domena bi trebalo da bude što obuhvatnija kolekcija dokumenata koji opisuju postojeći domen.

Pored prikupljanja informacija o postojećem stanju domena, drugi aspekt istraživanja domena je prikupljanje informacija o željama, potrebama i očekivanjima u odnosu na nov softver. U poduhvat planiranja i projektovanja novog softvera se uazi zato što postoje neke potrebe, koje postojeće stanje domena ne zadovoljava. Zato je neophodno precizno i temeljno razumevanje tih želja, potreba i očekivanja da bi kasnije mogli da se ispravno definišu projektni zahtevi.

Na primer, pretpostavimo da je na nekom fakultetu potrebno da se napravi softver za podršku procesa upisivanja novih studenata. Ako na fakultetu već postoje neki elementi informacionog sistema, onda bi novi softver trebalo da se poveže sa njima tako da čini celinu sa postojećim sistemom. Da bi to bilo moguće, istraživanje i analiziranje domena moraju da imaju za rezultat iscrpnu sliku postojećeg stanja domena. Posmatra se najpre domen u širem obimu, da bi se dobila celovita slika funkcionisanja fakulteta, ali i domen u užem obimu, da bi se dobila što preciznija slika onog dela fakulteta koji ima neposrednog dodira sa procesom upisivanja novih studenata. Istraživanje domena bi trebalo da obuhvati, između ostalog:

- pregled osnovnih aktivnosti na fakultetu;
- pregled osnovnih elemenata koji su već podržani postojećim informacionim sistemom;
- opis svih postupaka koji se sada (bez planiranog softvera) odvijaju u procesu upisa;
- pregled svih podataka koji se u tom postupku prikupljaju, koriste ili izdaju, uz posebno istaknute informacije o poreklu tih podataka:
- šta se dobija od kandidata;
- šta se povlači iz postojećeg informacionog sistema;
- šta se izdaje i u kojoj formi;
- šta se, kada i kako unosi u postojeći informacioni sistem
- i drugo;
- pregled svih različitih vrsta službenika koji učestvuju u procesu, sa jasnom evidencijom nihovih odgovornosti i privilegija;
- zvaničnu upisnu dokumentaciju iz prethodnih godina;
- pravila i kriterijume odlučivanja u različitim koracima procesa;
- pregled tehničkih protokola i interfejsa za komunikaciju sa postojećim informacionim sistemom;
- i još mnogo toga.

U ovom slučaju možda nije neophodno da se unapred detaljno prikupljaju informacije o željama i očekivanjima, zato što bi analiza trebalo da pokaže šta je i kako moguće da se automatizuje – nov softver neće da služi za neki nov posao nego za što bolju podršku procesu koji već postoji ali se do sada odvijao manuelno.

Analiziranje domena

Informacije, koje su prikupljene istraživanjem domena, se analiziraju i na osnovu njih se prave različiti modeli koji opisuju strukturu i način funkcionisanja domena. Taj korak se naziva *analiziranje domena*. Modeli koji nastaju kao rezultat analize domena se obično nazivaju *modeli domena* ili *analitički modeli*, pa se zbog toga ovaj korak često naziva i *modeliranje domena*.

Strukturni model domena, na primer, može da ima formu dijagrama klasa domena, dok funkcionalni model domena može da ima formu dijagrama aktivnosti, dijagrama BPMN, ili da obuhvata različite tablice odlučivanja ili korelacija[14]. Sa

[14] Neke od navedenih vrsta dijagrama ćemo upoznati u narednim poglavljima, a neke druge izlaze iz okvira ove knjige.

porastom složenosti domena, povećava se i broj različitih modela domena, koje je neophodno da napravimo da bi stanje (tj. struktura i ponašanje) tog domena bilo precizno opisano.

Model domena mora da jasno opisuje strukturu domena i funkcionalnosti pojedinačnih elemenata te strukture. Obično se domen najpre posmatra kao jedinstvena celina, pa se postepeno raspoznaju delovi te celine koji u njoj imaju neku zaokruženu i jasno prepoznatu ulogu. Preduzima se tzv. funkcionalna dekompozicija celovitog sistema, kako bi se prepoznale i razdvojile pojedinačne komponente i njihove funkcije. Pri pravljenu modela domena osnovni cilj je da se *što vernije predstavi domen*. Zato se tom prilikom obično ne preduzimaju veliki koraci apstrahovanja, kao što će to biti slučaj u kasnijim fazama projektovanja[15].

Posmatranje se zatim lokalizuje, u meri koja je potrebna za konkretan projekat, kako bi se važniji delovi sistema detaljnije opisali odgovarajućim modelima. Ako se u toku analiziranja prikupljenih informacija otvori neko pitanje na koje odgovor ne može da se pronađe u već prikupljenim informacijama, onda je obično neophodno da se te informacije dopune. Zato se istraživanje i analiziranje domena u praksi često prepliću.

Definisanje zahteva

Kada postoji jasan model domena, u narednom koraku je potrebno da se sagleda i precizira šta je potrebno da se pravi. Taj korak se obično zove *definisanje zahteva*.

Definisanje zahteva bi trebalo da se radi na osnovu modela postojećeg stanja domena i prikupljenih informacija o željama i potrebama u odnosu na novi softver. U nekim slučajevima, kao pomoć pri definisanju zahteva, može da se napravi tzv. *ciljni model domena*, koji predstavlja model modifikovanog domena, u obliku u kome će da postoji (ili se očekuje da postoji) nakon što se dovrši implementacija novog softvera. Ciljni model domena ne bi trebalo da ima elemente koji ukazuju na konkretne pojedinosti implementacije, već samo da predstavi mesto novog softvera u celovitom sistemu i njegove osnovne funkcije. Važno je da se ustanove dodirne tačke novog softvera i domena, kao i osnovne karakteristike najznačajnijih interfejsa.

Definisanje zahteva se obično prepoznaje kao posebna disciplina razvoja softvera, zato što zahteva specifične aktivnosti i sposobnosti razvijalaca. Da bi neko mogao da ispravno definiše zahteve, mora da dobro poznaje domen i razume način razmišljanja subjekata u domenu, a da istovremeno ima visok nivo tehničkih znanja koja mu omogućavaju da oblikuje zahteve tako da oni odgovaraju potrebama domena i doprinose čitavom sistemu, a uz to su i tehnički ostvarivi.

---

[15] O apstrahovanju i dekomponovanju, kao osnovnim postupcima strukturnog projektovanja biće više reči u narednim odeljcima.

Projektovanje implementacije

Nakon definisanja zahteva fokus se prebacuje iz prostora domena u prostor implementacije. U idealnom slučaju, od tog trenutka se prostor domena više uopšte ne razmatra neposredno, nego se implementacija projektuje i ostvaruje isključivo na osnovu precizno definisanih zahteva. Naravno, u praksi to ne biva uvek tako, zato što se često dešava da se pri projektovanju implementacije primeti da nedostaje neka informacija i da se onda zahteva da se informacije dopune bilo dodatnim preciziranjem zahteva ili možda još i analiziranjem domena ili čak i dodatnim istraživanjem domena.

Projektovanje implementacije započinje slično modeliranju domena. Proučavaju se definisani zahtevi i apstrahuje se osnovna ideja o funkcionalnosti (novog ili izmenjenog) celovitog sistema. Zatim se sistem funkcionalno dekomponuje na osnovne strukturne celine koje opisuju kako će se funkcionalnost sistema ostvariti u implementaciji. Potom se nivo posmatranja spušta na pojedinačne prepoznate delove i svaki deo se analizira i modelira da bi mu se što tačnije odredili funkcionalnost i struktura. Procesi analiziranja i strukturiranja se ponavljaju iterativno, sve dok se ne dođe do nivoa implementacije, odnosno do nivoa klasa i njihovih interfejsa i tačne strukture i implementacije.

Rezultate i međurezultate projektovanja možemo da podelimo po slojevima. Na najvišim nivoima imamo samo jedan posmatran element – ceo posmatran domen ili ceo softverski sistem koji pravimo. Na sledećem nivou dekomponujemo softverski sistem na njegove glavne funkcionalne delove – *komponente*. Zatim možemo da posmatramo svaku od komponenti i da pravimo njenu funkcionalnu dekompoziciju na sastavne delove – potkomponente. U nekom trenutku ćemo doći do nivoa na kome više nema prostora za dalje dekomponovanje na osnovu funkcionalnosti. Umesto toga, daljom strukturnom dekompozicijom delimo komponente na *pakete* ili konkretne *klase*, a ne na manje komponente. Završnica strukturnog projektovanja je određivanje interfejsa, strukture, internih metoda i međusobnih odnosa prepoznatih klasa.

Za razliku od komponenti, koje predstavljaju izdvojene funkcionalne celine, paketi predstavljaju logički grupisane kolekcije klasa (funkcija, potprograma i sl.), koje koristimo zajedno radi obavljanja nekog posla ili koje čine neku strukturnu celinu softvera. Na primer, ako bismo pravili interpretator za neki programski jezik, onda bismo u jedan paket mogli da stavimo hijerarhiju klasa koje predstavljaju apstraktno sintaksno drvo[16] – sve klase te hijerarhije čine jednu logičku celinu i među

[16] Apstraktno sintaksno drvo je drvo kojim se predstavlja sintaksa nekog izraza. U korenu drveta je čvor koji predstavlja operator najnižeg prioriteta, koji će se poslednji izračunavati. Listove drveta predstavljaju konstante i promenljive. Sve različite vrste

njima postoji visok stepen međuzavisnosti, ali sa druge strane one ne čine komponentu, koja ima jasno prepoznatu funkcionalnost i neki jednostavno upotrebljiv interfejs. Paketi predstavljaju važne strukturne elemente softvera, ali za razliku od komponenti ne predstavljaju elemente funkcionalne dekompozicije.

Pri detaljnijem projektovanju paketa takođe se radi iterativno. Najpre se prepoznaju ključne klase paketa, njihovi odnosi, interfejsi i struktura. U slučaju hijerarhija klasa, pod ključnim elementima se obično podrazumevaju bazna klasa hijerarhije, njen interfejs i još neke ključne klase hijerarhije – one od kojih postoji značajno grananje, one koje dodaju važno novo ponašanje, one koje imaju neku karakterističnu ulogu i slično. Nakon određivanja ključnih elemenata paketa, iterativno se određuju i svi ostali elementi i njihovi odnosi i struktura.

Relativno često se dešava da pri detaljnijem projektovanju uvidimo da različite komponente (i/ili paketi) imaju neke zajedničke funkcionalnosti, koje onda apstrahujemo i izdvajamo kao nezavisne celine. Dešava se i da nakon detaljnije analize ustanovimo da dekompozicija izvedena na prethodnom nivou nije dobra – na primer, da nisu prepoznate sve različite funkcije i komponente, ili da nisu dobro podeljene odgovornosti po komponentama, ili da interfejsi komponenti nisu dobro oblikovani i drugo. Zbog toga je veoma važno da razumemo da u svakom koraku možemo i moramo da preispitujemo prethodne korake.

Poseban aspekt projektovanja implementacije je prepoznavanje razvojnih zadataka. Na osnovu prepoznatih komponenti, paketa i klasa mogu da se oblikuju pojedinačni zadaci i njihove međuzavisnosti. Što je detaljnije izvedeno projektovanje, to je veća preciznost sa kojom se mogu oblikovati i raspoređivati zadaci. Na primer, na najvišem nivou, kao zadaci mogu da se razmatraju definisani projektni zahtevi. Kada se prepoznaju strukturni elementi, onda se projektni zahtevi prevode u formalne specifikacije planiranih strukturnih elemenata. Ako je projektovanje stiglo do nivoa apstraktnih komponenti, onda imamo nekoliko velikih zadataka koje možemo da prepoznamo i rasporedimo, ali ako je projektovanje već detaljno izvedeno, do nivoa klasa, onda možemo da oblikujemo veliki broj malih zadataka, čija se realizacija lakše ostvaruje i prati.

Sve do sada opisano čini strukturni model implementacije. U nastavku ovog poglavlja ćemo detaljnije razjasniti problem oblikovanja strukturnog modela implementacije, tj. posvetićemo se *strukturnom projektovanju implementacije*, ili kako se to obično kraće kaže *strukturnom projektovanju*.

čvorova (i listova) implementiraju se kao klase jedne hijerarhije, čija je bazna klasa obično uopšteni `Izraz`. Na više mesta u ovoj knjizi razmatramo primer klasa koje grade apstraktno sintaksno drvo.

Pored strukturnih elemenata, projekat implementacije obuhvata i razne druge elemente. Na primer, prave se dodatni planovi koji omogućavaju da se sagleda kako će tim (ili timovi) raditi na implementaciji i da se prati i vodi proces razvoja:

- plan životnog ciklusa;
- plan organizacije timova;
- plan angažovanja timova;
- raspored zadataka po timovima;
- plan infrastrukture;
- plan komunikacije;
- plan izrade verzija
- i drugo.

Svi ti elementi su tesno povezani sa specifičnim aspektima softverskog inženjerstva. Za razliku od strukturnog projektovanja, za koje je odgovornost podeljena između svih učesnika u razvoju (iako ne baš u podjednakoj meri), za ove elemente projekta je odgovornost prevashodno na ograničenom broju rukovodećih članova tima. Njima se nećemo detaljnije baviti u okviru ove knjige. Više informacija o tim aspektima projektovanja može se pročitati, na primer, u [*Pfleeger 2006*].

4.3 Pristupi projektovanju softvera

U zavisnosti od primenjene metodologije, sadržaj i redosled aktivnosti tokom projektovanja softvera mogu da se značajno razlikuju. Jedan od najvažnijih ciljeva razvojnih metodologija je upravo da odrede šta će se, kako i kojim redom raditi tokom razvoja softvera, a posebno tokom njegovog projektovanja. Istorija razvojnih metodologija je relativno bogata, ali je za ispravno razumevanje savremenog razvoja softvera i savremenih razvojnih metodologija uglavnom dovoljno da istaknemo tri značajna perioda, odnosno grupe metodologija koje su se primenjivale u tim periodima: (1) metodologije koje prethode OO metodologijama (u daljem tekstu ćemo ih nazivati *klasičnim metodologijama*), (2) ne-agilne OO metodologije i (3) agilne metodologije.

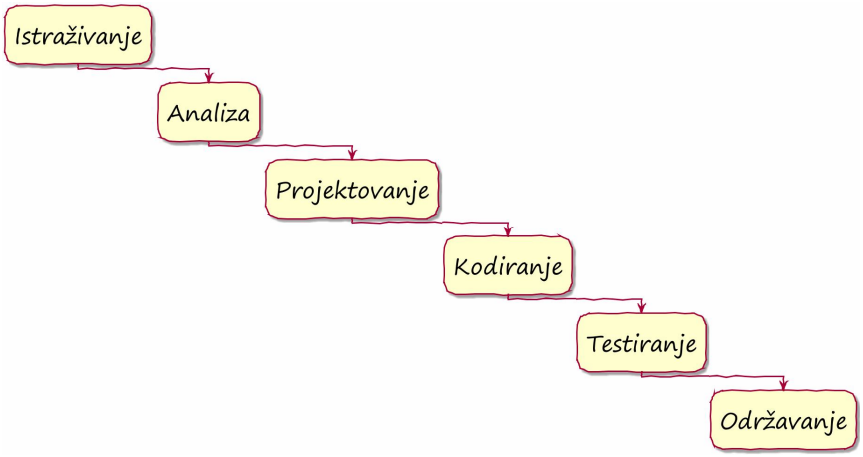
Projektovanje softvera u klasičnim metodologijama

Klasične razvojne metodologije su na različite načine pokušavale da uvedu red u razvojni proces, ali nećemo mnogo pogrešiti ako primetimo da su praktično sve stavljale fokus na (1) oblikovanje toka životnog ciklusa projekta kroz striktno određivanje redosleda obavljanja određenih poslova, (2) opisivanje poslovnih procesa i (3) strukturiranje elemenata projekta i samog razvojnog procesa.

Neke od klasičnih metodologija su se više fokusirale na samo neke od navedenih aspekata, dok su neke druge pokušavale da im daju relativno ujednačen značaj. Tako je opisivanje poslovnih procesa, na način koji je po mnogo čemu sličan opisivanju

algoritama, bio ključni element projekata u tzv. *procesnim* metodologijama. Takav pristup je bio posledica pretpostavke da se softverski sistem izrađuje da bi podržao obavljanje nekih poslovnih procesa u poslovnom domenu. Sa povećanjem složenosti rešavanih problema povećavala se i kompleksnost strukture softvera. Zato su se pojavile *strukturne* metodologije, koje su fokus prebacile na strukturu, kao i *kombinovane* metodologije, koje su pokušavale da fokus ravnomerno rasporede na strukturu i procese.

Kod većine klasičnih metodologija se posmatranje problema usmerava najpre na posmatranje procesa koji ga čine, da bi se zatim postepenom dekompozicijom dolazilo do strukture. Na primer, ako bi se pravio softver za podršku upisivanja novih studenata na fakultet, onda bi se najpre prepoznavali, analizirali i modelirali poslovni procesi, poput prijavljivanja kandidata za upis, održavanja prijemnog ispita, rangiranja kandidata ili upisivanja kandidata. Zatim bi se čitav projekat zasnovao na prepoznatim ključnim procesima.



Slika 1 – Životni ciklus projekta po modelu vodopada

Jedna od prepoznatljivih karakteristika klasičnih metodologija je bilo jasno razdvajanje faze projektovanja od faze implementacije. Štaviše, prepoznavao se niz faza kroz koje je razvojni poduhvat prolazio od započinjanja do završavanja, među kojima su se, između ostalog, nalazile i faze projektovanja i implementiranja. Broj faza i njihov sadržaj nije uvek bio isti (u različitim metodologijama ili kod različitih autora), ali je bio zajednički princip prelaženja iz jedne u drugu fazu *u jednom smeru*, bez vraćanja unazad. Zbog toga je takav model životnog ciklusa projekta obično nazivan *modelom vodopada*.

Slika 1 predstavlja jedan primer životnog ciklusa po modelu vodopada. Iz primera možemo da vidimo osnovnu ideju pristupa i osnovne razvojne celine.

Naravno, mogli bismo da podelimo neke korake na više manjih, ili možda čak da neke objedinimo većim koracima, ali osnovni princip se time ne menja.

Takav pristup životnom ciklusu projekta je bio tesno povezan sa pretpostavkom da razvijaoci softvera imaju relativno uske specijalnosti i da zbog toga u različitim poslovima i različitim fazama životnog ciklusa učestvuju različiti profili razvijalaca. Tako su se, na primer, prikupljanjem informacija o domenu i analizom domena bavili *sistem-analitičari*, projektovanjem *programeri analitičari* ili *sistemski programeri*, dok su se kodiranjem bavili *programeri*.

Jedno od glavnih ograničenja modela vodopada je to što on ne predviđa vraćanje na prethodne korake. Ako bi se tokom realizacije neke od kasnijih faza ispostavilo da neka od prethodnih faza nije dobro urađena, razvojni tim bi se našao pred ozbiljnom dilemom – da li da nastavi dalje onako kako je predviđeno ili da se vrati i ponavlja prethodne faze?

Ako bi se poštovao strogi model životnog ciklusa, to bi često u praksi imalo za posledicu da bi krajnji rezultat možda bio u skladu sa planovima, ali ne i u skladu sa potrebama. Da bi se softver doveo do upotrebljivog oblika, moralo bi ponovo da se prođe kroz čitav razvojni proces, kako bi se popravilo sve ono što je uočeno da nije dobro, ali i da bi se možda dodalo nešto što je u međuvremenu postalo potrebno. To praktično znači da bi se razvoj odvijao iterativno, a da ni razvojni tim ni primenjena metodologija nisu za to pripremljeni. Rezultat takvog rada je obično bio značajan gubitak vremena, kako zbog dodatnih iteracija tako i zbog odloženog menjanja već razvijenih delova softvera.

Alternativa je da se u trenutku uočavanja problema, iako metodologija to ne predviđa, prekine tekuća faza i ponovi prethodna. Naravno, vraćanje može da bude i za više od jedne faze unazad. Problem sa ovakvim pristupom je što ni metodologija ni razvojni timovi nisu na njega spremni. Ispostavlja se da je takvo vraćanje veoma skup postupak, zato što se čitav tim koji radi na tekućoj fazi praktično „deaktivira“, a tim koji je radio na prethodnoj fazi (i sada verovatno radi na nekom drugom poslu ili je čak rasformiran) mora da se ponovo okupi i vrati na ovaj posao. Obično se tu gubi dosta vremena i razvojni proces se značajno usporava.

Osnovni problem sa primenom strogih životnih ciklusa po modelu vodopada je što sa povećanjem obima i složenosti razvojnog projekta dolazi i do značajnog porasta rizika da projekat ne može da se završi u jednom prolazu, onako kako to propisuje metodologija. Ako je projekat obiman i složen, onda postoji ozbiljan rizik da u nekoj od faza bude načinjena greška, ili čak da sve teče i završava se kako je planirano, ali se zbog dužine realizacije projekta u međuvremenu izmene poslovne okolnosti i postane neophodno da se preduzmu odgovarajuće izmene u projektu. U takvim okolnostima, kao što smo već istakli, nije dobro ni da se vraćamo i menjamo projekat, ni da nastavimo dalje pa ga menjamo tek u narednoj iteraciji.

Projektovanje softvera u OO metodologijama

Sa prelaskom na objektno-orijentisane metodologije, sve veći značaj se pridaje povezivanju strukture i ponašanja softvera. OO razvojne metodologije teže da OO način posmatranja i modeliranja prenesu sa nivoa kodiranja programa na nivo modeliranja problema, pa čak i samog razvojnog procesa. Zbog toga se postepeno razvijaju novi pristupi posmatranju domena i softvera, a sa njima i nove tehnike modeliranja, što je na kraju dovelo i do pravljenja Objedinjenog jezika za modeliranje (*UML*)[17].

Osnovna promena koju su donele nove OO metodologije se odnosila na primarni predmet modeliranja – fokus se preneo sa procesa na objekte. Umesto da se kao najvažniji deo sistema posmatraju procesi i neke sa njima povezane strukture, OO način modeliranja je preneo pažnju projektanata na strukturu sa enkapsuliranim ponašanjem. Dok je u klasičnim metodologijama *struktura* predstavljala tek model podataka (u memoriji, bazi podataka ili stvarnom domenu), *struktura* u OO modelu predstavlja mnogo više od toga, zato što pruža objedinjenu sliku strukture podataka i ponašanja koje se na nju odnosi.

Često se kaže da je OO pristup bolji od procesnog, zato što su *objekti stabilniji od procesa*. Kada se govori o stabilnosti, tu se pre svega misli na stabilnost nekog koncepta na kome počiva projekat softvera. Naravno, ako je potrebno da na nečemu zasnujemo projekat, onda ne želimo da to bude nešto što je nestabilno, što se menja u vremenu i može da nas dovede u situaciju da ceo projekat mora da trpi izmene. Međutim, kao što mogu da se promene poslovni procesi, tako mogu da se promene i objekti (klase) kojima smo te procese modelirali, tj. ponašanje tih objekata – pa gde je onda prednost fokusiranja na objekte?

Suština veće stabilnosti objekata je u tome što (1) procesi imaju veću tendenciju nestajanja i nastajanja nego objekti, ali i što (2) promene u toku procesa mogu da impliciraju mnogo dublje promene nego promene u strukturi ili ponašanju objekata. Značajnu ulogu u tome ima enkapsulacija. Ako bi u poslovnom domenu prestala potreba za nekim od osnovnih procesa, to bi obično vodilo relativno velikim izmenama u odgovarajućim modelima aktivnosti. Sa druge strane, objektni pristup podrazumeva enkapsulaciju, pa eventualno odbacivanje nekih vrsta objekata ili postupaka, zahvaljujući enkapsulaciji, obično ima lokalnije posledice nego u slučaju procesa.

Iako su OO metodologije značajno promenile pristup problemu projektovanja, sam životni ciklus projekta nije nužno bio izmenjen. U početnim fazama razvoja OO metodologija faze projektovanja i implementacije su još uvek bile jasno razdvojene,

[17] Zbog njegovog izuzetno velikog značaja, *UML*-u je posvećeno posebno poglavlje *5 - UML*, na strani 87.

kako vremenski tako i prema specijalnostima razvijalaca. I dalje je životni ciklus obično imao odlike (unapređenog) modela vodopada.

Projektovanje softvera u agilnim metodologijama

Agilni pristup razvoju softvera ćemo detaljnije razmotriti u posebnom poglavlju (*8 - Agilni razvoj softvera*). Tamo ćemo bolje upoznati i karakteristike procesa projektovanja i koncepte na kojima počiva projektovanje u agilnim metodologijama. Ovde ćemo samo pokušati da, u relativno pojednostavljenom obliku, istaknemo neke od značajnih promena koje su agilne metodologije uvele u razvojni proces i posebno u projektovanje softvera.

Agilne metodologije su ostvarile najveći uticaj na projektovanje time što su (1) praktično obrisale ranije postojeću oštru granicu između različitih faza razvoja softvera, a time i između faza projektovanja i implementacije i kao posledicu toga (2) stvorile potrebu da većina razvijalaca bude osposobljena i za projektovanje i za implementiranje softvera. Specijalnost *programera analitičara*, koji su u stanju i da analiziraju probleme i projektuju rešenja i da ta rešenja implementiraju, ranije je bila relativno skromno zastupljena i pretežno ograničena na projektantske timove, a sa prelaskom na agilne metodologije značajno raste njena zastupljenost u razvojnim timovima.

Jedna od osnovnih pretpostavki agilnog razvoja je da su *promene zahteva* tokom razvoja nešto što je neminovno i samim tim sasvim normalno, pa da zato razvojni proces mora da se prilagodi tako da se te promene što bezbolnije prihvate i uklope u tok razvoja. Ali ako želimo da razvojni tim bude u prilici da brzo i efikasno reaguje na promene, onda je neophodno da se prekine sa praksom strogog razdvajanja faza projektovanja i implementiranja (kao i drugih faza) i da se predvidi da će analiziranje, projektovanje i implementiranje izmena doći onda kada je potrebno, a ne u unapred propisanim periodima. Štaviše, to se prenosi na sve segmente projekta, pa se čak i analiziranje i projektovanje delova softvera odlaže do trenutka njihove implementacije.

Da bi razvojni tim bio sposoban za takvu organizaciju rada, neophodno je da u timu imamo zastupljene sve specijalnosti koje mogu da budu potrebne pri reagovanju na promene (ili odloženom analiziranju i projektovanju) – od analitičara, preko projektanata do implementatora i ostalih specijalnosti. Jedan način da to ostvarimo je da u svakom timu obezbedimo različite kadrove, koji su specijalizovani za odgovarajuće aktivnosti, ali u tom slučaju bi neki od njih (pre svega analitičari i projektanti) mogli često da dolaze u situaciju da budu neaktivni tokom nekog perioda vremena, dok čekaju da se pojave neki zahtevi za promenama ili da se započne rad na novom delu softvera. Zato se teži da svi članovi tima budu što šire i bolje obučeni, tako da mogu da obavljaju različite poslove, čime se izbegava (ili bar smanjuje verovatnoća) da neki od njih imaju periode neaktivnosti.

4.4 Arhitektura i dizajn

U domenu razvoja softvera se umesto termina *projekat softvera* često upotrebljavaju termini *dizajn softvera* i *arhitektura softvera*. Sa druge strane, termin *projekat* se sve češće odnosi na kompletan poduhvat izrade i primene softvera, koji obuhvata planiranje, projektovanje, izradu, distribuciju, marketing i sve ostale aspekte tog poduhvata.

Termin *dizajn softvera* potiče od engleskog termina za *projekat* (engl. *design*) i samim tim doslovno predstavlja sinonim za *projekat softvera*. Upotrebljava se podjednako za označavanje projekta softvera, kao i za označavanje discipline razvoja softvera, koja se bavi projektovanjem. Danas je uobičajeno da se koristi u nešto užem smislu, tako da pojam *dizajn softvera* često ne obuhvata elemente projektovanja koji su bliži poslovnoj strani problema (kao što su modeliranje stanja domena, koncipiranje vizije i formalizaciju zahteva), kao ni elemente planiranja samog razvojnog procesa i infrastrukture (tj. specifične elemente softverskog inženjerstva), već se ograničava (ili bar fokusira) na opisivanje i modeliranje funkcionalnih i strukturnih elemenata softvera, tj. na modeliranje implementacije.

Termin *arhitektura softvera* uobičajeno predstavlja strukturu softverskog sistema posmatranu na relativno visokom (apstraktnom) nivou. Slično kao termin *dizajn* i termin *arhitektura* se podjednako koristi za označavanje projekta (na relativno apstraktnom nivou), kao i za označavanje discipline razvoja softvera koja se bavi izradom arhitekture.

Vidimo da je *dizajn* nešto opštiji pojam od *arhitekture*, tj. da svaka arhitektura u suštini predstavlja dizajn (projekat), ali da nije svaki dizajn istovremeno i arhitektura. Pri planiranju arhitekture softverskog sistema se uzimaju u obzir pre svega oni aspekti problema koji imaju uticaj na veći deo projekta, dok se uglavnom ne razmatraju pojedinosti čiji je uticaj relativno ograničen. Razliku između arhitekture i dizajna nije uvek lako ustanoviti, ali se obično raspoznaje na osnovu posmatranja softverskog sistema na različitim nivoima apstrakcije ili složenosti, ili na osnovu posmatranja konkretnih metodoloških postupaka, pa čak i faza razvoja.

Arhitektura se bavi važnim stvarima.
Šta god one bile.
Ralf Džonson

Možda je najnezgodniji aspekt razlikovanja arhitekture i dizajna u tome što granice između njih ne mogu da se jednoznačno postave. Različite organizacije i timovi imaju i različite kriterijume za njihovo postavljanje, ali je veoma važno da razumemo da pri tome najznačajniju ulogu imaju dva kriterijuma: (1) stepen funkcionalne dekompozicije i (2) procenjen značaj koji bi detaljnije projektovanje

imalo na ceo sistem. Neophodno je da težimo da arhitektura obuhvati sve ključne elemente funkcionalne dekompozicije i sve elemente strukturne dekompozicije koji imaju širok uticaj na različite delove softvera, ali i da, sa druge strane, ne obuhvati elemente koji se odnose na neke uske specifičnosti pojedinačnih strukturnih elemenata.

Svaki deo softvera se može učiniti fleksibilnim, ali svako omogućavanje fleksibilnosti unosi dodatnu složenost u sistem. Zato je pri oblikovanju arhitekture neophodno da pronađemo dobru ravnotežu između fleksibilnosti sistema i njegove složenosti. Jedan od važnijih kriterijuma koji se pri tome razmatraju jeste i cena eventualnih izmena, tj. cena koju ćemo platiti za eventualno naknadno uvođenje dodatne fleksibilnosti u sistem.

> *Arhitektura predstavlja značajne odluke o dizajnu, koje daju oblik sistemu, gde je značajnost određena cenom pravljenja izmena*
>
> *Grejdi Buč*

U narednim poglavljima ćemo videti da se u savremenom razvoju planiranje arhitekture i planiranje dizajna uglavnom vremenski razdvajaju, tako da se arhitektura relativno čvrsto ustanovljava u ranim fazama razvoja, dok se dizajn svakog pojedinačnog elementa softvera obično definiše tek tokom njegovog razvoja. Ovakav pristup je posledica potrebe da se uspostavi određena ravnoteža između dominantne zastupljenosti agilnog razvoja, koji podstiče odlaganje detaljnog projektovanja sve do trenutka razvoja konkretnog dela softvera, i relativno visoke cene promene arhitekture, koja nas motiviše da arhitekturu što ranije definišemo i da je što ređe menjamo. O tome će biti nešto više reči u poglavlju o agilnim metodologijama.

Termini *dizajn* i *arhitektura* se u nekim slučajevima koriste u nešto užem smislu, samo za označavanje strukturnih elemenata projekta. Funkcionalni elementi projekta (zajedno sa dodatnim nefunkcionalnim zahtevima) se često nazivaju *specifikacijom* (ili projektnim zahtevom) i često se ne smatraju delom *dizajna* ili *arhitekture*, već pretpostavkom za njihovo pravljenje. To može da bude veoma pogrešno, zato što tada specifikacije postaju unapred zacrtane i nisu podložne menjanju i prilagođavanju. Pravu ocenu kvaliteta specifikacije možemo da dobijemo tek kada pristupimo detaljnijem projektovanju, kako same komponente o čijoj se specifikaciji radi tako i drugih komponenti koje bi ovu trebalo da koriste. Uobičajeno je da se u okviru dizajna govori o specifikacijama strukturnih elemenata implementacije na svim nivoima (tj. o formalnim definicijama ponašanja, interfejsa pa čak i unutrašnje strukture pojedinačnih klasa, paketa i komponenti), dok se u okviru arhitekture kao ulazni parametri razmatraju specifikacije projektnih zahteva i elemenata modela

domena, a kao specifikacije strukturnih elemenata implementacije na najvišim nivoima apstrakcije, tj. specifikacije komponenti.

U praksi se oblikovanje arhitekture softvera relativno često suviše fokusira pa čak i ograničava samo na funkcionalne aspekte analiziranja i dekomponovanje sistema, dok se svi strukturni aspekti projektovanja ostavljaju za dizajn. Takav pristup može da ima veoma neugodne posledice i trebalo bi da se izbegava. Ponekad može da nam se učini da je to i čisto i efikasno, ali moramo da primetimo da se tako obično zanemaruje značajnost uticaja koji bi detaljnije projektovanje moglo da ima na ceo sistem. Kao rezultat takvog pristupa može da se dobije neprilagodljiva arhitektura, koja će tokom daljeg razvoja morati da trpi značajne i skupe promene. Zbog toga bi pri oblikovanju arhitekture trebalo da odredimo najviši nivo strukturne organizacije softvera i to ne samo na osnovu funkcionalnih aspekata nego i na osnovu svih drugih informacija koje su nam raspoložive u tom trenutku.

4.5 Apstrahovanje i dekomponovanje

Proces projektovanja u velikoj meri počiva na dva osnovna postupka – apstrahovanju i dekomponovanju. Usklađena i naizmenična primena apstrahovanja i dekomponovanja čini kostur puta do dobrog dizajna – apstrahovanjem dobijamo na opštosti rešenja, a dekomponovanjem preciziramo njegovu strukturu.

Apstrahovanje

Apstrahovanje predstavlja uopštavanje karakteristika nekog posmatranog problema i njegovih mogućih rešenja. Apstrahovanjem težimo da rešenje problema odvojimo od konkretnih okvira i specifičnosti, tako da dobijemo rešenje na relativno visokom konceptualnom nivou, u obliku u kome može da se primeni ne samo na konkretan razmatran problem već i na širi skup srodnih problema. Jedna od osnovnih karakteristika apstrahovanja je zanemarivanje pojedinosti koje su značajne za pojedinačne slučajeve, ali ne utiču značajno na opšte konceptualno rešenje. Pri tome je važno da se razmatraju i pojedinačni slučajevi, zato što bi u suprotnom moglo da se desi da napravljeni opšti model ne obuhvata neke od njih. Nije retkost ni da zbog različitih vidova ograničenja pri posmatranju problema (vreme, raspoloživost dokumentacije, poverljivost i sl.) projektant bude primoran da apstrahuje na osnovu suženog broja dobro dokumentovanih slučajeva.

Apstrahovanje može da ima nekoliko različitih vidova, među kojima se kao najvažniji izdvajaju:

- klasifikacija,
- generalizacija,
- oblikovanje servisa i
- komponovanje.

Klasifikacija je prepoznavanje zajedničkih karakteristika u nekom skupu posmatranih objekata i definisanje *klase* koja modelira sve te objekte. Najčešće se odnosi na prepoznavanje klasa u domenu implementacije (tj. u kontekstu OOP), ali može da se radi i o nekom apstraktnijem konceptu klasifikovanja, kao što su na primer klasifikovanje komponenti, klasifikovanje procesa, klasifikovanje slučajeva upotrebe, klasifikovanje korisnika softvera i drugo.

Postupak suprotan klasifikaciji je *instanciranje* – pravljenje (ili izdvajanje) jednog konkretnog objekta koji je primerak posmatrane klase. Kao i u slučaju klasifikacije, i instanciranje može da se posmatra opštije i da se odnosi na uočavanje konkretnih elemenata nekih klasa, na primer komponenti, procesa, korisnika i slično.

Generalizacija je prepoznavanje zajedničkih karakteristika u nekom skupu posmatranih klasa i izvođenje njihove zajedničke bazne klase, kao opštijeg modela koji opisuje zajedničke osobine svih instanci posmatranih klasa. Predstavlja osnovu izgradnje hijerarhije klasa. Isto kao i klasifikacija, odnosi se prevashodno na modeliranje klasa objekata na nivou implementacije, ali može da se odnosi i na druge apstraktnije koncepte klasa.

Suprotno od generalizacije je *specijalizacija* – oblikovanje klase koja predstavlja jedan specijalniji tip objekata od prethodno posmatrane opštije klase.

Oblikovanje servisa predstavlja prepoznavanje najvažnijih odlika usluga, koje bi neki servis trebalo da pruža svojim korisnicima, uz definisanje što apstraktnijeg interfejsa, koji (1) omogućava zahtevanje svih potrebnih usluga, ali (2) ni na koji način ne otkriva (i samim tim ne ograničava niti prejudicira) način implementacije servisa (tj. njegovu internu strukturu i enkapsulirani deo ponašanja). Oblikovanje servisa se radi na svim nivoima projektovanja. Na višim nivoima projektovanja se oblikovanje servisa odnosi na definisanje interfejsa komponenti, dok se na nižim nivoima projektovanja ovaj vid apstrahovanja odnosi na oblikovanje interfejsa klasa. Dobro oblikovanje servisa omogućava da se kasnije uspešno sprovedu implementacija i enkapsulacija.

Suprotan postupak je *otvaranje servisa*, tj. otkrivanje specifičnosti implementacije servisa, da bi detalji implementacije mogli da se upotrebe u specifičnim kontekstima. Na taj način posmatrani servis praktično prestaje da bude servis u punom smislu te reči i poprima neke odlike biblioteke. Smanjuje se nivo apstrakcije i obično postaje neophodno da se revidira prethodno izvršena dekompozicija.

Komponovanje je vid apstrahovanja kojim se prepoznaje da neki skup objekata može da se posmatra kao celina iz „spoljnog“ sveta. Podrazumeva grupisanje objekata u celinu i definisanje granica te celine prema spoljnom svetu. Nakon definisanja granica obično je neophodno, a skoro uvek je preporučljivo, da se primeni oblikovanje servisa. Komponovanje može da se odnosi na više klasa, koje sadrađuju u obavljanju nekog posla, kojom prilikom se pravi tzv. *fasada*, koja predstavlja interfejs celine i zaklanja ostale klase koje tu celinu implementiraju.

Slično, komponovanje može da se odnosi i na komponente ili servise, kada se oblikovanjem nove komponente definiše opštiji ili suženi interfejs prema nekim složenijim upotrebama manjih servisa, da bi se njihova upotreba učinila jednostavnijom.

Postupak suprotan komponovanju je *dekomponovanje* (u užem smislu). Kao što ćemo videti u nastavku, odnos suprotstavljenosti apstrahovanja (u vidu komponovanja) i dekomponovanja najviše dolazi do izražaja kada se radi o *funkcionalnom* komponovanju i dekomponovanju.

Kao što možemo da primetimo da klasifikacija, generalizacija, oblikovanje servisa i komponovanje predstavljaju različite vidove apstrahovanja, tako možemo da kažemo i da instanciranje, specijalizacija i otvaranje servisa predstavljaju različite vidove dekomponovanja.

U početnim koracima projektovanja, kada se prepoznaju i oblikuju komponente i njihovi odnosi, apstrahovanje je najprisutnije u vidu komponovanja i oblikovanja servisa, mada može da bude mesta i za klasifikaciju i generalizaciju. Nasuprot tome, u kasnijim fazama projektovanja, kada se određuje unutrašnja struktura (dizajn) komponenti, glavnu ulogu imaju klasifikacija i generalizacija, dok se oblikovanje servisa primenjuje u sklopu definisanja interfejsa klasa, a komponovanje se tada nešto manje koristi.

Dekomponovanje

Dekomponovanje je postupak postepenog uvođenja sve više pojedinosti u uopšteni model softvera, kroz prepoznavanje manjih celina koje čine taj sistem. Dekompozicijom nekog sistema se dobija model tog sistema na nešto nižem nivou apstrakcije. Dobijeni model bi trebalo da ima konkretniju i jasniju strukturu, tako da bude nešto bliži implementaciji nego što je to bio polazni model. Dekomponovanje nazivamo i *razlaganjem* rešenja. Razlaganje celine na manje delove se preduzima iz različitih motiva. U zavisnosti od kriterijuma razdvajanja celina imamo dva osnovna vida dekomponovanja: *funkcionalno dekomponovanje* i *logičko dekomponovanje*. Veoma je značajno i tzv. *dekomponovanje prema promenljivosti*.

Funkcionalna dekompozicija predstavlja podelu celine na *komponente*, koje predstavljaju manje funkcionalne celine. Svaka komponenta bi trebalo da ima jasno prepoznatljivu ulogu u većem sistemu. U idealnom slučaju jedna komponenta obavlja samo jednu funkciju i za nju je u potpunosti odgovorna. Komponente međusobno sarađuju da bi sistem ispravno funkcionisao, ali pri tome težimo da njihova saradnja bude što manjeg obima i da bude što preciznije definisana. Dodirne tačke komponenti, putem kojih se odvija njihova međusobna komunikacija, nazivamo *interfejsima*.

Funkcionalna dekompozicija projekta se obično odvija u ranijim fazama projektovanja, kada je važno da se prepoznaju sve potrebne funkcije i komponente

koje su za njih zadužene. Od kvaliteta dekomponovanja i dobijene dekompozicije veoma često neposredno zavisi i kvalitet projekta. Pri prepoznavanju komponenti je od suštinskog značaja da se teži njihovoj što većoj samostalnosti. Komponenta bi trebalo da nema posebne pretpostavke u odnosu na kontekst u kome se koriste njene usluge, zato što bilo kakve pretpostavke o kontekstu neminovno umanjuju upotrebljivost komponente, kako u slučaju promene okolnosti (uslova poslovanja ili rada) tako i u slučaju eventualne primene komponente u nekom drugom projektu.

Pri funkcionalnom dekomponovanju često smo u prilici da *izmišljamo* nove koncepte da bismo lakše podelili celinu na delove. To je obično dobar pristup i malo ćemo ga detaljnije razmotriti u poglavlju *6 - Principi projektovanja softvera*, kada budemo govorili o principu *Izmišljotina*. Međutim, izmišljanje može projektanta da uvuče u zamku *umišljanja* funkcionalnosti radi *lepše* podele celine na delove. Tipičan oblik upadanja u ovu zamku je prepoznavanje komponenti uz odlaganje definisanja interfejsa. Takav pristup može da izgleda privlačno, zato što nam omogućava da brže dođemo do grube funkcionalne strukture softverskog sistema, dok detalje (interfejse) ostavljamo za kasnije zato što nam se čini da će to biti jednostavniji deo posla. Međutim, to može i da nam napravi ozbiljne probleme. Ako neka komponenta nema jasan interfejs, onda to obično znači da ona nema jasnu funkciju ili jasnu razdvojenost od ostalih komponenti, odnosno da smo pri njenom prepoznavanju u manjoj ili većoj meri *umislili* njenu funkcionalnost. Ako imamo u vidu da razvoj različitih komponenti može da bude predmet rada različitih timova, onda možemo da razumemo i koliko su dalekosežne posledice takvih grešaka – one imaju potencijal da se odraze na kvalitet i efikasnost rada velikog broja ljudi.

Radi ilustracije funkcionalne dekompozicije možemo da razmotrimo primer elektronskog naručivanja proizvoda, bilo putem veba ili preko telefona. Nakon što korisnik izabere proizvode koje želi da kupi i unese podatke o isporuci i plaćanju, implementacija naručivanja mora da izvede više operacija, među kojima su rezervisanje proizvoda u magacinu, elektronsko plaćanje, zakazivanje isporuke i drugo. Neke od tih operacije su međusobno potpuno nezavisne, pa ima smisla da se prepoznaju različite komponente, sa odgovarajućim funkcionalnostima. Na primer, mogli bismo najpre da uvedemo komponentu `Naručivanje`, koja je zadužena da sprovede kompletan posao naručivanja:

- Komponenta `Naručivanje` je odgovorna za naručivanje proizvoda. Interfejs `INaručivanje` je:

```
int naručiProizvode( ...podaci o kupcu...,
                     ...adresa...,
                     Narudžbenica& n )
```

U narednom koraku bismo primenili funkcionalnu dekompoziciju, da iz tako zamišljene komponente `Naručivanje` izdvojimo neke zaokružene relativno

nezavisne celine. Na primer, možemo da izdvojimo staranje o magacinu, plaćanju i isporučivanju. To su tri potpuno odvojene vrste poslova, pa možemo da oblikujemo tri različite komponente koje bi imale odgovarajuće funkcionalnosti:

- Komponenta `Magacin` je odgovorna za proveru stanja proizvoda u magacinu, rezervisanje proizvoda, evidentiranje prijema robe i evidentiranje isporuke. Jedina usluga ove komponente, koja se koristi neposredno pri naručivanju, je rezervisanje proizvoda. Interfejs `IMagacin` bi mogao da izgleda otprilike ovako:

```
int dodajProizvod( int idProizvoda, int količina )
int rezerviši Proizvod( int idNarudžbenice,
                        int idProizvoda,
                        int količina )
int poništiRezervaciju( int idNarudžbenice )
int evidentirajIsporuku( int idNarudžbenice,
                         int idProizvoda,
                         int količina )
int prijemRobe( ... )
```

- Komponenta `Plaćanje` je odgovorna za obavljanje elektronskog plaćanja u ime kupca, putem odgovarajućeg servisa poslovne banke. Interfejs `IPlaćanje` bi mogao da bude:

```
int naplati( ...podaci o kupcu i platnoj kartici...,
             int idNarudžbenice,
             double iznos )
```

- Komponenta `Isporučivanje` je odgovorna za procenu cene isporuke prilikom naručivanja i za razmenu podataka sa servisom za isporučivanje. Interfejs `IIsporučivanje` bi mogao da bude:

```
double proceniVrednostIsporuke( double ukupnaZapremina,
                                double ukupnaMasa,
                                double vrednost )
time_point zakažiIsporuku( ...adresa...,
                           ...paketi... )
```

Na kraju, potrebno je da popravimo opis komponente `Naručivanje`, tako da se vidi da se ona implementira korišćenjem ostalih opisanih komponenti:

- Komponenta `Naručivanje` je odgovorna za naručivanje proizvoda. Jedini podaci koje ona neposredno menja su podaci o stanju narudžbenice. Za sve ostalo koristi usluge drugih komponenti. Interfejs `INaručivanje` se ne menja.

Opisana dekompozicija predstavlja primer funkcionalne dekompozicije, zato što je kao osnovni kriterijum za razdvajanje odgovornosti razmatrana njihova funkcionalnost – svaka od komponenti ima jasno definisanu funkciju, koja se ne preklapa sa drugim komponentama.

Dekompozicija na komponente se ilustruje dijagramom komponenti UML-a. Dijagrame UML-a ćemo upoznati u narednom poglavlju, pa će tamo biti naveden i primer dijagrama komponenti koji ilustruje opisanu dekompoziciju (*Slika 16*, na strani 100).

Funkcionalna dekompozicija se odvija i na nižem nivou, pri dizajniranju delova softvera. Prepoznavanje različitih „pomoćnih" funkcija koje su nam potrebne da bismo mogli da implementiramo „glavnu" funkciju obično ima za rezultat dodavanje novih strukturnih elemenata softvera, koji kroz međusobnu komunikaciju i kroz obavljanje svojih pojedinačnih funkcija, omogućavaju implementaciju „glavnog" posla. Funkcionalna dekompozicija pri dizajniranju softvera često može da se osloni na neke uobičajene obrasce dekomponovanja, o čemu će biti više reči u poglavlju *7 - Obrasci za projektovanje* i posebno u delu o obrascima ponašanja.

Greške pri funkcionalnom dekomponovanju uopšte nisu retke. Štaviše, one su do te mere i česte i ozbiljne da neki istraživači smatraju da funkcionalno dekomponovanje treba izbegavati [*Lowy 2019*]. Umesto toga se predlaže *dekomponovanje prema promenljivosti*. Ideja je da se prepoznaju glavne *tačke promenljivosti* (mesta na kojima može doći do promena usled promena okolnosti ili prilagođavanja dela softvera za druge primene) i glavne *ose promenljivosti* (pravci distribuiranja promena, koji su obično određeni zavisnostima delova sistema od tačaka promenljivosti) i da se zatim vrši dekomponovanje sa ciljem da se tačke promenljivosti međusobno razdvoje, a ose promenljivosti grupišu, tako da eventualno nastupanje jednog uzroka promena najčešće ne može da proizvede kao posledicu izazivanje većeg broja novih promena.

Dobra strana ovakvog pristupa je što pruža potencijal da se kao rezultat dobije čvršća arhitektura, koja je veoma otporna na promene. Međutim, ni takav pristup nije bez potencijalnih slabosti. Jedna od slabosti je da je za utvrđivanje tačaka i osa promenljivosti potrebno da se makar započne detaljnije projektovanje, pa je zato za uspešno dekomponovanje prema promenljivosti poželjno da se obezbede ili veliko iskustvo projektanata ili učestalo smenjivanje opštijeg i detaljnijeg projektovanja. Druga slabost je u tome što tačke i ose promenljivosti mogu da se prepoznaju praktično bilo gde i bilo kad, pri čemu procenjivanje njihovog značaja nije uvek jednostavno. Ako istaknemo previše tačaka i osa promenljivosti, među kojima neke imaju sasvim ograničen značaj, dobićemo mnogo komplikovaniji model nego što je potrebno. Sa druge strane, ako propustimo da istaknemo neku značajnu tačku ili osu promenljivosti, onda će model biti osetljiviji na promene nego što bi trebalo da bude.

Zbog toga je neophodno da pri funkcionalnom dekomponovanju veoma pažljivo vodimo računa pre svega o jasnom prepoznavanju i razdvajanju odgovornosti komponenti i njihovoj što manjoj međusobnoj zavisnosti (*4.7 Kohezija i spregnutost*), ali i o drugim osnovnim principima projektovanja (*6 - Principi projektovanja softvera*). U cilju postizanja veće stabilnosti rešenja, dobro je da se funkcionalno dekomponovanje kombinuje sa dekomponovanjem prema promenljivosti. Osnovna ideja takvog pristupa je da se rezultat funkcionalne dekompozicije u svakom koraku posmatra kritički iz ugla dekomponovanja prema promenljivosti. Najpre se u dobijenom modelu prepoznaju i lokalizuju ključne tačke i ose promenljivosti, tj. identifikuju se tačke promenljivosti u različitim komponentama i ustanovi se koja osa promenljivosti povezuje koje tačke i koje komponente. Zatim se dobijeni model dalje preoblikuje, pri čemu je cilj da se tačke promenljivosti što bolje međusobno razdvoje u različite komponente (da bi bilo što manje mogućih uzroka za menjanje svake pojedinačne komponente), a da se ose promenljivosti koje polaze iz iste tačke grupišu (da bi se što više smanjio broj komponenti, koje će morati da se menjaju ako se menja ona komponenta u kojoj je ta tačka promenljivosti). Pri tome često moramo da se oslonimo na detaljnije analiziranje i bar delimično strukturno modeliranje posmatranih celina.

U narednom poglavlju ćemo se posvetiti principima projektovanja. Kao što ćemo videti, značajan deo tih principa se odnosi upravo na različite kriterijume za određivanje granica između celina (na primer, komponenti ili klasa). Neki od principa se neposredno tiču tačaka ili osa promenljivosti, dok se neki drugi tiču odgovornosti ili funkcija koje ima neka celina. Primena principa projektovanja ima za cilj da nam pomogne da se fokusiramo na najvažnije aspekte modeliranja, kao i da nam omogući da sprečimo ili bar na vreme prepoznamo neke od uobičajenih grešaka koje se prave pri projektovanju. Videćemo da nas principi projektovanja upućuju da izvodimo dekomponovanje upravo u skladu sa dobrom praksom funkcionalne dekompozicije i dekompozicije prema promenljivosti.

Logička dekompozicija predstavlja podelu celine na *pakete* (ili druge strukturne celine), koji grupišu delove implementacije koji se zajedno razvijaju ili imaju izražene unutrašnje međuzavisnosti strukture ili ponašanja. Delovi implementacije koji čine jedan paket ne moraju da budu funkcionalno povezani, mada je to čest slučaj. Takođe, ne moraju da predstavljaju ni potpuno zaokruženu funkcionalnu celinu. Uobičajena praksa je da se elementi paketa povezani logički a ne funkcionalno. Primer paketa logički povezanih elemenata je paket klasa koje implementiraju hijerarhiju čvorova apstraktnog drveta izraza. Sve te klase su strukturno povezane, imaju jaku logičku vezu i koriste se zajedno, ali među njima ne postoji jaka funkcionalna veza, osim što su sve klase hijerarhije funkcionalno zavisne od bazne klase.

Osim funkcionalne i logičke povezanosti, paketi relativno često mogu da sadrže i delove različitih komponenti koji se na sličan način implementiraju. Možemo reći da

se tu radi o vidu strukturne povezanosti elemenata paketa. To je svakako manje poželjno od funkcionalne i logičke povezanosti ali može da bude opravdano. Na primer, paket može da sadrži nekoliko osnovnih klasa hijerarhije koja se koristi za obezbeđivanje komunikacije među komponentama.

Logička dekompozicija projekta se odvija u svim fazama projektovanja, ali je posebno značajna u fazi planiranja implementacije, kada se prepoznaju i oblikuju paketi. Nije pogrešno reći ni da pravljenje paketa predstavlja logičko *komponovanje* prepoznatih klasa u celine koje će se zajedno implementirati. Razlika je u tome u kojoj fazi oblikujemo pakete. Ako pakete prepoznajemo i definišemo na osnovu prepoznatih komponenti i površnih ali još uvek nedovoljno razrađenih ideja o unutrašnjoj strukturi komponenti, onda možemo da kažemo da je to *logička dekompozicija*, zato što delimo celinu na pakete. Sa druge strane, ako pakete pravimo nakon što napravimo detaljan strukturni model implementacije komponenti, onda je to *logička kompozicija*, zato što grupišemo delove (klase) u celine (pakete).

Za sve vidove dekomponovanja je zajedničko da se celina deli na manje delove (obično se nazivaju *delovi* ili *moduli*), čijim dobrim *razdvajanjem* bi trebalo da se spušta nivo složenosti delova projekta, a čijim *povezivanjem* bi trebalo da se ostvaruje celovitost i puna funkcionalnost softvera. Od uspešnosti ostvarivanja ravnoteže između razdvajanja i povezivanja modula (bilo da su u pitanju komponente, paketi ili neke druge vrste delova) često presudno zavise kvalitet projekta i uspeh implementacije.

Iterativno sprovedeno dekomponovanje nam postepeno daje za rezultat sve preciznije i konkretnije oblikovanu strukturu softvera. Dizajn i arhitektura softverskog sistema nam tako opisuju dekompoziciju softvera na njegove strukturne elemente, na različitim nivoima apstrakcije. Pri tome pod strukturnim elementima obično podrazumevamo komponente, module, servise, klase, metode, funkcije, pakete, strukture podataka i sve druge elemente koji čine razvijeni softver.

4.6 Procenjivanje kvaliteta projekta

Kvalitet projekta možemo da procenjujemo kvalitativno i kvantitativno. Kvalitativno procenjivanje počiva na analiziranju projekta i razvijenog softvera i razmatranju njihovih karakteristika. Neke od karakteristika softvera prepoznajemo kao dobre ili poželjne, dok neke druge smatramo za loše ili nepoželjne. Sagledavanjem poželjnih i nepoželjnih karakteristika nekog projekta i njihovim upoređivanjem sa drugim projektima ili nekim prethodno utvrđenim planom, možemo da izvedemo kvalitativnu procenu projekta.

Kvantitativno procenjivanje počiva na definisanju i izračunavanju različitih numeričkih ocena i njihovom analiziranju. Pri kvalitativnom i kvantitativnom procenjivanju projekta posmatraju se uglavnom iste karakteristike, ali na različite

načine – u jednom slučaju ih opisujemo i upoređujemo prema načinu i obliku ostvarivanja, a u drugom ih predstavljamo numerički i poredimo dobijene brojeve.

Problemom *merenja* različitih karakteristika projekata, ali i drugih elemenata razvojnog procesa, bavi se disciplina *softverske metrike*. Razlikujemo dve osnovne vrste metrika: metrike razvoja, koje nam služe da pratimo različite parametre toka razvojnog procesa, i metrike dizajna, koje nam pomažu da u numeričke vrednosti prevedemo neke karakteristike dizajna softvera, da bismo mogli da ih lakše upoređujemo. Merenje karakteristika softvera jeste moguće ali nam u praksi ne pruža odgovore na sva pitanja sa kojima se susrećemo. U idealnom slučaju bismo mogli da egzaktno izmerimo kvalitet projekta, ali to najčešće ili nije moguće, ili nije lako, ili nije sasvim jasno kako da tumačimo tako dobijenu ocenu.

Poželjne karakteristike projekta su povezane sa poželjnim karakteristikama softvera koji se razvija. Kao najvažnije karakteristike softverskog proizvoda se obično prepoznaju karakteristike koje su vidljive kako iz ugla razvojnog tima tako i iz ugla korisnika:

- ispravnost i
- efikasnost.

Ispravnost softvera podrazumeva da se softver ponaša u potpunosti u skladu sa funkcionalnom specifikacijom, tj. da sve funkcije softvera rade onako kako je dokumentovano, bez neispravnih rezultata ili otkazivanja. *Efikasnost* podrazumeva da softver radi uz planirano zauzeće resursa, uključujući ne samo procesorsko vreme nego i radnu memoriju, komunikacione linije, skladišni prostor i sve druge resurse računarskog sistema na kome se izvršava. Od početka XXI veka se efikasnost sve više razmatra i iz ugla utroška energije i prirodnih resursa[18].

Nesumnjiv je značaj ispravnosti i efikasnosti u procenjivanju vrednosti nekog softvera. Softver koji ne radi ispravno ili dovoljno efikasno, obično zbog toga ne ostvaruje ni ciljeve zbog kojih je razvijan, što je dovoljan razlog da ove dve karakteristike obično stavljamo u prvi plan. Sa druge strane, pri procenjivanju kvaliteta softvera se često pravi greška tako što se razmatranje i procenjivanje zadržavaju samo na onim karakteristikama softvera koje mogu (lakše ili teže) da se

[18] Utrošak energije tokom izračunavanja utiče na njegovu cenu, ali ima za posledicu i zagađenje prirode (na primer stvaranje CO_2) ili utrošak nekog prirodnog resursa. Zbog toga u slučaju zahtevnih izračunavanja može da bude važno da se uspostavi odgovarajući balans između brzine i utroška energije ili drugih resursa. Time se bave *računarstvo svesno energije* (engl. *energy-aware computing*) i *računarstvo svesno resursa* (engl. *resource-aware computing*) [*Pereira 2017*].

sagledaju od strane korisnika, što se često svodi na ispravnost i efikasnost[19]. Iako to nije uvek očigledno, ipak moramo da imamo u vidu da nije jedina namena razvijenog softvera da nešto radi ispravno i efikasno, pa zbog toga moramo da razmatramo i neke druge karakteristike softvera.

Kada razvojni tim (ili neko preduzeće za razvoj softvera) napravi neki softver, to obično radi da bi se ispunili zahtevi ugovora sa nekim klijentom, ili da bi se taj softver stavio na tržište i na taj način doneo prihode ili da bi primenom tog softvera mogla da se ostvari neka korist, ili da bi se ostvarila korist za opštu zajednicu, ili radi nekih drugih benefita kao što su prestiž, slava, lično zadovoljstvo i drugo. U svakom od ovih slučajeva, naravno, softver mora da radi ispravno i dovoljno efikasno – to je osnovni preduslov da bi bio upotrebljiv. Ali tu nije kraj – nakon što je konkretan softverski proizvod razvijen, a često čak i pre toga, obično postoji prostor da se u softveru kao celini ili u nekim njegovim delovima prepozna potencijal za ostvarivanje dodatne koristi. Na primer, možda bi modifikovana verzija softvera mogla da ima neke dodatne primene, ili produženu upotrebljivost, ili bi možda mogla da ima šire potencijalno tržište, ili bi možda neki delovi razvijenog softvera mogli da se upotrebe u nekom drugom projektu? Znači, ako bismo bili u stanju da relativno lako ponovo upotrebimo bilo ceo projekat bilo neki njegov deo, onda bismo bili u prilici da uz relativno malo dodatnog uloženog napora dobijemo relativno veliku korist.

Ponovna upotrebljivost softvera ili delova softvera je povezana sa nekim veoma važnim karakteristikama, koje se podjednako odnose na celovit softverski proizvod i na njegov projekat (odnosno dizajn). To su, pre svega:

- fleksibilnost;
- proširivost i
- modularnost.

Kažemo da je softver *fleksibilan* ako postoji potencijal da se čitav softver ili njegovi delovi uz relativno male izmene, ili čak bez izmena, prilagođavaju promenama u okruženju ili primenjuju u sasvim novim okolnostima. Softver je *proširiv* ako je značajan broj sagledivih eventualnih dopuna moguće implementirati samo dodavanjem novih elemenata softvera ili uz eventualno minimalno menjanje njegovih postojećih delova.

Fleksibilnost i proširivost softvera u velikoj meri zavise od kvaliteta funkcionalne dekompozicije softvera. Za softver kažemo da je *modularan* ako se sastoji od dobro

[19] Uz ispravnost i efikasnost tu su još i neki estetski i upotrebni elementi softvera, koji svakako imaju važnu ulogu, posebno iz ugla korisnika, ali koji nisu u našem fokusu.

dekomponovanog skupa komponenti (tj. modula), što znači da je složena celina podeljena na manje delove, čija je funkcionalnost jasno definisana. Ako je softverski sistem modularan, onda očekujemo da, umesto da razvijamo softver kao jednu veliku celinu, možemo da pristupimo razvoju svakog modula nezavisno i da zatim razvijene module možemo relativno jednostavno da povežemo u celinu. Osim što je modularnost neophodna da bismo ostvarili fleksibilnost i proširivost, ona je neophodna i za agilan razvoj. Štaviše, savremeni razvoj teži tome da modularizacija (tj. dekompozicija na module) ide toliko daleko (tj. da bude toliko detaljna) da svaki pojedinačan prepoznat deo softvera može da se razvije u okviru jedne iteracije, tj. u toku jednog relativno kratkog razvojnog ciklusa.

Modularizacija nije uvek dovoljna da bi se obezbedilo da povezivanje napravljenih delova bude dovoljno jednostavno, pa čak ni ostvarivo. Zbog toga se uvode dodatni zahtevi, u vidu prepoznavanja dodatnih važnih karakteristika koje želimo da imaju komponente softverskog proizvoda:

- razdvojenost odgovornosti komponenti;
- preciznost i jasnoća interfejsa;
- doslednost enkapsulacije;
- visoka kohezija i
- niska spregnutost.

Jasna razdvojenost odgovornosti komponenti podrazumeva da se odgovornosti pojedinačnih komponenti ne preklapaju, odnosno da jedna komponenta potpuno samostalno obavlja posao za koji je zadužena[20]. Eventualni izostanak razdvojenosti odgovornosti može da ima za posledicu da neki deo posla ne obavlja jedna komponenta samostalno, već više komponenti zajedno. To može da značajno oteža kako razvoj tih pojedinačnih komponenti (zato što funkcionalnost i implementacija jedne komponente potencijalno značajno utiču na rad druge) tako i njihovo povezivanje u celinu.

Razdvajanje odgovornosti je obično veoma tesno povezano sa definisanjem preciznih i čistih interfejsa komponenti. Interfejs neke komponente predstavlja sredstvo putem koga druge komponente mogu da koriste funkcionalnosti te komponente. Interfejs može da se posmatra i kao deklaracija ponašanja (tj. deklaracija funkcionalnosti) jedne komponente. Ako su odgovornosti dobro razdvojene, onda interfejsi mogu da se definišu tako da budu uski, precizni i jasni.

[20] Kroz razmatranje nekih osnovnih principa projektovanja softvera u narednom poglavlju, videćemo da ćemo u praksi da očekujemo i zahtevamo i neke druge stvari, kao na primer da jedna komponenta, pored toga što obavlja samostalno posao za koji je zadužena, ne radi ništa drugo osim tog posla.

Pod širinom (ili veličinom) interfejsa podrazumevamo broj tačaka povezivanja (tj. broj funkcija ili klasa koje se vide od strane drugih komponenti) i broj parametara koji se pri povezivanju razmenjuju. Ako je interfejs širok, onda to obično znači da komponenta radi više od jednog posla ili da je povezivanje komponenti putem tog interfejsa intenzivnije nego što bi trebalo da bude, zato što je obavljanje jednog posla podeljeno između komponenti i nisu dobro razdvojene nadležnosti. Velika širina interfejsa često može da bude posledica postojanja više sličnih funkcija ili metoda, koji služe za obavljanje različitih vrsta nekog posla ili radi prenošenja parametara u različitim oblicima. Za takav interfejs kažemo da je *neprecizan* i to može da bude znak loše podele odgovornosti. Obično je bolje da se obezbedi uži i precizniji interfejs, a da se načini upotrebe parametrizuju prenošenjem strukture podataka koja sadrži potrebne informacije.

Jasnoća (ili čistoća) interfejsa je takođe povezana sa razdvajanjem nadležnosti. Ako komponenta obavlja samo jedan posao, onda je njen interfejs obično vrlo jasan i svi elementi interfejsa su međusobno međuzavisni, u smislu da omogućavaju da neka druga komponenta upotrebi ovu komponentu radi obavljanja tog jednog posla. Međutim, ako komponenta obavlja više poslova, onda povezanost delova interfejsa može da ne bude očigledna ili da čak uopšte ne bude jasno da li među delovima interfejsa postoji ikakva veza.

Pojam enkapsulacije je čitaocima poznat iz OOP, gde se vezivao prvenstveno za klase. U kontekstu projektovanja softvera, klasa nije ništa drugo do jedan vid komponente (ili modula), tako da mnogo toga što je ranije upoznato na primerima klasa, sada može (i mora) da se primeni i na opštiji slučaj komponente. Svaka komponenta predstavlja jednu celinu koja ima svoju funkcionalnost, koja je opisana kroz ponašanje ali i kroz sadržaj komponente. Kao što se teži da sadržaj objekata klase ne bude dostupan nikome van konkretnog objekta, tako se isto teži i da sadržaj komponente bude enkapsuliran. Dobra enkapsulacija je tesno povezana sa dobrim interfejsima i dobrim razdvajanjem nadležnosti komponenti – ako su nadležnosti dobro razdvojene onda obično nije teško definisati dobre interfejse putem kojih ne može da se pristupa enkapsuliranom sadržaju komponenti.

Kohezija i spregnutost predstavljaju verovatno dve najvažnije karakteristike softverskih projekata. Imaju i prilično složene aspekte i veoma veliki uticaj. U narednim poglavljima ćemo videti da je značajan broj principa projektovanja i obrazaca za projektovanje neposredno povezan sa ovim pojmovima, a praktično svi imaju makar posrednu vezu sa njima. Zbog toga ćemo ovim pojmovima posvetiti posebnu pažnju i poveći odeljak.

4.7 Kohezija i spregnutost

Kao što smo već naglasili, kvalitet projekta i uspešnost implementacije često presudno zavise od uspešnosti ostvarivanja ravnoteže između razdvajanja i

povezivanja modula. Da bismo mogli da ocenimo kvalitet dekompozicije, neophodno je da posmatramo i da na neki način merimo i procenjujemo kvalitet povezanosti odnosno razdvojenost modula. Dve osnovne mere koje pri tome uvodimo su *kohezija* i *spregnutost*:

- kohezija je stepen međusobne povezanosti elemenata jednog modula;
- spregnutost (engl. *coupling*) je stepen međusobne povezanosti elemenata različitih modula.

Dobro dizajniran sistem se odlikuje *visokom kohezijom* i *niskom spregnutošću*. Visoka kohezija u nekoj strukturnoj celini znači da su svi delovi te celine neophodni da bi ona mogla da ispunjava svoju svrhu, a uz to su još i čvrsto međusobno povezani. Sa druge strane, između različitih strukturnih celina postoji niska spregnutost ako su te celine relativno slabo međusobno povezane.

Praktično svaka priča o principima dobrog projektovanja i sve preporuke koje nas upućuju na to kako da pravimo dobre projekte, na kraju se svode na pronalaženje balansa između kohezije i spregnutosti. Svaka priča o primeni apstrahovanja i dekompozicije takođe se na kraju svodi na koheziju i spregnutost. Bez imalo preterivanja se može reći da kohezija i spregnutost predstavljaju ključne koncepte u oblasti projektovanja softvera. U ovom odeljku ćemo im posvetiti zasluženu pažnju, dok ćemo se praktičnim načinima uspostavljanja neophodne ravnoteže među njima baviti u poglavlju *6 - Principi projektovanja softvera*.

Koheziju i spregnutost možemo (i moramo) da razmatramo na različitim nivoima funkcionalnog i strukturnog dekomponovanja sistema. Uobičajeno je da ih procenjujemo i na osnovu njih ocenjujemo kvalitet dekompozicije softverskog projekta na komponente, pakete, pa i na klase. Kohezija i spregnutost su apstraktne mere, koje se veoma često izražavaju samo opisno. Njihov veliki značaj i potreba da ih merimo i upoređujemo su motivisali istraživače da pronađu načine za njihovo numeričko izražavanje, čime se bavi disciplina softverske metrike.

4.7.1 Kohezija

Visoka kohezija jednog modula znači da su svi delovi tog modula međusobno snažno povezani. Ako posmatramo jedan modul kao deo neke šire celine, koji okuplja neke srodne elemente, onda je očekivano da ti elementi budu nekako povezani, bilo da je ta povezanost funkcionalna (na primer, delovi jedne komponente sarađuju radi ostvarivanja funkcije komponente) ili logička (na primer, delovi jednog paketa su implementaciono međuzavisni).

Sa druge strane, niska kohezija u nekom modulu znači da pojedini delovi tog modula nisu međusobno povezani ili su vrlo slabo međusobno povezani. Niska kohezija nam obično ukazuje na to da dekompozicija možda nije izvedena dovoljno

dobro i da granice modula nisu ispravno određene. Na primer, ako u modulu možemo da prepoznamo manje grupe njegovih elemenata koji su međusobno snažno povezani, a pri tome su relativno slabo povezani sa ostatkom modula, onda bi verovatno trebalo nastaviti dekomponovanje, tj. podeliti modul na više manjih modula. Niska kohezija može da bude posledica dodeljivanja više funkcija ili odgovornosti jednom modulu.

Razlikujemo nekoliko *vrsta* kohezije. Navešćemo ih redom od najviših i najpoželjnijih, prema onima koje su najslabije i najmanje poželjne, a zatim ćemo ih u istom poretku i razmatrati:

- funkcionalna kohezija;
- sekvencijalna kohezija;
- komunikaciona kohezija;
- proceduralna kohezija;
- vremenska kohezija;
- logička kohezija i
- koincidentna kohezija.

Funkcionalna kohezija

Funkcionalna kohezija se odnosi na povezanost delova modula na osnovu međusobne funkcionalne zavisnosti, a u cilju ostvarivanja funkcije za koju je modul odgovoran.

Funkcionalna kohezija predstavlja osnovu funkcionalne dekompozicije, tj. podele celine na komponente. Dobijamo je tako što dekompozicijom dolazimo do komponente koja ima neku jasno prepoznatu funkciju, koja se ostvaruje kroz međusobnu saradnju elemenata koji čine tu komponentu. Svi delovi jedne komponente su prikupljeni da zajedno čine tu komponentu sa jednim istim osnovnim ciljem – da bi komponenta mogla da ispunjava svoju funkciju. Pri tome elementi modula međusobno sarađuju, tj. među njima postoji funkcionalna zavisnost. Važna karakteristika funkcionalne kohezije je da je svaki od delova komponente neophodan za ostvarivanje njene funkcije, tj. nijedan deo komponente nije suvišan.

U idealnom slučaju, funkcionalna dekompozicija bi trebalo da se zaustavi na modulima u kojima postoji samo funkcionalna kohezija.

Na nivou klase, možemo da kažemo da je idealan slučaj funkcionalne kohezije kada (1) klasa ima dobro oblikovan interfejs koji omogućava da klasa obavlja svoj posao i (2) svi ostali metodi su neophodni da bi metodi interfejsa mogli da se implementiraju, tj. upotrebljavaju se u njihovoj implementaciji.

Sekvencijalna kohezija

Kohezija je *sekvencijalna* ako su elementi modula projektovani tako da rezultat jednog elementa predstavlja ulazne podatke drugog elementa.

Pri sekvencijalnoj koheziji ne postoji puna funkcionalna zavisnost elemenata modula. Delovi sekvencijalne obrade mogu da predstavljaju potpuno različite i međusobno nezavisne poslove. Samim tim je moguće i da postoji problem nejasnog definisanja odgovornosti elemenata modula u odnosu na posao kao celinu. Takođe, ako pri tome neki od elemenata imaju javne interfejse koje mogu da koriste i drugi moduli, onda bi možda moglo da bude bolje da se takvi elementi izdvoje u posebne module.

Ipak, ako su zaobiđeni navedeni problemi, onda sekvencijalna kohezija najčešće predstavlja sasvim prihvatljiv vid kohezije, pa se zato relativno često susreće u praksi. Kao što ćemo videti u daljem tekstu, neki od principa projektovanja (pre svega principi grupisanja) podstiču grupisanje strukturnih elemenata u veće celine upravo na osnovu sekvencijalne kohezije.

Sekvencijalna kohezija ima sličnosti sa proceduralnom kohezijom. U oba slučaja je uobičajeno da neka druga komponenta upravlja tokom postupka, dok elementi posmatrane celine služe za implementiranje pojedinačnih koraka. Osnovna razlika je u tome što se kod sekvencijalne kohezije očekuje da upravljanje postupkom bude praktično trivijalno – radi se o nekoj predefinisanoj sekvenci koraka, dok kod proceduralne kohezije upravljanje može da bude veoma složeno. U oba slučaja se upravljanje postupkom ne izvodi iz istog modula, zato što bismo onda imali funkcionalnu koheziju.

Na nivou klase, za razliku od funkcionalne kohezije, ovde interfejs može da bude nešto širi i da se ne koristi uvek u celosti. Elementi interfejsa predstavljaju korake koji se veoma često (ili čak uvek) koriste u predefinisanom redosledu. Oni ne koriste jedni druge, inače bi to bila funkcionalna kohezija.

Komunikaciona kohezija

Kohezija je *komunikaciona* ako su elementi modula prikupljeni u istu celinu zato što koriste iste podatke.

Kod komunikacione kohezije ne postoji jasna funkcionalna zavisnost između elemenata modula. Slično kao i kod sekvencijalne kohezije, elementi modula obavljaju različite poslove, ali je to ovde još više izraženo, zato što ti poslovi ne moraju da čine korake nekog većeg posla. Zbog toga se kod komunikacione kohezije praktično uvek postavlja pitanje ispravnosti razdvajanja odgovornosti. Moduli oblikovani na osnovu komunikacione kohezije često imaju više odgovornosti, pa i te odgovornosti mogu da budu podeljene na više modula.

Slično kao i u slučaju sekvencijalne kohezije, komunikaciona kohezija je često sasvim prihvatljiva i relativno često se susreće u praksi. I ona se podstiče nekim od principa projektovanja.

Jedan prihvatljiv primer komunikacione kohezije bi bio ako se neka struktura podataka često upotrebljava u većim poslovima, pa se onda u jednoj klasi implementira veći broj operacija nad tom strukturom, koje su često potrebne. Ako među metodima te klase ne bi postojala ni funkcionalna ni sekvencijalna kohezija, onda bi se tu radilo o komunikacionoj koheziji.

Proceduralna kohezija

Kohezija je *proceduralna* ako su elementi modula prikupljeni zato što se koriste pri obavljanju nekog celovitog posla.

Razlika u odnosu na sekvencijalnu koheziju je što sada elementi ne rade kao koraci nekog sekvencijalnog postupka, već može da se radi i o složenijim vidovima saradnje. Kao i u slučaju sekvencijalne kohezije, povezivanje delova posla u celinu se ne odvija u okviru istog modula. Na primer, proceduralna kohezija je na delu ako se u jedan modul stave različite operacije sa datotekama (npr. otvaranje datoteke, proveravanje ispravnosti,...), koje su međusobno relativno nezavisne, ali se često koriste u okviru većih poslova. Slično, mogli bismo reći da klasa `string` okuplja različite metode primenom proceduralne kohezije – iako imamo neke manje skupove metoda koji su interno funkcionalno zavisni, svi zajedno su okupljeni samo zato što predstavljaju operacije koje se često koriste zajedno pri radu sa niskama.

U ovom slučaju najčešće ne postoje funkcionalne zavisnosti među elementima modula, već oni obavljaju potpuno raznorodne poslove. Proceduralna kohezija može da nam ukazuje na nejasno razdvojene odgovornosti, zato što posmatrani modul obično okuplja delove različitih poslova, koji su podeljeni u više modula.

Proceduralna kohezija može da bude opravdana pri pravljenju logičkih celina, kao što su paketi ili klase, ako se na taj način postiže da se jedna klasa potencijalnih promena lokalizuje u tako oblikovanoj celini. Takođe, onda može da pojednostavi upotrebu nekih elemenata njihovim okupljanjem na jedom mestu (na primer, klasa `string`). Takvu praksu podstiču i neki od principa projektovanja (grupisanja), ali bi je trebalo primenjivati samo ako nema osnova da se uspostavi grupisanje uz ostvarivanje nekog poželjnijeg vida kohezije.

Proceduralna kohezija se često primenjuje u obliku para *Kontroler – Kontrolisan*. U okviru kontrolisane klase se implementira veći broj metoda koji su slabo međuzavisni, tj. među njima postoji tek proceduralna kohezija. U okviru kontrolera (klasa, funkcija ili više njih) se kontrolisani objekat i njegovi metodi upotrebljavaju da bi se implementirala neka složenija operacija. Na primer, kontrolisana klasa može da bude `string`, a kontroleri različite funkcije (metodi, klase i sl.) koje korišćenjem objekata i metoda klase `string` obavljaju neki složeni posao. Slično, kontrolisana klasa bi mogla da bude `Slika`, sa implementiranim osnovnim metodima za crtanje ili

obradu slike, a kontroler bi mogao da koristi objekte i metode klase `Slika` da bi nacrtao neki UML dijagram. Kao što ćemo videti u narednim odeljcima, između kontrolera i kontrolisane klase postoji *spregnutost preko kontrole*.

Vremenska kohezija

Vremenska kohezija nastaje kada su elementi modula prikupljeni zajedno zato što se koriste u "istom" periodu vremena, tj. u nekom prepoznatom vremenskom okviru.

Na primer, ako u jednom modulu lociramo različite elemente inicijalizacije nekog sistema, koji obavljaju svoj posao tokom perioda inicijalizacije, ali među njima ne postoji funkcionalna zavisnost, onda se tu radi o vremenskoj koheziji. Kao i u prethodnim slučajevima, ali ovde je to još izraženije, elementi koje povezuje vremenska kohezija obično obavljaju potpuno raznorodne poslove. I ovde se vrlo verovatno radi o nejasnoj podeli odgovornosti. Štaviše, vremenska kohezija nam obično ukazuje da dekompozicija sistema (ili grupisanje elemenata) nije bila zasnovana ni na odgovornostima ni na funkcionalnostima.

Vremenska kohezija ne spada u poželjne vrste kohezije.

Logička kohezija

Kohezija je *logička* ako su elementi modula prikupljeni u celinu zato što imaju logički sličnu (ili čak istu) ulogu u sistemu.

Logička kohezija je, na primer, ako se u jedan modul stave različite transakcije koje se obavljaju u sistemu, ili ako se u jedan modul stave različite operacije čitanje i parsiranja različitih formata dokumenata. Obično se prepoznaje tako što se elementi modula koriste od strane drugih modula u konceptualno sličnim, ali u suštini različitim poslovima. Odlikuje se odsustvom funkcionalne zavisnosti između elemenata, koji obično obavljaju potpuno raznorodne poslove. Takvi elementi često predstavljaju alternativu jedni drugima (npr. čitanje slike u formatu JPEG i čitanje slike u formatu TIFF). Karakteristika ovako napravljenih modula je da imaju veći broj odgovornosti, koje mogu ali ne moraju istovremeno da budu podeljene na više modula.

Logička kohezija može da bude prihvatljiva u paketima, ali je obično vrlo nepoželjna u komponentama i drugim funkcionalnim modulima.

Koincidentna kohezija

Kohezija je *koincidentna*: ako su elementi modula međusobno ili potpuno nezavisni ili su samo veoma slabo povezani. Ona predstavlja najniži nivo kohezije.

Koincidentna kohezija je dobila ime po tome što ona nastaje kada su elementi modula praktično *slučajno* okupljeni u isti modul. Presudan faktor za takvo grupisanje može da bude to što su možda pisani istog dana, ili istim alatom, ili što oni koriste neke slične interfejse i sl. Za ovu vrstu kohezije je karakteristično da među elementima modula ne postoji nikakva ili tek veoma slaba funkcionalna zavisnost.

Odgovornosti uopšte nisu razdvojene, a često ne mogu ni da se jasno raspoznaju, zato što delovi modula rade potpuno raznorodne poslove.

Prisustvo koincidentne kohezije nam ukazuje na to da se nije dovoljno pažnje posvetilo dizajnu odgovarajućih delova softvera. Ona predstavlja najnepoželjniji oblik kohezije.

4.7.2 Spregnutost

Niska spregnutost predstavlja nizak nivo međusobnih zavisnosti između različitih modula. U slučaju dobre dekompozicije, svaki modul predstavlja samostalnu celinu, koja može da koristi usluge drugog modula ili da pruža usluge drugom modulu. Cilj dobrog projektovanja je da se ostvari što bolja enkapsulacija implementacije svake od celina i da se njihovo povezivanje ostvaruje putem što manjih i jednostavnijih interfejsa, što je upravo ekvivalentno niskoj spregnutosti.

Visoka spregnutost predstavlja snažnu povezanost nekog modula sa drugim modulima. Ako jedan modul intenzivno koristi usluge nekog drugog modula, ali kroz veoma čist i jednostavan interfejs, to obično ne predstavlja problem. Međutim, ako je interfejs širok (tj. sastoji se od velikog broja funkcija, metoda ili klasa, ili pruža sasvim različite funkcije), onda je to obično signal da imamo loše izvedenu dekompoziciju. Širok interfejs nekog modula nam obično ukazuje na to da taj modul ima više različitih odgovornosti ili nejasno definisanu odgovornost. Visoka spregnutost između dva modula nam ukazuje na to da možda nije dobro postavljena granica između posmatranih modula, tj. da su neke odgovornosti nejasno ili pogrešno podeljene između njih. Takav slučaj obično može da se popravi ako se moduli spoje u jedan veći, ili se delovi jednog od modula premeste u drugi, ili se potpuno izmeni način dekomponovanja celine kojoj oni pripadaju.

Primetimo da je spregnutost neminovna. Celovit sistem delimo na module radi lakšeg razumevanja, implementiranja i održavanja, ali da bi sistem mogao da funkcioniše kao celina, neophodno je da svaki od njegovih delova bude na neki način povezan sa drugim delovima. U suprotnom, sistem ne bi mogao da se ponaša kao celina, već bi bio skup potpuno odvojenih delova.

To znači da komponente mogu da međusobno sarađuju samo ako postoji neki oblik spregnutosti među njima, bilo neposredno ili posredno. Odsustvo spregnutosti neke komponente je ekvivalentno izolovanosti te komponente. Zbog toga, kada govorimo o niskoj spregnutosti, to nikako ne znači da težimo potpunom eliminisanju spregnutosti, već da težimo da spregnutost između različitih modula ima neke poželjne karakteristike. Moramo da se trudimo da eliminišemo nepoželjne karakteristike spregnutosti, ali ne i samu spregnutost. U narednim odeljcima ćemo da razmotrimo neke od najvažnijih karakteristika spregnutosti i da istaknemo karakteristike koje su poželjne, kao i one koje nisu.

Aksiome spregnutosti

Pri povezivanju komponenti težimo da vodimo računa o tzv. *aksiomama spregnutosti*:

- Što je komponenta složenija, to je važnije da bude što manje spregnuta i
- Spregu je poželjno uvoditi na što jednostavnijim komponentama.

Motivacija za ova pravila se tiče težnje da sprečimo da eventualne promene neke od komponenti indukuju potrebu za menjanje drugih komponenti. U slučaju narušavanja prve aksiome doći ćemo u situaciju da složena komponenta zavisi od mnogo drugih komponenti, pa će zbog tih zavisnosti biti povećana i verovatnoća da bude potrebno da je menjamo – a prirodno je da ne želimo da često menjamo složene komponente. Sa druge strane, ako je komponenta složena to onda znači da postoji više potencijalnih uzroka za njene promene nego ako je jednostavna. Posledica toga je da zavisnosti drugih komponenti od neke složene komponente predstavljaju rizik za propagaciju promena.

Druga aksioma se nadopunjuje sa prvom – ako želimo da složene komponente budu što manje spregnute, a neophodno nam je da nekako povežemo komponente sistema, onda ćemo težiti da neposredno povezujemo što jednostavnije komponente.

Aksiome spregnutosti su povezane sa razmatranjem tačaka i osa promenljivosti u procesu dekomponovanja sistema. Složena komponenta obično sadrži veći broj tačaka promenljivosti. Samim tim, težićemo da ose promenljivosti koje polaze od tih tačaka zadržimo u okviru iste komponente.

Vrste spregnutosti

Osnovne vrste spregnutosti su sprega logike, sprega tipova i sprega specifikacije. Sprega logike je najnepoželjnija, sprega tipova je ponekad prihvatljiva, a sprega specifikacije je prihvatljiva u većini slučajeva.

Sprega logike

Sprega logike se odlikuje deljenjem informacija ili pretpostavki između modula. Jedan modul „zna“ neke stvari o implementaciji drugog modula i to znanje je iskorišćeno pri implementaciji njegovih elemenata.

Deljenje informacija se odnosi na slučaj kada jedan modul neposredno pristupa informacijama koje se održavaju od strane drugog modula. Deljenje pretpostavki se odnosi na slučaj kada pri implementaciji jednog modula moramo da vodimo računa o pretpostavkama o načinu implementacije delova drugog modula. U oba slučaja se radi o grubom narušavanju enkapsulacije.

Na primer, ako dve komponente A i B obavljaju međusobno komplementarne poslove (recimo da A obavlja kodiranje a B dekodiranje po istom algoritmu, ili da A

obavlja zapisivanje a B čitanje podataka u istom formatu), onda je implementacija delova modula A čvrsto povezana sa implementacijom delova modula B. Ova dva modula *dele pretpostavke* o tome kako se neki posao obavlja u drugom modulu.

Sprega logike je veoma problematična i nepoželjna. Osnovni razlog za to je u postojanju jakih osa promenljivosti između komponenti. Promene u implementaciji jedne od njih često imaju za posledicu da mora da se menja i druga komponenta. Kada imamo spregu logike između nekih komponenti, onda moramo da se zapitamo da li bi te komponente trebalo da ostanu razdvojene, ili je možda bolje da ih spojimo u jednu?

Sprega tipova

Sprega tipova nastupa kada jedna komponenta koristi neki tip koji je definisan u okviru druge komponente. Jedan modul definiše i implementira neke tipove, a drugi ih koristi uz puno poznavanje njihovih specifičnosti, pa možda čak i nekih elemenata implementacije. Važna karakteristika ove vrste spregnutosti je već istaknuto *puno poznavanje* specifičnosti upotrebljenih konkretnih tipova podataka.

Sprega tipova može da bude *određena* i *neodređena*. Određena je kada komponenta koja koristi posmatrani tip čak i pravi objekte tog tipa, a neodređena je ako se objekti samo koriste, a pravljenje se prepušta matičnoj komponenti.

Ako je konkretan tip dobro dizajniran, tj. ako je njegov sadržaj dobro enkapsuliran i ako je obezbeđen dobar interfejs, onda možemo da kažemo da je sprega tipova dobro implementirana i takva sprega tipova nam obično ne predstavlja problem. Neodređena sprega tipova je poželjnija od određene.

Sprega specifikacije

Sprega specifikacije je još apstraktnija nego sprega tipova – pretpostavlja se da nije poznat konkretan tip koji se koristi već samo neke pretpostavke o njegovom interfejsu. Sprega specifikacija nastupa kada modul koji koristi neki tip zapravo ne koristi konkretne tipove nego apstraktne specifikacije tipova i u stanju je da ispravno radi sa bilo kojim konkretnim tipovima koji odgovaraju datim specifikacijama.

Sprega specifikacija se u praksi ostvaruje primenom nekog od vidova polimorfizma. U slučaju hijerarhijskog polimorfizma se modul koji koristi druge tipove implementira tako da koristi apstraktnu klasu[21]. Vid sprege specifikacija je i kada jedan modul definiše implementaciju interfejsa koji je definisan u drugom modulu[22].

---

[21] Ako je u pitanju C++, onda imamo i dodatni zahtev da mora da koristi objekte isključivo putem referenci ili pokazivača.

[22] Na primer, klasa `Drajver` definiše interfejs drajvera za bazu podataka, a svaki konkretan modul drajvera (obično u vidu dinamičke biblioteke) implementira jedan konkretan drajver za neku konkretnu bazu podataka.

U slučaju drugih vidova polimorfizma se modul-korisnik implementira nad apstraktnim parametrizovanim tipovima ili čak bez navođenja tipova. U slučaju programskog jezika C++ to se ostvaruje implementiranjem šablona klasa ili šablona funkcija. Upotreba šablona u C++-u je dugo podrazumevala samo implicitnu specifikaciju interfejsa, zato što osnovna sintaksa šablona ne obuhvata njegovu formalnu eksplicitnu specifikaciju. Od verzije C++20 programerima su na raspolaganju *koncepti* kao sredstvo za bliže opisivanje dopuštenih parametarskih tipova. Koncepti su uveli sintaksu za formalno specificiranje interfejsa šablona, tako što su pružili mogućnost da se precizno navedu uslovi koje parametri šablona moraju da zadovoljavaju. Kao vid eksplicitne specifikacije, koncepti su omogućili i jednostavnije i efikasnije prevođenje šablona, ali i preciznije izveštavanje o greškama usled njihove neispravne upotrebe.

Dobro implementirana sprega specifikacije ne predstavlja problem. Štaviše, teži se da spregnutost uvek bude ostvarena kao sprega specifikacije.

Nivoi spregnutosti

Nivo spregnutosti se odnosi na složenost sadržaja koje različiti moduli (ili različite funkcije ili metodi) razmenjuju pri svom radu. Nivoi spregnutosti, od najjačeg prema najslabijem, su:

- spregnutost po sadržaju;
- spregnutost preko zajedničkih delova;
- spoljašnja spregnutost;
- spregnutost preko kontrole;
- spregnutost preko markera;
- spregnutost preko podataka i
- spregnutost preko poruka.

Različite nivoe spregnutosti ćemo ilustrovati primerima u kojima ulogu komponenti (modula) imaju različite klase.

Spregnutost po sadržaju

Spregnutost po sadržaju predstavlja otvoreno i neposredno pristupanje sadržaju jedne komponente od strane druge komponente, bilo da se radi o čitanju ili menjanju.

Pojednostavljen primer:

```
... A::metod(...)
{
    ... objB->podatak ...
}
```

U slučaju spregnutosti po sadržaju je ozbiljno narušena enkapsulacija. Održavanje komponenti koje su spregnute po sadržaju je često vrlo nezahvalan posao, zato što komponente zavise od internih elemenata drugih komponenti, pa svaka izmena implementacije tih drugih komponenti preti da zahteva izmene i u komponentama koje od njih zavise. Dovode se u pitanje odgovornosti komponenti i uspostavljene granice među komponentama.

Spregnutost po sadržaju je najsnažnija i najnepoželjnija vrsta spregnutosti.

Spregnutost preko zajedničkih delova

Spregnutost *preko zajedničkih delova* imamo kada dve ili više komponenti neposredno pristupaju nekim podacima koji nisu njihov sastavni deo. Obično su to podaci sadržani u nekoj odvojenoj strukturi.

Pojednostavljen primer:

```
struct ZajednickiDelovi {...};
... A::metod1(...){
    ... zajednickiDelovi->podatak ...
}
... B::metod2(...){
    ... zajednickiDelovi->podatak ...
}
```

U slučaju spregnutosti preko zajedničkih delova imamo razdvajanje podataka (zajednički delovi) od ponašanja (komponente A i B). Iako naizgled nije narušena enkapsulacija, ona zapravo nije ni uspostavljena. Iako komponente A i B formalno ne pristupaju jedna drugoj po sadržaju, one u suštini to ipak čine i to obostrano, zato što i jedna i druga tretiraju zajedničke delove kao svoje podatke (tj. kao deo sopstvene strukture), pa možemo da kažemo da ni u ovom slučaju nisu uspostavljene granice odgovornosti između komponenti.

Ovo je veoma visok nivo spregnutosti koji može da značajno otežava održavanje softvera. Zato je jednako nepoželjan kao i spregnutost po sadržaju.

Spoljašnja spregnutost

Spoljašnja spregnutost predstavlja upotrebu deljenog i sa strane nametnutog koncepta (interfejs, format podataka, komunikacioni protokol,...) od strane više komponenti.

Pojednostavljen primer: komponente A, B i C dele koncept definisan u komponenti `external`, koja nije deo nijedne od njih:

```
... A::metod1 {
    ...external::oper1(...);
    ...external::oper2(...);
};
```

```
... B::metod2(...){
    ...external::oper1(...);
    ...external::oper2(...);
}
... C::metod3(...){
    ...external::oper3(...);
    ...external::oper4(...);
}
```

Problem sa ovim nivoom spregnutosti je u tome što često dolazi do ponavljanja istog ili veoma sličnog programskog koda, pa ako dođe do nekih promena u upotrebljenom konceptu, onda te promene indukuju nove izmene u svim komponentama koje upotrebljavaju taj koncept. Takođe, ako interfejs nije sasvim uzak, onda ovde imamo i problem podeljene odgovornosti – ko je odgovoran da od delova koncepta napravi složeniju operaciju?

Primetimo da spoljašnja spregnutost obično počiva na korišćenju interfejsa komponente u kojoj je ostvarena proceduralna kohezija. Ni proceduralna kohezija ni spoljašnja spregnutost ne spadaju u poželjne karakteristike dizajna programa, ali ako je spoljašnji koncept relativno stabilan i pouzdan i ima relativno uzak interfejs (na primer, neka dobro oblikovana spoljašnja biblioteka), onda spoljašnja spregnutost može da bude sasvim prihvatljiva. U suprotnom bi trebalo da se pokuša sa umotavanjem spoljašnjeg koncepta u novu komponentu, koja bi sakrila veći deo složenosti upotrebe i stavila na raspolaganje samo vrlo uzak interfejs, čime se praktično dobija spregnutost preko kontrole.

Spregnutost preko kontrole

Spregnutost preko kontrole je na delu kada jedna komponenta (tzv. kontroler) ultimativno upravlja radom druge komponente, pri čemu upravljana komponenta nije u stanju da funkcioniše bez spoljašnje kontrole (tj. bez kontrolera) ili od kontrolera očekuje sva ili bar najvažnija uputstva za rad po kojima zatim može da samostalno postupa.

Spregnutost preko kontrole je slična spoljašnjoj spregnutosti. Razlika je u nivou apstraktnosti i složenosti pojedinačnih elemenata spoljašnjeg koncepta. Kod spoljašnje spregnutosti se obično radi o većem broju vrlo jednostavnih elemenata, dok se kod spregnutosti preko kontrole obično radi o jasno definisanim atomičnim postupcima koji imaju veći semantički značaj.

Pojednostavljen primer:

```
class Kontrolisana {
    ...oper1(...);
    ...oper2(...);
};
```

```
... Kontroler::metod(...){
    ...kontrolisana->oper1(...)...
    ...kontrolisana->oper2(...)...
}
```

Spregnutost preko kontrole ima i dobre i loše strane. Ona je relativno jaka, ali to često ne predstavlja smetnju. Primer dobre primene ovakve spregnutosti je kada kontrolisana komponenta implementira niži nivo neke klase operacija, a viši nivo upravljanja prepušta svojim kontrolerima. Na primer, kontrolisana komponenta bi mogla da implementira niži nivo operacija sa slikama, a različiti kontroleri bi mogli da implementiraju složenije operacije sa slikama primenjujući operacije kontrolisane komponente.

U okviru kontrolisane komponente postoji proceduralna kohezija, koja nije među najpoželjnijim kohezijama. Problem može da predstavlja potencijalno nejasna podela odgovornosti, do koje dolazi ako komponenta kontroler ima i druge odgovornosti osim implementiranja konkretne složene operacije, koju ostvaruje pomoću kontrolisane komponente. Ako neka komponenta ima ulogu kontrolera, tj. ako je odgovorna za viši nivo operacija, onda bi to trebalo da bude i njena jedina odgovornost.

Drugi potencijalan problem je sličan kao i u slučaju spoljašnje spregnutosti – ako jednu kontrolisanu komponentu kontroliše veći broj kontrolera, onda u slučaju promene interfejsa kontrolisane komponente može da se zahteva i veći broj izmena u kontrolerima. Zbog toga ovaj nivo spregnutosti može da predstavlja rizik ako postoji veći broj kontrolera.

Spregnutost preko markera

Spregnutost je *preko markera* ako više komponenti međusobno razmenjuje neku složenu strukturu podataka (marker) koju upotrebljavaju na različite načine.

Primetimo da, ako marker nema definisano ponašanje niti je njegov sadržaj enkapsuliran, već se samo neposredno koriste podaci markera, onda je zapravo u pitanju spregnutost preko zajedničkih delova, a ne preko markera. Slično tome, ako marker ima relativno složeno ponašanje i može da se koristi na različite načine, onda to može da bude spregnutost preko kontrole.

Kod spregnutosti preko markera podrazumevamo da marker ima neko definisano ponašanje, kao i neki nivo enkapsulacije sadržaja, te da se koristi kao predmet na kome se neki posao obavlja ili kao nosilac informacija između delova posla.

Pojednostavljen primer:

```
...
parser->fillRequest( request );
environment->prepareRequest( request );
processor->performTransaction( request );
response = reporter->prepareReport( request );
...
```

Problem sa ovim nivoom spregnutosti je u potencijalno nedefinisanoj odgovornosti markera – zašto su neki aspekti ponašanja prepušteni drugim komponentama i to obično većem broju komponenti? Marker obično nije do kraja strogo enkapsuliran i deo odgovornosti se deli među njegovim korisnicima, pa postoji i opasnost od sukoba nadležnosti nad nekim podacima ili delovima posla.

Spregnutost preko markera može da bude sasvim prihvatljivo rešenje u nekim slučajevima. Na primer, ako marker nema složeno ponašanje i nije spregnut na drugi način osim ovim putem, a pri svemu tome postupak obrade u kome marker učestvuje ili nije stabilan ili možda realno zavisi od više različitih komponenti, onda ovakvo rešenje može da bude sasvim prihvatljivo.

Spregnutost preko podataka

Spregnutost preko podataka je kada jedna komponenta koristi interfejs druge komponente, putem koga mu prenosi pojedinačne podatke. Ako su podaci koji se prenose relativno složeni, onda može da bude reč o spregnutosti preko zajedničkih delova.

Pojednostavljen primer:

```
... simulacija::promenaSmera(...)
{
    ...
    automobil->skreniLevo( 5 );
    ...
}
```

Spregnutost preko podataka počiva na međusobnoj zavisnosti na nivou interfejsa, pa je ovaj nivo spregnutosti među najnižim i obično je prihvatljiv, pa čak i poželjan. Eventualni problemi mogu da budu posledica loše definisanih interfejsa ili različitog tumačenja podataka koji se razmenjuju, ali ne i samog nivoa spregnutosti.

Spregnutost preko poruka

Spregnutost preko poruka je kada jedna komponenta koristi interfejs druge komponente, putem koga mu prenosi samo poruke, bez ikakvih podataka.

Pojednostavljen primer:

```
... simulacija::promenaSmera(...)
{
    ...
    automobil->skreniLevo();
    ...
}
```

Spregnutost preko poruka predstavlja naniži nivo spregnutosti, pa je takva spregnutost najpoželjnija. Radi se o zavisnosti na nivou interfejsa i eventualni problemi mogu da nastanu samo kao posledica loših interfejsa.

Smer spregnutosti

Spregnutost između dveju komponenti može biti jednosmerna i dvosmerna. U slučaju jednosmerne spregnutosti jedna od komponenti upotrebljava elemente druge komponente, ali ne i obratno. U slučaju dvosmerne spregnutosti svaka od dveju komponenti upotrebljava elemente one druge komponente.

U slučaju međusobne spregnutosti više komponenti može biti reči i o cirkularnoj spregnutosti, kada ne postoji neposredno dvosmerna sprega dveju komponenti, ali postoji tranzitivna cirkularna spregnutost više komponenti.

Poželjno je da spregnutost bude jednosmerna, zato što dvosmerna i cirkularna spregnutost uvećavaju složenost sistema. U takvim slučajevima često dolazi do značajno složenije podele odgovornosti među komponentama. Takođe, postavlja se i pitanje granica među komponentama i kvaliteta izvedene enkapsulacije.

Širina sprege

Širina sprege između dveju komponenti odgovara broju elemenata jedne komponente (objekata, podataka, metoda, poruka,...) koje upotrebljava druga komponenta. Ako je spregnutost dvosmerna, onda širina sprege zavisi i od smera koji se posmatra, tj. širina sprege je retko ista u oba smera.

Naravno, poželjno je da sprega bude što uža. Što je manje elemenata neke komponente A koji se koriste u implementaciji neke druge komponente B, to će biti manja i verovatnoća da zbog izmena implementacije ili interfejsa komponente A mora da se menja i komponenta B.

Široka sprega može da bude znak da podela odgovornosti između dve komponente nije izvedena dovoljno jasno ili da su odgovornosti neke od komponenti preširoke. U prvom slučaju sprega možda može da se suzi implementiranjem nekih složenijih celina u okviru prve komponente, tako da se omogući sužavanje njenog interfejsa. U drugom slučaju je možda potrebno da se komponenta podeli na više manjih.

Način ostvarivanja sprege

Spregnutost između različitih modula može da se ostvaruje statički i dinamički

Statička spregnutost je eksplicitno izražena u programskom kodu, tj. pre samog izvršavanja programa. Neki od primera statičke spregnutosti mogu biti:

- klasa `B` nasleđuje klasu `A`;
- podatak `b` klase `A` je objekat klase `B`;

- argument metoda klase `A` je objekat klase `B`;
- u metodima klase `A` se pravi ili upotrebljava objekat klase `B`.

Dinamička spregnutost se uspostavlja u toku izvršavanja programa. Ona nije eksplicitno iskazana u programskom kodu koji opisuje module A i B, već može da zavisi od konfiguracije, ulaznih parametara i drugih faktora.

Statička sprega je intenzivnija i nepoželjnija od dinamičke. Zbog toga što je uspostavljena relativno čvrsto u programskom kodu, ona obično ima odlike sprege tipova ili čak sprege logike. Sa druge strane, dinamička spregnutost obično ima odlike sprege specifikacije.

U savremenom razvoju softvera se teži arhitekturama koje omogućavaju dinamičku spregnutost, da bi se dobio stabilniji i istovremeno fleksibilniji softver. Među najpoznatije primere arhitektura sa dinamičkom spregnutošću spada razvoj vođen događajima, gde je osnovna ideja da se pokretač neke operacije potpuno odvoji od implementacije sprovođenja te operacije. Primer implementacije te arhitekture je koncept signala i slotova koji se koristi u radnom okviru *Qt*. Klasama se dodaju dve vrste metoda – *signali* koji se ne implementiraju nego služe za *iniciranje događaja*, i *slotovi* koji služe za obradu događaja. Pri tome se povezivanje signala i slotova koji će reagovati na njih odvija dinamički [*Qt*].

Intenzitet spregnutosti

Intenzitet spregnutosti je složena karakteristika koja uzima u obzir sve prethodno navedene karakteristike spregnutosti. Pri tome uzimamo u obzir da:

- logička sprega ima veći intenzitet od sprege tipova i specifikacija;
- viši nivo spregnutosti ima veći intenzitet;
- šira sprega ima veći intenzitet;
- cirkularna sprega ima veći intenzitet od dvosmerne, a ona od jednosmerne i
- statička sprega ima veći intenzitet od dinamičke.

Intenzitet spregnutosti može da se predstavi i numerički, tj. da se *meri* na različite načine. Na primer, možemo da brojimo:

- koliko drugih elemenata se koristi iz posmatranog;
- koliko drugih elemenata koristi posmatrani element;
- na koliko mesta u posmatranom elementu de koriste drugi elementi;
- na koliko mesta u drugim elementima se koristi posmatrani element i
- razne druge vidove zavisnosti.

Pri tome pod elementima možemo da posmatramo komponente, klase, pakete, metode i druge strukturne elemente programa. Pored brojanja možemo da uvedemo i neke težinske faktore, kojima bismo predstavili karakteristike posmatranih zavisnosti. U zavisnosti od toga da li posmatramo različite celine ili samo delove jedne celine, merenje bi se odnosilo na spregnutost ili na koheziju.

Na primer, spregnutost dveju komponenti bismo mogli da računamo kao:

```
coefSprege(A,B) = coefVrste(A,B) + coefNivoa(A,B) + ...
```

Kao nadgradnju toga, spregnutost jedne komponente sa ostatkom programa ili nekim izabranim podsistemom bismo mogli da računamo kao:

$$coefSprege(A) = sum_B(coefSprege(A,B))$$

U istom duhu, ukupnu međusobnu spregnutost delova nekog sistema ili podsistema, možemo da izračunamo kao:

$$coefSpregeSistema = sum_A(coefSprege(A))$$

Svođenje spregnutosti, kao kvalitativnog svojstva strukture softvera, na numeričke vrednosti, predstavlja primer uvođenja mera i metrika u prostor dizajna softvera. Metrike dizajna softvera su jedna od značajnih, ali i zahtevnih tema, za koju se nije našlo dovoljno prostora u ovoj knjizi. Intenzitet spregnutosti smo iskoristili kao primer koji bi mogao da čitaoce zainteresuje za dalje izučavanje metrika. Više informacija o metrikama i merenju softvera može da se pročita, na primer, u [*Fenton 2014*].

4.8 Umesto zaključka

Pristup projektovanju softvera se vremenom menjao od planiranja i oblikovanja algoritama, u ranim fazama razvoja računarstva, preko klasičnog stava o neophodnosti detaljnog projekta pre započinjanja kodiranja, pa sve do ekstremnog stava da je potpuno nepotrebno da se pravi projekat pre pisanja programa. Takve promene su se odvijale paralelno sa promenama u domenu primene računarstva, kao i sa promenama u načinu i nivou obrazovanja razvijalaca softvera.

Savremena praksa pokušava da pronađe ravnotežu između različitih pristupa tako što teži da inicijalno projektovanje uglavnom svede na oblikovanje arhitekture, dok se detaljno projektovanje praktično integriše sa implementacijom, tj. sa programiranjem. Ovim pitanjem ćemo se baviti i na kraju poglavlja *8 - Agilni razvoj softvera*. Nakon što upoznamo osnovne koncepte aglinog razvoja, razmotrićemo i odnos agilnog razvoja i projektovanja softvera.

Među prvima koji su javno zastupali stav da programiranje predstavlja čin projektovanja softvera je bio Džek Rivs, koji je objavio tekst o tome u časopisu *C++*

Journal [*Reeves 1992*]. Taj tekst je izazvao burne reakcije, ali je ostvario veliki uticaj na nadolazeći talas razvoja agilnih metodologija. Sa druge strane, stav o važnosti stabilne i robusne arhitekture možda najjače promoviše metodologija *RUP* [*Kruchten 2003*]. Odnos arhitekture i dizajna je do danas ostao relativno neprecizan, o čemu na veoma zanimljiv način piše Martin Fauler [*Fowler 2003b*].

5 - UML

„Ti, koji me čitaš,
jesi li siguran da razumeš moj jezik?“
Horhe Luis Borhes, Vavilonska biblioteka

5.1 Objedinjeni jezik za modeliranje

Objedinjeni jezik za modeliranje (engl. *Unified Modeling Language – UML*) predstavlja jednu od najvažnijih savremenih razvojnih tehnologija. Kao što je već naglašeno u poglavlju *3.3 – Objektno-orijentisane metodologije*, *UML* je nastao uglavnom u periodu od 1995. do 2000. godine, nakon čega se neprekidno dalje unapređuje. Uticaj koji je *UML* ostvario na industriju razvoja softvera može da se meri sa uticajem strukturnog programiranja, objektno-orijentisanog programiranja ili relacionih baza podataka – sve ove tehnologije su danas praktično svuda prisutne i bez njih je nezamisliv razvoj softvera.

Održavanjem standarda *UML*-a se bavi *OMG*. Na njihovoj veb lokaciji može da se pronađe zvanična dokumentacija o standardu, kao i mnogo drugih izvora informacija o *UML*-u [*OMG*] [*UML*]. Zanimljivo je da se kao jezik za predstavljanje apstraktne sintakse i strukture *UML*-a koristi upravo *UML*.

Iscrpno predstavljanje *UML*-a bi zahtevalo mnogo više prostora nego što nam je ovde na raspolaganju. Zbog toga ćemo da napravimo osvrt na osnovne elemente jezika, dok ćemo se temeljnije posvetiti samo nekim vrstama dijagrama. Na srpski jezik je prevedeno nekoliko knjiga o *UML*-u. Iako one često nisu najsvežije (uglavnom su prevođene knjige sa početka 2000-ih godina, koje se bave prvim

verzijama jezika), sasvim su dobre za početno upoznavanje *UML*-a i toplo ih preporučujemo [*Booch 1999*][23] [*Fowler 2003a*].

UML je prvenstveno grafički jezik. Najveći deo elemenata jezika se odnosi na različite dijagramske tehnike, dok se tekstom samo detaljnije opisuju ili komentarišu određeni elementi. Samo pojedini delovi jezika se više oslanjaju na tekstualne nego na grafičke opise elemenata modela.

Svi dijagrami *UML*-a mogu da se podele na:

- strukturne dijagrame i
- dijagrame ponašanja.

U nastavku ovog poglavlja ćemo da ukratko predstavimo vrste dijagrama i da malo temeljnije opišemo one koje se koriste u ovoj knjizi.

Strukturni dijagrami

Strukturni dijagrami opisuju strukturu softverskog sistema i njegovih delova. Različiti dijagrami mogu da imaju različit nivo apstraktnosti i mogu da odgovaraju različitim fazama modeliranja ili implementacije softvera. Elementi strukturnih dijagrama nemaju dodira sa konceptom protoka vremena ili povezanim konceptima, kao što su tok komunikacije među objektima ili promene stanja objekata.

U strukturne dijagrame spadaju:

- dijagram klasa (engl. *class diagram*);
- dijagram objekata (engl. *object diagram*);
- dijagram paketa (engl. *package diagram*);
- dijagram složene strukture (engl. *composite structure diagram*);
- dijagram komponenti (engl. *component diagram*);
- dijagram raspoređivanja (engl. *deployment diagram*, naziva se i *dijagram isporučivanja*) i
- dijagram profila (engl. *profile diagram*).

Dijagram klasa je jedan od najvažnijih strukturnih dijagrama. On opisuje elemente statičkog modela softvera, a pre svega klase, njihovu strukturu i međusobne odnose. Dijagram klasa predstavlja klase kao osnovne statičke elemente sistema.

[23] Postoji novije izdanje [*Booch 2005*] koje pokriva *UML* 2.0, ali ono nije prevedeno na srpski jezik do trenutka pisanja ove knjige.

Dijagram objekata nam pomaže da bolje razumemo dinamičku prirodu odnosa među objektima. Koristi se kao dopuna dijagrama klasa i komunikacije, za opisivanje dinamičkih sistema. Dijagram objekata predstavlja izabrani skup objekata, koji svojim sadržajem i odnosima ilustruje primer jedne konkretne slike stanja sistema u nekom izabranom trenutku. Sadrži nazive objekata i njihovih klasa, imena i vrednosti atributa i odnose među objektima.

Dijagram paketa opisuje kako su elementi logičkog modela organizovani u pakete, kao i međuzavisnosti paketa. U savremenim OO programskim jezicima, *paket* je često sinonim za *prostor imena*. Paket okuplja elemente (klase, komponente, slučajeve upotrebe i sl.) koji su semantički povezani i očekuje se da se zajedno menjaju. Dijagram sadrži nazive i granice paketa i međusobne zavisnosti paketa. Može da sadrži i klase u paketima, kao i njihove međusobne odnose, predstavljene na isti način kao što se predstavljaju u dijagramima klasa.

Dijagram složene strukture ima za cilj da bliže objasni unutrašnju strukturu elemenata neke strukture (komponente ili glavne klase), predstavljanjem njenih sastavnih delova i njihovih međusobnih odnosa.

Dijagram komponenti opisuje komponente koje čine aplikaciju, podsistem ili veću komponentu. Za razliku od dijagrama paketa, koji predstavlja logičku dekompoziciju softverskog sistema, dijagram komponenti predstavlja funkcionalnu dekompoziciju sistema. Sadrži nazive komponenti, njihove javne interfejse i međusobne odnose. Svaka komponenta bi trebalo da ima jasno prepoznatu i oblikovanu funkciju, koju druge komponente koriste putem njenog interfejsa.

Dijagram raspoređivanja predstavlja elemente fizičke arhitekture softverskog sistema i način raspoređivanja softverskih komponenti na te fizičke elemente. Sadrži čvorove (servere, klijente i druge uređaje), softverske i hardverske podsisteme ili komponente, linije komunikacije među uređajima i međusobne veze podsistema. Na svakom od čvorova se predstavljaju softverske komponente koje će njima raditi, kao i osnovne linije komunikacije (odnosi) među njima.

Dijagram profila predstavlja osnovu za prilagođavanje *UML* dijagrama specifičnim domenima problema, dodavanjem novih vrsta elemenata, pravljenjem novih svojstava i određivanjem nove specifične semantike. Počiva na primeni tri mehanizma za proširivanje: stereotipovima, označenim vrednostima i uslovnim ograničenjima. Dijagramima profila se definišu nove vrste stereotipova, metaklasa i slično.

Dijagrami ponašanja

Dijagrami ponašanja se bave dinamičkom prirodom softverskog sistema. Oni opisuju ponašanje pojedinačnih objekata ili podsistema. Ponašanje se predstavlja u kontekstu toka vremena, u obliku razmene poruka, toka komunikacije ili toka promene stanja.

Dijagrami interakcije su posebna podgrupa dijagrama ponašanja, ali se ponekad razmatraju posebno. Razlikuju se od ostalih dijagrama ponašanja po tome što je na njima vremenski tok opisan eksplicitnije nego na ostalim dijagramima.

U dijagrame ponašanja spadaju:

- dijagram slučajeva upotrebe (engl. *use case diagram*);
- dijagram aktivnosti (engl. *activity diagram*);
- dijagram stanja (engl. *state machine diagram*) i
- dijagrami interakcije:
  - dijagram sekvence (engl. *sequence diagram*);
  - dijagram komunikacije (engl. *communication diagram*, do verzije 2 je bio u upotrebi naziv *dijagram saradnje,* engl. *collaboration diagram*);
  - dijagram vremena (engl. *timing diagram*) i
  - pregledni dijagram interakcija (engl. *interaction overview diagram*).

Dijagram slučajeva upotrebe predstavlja slučajeve upotrebe softverskog sistema, aktere (učesnike) koji u njima učestvuju i njihove međusobne odnose. Slučajevi upotrebe mogu da budu organizovani u pakete ili podsisteme, pa ovi dijagrami mogu da sadrže i njihove međusobne odnose. Za razliku od ostalih dijagrama, svaki slučaj upotrebe mora da bude propraćen opsežnom tekstualnom dokumentacijom. Dijagram je samo okvirna ilustracija čiji cilj je da predstavi ključne elemente i odnose među njima, dok se sve važne informacije o pojedinačnim slučajevima upotrebe opisuju tekstualno.

Dijagram aktivnosti opisuje poslovne procese višeg nivoa i tokove podataka a može da predstavlja i složene logičke elemente sistema. Sadrži procese, tokove podataka između procesa, grananja i čvorišta, uslovne tačke i početne i završne tačke. Može da bude grupisan po trakama aktera, tako da se vidi ko je od subjekata zadužen za koju aktivnost. Ima sličnosti sa klasičnim dijagramskim tehnikama za opisivanje algoritama.

Dijagram stanja opisuje kako se stanje jednog objekta menja u zavisnosti od interakcija u koje objekat ulazi tokom svog života. Dijagram stanja sadrži sva stanja u kojima posmatrani objekat može da bude (pri čemu su posebno označena početno i završno stanje), zatim sve moguće prelaske iz jednog u drugo stanje, kao i nazive događaja koji menjaju stanje objekata.

Dijagram sekvence služi za preciznije opisivanje toka odvijanja slučajeva upotrebe, kao i za jasno prepoznavanje odgovornosti subjekata i objekata za pojedinačne korake. Sadrži slučajeve upotrebe, aktere, pakete, podsisteme, poruke i međusobne odnose. Dijagram sekvence često može da se neposredno prevede u implementaciju metoda ili skupa metoda.

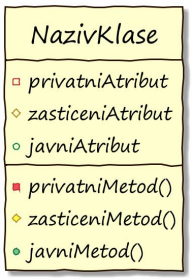
Dijagram komunikacije ilustruje objekte, njihove međusobne odnose i poruke koje razmenjuju. Posvećuje se posebna pažnja strukturnoj organizaciji objekata koji učestvuju u razmeni poruka. Sadrži objekte (među kojima mogu da se nađu i subjekti), poruke koje oni razmenjuju, komentare i napomene.

Dijagram vremena predstavlja promene stanja objekata tokom vremena. koristi se za opisivanje zavisnosti promena stanja od spoljašnjih događaja. Sadrži protok vremena, spoljašnje događaje i promene stanja objekata.

Pregledni dijagram interakcija je varijanta dijagrama aktivnosti u kojoj je akcenat na upravljanju procesima ili sistemom. Svaki čvor/aktivnost u dijagramu može da predstavlja neki drugi dijagram interakcija ili aktivnosti. Sadrži objekte, manje dijagrame aktivnosti ili interakcija, slučajeve upotrebe, tok odvijanja procesa (protoka podataka), grananja i spajanja, početak i kraj.

5.2 Dijagram klasa

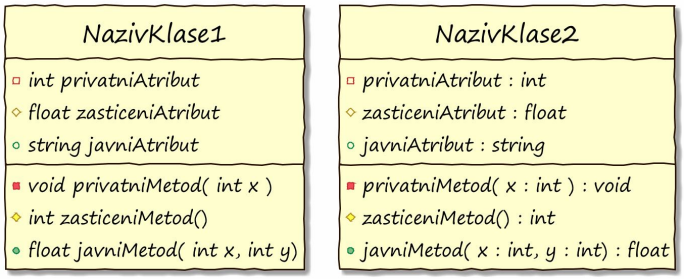
Dijagram klasa je jedan od najčešće korišćenih dijagrama *UML*-a. Njegova osnovna namena je da opiše strukturu statičkog modela softvera – strukturu klasa i njihove međusobne odnose. Dijagram klasa je postao osnovna tehnika kojom se dokumentuju poslovi koje treba implementirati ali i ostvareni rezultati rada.



Slika 2 – Primer predstavljanja klase na dijagramu klasa

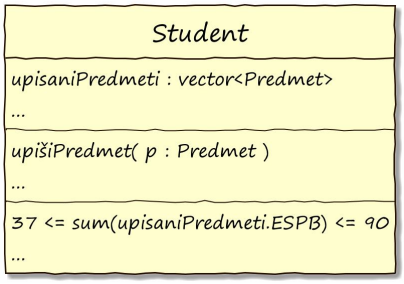
Osnovni elementi dijagrama su *klase*, koje se predstavljaju pravougaonicima. Klasa može da bude podeljena horizontalnim linijama na odeljke – prvi odeljak sadrži naziv klase, drugi sadrži spisak atributa, a treći sadrži spisak metoda klase. Vidljivost atributa i metoda se može predstavljati na različite načine, ali je uobičajeno da se privatni članovi označavaju znakom „–“, javni znakom „+“, a zaštićeni znakom „#“. Dopušteno je da umesto toga postoji neka jasna grafička notacija. U primeru (*Slika 2*) je vidljivost opisana bojama i oblicima – crveni kvadrat označava privatne, zeleni krug javne, a žuti romb zaštićene elemente. Pritom su atributi označeni praznim a metodi popunjenim oznakama.

Tipovi atributa i metoda mogu, ali ne moraju da se navode. Ako se navode, onda se to obično radi u skladu sa sintaksom izabranog programskog jezika ili tako da se tipovi navode iza imena, odvojeni simbolom „:“ (*Slika 3*).



Slika 3 – Dva načina predstavljanja tipova atributa i metoda na dijagramu klasa

U zavisnosti od vrste modela koji se predstavlja dijagramom klasa, neki elementi strukture klasa mogu da se izostavljaju. Tako se, na primer, u dokumentaciji koja služi korisnicima nekog dela softvera, u klasama navode samo elementi interfejsa, tj. javni metodi, dok se sve ostalo sakriva da ne bi skretalo pažnju korisnika. Ako je potrebno da stanje klase ili način upotrebe klase poštuju neka posebna pravila ili ograničenja (engl. *constraints*) onda može da se doda i četvrti deo opisa klase, u kome se navode ta ograničenja (*Slika 4*). Ograničenja i dodatni uslovi mogu da se zapisuju rečima ili formulama.

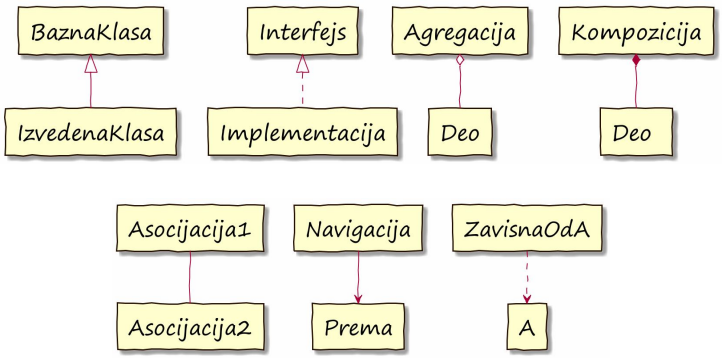


Slika 4 – Primer predstavljanja ograničenja na dijagramu klasa

Navođenjem tipova atributa i metoda se dijagram klasa dodatno opterećuje informacijama koje mogu da se vide i u tekstualnom opisu, a mogu da otežavaju razumevanje dijagrama. Zbog toga se, u slučaju kada na dijagramima imamo mnogo klasa i želimo da pre svega istaknemo njihove međusobne odnose, često odlučujemo da ne predstavljamo tipove, a ponekad čak ni interfejse, već da opise klasa svedemo samo na nazive (*Slika 5*).

Odnosi među klasama se predstavljaju linijama, koje na krajevima mogu da imaju određene simbole:

- nasleđivanje (engl. *inheritance*) se predstavlja linijom koja se na strani bazne klase (tj. na strani *generalizacije*) završava strelicom u obliku praznog trougla;
- implementacija interfejsa se predstavlja slično nasleđivanju, ali sa isprekidanom linijom[24];
- agregacija se predstavlja linijom koja se završava praznim rombom na strani klase koja sadrži delove;
- kompozicija se predstavlja linijom koja se završava popunjenim rombom na strani klase koja sadrži delove;
- asocijacija nema dodatne simbole na krajevima, osim eventualno strelice koja označava smer navigacije, i
- zavisnost (engl. *dependency*) se predstavlja isprekidanom linijom.



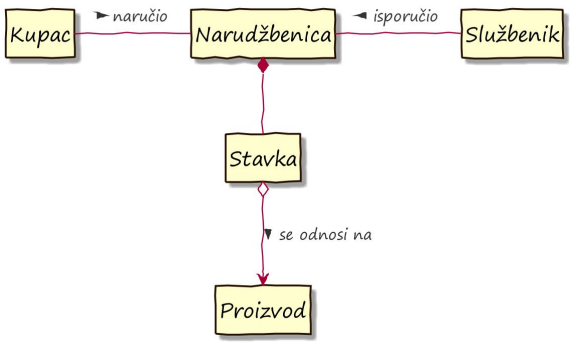
Slika 5 – Predstavljanje odnosa među klasama

Agregacija predstavlja odnos dveju klasa u kome objekat jedne klase (tzv. *agregacija* ili *složeni objekat*) sadrži referencu na jedan ili više objekata druge klase (tzv. *deo*), ali je odnos takve prirode da delovi mogu da postoje i nezavisno od objekta koji ih okuplja – ako se uništi složeni objekat to ne znači da će biti uništeni i njegovi sastavni delovi. Zato se kaže da agregacija predstavlja *slabu integraciju* delova u celinu. Uobičajen primer agregacije je povezivanje automobila i točkova – točkovi jesu delovi automobila, ali mogu da postoje i bez njega. Primer agregacije je i odnos

[24] U opštem slučaju interfejs nije klasa, pa se u dijagramima predstavlja uz navođenje stereotipa „<<interface>>“ iznad imena.

stavke narudžbenice i proizvoda na koji se stavka odnosi (*Slika 6*) – jedna stavka se uvek odnosi na jedan konkretan proizvod, ali taj proizvod postoji i bez te stavke (pa i narudžbenice), a može i da predstavlja sadržaj više različitih stavki.

Nasuprot tome, u slučaju kompozicije, podrazumeva se da se radi o tzv. *jakoj integraciji* – ako se uništi složeni objekat, automatski će se uništiti i njegovi delovi. Štaviše, objekti koji predstavljaju delove kompozicije ne mogu ni da se naprave a da već tom prilikom ne bude jasno određeno kom složenom objektu pripadaju. Primer kompozicije je odnos narudžbenice i stavki narudžbenice (*Slika 6*) – stavke narudžbenice ne mogu da postoje bez narudžbenice kojoj pripadaju.



Slika 6 – Dijagram klasa za rad sa narudžbenicama

Najopštiji odnosi, kod kojih postoji neposredna veza između objekata, nazivaju se asocijacijama. Asocijacije znače da su neki konkretni objekti međusobno povezani, ali da ta veza može biti ostvarena na više načina. Ako jedan od objekata sadrži referencu na drugi objekat i može da se praćenjem te reference dođe do drugog objekta, ali obrnuto ne važi, onda je u pitanju jednosmerna asocijacija, koja se označava strelicom u smeru u kome se može ostvariti pristupanje objektu, tzv. *navigacija*. Ako svaki od dva asocirana objekta sadrži referencu na drugi objekat (primetimo da je to redundantno i da se ne preporučuje, ali se ipak dešava), ili postoji neki spoljašnji mehanizam za pronalaženje povezanih objekata (na primer katalog parova i sl.), onda kažemo da je asocijacija dvosmerna i ne označavamo je na poseban način.

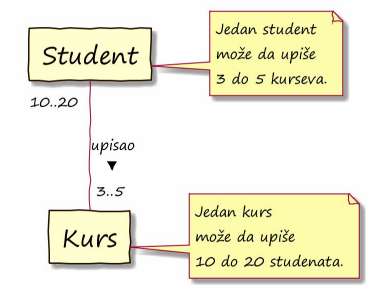
Asocijacija i agregacija često mogu da se koriste relativno ravnopravno. U prethodnom primeru sa narudžbenicama je upotrebljena agregacija da se opiše odnos stavke i proizvoda na koji se ona odnosi. Ako posmatramo realan domen, možemo da se držimo toga da proizvod nije „deo“ stavke, pa da je zato bolje da se agregacija zameni asocijacijom. Sa druge strane, na nivou implementacije, možemo da uočimo da se stavka „sastoji“ od proizvoda, pa da nam agregacija bolje opisuje taj odnos. Imajući u vidu da je agregacija specifičniji odnos od asocijacije (tj. nosi nešto više informacija), često je model informativniji ako se jednosmerna asocijacija zameni agregacijom.

Ako se među objektima klasa ne uspostavlja nikakva trajna povezanost, ali jedna klasa mora da zna za drugu klasu, na primer zato što se ona koristi u opisu ili implementaciji njenih podataka ili metoda, onda je to najslabiji oblik povezanosti, koji se naziva *zavisnost* klasa. Zavisnost klasa se razlikuje od ostalih navedenih odnosa po tome što ona ima više deklarativni nego strukturni karakter. Zavisnost klasa se označava isprekidanom linijom, a u slučaju jednosmerne zavisnosti i strelicom na kraju linije, prema klasi od koje ona druga zavisi. Veoma je poželjno da zavisnost bude jednosmerna. Primer zavisnosti predstavlja odnos klase *Lista* i klase *Iterator* (*Slika 9*), gde objekti klase *Lista* ne sadrže objekte klase *Iterator*, niti čuvaju neke trajne reference na njih, ali ih koriste u komunikaciji, tj. objekti tipa *Iterator* se koriste kao argumenti ili rezultati metoda klase *Lista*.

Svaki odnos može da se označi imenom. Ime odnosa se navodi što bliže sredini linije. Pored imena može da se navede i koju ulogu u odnosu ima koji od učesnika, što se označava navođenjem kratkog naziva uz liniju bliže klasi na koju se odnosi. Da se dijagram ne bi suviše vizualno opterećivao, alternativa je da se navodi samo predikat koji opisuje odnos subjekta i objekta, pa se onda navodi i strelica u obliku trougla, kojom se označava smer čitanja od subjekta prema objektu (*Slika 6*).

Važan element opisa odnosa je i kardinalnost, tj. broj objekata koji mogu da učestvuju u odnosu. Broj koji se navodi bliže jednoj klasi označava koliko objekata te klase može da učestvuje u datom odnosu sa jednim objektom druge klase (*Slika 7*).

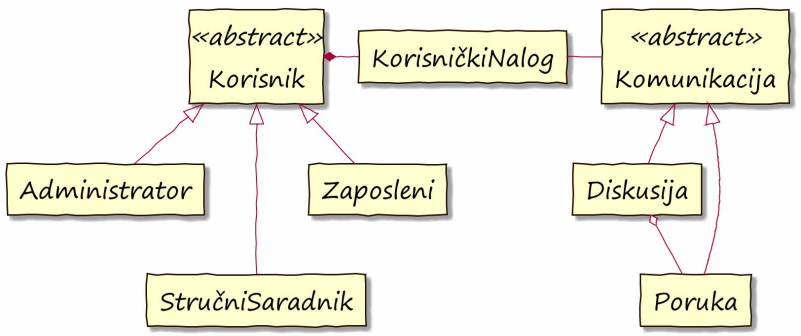
Dodatne zabeleške (poput komentara i detaljnijih opisa) mogu da se navode u pravougaonicima koji se povezuju sa elementima na koje se odnose. Komentar se povezuje isprekidanom linijom sa elementom na koji se odnose (najčešće je to klasa, ali može da bude u pitanju i neki njen deo ili odnos između dve klase), ali se često koriste i alternativne vizualne reprezentacije (*Slika 7*).



Slika 7 – Opisivanje kardinalnosti odnosa *upisanKurs*

Za bliže označavanje prirode ili namene neke klase uobičajena je upotreba tzv. *stereotipova* (*Slika 8*, *Slika 11*). Stereotip se navodi u obliku „<< *naziv stereotipa* >>“ ispred naziva klase. Neki od uobičajenih stereotipova su:

- *abstract* – klasa je apstraktna;
- *auxiliary* – u pitanju je pomoćna klasa, koju korisnici obično ne vide;
- *entity* – klasa predstavlja trajni podatak, obično zapisan u bazi podataka;
- *enumeration* – ne radi se o pravoj klasi već o novom tipu čije se dopuštene vrednosti navode (ili opisuju) u delu namenjenom za atribute;
- *focus* – u pitanju je najvažnija klasa na dijagramu; ako dijagram predstavlja implementaciju neke komponente, ovako je obično označena klasa koja implementira interfejs ili poslovnu logiku komponente;
- *interface* – u pitanju je specifikacija interfejsa koji neke druge klase implementiraju.



Slika 8 – Dijagram klasa podsistema za komunikaciju zaposlenih

Vrste dijagrama klasa

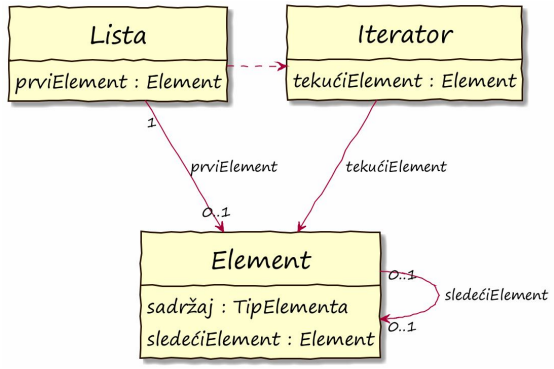
Struktura jednog podsistema može da bude veoma složena, pa predstavljanje svih relevantnih elemenata podsistema na jednom dijagramu može da ima za rezultat dijagram koji je pretrpan informacijama i prilično težak za razumevanje. Zato je većina dijagrama klasa fokusirana na samo neke od aspekata posmatranog dela sistema. Relativno retko se na jednom dijagramu klasa predstavljaju svi atributi, metodi, ograničenja i odnosi svih klasa, već je uobičajeno da se predstavljaju samo oni elementi koji su značajni u nekom kontekstu. Pri tome mora uvek da bude potpuno jasno naznačeno ako neki elementi nisu predstavljeni, da ne bi neko mogao da pomisli da oni uopšte ne postoje.

Razlikujemo sledeće podvrste dijagrama klasa:

- dijagram klasa domena;
- dijagram interfejsa klasa;
- dijagram implementacije klasa i
- detaljan dijagram klasa.

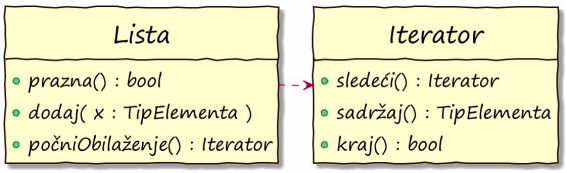
Dijagram klasa koji se prvenstveno bavi strukturom objekata, tj. informacijama koje su karakteristične za sadržaj objekata i njihove odnose, a zapostavlja njihovo

ponašanje tako što uopšte ne sadrži metode, obično se naziva *dijagram klasa domena* ili *dijagram modela domena* (*Slika 9*). Namena dijagrama klasa domena je da predstavi koje su sve informacije ili podstrukture sadržane u posmatranom domenu ili delu domena, kao i kakav je statički odnos između tih informacija. Takav dijagram može da se koristi i u fazi projektovanja baza podataka, kada se često naziva i *dijagram klasa podataka*.



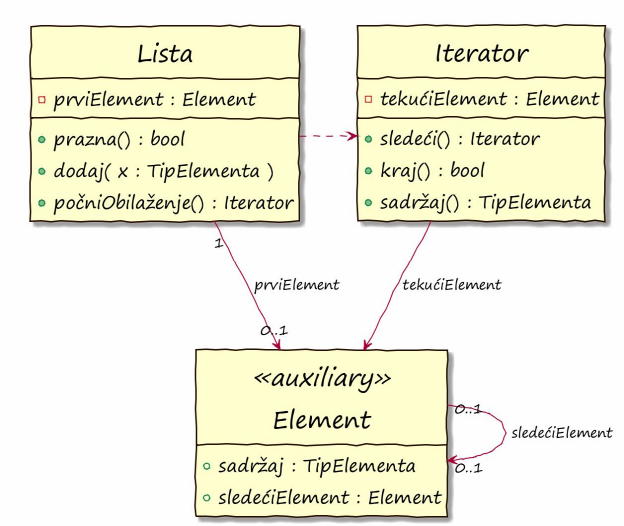
Slika 9 – Dijagram modela domena *Lista*

Ako se dijagram klasa bavi prvenstveno ponašanjem, onda može da sadrži metode a da zanemari atribute klasa. Ako sadrži samo javne metode onda se naziva *dijagram interfejsa klasa* (*Slika 10*), a ako sadrži neki izabran skup metoda onda se imenuje tako da bude jasno šta sadrži. Dijagram interfejsa klasa često ne sadrži neke klase koje su značajne za internu strukturu ili implementaciju nekih od predstavljenih klasa, ako se ne pojavljuju u interfejsu pa korisnik ne mora da zna za njihovo postojanje. Predstavljen primer dijagrama interfejsa klasa (*Slika 10*) ne sadrži klasu *Element*, zato što se ona ne pojavljuje u interfejsu klase *Lista*, ali zato sadrži klasu *Iterator*, koja predstavlja tip rezultata metoda *počniObilaženje*.



Slika 10 – Dijagram interfejsa *Lista*

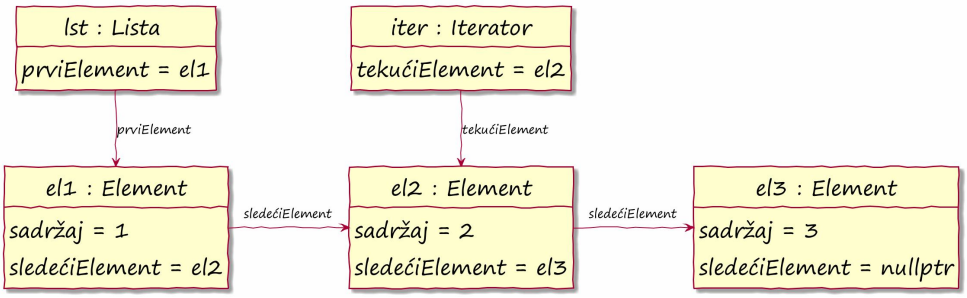
Dijagram koji sadrži sve metode, bez atributa, često se naziva *dijagram implementacije*, mada se isti naziv često koristi i za kompletne dijagrame, sa svim metodima i atributima, a koji se nazivaju i *detaljnim dijagramom klasa*. Primer detaljnog dijagrama klasa (*Slika 11*) opisuje sve elemente implementacije dinamičke strukture podataka *Lista*.



Slika 11 – Dijagram klasa *Lista*

5.3 Dijagram objekata

Dijagram objekata nam pomaže da bolje razumemo dinamičku prirodu odnosa među objektima. Koristi se kao dopuna dijagrama klasa i dijagrama komunikacije za opisivanje dinamičkih sistema. Sadrži nazive klasa, nazive objekata, imena i vrednosti atributa i odnose među objektima. On predstavlja objekte i njihove odnose u jednom trenutku vremena, pa je zato za ilustraciju složenije dinamičke prirode nekih klasa često potrebno da se napravi više odgovarajućih dijagrama objekata.



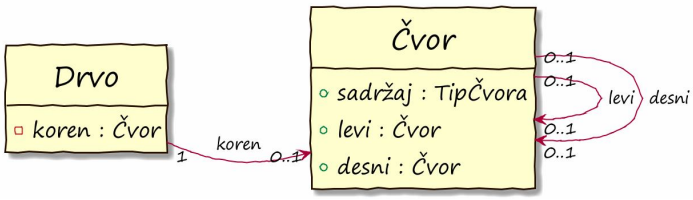
Slika 12 – Dijagram objekata koji ilustruje implementaciju liste

Na dijagramu se predstavlja skup objekata koji postoje u nekom izabranom trenutku vremena. Objekti se predstavljaju pravougaonicima koji u prvom delu sadrže naziv objekta i njegovu klasu (tip), a u drugom delu imena i vrednosti atributa. Dijagram ne mora da sadrži vrednosti svih atributa, već je dovoljno da se predstave oni koji su nam od značaja za konkretan dijagram.

Na primer, *Slika 11* sadrži dijagram klasa koji opisuje odnose klasa koje implementiraju dinamičku strukturu liste. Međutim, iz tih opisa može da ne bude sasvim jasno kako izgleda lista u memoriji. Dodavanjem dijagrama objekata (*Slika 12*), koji opisuje stanje jedne izabrane liste u nekom trenutku, možemo da pomognemo da se lakše razume kako su elementi interno povezani. Dodatno možemo da predstavimo i dijagram objekata koji opisuje praznu listu i završni iterator (*Slika 13*). Ako bi bile dopuštene i kružne liste, to bi takođe moglo da se predstavi posebnim dijagramom objekata.

Slika 13 – Dijagram objekata koji ilustruje implementaciju prazne liste

Slično prethodnom primeru, ako napravimo dijagram klasa koje opisuju strukturu binarnog drveta (*Slika 14*), onda odnosi između čvorova drveta mogu da budu još teže razumljivi nego u slučaju liste. Zato bi i u ovom slučaju mogao da bude od pomoći dodatni dijagram objekata (*Slika 15*).



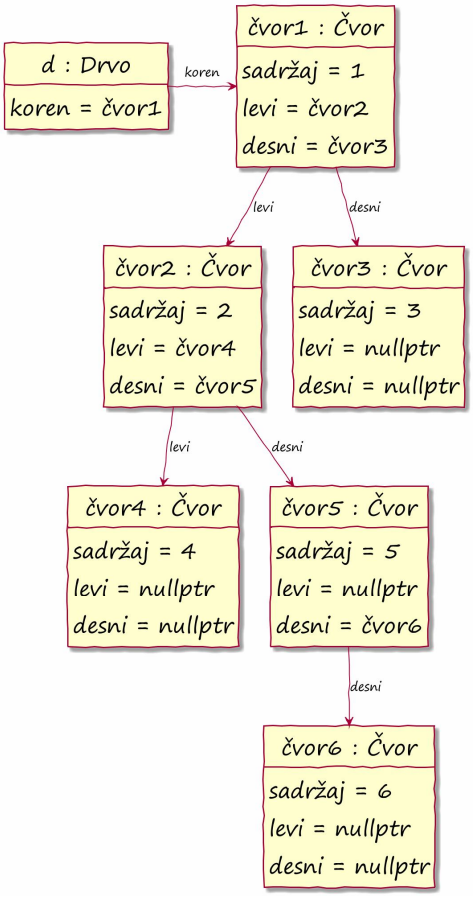
Slika 14 – Dijagram modela domena *Drvo*, koji predstavlja binarno stablo

5.4 Dijagram komponenti

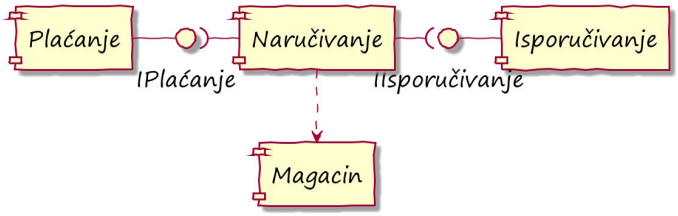
Komponenta predstavlja funkcionalnu ili fizičku celinu softvera, koja ima prepoznatu i zaokruženu ulogu ili funkciju. Za komponente važe veoma slična pravila projektovanja kao za klase – komponenta bi trebalo da ima jednu odgovornost i da tu odgovornost obavlja samostalno, koliko je god to moguće. Na komponente se takođe primenjuju pravila enkapsulacije i izdvajanja interfejsa. Komponente bi trebalo da se međusobno povezuju putem interfejsa. Jedna komponenta može da ima više interfejsa, na primer za različite protokole komunikacije ili za različite vrste korisnika.

Komponenta se predstavlja pravougaonikom sa dodatnim oznakama koje ukazuju na to da se radi o komponenti. U verziji *UML* 1 se kao oznaka koristi par ispusta sa kojima bi komponenta trebalo da liči na slagalicu. U verziji *UML* 2 se

umesto toga koristi pravougaonik koji u desnom gornjem uglu ima simbol slagalice kao oznaku stereotipa. Obe verzije se ravnopravno koriste.



Slika 15 – Dijagram objekata koji ilustruje implementaciju binarnog drveta



Slika 16 – Primer dijagrama komponenti

Javni interfejs se predstavlja malim krugom koji može da ima ime i koji je povezan pravom linijom sa odgovarajućom komponentom. Upotreba interfejsa se označava linijom sa polukrugom ili strelicom prema interfejsu (*Slika 16*). Ako jedna

komponenta zavisi od druge, ali ne želimo da ističemo interfejse, onda taj odnos može da se predstavi isprekidanom strelicom.

Interfejsi moraju da se predstavljaju ako u projektu potencijalno imamo više komponenti koje pružaju isti interfejs, ili ako je fokus upravo na interfejsima. U ostalim slučajevima, a posebno ako na dijagramu ima mnogo komponenti, onda je nekada korisno da se interfejsi sakriju i da se ostave samo neposredne zavisnosti.

5.5 Dijagram slučajeva upotrebe

Dijagram slučajeva upotrebe ilustruje na koji način neke grupe korisnika mogu da stupaju u interakciju sa softverom ili delom softvera. Osnovni elementi ovog dijagrama su *slučajevi upotrebe*, *akteri* koji u njima učestvuju i njihovi međusobni odnosi. Svaka celovita funkcionalna interakcija korisnika se predstavlja kao jedan zaokružen *slučaj upotrebe*. Različite vrste korisnika se predstavljaju kao različite vrste aktera. Pored toga, dijagram može da sadrži i pakete i podsisteme.

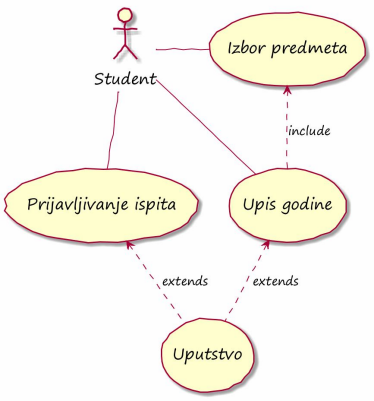
Za razliku od ostalih dijagrama *UML*-a, ova vrsta dijagrama služi samo da nam pruži površan osnovni uvid u elemente i odnose među njima. Glavni deo specifikacije odgovarajućih funkcionalnih celina čine tekstualni opisi slučajeva upotrebe. Svaki slučaj upotrebe mora da bude propraćen opsežnom tekstualnom dokumentacijom, koja ga opisuje do pojednosti.

U zavisnosti od nivoa dekompozicije sistema, slučajevi upotrebe mogu da budu na višem ili nižem nivou apstrahovanja. Pri predstavljanju konceptualnog nivoa softvera teži se da se sa što manje slučajeva upotrebe opiše ceo sistem. Tada su slučajevi upotrebe vrlo apstraktni i opisuju čitav model upotrebe sistema od strane jedne vrste korisnika. Takvi slučajevi upotrebe se obično nazivaju *poslovni slučajevi upotrebe* i uobičajeno se teži da jedan korisnik učestvuje u što manje poslovnih slučajeva, ili čak u samo po jednom. Na primer, *Student studira*, *Nastavnik podučava* i sl.

Na nižim nivoima apstrahovanja, teži se da jedan slučaj upotrebe odgovara jednoj celovitoj aktivnosti korisnika, koja je istovremeno dovoljno velika da predstavlja značajnu celinu, ali i dovoljno mala da može relativno brzo da se implementira. Kao donja granica dekompozicije se obično zahteva da slučaj upotrebe ima *prepoznatljiv rezultat*, tj. da nakon što akter upotrebi sistem na opisan način, kao rezultat ostaje neki trajan trag, koji predstavlja ili završni rezultat tog slučaja upotrebe ili osnovu za započinjanje ili obavljanje nekog drugog slučaja upotrebe. Tako oblikovani slučajevi upotrebe kasnije predstavljaju osnovne elemente planiranja i implementiranja softvera.

Akteri se predstavljaju jednostavnim simbolima koji označavaju čoveka, ali mogu da se koriste i drugačiji simboli. Naziv aktera se navodi ispod simbola. Slučajevi upotrebe se predstavljaju elipsama u kojima su navedeni nazivi. Akteri i slučajevi

upotrebe u kojima oni učestvuju se povezuju (obično pravim) linijama bez strelica na krajevima (*Slika 17*).



Slika 17 – Dijagrama slučajeva upotrebe – Studiranje

Odnosi između slučajeva upotrebe se označavaju strelicama sa isprekidanim linijama (*Slika 17*). Svaki odnos je označen stereotipom koji opisuje prirodu odnosa. Uobičajeni odnosi su:

- *include* – slučaj obuhvata čitav slučaj prema kome ide strelica; obično se koristi za opisivanje delova posla koji se često ponavljaju u drugim slučajevima;
- *extends* – slučaj od koga ide strelica predstavlja *moguće* tj. *opcionalno* proširenje drugog slučaja, u vidu dodatka, ali ne i njegov obavezan deo.

Tekstualna dokumentacija slučajeva upotrebe mora da bude dovoljna da na osnovu nje može da se pristupi planiranju i izradi implementacije. U ranim fazama projektovanja se obično ne navode sve pojedinosti koje su potrebne za implementaciju, ali je potrebno da se navede dovoljno informacija da mogu da se okvirno procene potreban obim i trajanje poslova na implementaciji, kao i eventualne međuzavisnosti sa drugim slučajevima upotrebe. Kasnije, pri detaljnom planiranju implementacije, dokumentacija mora da se dopuni tako da omogući da slučaj upotrebe može da se implementira bez dodatnih zahteva za informacijama.

Opis slučaja upotrebe može da sadrži različite vrste informacija, ali po pravilu mora da sadrži bar sledeće elemente:

- naziv – mora da dobro ilustruje slučaj upotrebe, ali je dobro da bude što kraći;
- aktere – učesnici u slučaju upotrebe; ako različiti subjekti imaju različite uloge, to je potrebno da se objasni;

- kratak opis – suština slučaja upotrebe se objašnjava ukratko, u svega nekoliko rečenica, često i samo jednom rečenicom;
- preduslove – koji preduslovi moraju da budu ispunjeni da bi akteri mogli da započnu slučaj upotrebe;
- postuslove – koji uslovi moraju da budu ispunjeni posle odvijanja slučaja upotrebe;
- opis toka slučaja upotrebe – detaljan opis koraka koji se izvode tokom odvijanja slučaja upotrebe, uključujući i njihov redosled i sve moguće osnovne tokove koraka;
- opis posebnih slučajeva – ako postoje posebni slučajevi, potrebno je da se opiše kada nastupaju i po čemu se odvijanje postupka razlikuje od uobičajenog (npr. greške pri unosu i sl.);
- dijagrame koji tačnije opisuju slučaj upotrebe – često se slučaj upotrebe najbolje opisuje dijagramima aktivnosti ili sekvence, ali mogu da budu od koristi i svi drugi dijagrami *UML*-a;

Dodatni neobavezni elementi obuhvataju sve one informacije koje mogu da budu od značaja pri planiranju ili implementiranju slučaja upotrebe. Na primer, mogu da obuhvataju:

- klasifikaciju – bilo po vrsti slučaja upotrebe ili logičkoj ili poslovnoj celini kojoj pripadaju;
- podaci – ako slučaj upotrebe obrađuje ili proizvodi veću količinu podataka, onda je dobro da se ti podaci opišu i izdvojeno, a ne samo u okviru opisa toka;
- dodatne nefunkcionalne zahteve – na primer, ciljne performanse;
- bezbednosne karakteristike – nivo poverljivosti podataka ili postupka, način pristupanja poverljivim podacima, način kodiranja ili proveravanja ispravnosti podataka i sve drugo što može da ima veze sa bezbednošću podataka;
- pregled najvažnijih rizika – spisak rizika za koje je prepoznato da mogu da nastupe, eventualno alternativne karakteristike ili moguće očekivane izmene u slučaju upotrebe, moguća potencijalna unapređenja i slično;
- jasno objašnjen cilj slučaja upotrebe – ako iz naziva, kratkog opisa i opisa toka izvođenja možda nije sasvim jasno koja je svrha konkretnog slučaja upotrebe ili koju ulogu on ima u okviru softvera kao celine, onda je to potrebno da se objasni;

- prilozi – različiti prilozi koji omogućavaju da se bolje razumeju svrha i tok slučaja upotrebe: primeri formulara ili dokumenata, uzorci realnih podataka i slično.

Radi ilustracije, navešćemo jedan primer tekstualnog opisa slučaja upotrebe (*Slika 18*). Detaljnost odgovara ranim fazama planiranja.

Naziv: Prijavljivanje na konkurs

Akteri: Kandidat koji se prijavljuje na konkurs za upis na fakultet, službenik koji radi na evidentiranju kandidata.

Kratak opis: Kandidat podnosi neophodna dokumenta i formulare na konkurs za upis na fakultet.

Tok događaja:

Osnovni tok:

1. Kandidat donosi rukom popunjen obrazac P1 kao i sledeća dokumenta:
   - fotokopiju izvoda iz matične knjige rođenih (original na uvid),
   - fotokopije svedočanstva sva četiri razreda srednje škole (originali na uvid),
   - fotokopiju svedočanstva o završenoj srednjoj školi (original na uvid),
   - potvrdu o uplati odgovarajućeg iznosa za prijavu.
2. Službenik proverava dokumenta i unosi podatke u sistem
3. Službenik štampa popunjene formulare
4. Kandidat potpisuje i predaje formulare
5. Službenik štampa i potpisuje obrazac P2 i uručuje kandidatu.

Alternativni tokovi:

1a. Za strane državljane (osim za državljane BiH koji su završili školu u RS), potrebna su i nostrifikovana školska dokumenta ili potvrda o započetoj nostrifikaciji

1b. Ako je kandidat položio prijemni ispit na drugom fakultetu, potrebna je i potvrda o položenom prijemnom ispitu na drugom fakultetu

1c. Ako je kandidat nosilac nagrada sa takmičenja koje mogu da doprinesu oslobađanju od prijemnog ispita, onda popunjava i formular P3.

...

Preduslovi: Kandidat je završio srednju školu i dobio odgovarajuća dokumenta.

Postuslovi: Kandidat je prijavljen i dobio je potvrdu koja sadrži redni broj prijave.

Prilozi: sadržaj i predložen izgled formulara P1, P2 i P3

P1. Prijavni list sa ličnim podacima kandidata, podacima o mestu rođenja, državljanstvu, roditeljima, prebivalištu, prethodno završenoj instituciji i studijskim programima koje kandidat želi da upiše.

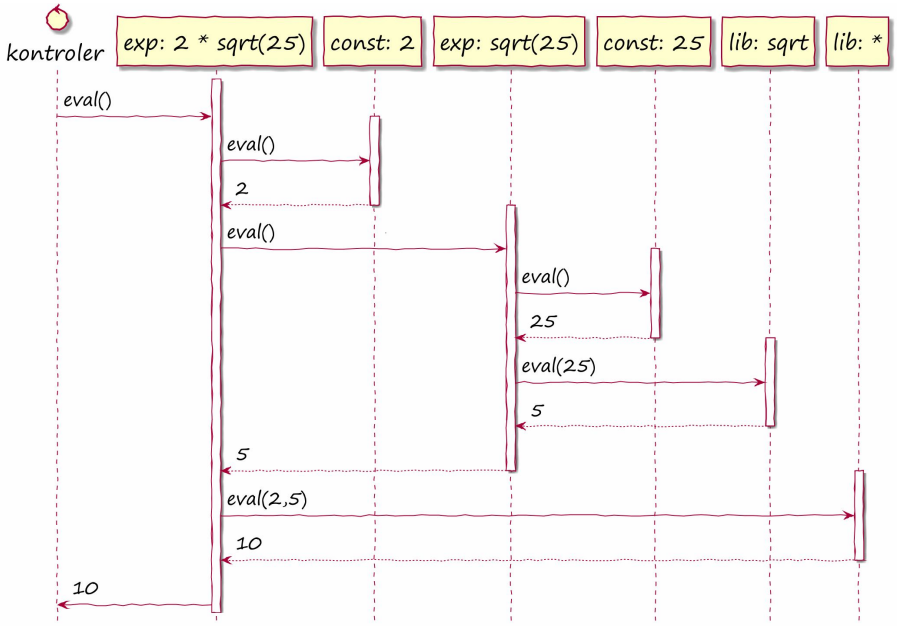
P2. Identifikacioni list sa imenom, prezimenom i rednim brojem prijave.

P3. Formular za diplome koje kandidat prijavljuje i na osnovu kojih želi da bude oslobođen od prijemnog ispita

Slika 18 – Primer tekstualne dokumentacije slučaja upotrebe

5.6 Dijagram sekvence

Dijagram sekvence služi za opisivanje toka odvijanja slučaja upotrebe. On omogućava da se predstavi koji subjekat ili objekat je zadužen za koji korak slučaja upotrebe, kao i kojim redom se odvijaju koraci i prenose odgovornosti sa jednog na drugi subjekat ili objekat. Na najnižem nivou apstrakcije, dijagram sekvence predstavlja način implementiranja nekog postupka, sve do nivoa pozivanja konkretnih metoda, odnosno slanja poruka između objekata.



Slika 19 – Dijagram sekvence koji opisuje korake izračunavanja grafa izraza

Vertikalna dimenzija predstavlja tok vremena, od vrha prema dnu dijagrama. Horizontalno se dijagram deli na prostore koji odgovaraju pojedinačnim objektima. U svakoj koloni je u vrhu identifikacija objekta, a zatim se ispod crta isprekidana vertikalna linija koja predstavlja životni vek objekta.

Na toj liniji se crtaju uski pravougaonici koji predstavljaju okvire aktivnosti. Okvir aktivnosti objekta započinje prijemom poruke, a završava se kada se primljena poruka obradi. Ako se objekat pravi u toku sekvence, onda je prva poruka koja mu se šalje označena sa *napravi* (ili *create* ili slično). Ako se objekat u nekom trenutku uništava, onda se na dnu odgovarajućeg okvira aktivnosti crta „X".

Sekvenca se predstavlja nizom poruka koje se razmenjuju. Poruka kojom se vraća rezultat se predstavlja strelicom u suprotnom smeru. Poruka sa rezultatom ne mora da se navodi. Ipak, ako je komunikacija sinhrona, tako da objekat koji je uputio poruku ne radi ništa dok ne dobije odgovor, onda je uobičajeno da se i ona navodi. Takođe, ako je rezultat koji se vraća značajan za kontekst dijagrama, dobro je da se navede poruka sa tim rezultatom.

Dijagram sekvence može da se predstavlja na različitim nivoima apstrakcije. U principu, svaka poruka odgovara fizičkom slanju poruka između objekata, tj. pozivanju metoda. Na najnižem nivou apstrahovanja se predstavljaju doslovno sve poruke i onda dijagram sekvence praktično ilustruje kako je potrebno da se implementiraju odgovarajući metodi. Na višim nivoima se ne predstavljaju sve poruke, tj. ne opisuju se svi metodi.

U primeru (*Slika 19*) je predstavljen dijagram sekvence kojim se opisuje kako teče izračunavanje izraza koji je predstavljen grafom, u kome postoje konstante, aplikacije operacija i bibliotečke operacije.

5.7 Umesto zaključka

Ono što savremenom društvu nije uspelo sa *esperantom*, u domenu razvoja softvera je praktično uspelo sa *UML*-om – dobili smo jezik koji poznaju (ili bi bar trebalo da poznaju) praktično svi učesnici u razvoju softvera. Iscrpno predstavljanje *UML*-a bi zahtevalo mnogo više prostora nego što nam kontekst dopušta, pa smo se zato ograničili samo na predstavljanje osnovnih vrsta dijagrama, da bi čitaoci mogli da počnu da koriste *UML*, ali i da ih njegovo delimično upoznavanje motiviše da se dodatno angažuju i da ga bolje upoznaju.

Mnogi savremeni alati za razvoj softvera omogućavaju relativno visok nivo integrisanja *UML*-a i alata za programiranje. Većina njih omogućava automatsko pravljenje *UML*-dijagrama na osnovu postojećeg programskog koda. Manji broj naprednijih alata omogućava i obrnuto – pravljenje kostura implementacije programa na osnovu *UML* dijagrama, pri čemu dijagrami klasa mogu da posluže kao osnov za automatsko definisanje kostura klasa, a dijagrami ponašanja (dijagrami sekvence i drugi) za automatsko implementiranje metoda. Primeri takvih alata su *IBM Software Architect* i *Visual Paradigm*.

Na srpskom jeziku postoji solidna literatura o *UML*-u. Iako prevedene knjige uglavnom nisu najsvežije, one su ipak sasvim dobre za početak. Istakli bismo [*Booch 1999*] i [*Fowler 2003a*].

6 - Principi projektovanja softvera

Moramo da se borimo protiv haosa,
a najefikasniji način je da sprečimo njegovo nastajanje.
Edsher Daikstra

6.1 Pojam i motivacija

Pri projektovanju softvera se rukovodimo nekim načelima, koja nam omogućavaju da lakše razrešavamo različite dileme i donosimo odgovarajuće odluke. Takva načela se često obrađuju u literaturi i obično se nazivaju *principima projektovanja*. U različitim izvorima mogu da se pronađu različiti skupovi principa projektovanja. Obično se dele prema poreklu (agilni razvoj, OO metodologije i drugo), prema domenu primene (raspoređivanje odgovornosti, grupisanje, razdvajanje i sl.) i prema značaju. Ovde ćemo da razmotrimo neke od najvažnijih skupova principa projektovanja, koji su ostvarili najveći uticaj na oblast projektovanja softvera u XXI veku.

Savremeni razvoj softvera veoma često pretpostavlja agilni razvoj i primenu OO metodologija. Zbog toga se i problem projektovanja softvera danas obično posmatra u kontekstu agilnog razvoja i OO metodologija. Skupovi principa projektovanja softvera, koje ćemo da predstavimo u ovom poglavlju, takođe se odnose prvenstveno na agilne i OO metodologije, ali su u velikoj meri formulisani u odnosu na opšte pojmove (kao što su elementi programa, celine i delovi, zavisnosti i slično) pa bez mnogo prilagođavanja mogu da se primenjuju i na druge razvojne metodologije.

Principi projektovanja softvera obično imaju oblik *pravila* ili *preporuka*. Osnovni cilj svih principa je da nam pomognu da u određenim situacijama uočimo i izaberemo one kriterijume koji su značajniji i da zatim na osnovu tih kriterijuma odlučujemo kako da strukturiramo elemente programa koji projektujemo. Kada

razmatramo neki manji skup principa, onda možemo da primetimo da nam navedeni principi nisu dovoljni da razrešimo sve slučajeve sa kojima se susrećemo u praksi. Sa druge strane, kada razmatramo više skupova principa, onda možemo da dođemo u situaciju da je primena nekih principa delimično redundantna ili čak delimično suprotstavljena. Upravo zbog toga principe valja prihvatiti kao *savete* ili *lekcije*, a ne kao ultimativna pravila.

Principi koje ćemo predstaviti su oblikovani na osnovu praktičnog iskustva značajnih autoriteta u oblasti projektovanja softvera i dobro su provereni u širokoj programerskoj populaciji. Dobro poznavanje principa projektovanja bi početnicima u projektovanju trebalo da u određenoj meri nadoknadi nedostatak iskustva i omogući uspešnije početne korake u razvoju složenijih softverskih projekata. Ako poneki princip možda izgleda trivijalno ili očigledno, to ne znači da bi trebalo da mu posvetimo manje pažnje – takvi principi su formulisani i istaknuti upravo zato što se relativno često dešava da se trivijalne ili očigledne stvari previđaju.

Svi principi koje ćemo ovde upoznati odnose se na strukturno projektovanje. Zato se često nazivaju i principima strukturnog projektovanja ili principima dizajniranja softvera.

Sadržaj ovog poglavlja je tesno povezan sa sadržajem poglavlja *7* - Obrasci za projektovanje. Principi projektovanja nam daju preporuke, a obrasci predstavljaju dobre primere njihove primene. Zbog toga ćemo se na nekim mestima referisati na odeljke koji tek slede. Da bismo dobro razumeli principe i obrasce, neophodno je da upoznamo i jedno i drugo.

6.2 Osnovni principi OO dizajna

Takozvani *osnovni principi OO dizajna* u svom osnovnom obliku potiču iz OOP, ali se vrlo često primenjuju i u drugim domenima. Izvorno se odnose na načine oblikovanja klasa, ali se podjednako dobro primenjuju i na druge strukturne elemente softvera, kao što su funkcije, metodi, komponente, interfejsi, paketi i drugo.

Ovaj skup principa projektovanja se obično pripisuje Robertu Martinu. On nije lično osmislio sve pojedinačne principe, ali je uticao na njihovo formulisanje u obliku u kome su danas poznati [*Martin 2003*]. Osnovni princopi OO dizajna se često označavaju engleskom skraćenicom *SOLID*, koja se gradi od početnih slova njihovih engleskih naziva.

Princip jedinstvene odgovornosti (SRP)

„Klasa bi trebalo da ima samo jedan razlog da se menja.“

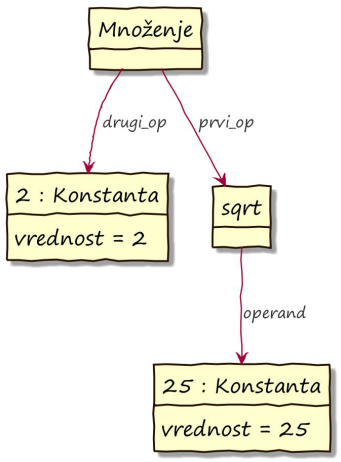
Princip jedinstvene odgovornosti (engl. *SRP – Single Responsibility Principle*) ima i alternativne oblike, poput:

- klasa bi trebalo da ima samo jednu odgovornost ili
- komponenta bi trebalo da ima samo jedan razlog da se menja.

U osnovi, umesto *klase* može da se nađe bilo koji drugi strukturni element. Sa druge strane, „jedan razlog da se menja" je nešto opštije od „jedne odgovornosti", ali u praksi oba oblika obično imaju isti smisao.

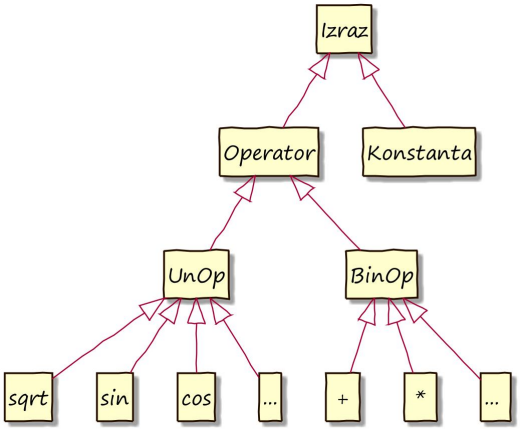
Značaj ovog osnovnog principa se ogleda u težnji prema čistoj i jasnoj modularizaciji softverskog sistema. Tesno je povezan sa funkcionalnom dekompozicijom i dekompozicijom prema promenljivosti. Smanjivanjem broja mogućih razloga za menjanje različitih strukturnih elemenata softvera, smanjuje se i mogućnost da neka manja promena izazove veliki broj drugih promena u programu. Na taj način se olakšavaju i pisanje i održavanje softvera.

Iako ovaj princip izgleda vrlo jasno i razumljivo, čak do te mere da se stiče utisak da može i da se intuitivno primenjuje, u praksi stvari stoje nešto drugačije i relativno često se nailazi na primere projekata u kojima ovaj princip nije dobro (ili čak nije uopšte) primenjen.



Slika 20 – Primer drveta izraza

Radi ilustracije, razmotrimo softverski projekat u kome se pojavljuje apstraktno sintaksno drvo izraza (ASD), gde čvorovi tog drveta predstavljaju objekte koji pripadaju klasama hijerarhije koja modelira različite vrste literala, operatora i drugih sintaksnih konstrukcija. Sintaksno drvo nam služi za pravljenje interne reprezentacije izraza i gradi se parsiranjem izraza. Na primer, Slika 20 predstavlja primer drveta izraza, koje bi se dobilo parsiranjem izraza `2*sqrt(25)`. U zavisnosti od sintakse jezika, ova hijerarhija može da sadrži veliki broj različitih klasa. Slika 21 predstavlja primer jednostavnije hijerarhije klasa čvorova ASD.



Slika 21 – Hijerarhija klasa koje predstavljaju čvorove ASD

Nakon izgradnje izraza, može da bude potrebno da se on prevede u neki drugi oblik (možda na mašinski jezik), ili da se ispiše u nekom drugom obliku, ili da se optimizuje, a možda i da se izračuna (tj. da se interpretira). Jedan od uobičajenih načina implementiranja ovakvih zahteva je dodavanje novog ponašanja kroz dodavanje novih metoda čvorovima hijerarhije izraza, tako da oni mogu da ispišu, izračunaju ili prevedu odgovarajući podizraz. Posledica takvog pristupa je da sve klase hijerarhije implementiraju veliki broj metoda, među kojima često ne postoji skoro nikakva kohezija:

```
class Izraz { ...
public:
    virtual MasinskiKod PrevediNaMasinskiKod() = 0;
    virtual string Ispisi() = 0;
    virtual double Izracunaj() = 0;
    ...
};
```

Iako takvo rešenje izgleda prirodno i relativno jednostavno, takvim pristupom ne samo da pravimo klasu sa slabom kohezijom, već i narušavamo princip jedinstvene odgovornosti (a kao što ćemo malo kasnije videti, i još neke druge važne principe projektovanja).

Klase apstraktnog drveta izraza već imaju jednu odgovornost – one služe da grade internu reprezentaciju izraza. Dodavanjem bilo koje druge odgovornosti (kao što je, na primer, izračunavanje vrednosti izraza) mi bismo svakoj od klasa povećali broj odgovornosti, a time i broj mogućih razloga za menjanje. Iako u slučaju dodavanja samo jedne nove operacije to može da nam izgleda kao dobar pristup, problem se u punoj meri ispoljava kada dođemo do toga da moramo da svakoj klasi dodamo više različitih aspekata ponašanja – pisanje, prevođenje, optimizovanje, izračuna-

vanje i drugo. Održavanje takvog programa preti da postane veoma nezahvalan posao. Umesto toga, možemo da primenimo obrazac za projektovanje *Posetilac* i da problem rešimo na mnogo čistiji način, bez nagomilavanja odgovornosti, njihovim raspoređivanjem u više različitih klasa posetilaca (*7.5 – Primer – Obrazac Posetilac*, na str. 153).

Obrascem *Posetilac* ćemo se baviti u narednom poglavlju, a ovde ćemo samo ukratko da istaknemo njegovu osnovnu ideju – klase hijerarhije zadržavaju svoju izvornu odgovornost modeliranja strukture, a za svaku dodatnu odgovornost se pravi po jedan novi posetilac. Posetilac obilazi strukturu i radi posao za koji je zadužen. Posetilac čak i ne zna kako se struktura obilazi, već samo šta mora da se radi sa kojom vrstom čvorova. Obilaženje strukture je ponašanje koje predstavlja sastavni deo strukture, tj. čini jedinu dodatnu odgovornost posmatrane hijerarhije klasa.

U konkretnom primeru, klase drveta izraza su odgovorne samo za održavanje strukture izraza i omogućavanje posetiocima da ih obilaze na odgovarajući način (na primer, konstante moraju da mogu da isporuče vrednost, a operacije moraju da mogu da isporuče operaciju za koju su zadužene, kao i konkretne operande). Sa druge strane, svaka klasa posetilac je odgovorna da na izrazima izvodi tačno jednu odgovarajuću operaciju. Na primer, jedan posetilac bi se bavio izračunavanjem, drugi ispisivanjem, treći prevođenjem i slično.

Princip otvorenosti i zatvorenosti (OCP)

„Elementi softvera bi trebalo da budu otvoreni za proširivanje i zatvoreni za menjanje.“

Princip otvorenosti i zatvorenosti (engl. *OCP – Open-Closed Principle*) se odnosi prvenstveno na klase, ali i na sve druge strukturne elemente softvera (komponente, pakete, funkcije, metode i drugo).

Otvorenost za proširivanje podrazumeva da bi strukturnom elementu trebalo da može relativno jednostavno da se doda novo ponašanje, ako se za tim ukaže potreba. Na taj način se obezbeđuje fleksibilnost projekta, posebno u kontekstu agilnog razvoja gde su promene i dopune specifikacija vrlo česte i prirodno dovode i do potrebe za novim (tj. proširenim) ponašanjem.

Sa druge strane, ako naš element već postoji, to znači da se on na nekim mestima već koristi. Zbog toga zatvorenost za menjanje podrazumeva da bi projekat trebalo da omogući da se proširivanje izvodi bez menjanja postojećeg programskog koda, a pre svega bez menjanja interfejsa i postojećeg ponašanja. Štaviše, u kontekstu distribucije softvera, ne samo da ne želimo da se menja već napisan i korišćen programski kod, već ne želimo ni da se menjaju isporučeni binarni moduli (npr. izvršni programi ili dinamičke biblioteke).

Put do primene ovog principa vodi preko tehnika koje su uobičajene za OOP – nasleđivanja, dinamičkog vezivanja metoda i definisanja apstraktnih klasa (i interfejsa). Ovaj princip nas podstiče da oblikujemo apstraktne hijerarhije klasa tako da se odgovarajuće ponašanje može po želji i potrebi dodavati pravljenjem novih izvedenih klasa, a bez menjanja već napisanih klasa.

Važno je da uvek imamo na umu da svaka apstrakcija doprinosi složenosti programskog koda. Ako bismo unapred predvideli i ugradili sve apstrakcije koje pretpostavljamo da bi nekada mogle da nam zatrebaju, to bi onda imalo za rezultat programski kod koji bi bio pun elemenata koji se ne upotrebljavaju, ali nam nepotrebno usložnjavaju održavanje. Imajući to u vidu, kao i princip agilnog razvoja da se nijedan deo programa ne piše ako nije sada i odmah potreban, zaključujemo da u praksi ne bi trebalo da pišemo apstraktne delove koda, sve dok ne budemo sigurni da su nam potrebni. Posledica takvog pristupa je da naš programski kod neće biti zatvoren za sve vrste izmena i da ćemo za neka proširenja ipak morati da prethodno izvršimo i neke izmene *zatvorenog* koda. U takvim slučajevima menjanje koda izvodimo u formi refaktorisanja, što nam obezbeđuje da u eventualnom narednom sličnom slučaju možemo da rešimo problem samo proširivanjem, bez menjanja.

Radi ilustracije možemo ponovo da razmotrimo primer sa posetiocima hijerarhije klasa apstraktnog drveta izraza (*7.5 – Primer – Obrazac Posetilac*, na str. 153), sa kraja odeljka o principu jedinstvene odgovornosti. Pri prvom dodavanju novog ponašanja u formi posetioca, morali bismo da dopunimo hijerarhiju čvorova izraza novom apstrakcijom, tako da se posetiocima omogući obilaženje strukture izraza. To predstavlja menjanje postojećeg koda, čime se nedvosmisleno narušava princip zatvorenosti za menjanje. Ali kada to jednom uradimo, onda će kasnije, pri dodavanju svakog sledećeg novog ponašanja, biti dovoljno da napravimo novu klasu posetioca koja će posećivati drvo izraza pomoću već postojećeg mehanizma, uz potpuno poštovanje principa otvorenosti i zatvorenosti.

Princip zamenljivosti (LSP)

„Neophodno je da podtipovi mogu da zamene bazne tipove."

Princip zamenljivosti (engl. *LSP – The Liskov Substitution Principle*) je dobio ime po Barbari Liskov, koja ga je izvorno definisala u nešto opširnijem obliku još 1988. godine: *Ako za svaki objekat o_1 tipa S postoji neki objekat o_2 tipa T takav da za sve programe P, u kojima se pojavljuje tip T, važi da se ponašanje programa P ne menja kada se objekat o_1 zameni objektom o_2, onda S predstavlja podtip tipa T*. U još opštijem i formalnijem obliku ovaj princip glasi: *Ako svako svojstvo A, koje važi za sve objekte tipa T, istovremeno važi i za sve objekte tipa S, onda S predstavlja podtip tipa T.* Primetimo da u takvom obliku ovaj princip predstavlja upravo definiciju pojma podtipa, kao tipa koji ispunjava uslov zamenljivosti.

Princip zamenljivosti predstavlja osnovu hijerarhijskog polimorfizma – ako on ne bi važio, onda ni hijerarhijski polimorfizam ne bi imao smisla. Međutim, iako može

da izgleda da ovaj principi važi „sam po sebi“, te da je zbog toga možda i *suvišan* ili *redundantan*, stvari zapravo stoje sasvim drugačije. Pri projektovanju klasa je često potrebno da se interfejs klase oblikuje veoma pažljivo i da se još pažljivije definiše semantika metoda klase, da ne bi došlo do narušavanja ovog principa. I pored toga, njegovo narušavanje nekada ne može da se izbegne na jednostavan način.

Dobra ilustracija problematičnog usaglašavanja sa ovim principom je predstavljena u poglavlju o OO metodologijama, u odeljku *3.4 – Slabosti objektno-orijentisanih koncepata*, na stranici 37. Podsetimo se, u baznoj klasi `Pravougaonik` je definisan metod `postaviSirinu`, koji se zatim predefiniše za klasu `Kvadrat`, tako da menja ne samo širinu nego i visinu, zato da bi objekat zadržao osnovnu osobinu kvadrata da su mu sve stranice podudarne.

Obratimo pažnju na naredni primer koda:

```
void promeniDimenzije( Pravougaonik& p )
{
    p.postaviVisinu( 3 );
    p.postaviSirinu( 5 );
    cout << "Povrsina je: " << p.povrsina() << endl;
}
```

Ako bismo ga primenili na objekat klase `Pravougaonik`, rezultat metoda `povrsina` bi bio u skladu sa očekivanjem i ispisala bi se površina 15. Međutim, ako bismo ga primenili na objekat klase `Kvadrat`, onda bismo dobili da je površina 25, zato što metod `postaviSirinu` menja i širinu i visinu kvadrata. Iz ugla kvadrata, to je u redu, ali iz ugla principa zamenljivosti moramo da primetimo da se naš primer programskog koda ponaša različito za kvadrat i pravougaonik, te da imamo slučaj narušavanja ovog principa.

Kao što je već ranije naglašeno, ne postoji savršeno rešenje ovog problema. Štaviše, na osnovu ovakvih problema možemo da zaključimo da ispravnost modela ne sme da se proverava izolovano, posmatranjem pojedinačnih klasa, već samo globalno, posmatranjem čitavog programa. U konkretnom slučaju je uzrok problema u tome što je klasa `Kvadrat` napisana tako da narušava pretpostavke o ponašanju metoda klase `Pravougaonik`. To što te pretpostavke nisu eksplicitno iskazane (u specifikacij problema ili dokumentaciji koja prati implementaciju) nimalo ne umanjuje problem. Naprotiv, neko će takve implicitne pretpostavke da podrazumeva i uzima ih u obzir, a neko neće, što stvara idealno okruženje za nastajanje veoma nezgodnih bagova.

Da bi bio poštovan princip zamenljivosti, neophodno je da svaki put kada u izvedenoj klasi predefinišemo neki metod bazne klase, pri tome važi da su (1) preduslovi tog metoda u izvedenoj klasi isti ili blaži nego u baznoj klasi, a da su (2) postuslovi tog metoda u izvedenoj klasi isti ili jači nego u baznoj klasi.

U konkretnom primeru, postuslov metoda `postaviSirinu` za `Pravougaonik` je da će biti postavljena data širina ali i da će biti očuvana visina, dok je postuslov za `Kvadrat` blaži, zato što se neće očuvati visina. Zbog toga je narušen princip zamenljivosti.

Da bismo izbegli narušavanje principa zamenljivosti, možemo da probamo da definišemo semantiku problematičnih metoda tako da u postuslovima ne pretpostavljamo očuvanje drugih podataka. U konkretnom primeru, mogli bismo da u baznoj klasi `Pravougaonik` definišemo semantiku metoda `postaviSirinu` i `postaviVisinu` (i dokumentujemo je) tako da se nakon primene jednog od njih ne očekuje da je i dalje na snazi rezultat eventualne ranije primene drugog metoda. Međutim, na osnovu tako definisane semantike ovih metoda, mogli bismo da zaključimo da je ponašanje funkcije `primer` nedefinisano i da ne može da se predvidi šta će ta funkcija da ispiše. Samim tim, ako je ponašanje nepredvidivo, onda ne možemo da kažemo ni da je izmenjeno u slučaju kvadrata, zato što je i dalje nepredvidivo. Iako može da izgleda da smo ovakvim rezonovanjem rešili problem, ono nam zapravo nije donelo praktičnu korist – i dalje ne možemo da napišemo funkciju `primer` tako da budemo sigurni da će da radi ono što nam je potrebno.

Moguće rešenje ovog problema je da značajnije izmenimo interfejs, tako da odgovarajući preduslovi i postuslovi mogu da se ispravno definišu. U našem primeru to bi moglo da se uradi tako da se umesto dva metoda `postaviSirinu` i `postaviVisinu` definiše samo jedan metod `postaviSirinuIVisinu`, koji menja i širinu i visinu, a izbacuje izuzetak ako dati argumenti nisu ispravni. U tom slučaju verzija za pravougaonik bi mogla da proverava da li su vrednosti širine i visine pozitivne i da izbacuje izuzetak ako nisu, a postavlja nove vrednosti ako jesu. Odgovarajuća verzija za kvadrat bi mogla da se definiše tako da proverava da li su date vrednosti visine i širine jednake i da izbacuje izuzetak ako nisu, a inače poziva odgovarajući metod pravougaonika. Ovakvo rešenje je daleko od idealnog, ali je verovatno najbolje što može da se uradi u ovakvom slučaju – poštuje se princip zamenljivosti, ali se plaća cena u vidu smanjivanja funkcionalnosti kroz sužavanje interfejsa.

Princip razdvajanja interfejsa (ISP)

„Klijenti (korisnici) ne bi trebalo da budu primorani da zavise od metoda (interfejsa) koje ne koriste."

Princip razdvajanja interfejsa (engl. *ISP – The Interface Segregation Principle*) pod *klijentima* podrazumeva strukturne elemente programa koji koriste neke interfejse, tako da se umesto *klijenti* u formulaciji ovog principa može naići i na *korisnici*. Slično, umesto *metoda* može se naići i na *interfejse*.

Pod *razdvajanjem interfejsa* se podrazumeva da složene interfejse, koji služe za obavljanje više poslova, valja podeliti na više manjih interfejsa. Ovaj princip može da

se naruši na različite načine, od kojih su najčešći (1) pravljenje klasa ili hijerarhija sa višestrukim odgovornostima i (2) kada celoj hijerarhiji dodajemo nove elemente interfejsa zato što su potrebni samo jednom delu te hijerarhije (tj. klasama u jednoj njenoj grani). Narušavanje ovog principa obično ukazuje na loše raspodeljene odgovornosti pojedinačnih klasa ili preplitanje odgovornosti između delova hijerarhije ili čak više različitih hijerarhija klasa.

U prvom slučaju nije uvek jednostavno prepoznati da li je složen interfejs posledica višestrukih odgovornosti ili je obrnuto, pa to zahteva pažljivu analizu. Međutim, rešenje je relativno jednostavno i očigledno – potrebno je podeliti klasu ili komponentu na više manjih, tako da svaka ima jasno definisane odgovornosti, a time i jasno definisan i sužen interfejs. Primetimo da taj slučaj predstavlja neposredno narušavanje principa jedinstvene odgovornosti.

Drugi slučaj je nešto nezgodniji, između ostalog i zato što imamo na raspolaganju više načina rešavanja. Najjednostavniji način rešavanja je ako možemo da metode koji su potrebni u jednoj grani hijerarhije sklonimo iz bazne klase i interfejsa cele hijerarhije i premestimo ih u baznu klasu grane hijerarhije u kojoj su potrebni. Takvo rešenje je relativno jednostavno i čisto, ali je problem što nije uvek ostvarivo ili preporučljivo. Na primer, ako se svi objekti hijerarhije koriste samo putem referenci ili pokazivača na baznu klasu, onda će biti potrebno da u programskom kodu, koji koristi problematične metode, proveravamo da li objekat koji koristimo pripada odgovarajućoj grani hijerarhije ili ne, što zahteva ili dinamičke provere tipova ili dodavanje novih metoda (opet u baznu klasu cele hijerarhije) koji omogućavaju takve provere i odgovarajuće konverzije tipova.

Drugi način rešavanja je da se problematični metodi izdvoje u posebnu klasu, koja će da predstavlja adapter do klasa one grane hijerarhije u kojoj su potrebni. Na taj način se deo ponašanja izmešta iz hijerarhije u pomoćne klase adaptere, ali i dalje deo problema mora da se rešava slično prvoj varijanti, zato što adapter mora da ima sredstvo komunikacije samo sa objektima koji su u jednog grani hijerarhije a ne u celoj hijerarhiji – tj. ponovo moramo da dodajemo neke metode u samo jednu granu hijerarhije (makar to bilo i trivijalno).

Treći način rešavanja je moguć ako problematični metodi mogu da se grupišu u novoj hijerarhiji klasa i da se zatim primeni višestruko nasleđivanje. To nije moguće u svim OO programskim jezicima, a u nekim je moguće samo ako se nova hijerarhija započne interfejsom a ne pravom baznom klasom.

Vratimo se ponovo na primer sa apstraktnim drvetom izraza. Ako bismo implementirali neka od ranije pominjanih dodatnih ponašanja (prevođenje, izračunavanje, ispisivanje i slično) u svim klasama hijerarhije izraza, onda bismo narušili i princip razdvajanja interfejsa, zato što bismo proširili interfejs tih klasa ponašanjima koja većini korisnika nisu potrebna. Sa druge strane, implementacija pomoću obrasca Posetilac je u skladu sa ovim principom, zato što i klase hijerarhije i posetioci

zadržavaju čist i relativno jednostavan interfejs, koji će biti potreban svakome kome je potrebna neka od tih klasa.

Pored navedenih oblika narušavanja ovog principa, postoji i jedan oblik koji se često toleriše – pravljenje širokog interfejsa klase koja predstavlja sastavni deo neke biblioteke. Dobar primer je klasa standardne biblioteke `std::string`. Ova klasa ima veoma širok interfejs, koji čine različite operacije nad niskama. Ako bismo analizirali interfejs ove klase, mogli bismo da primetimo da je samo manji broj metoda neophodan za njeno funkcionisanje, a da ostali metodi predstavljaju dodatne operacije, koje su mogle da se implementiraju i kao skup funkcija koje za argumente imaju objekte klase `string`. Suštinu funkcionisanja klase `string` obezbeđuju metodi koji omogućavaju pristupanje sadržaju niske i njeno menjanje. Sve ostale operacije (traženje ili izdvajanje podniske, čitanje niske iz toka i slično) bi trebalo da se implementiraju kao funkcije, a ne kao metodi ove klase. Analogno, možemo da razmotrimo pisanje hipotetičke klase `Float` – da li bi operacije kao `sqrt`, `sin`, `exp`, `log` i druge trebalo da budu metodi klase ili dodatne funkcije? Slično je stanje sa raznim bibliotekama za crtanje ili za rad sa slikama, gde se na osnovnim klasama isporučuje relativno bogat interfejs, koji tek malobrojni korisnici u celosti upotrebljavaju – većini je dovoljan samo manji deo. Ako je biblioteka relativno stabilna, onda takav pristup ne predstavlja problem, ali ako nije, onda se otvara prostor za brojne izmene u kodu koji koristi takve bibliotečke klase.

Princip inverzne zavisnosti (DIP)

„Moduli visokog nivoa ne smeju da zavise od modula niskog nivoa. I jedni i drugi bi trebalo da zavise od apstrakcija."

Princip inverzne zavisnosti (engl. *DIP – The Dependency-Inversion Principle*) ima i alternativne oblike:

- Apstrakcije ne smeju da zavise od pojedinosti. Pojedinosti bi trebalo da zavise od apstrakcija.
- Programirati prema interfejsima a ne prema implementacijama.

Često ćemo se susretati sa pojmom *smer zavisnosti*. Podrazumevaćemo da smer zavisnosti ide od zavisnog elementa prema elementu od koga on zavisi. Na primer, ako je smer zavisnosti od A prema B, to znači da A zavisi od B.

Svaka zavisnost komponenti predstavlja istovremeno i osu promenljivosti, ali u suprotnom smeru – ako komponenta A zavisi od komponente B, onda ćemo imati osu promenljivosti koja će potencijalno prenositi promene sa komponente B na komponentu A.

Pri uvođenju u osnovne pojmove projektovanja naglasili smo važnost apstrahovanja u postupku projektovanja. Istakli smo da se apstrahovanje vrši

posmatranjem pojedinačnih slučajeva i oblikovanjem karakteristika opšteg rešenja na osnovu uočenih zajedničkih karakteristika tih pojedinačnih slučajeva. Otuda navedena formulacija principa inverzne zavisnosti može da nam izgleda suprotstavljeno postupku apstrahovanja, zato što sa jedne strane apstrahujemo na osnovu pojedinačnih slučajeva, pa samim tim naše apstrakcije na određeni način moraju da zavise od pojedinačnih slučajeva, a sa druge strane zahtevamo da konkretni slučajevi zavise od apstrakcija. Međutim, tu se radi o potpuno različitim vrstama zavisnosti.

Pri apstrahovanju i dizajniranju strukture softvera analiziramo specifične slučajeve i na osnovu njih (tj. *zavisno od njih*) definišemo apstrakcije, ali kada ih jedanput definišemo, onda softver implementiramo tako da zavisnost ide u suprotnom smeru – specifične izvedene klase se implementiraju tako da zavise od apstraktne bazne klase. Otuda i potiče naziv ovog principa – *inverzna zavisnost*. Primetimo da se u nekim starijim metodologijama zaista težilo tome da zavisnosti idu u smeru od opštih rešenja prema pojedinačnim slučajevima. Međutim, danas je drugačije – agilne i OO metodologije zahtevaju upravo suprotno. I ta promena pristupa se ističe nazivom ovog principa.

Kao ilustraciju primene ovog principa ćemo da iskoristimo slučaj pravljenja izveštaja u različitim formatima (na primer: neformatirani tekst, HTML i TEX). Neka klasa `Podaci` sadrži podatke na osnovu kojih je potrebno da se napravi izveštaj i neka je klasa `Izvestavac` odgovorna za njegovo pravljenje. Pošto se izveštaji prave u različitim formatima, mogli bismo da za svaki od formata napravimo poseban metod, ali to nije dobra ideja zato što bi bilo mnogo ponavljanja – bez obzira na zahtevani format, izveštaj se pravi na osnovu istih podataka, koji se dohvataju na isti način, a čak je i struktura izveštaja uglavnom ista. Štaviše, interfejs bi mogao da ima različite metode za svaki od formata, ali bi ti metodi verovatno svi pozivali jedan isti metod sa različitim vrednostima parametara, koji bi opisivali kako se pravi izveštaj:

```
class Izvestavac {
...
    string napraviHtmlIzvestaj( const Podaci& p ){
        return napraviIzvestaj( p, "html" );
    }
    string napraviTxtIzvestaj( const Podaci& p ){
        return napraviIzvestaj( p, "txt" );
    }
    string napraviTexIzvestaj( const Podaci& p ){
        return napraviIzvestaj( p, "tex" );
    }
    string napraviIzvestaj( const Podaci& p, const string& format ){
        ...
    }
...
};
```

Neposredna (ili *neinverzna*) zavisnost bi bila primenjena kada bi metod `napraviIzvestaj`, na svakom mestu gde je potrebno različito ponašanje, proveravao koji se format zahteva i zatim radio odgovarajuću stvar, na primer:

```
...
string naslov = p.naslov();
if( format == "html" )
    izvestaj += htmlNaslov( naslov );
else
    ...
...
```

To je klasičan pristup. On ima nekoliko loših strana, od kojih su nam najvažnije dve: (1) pri pisanju metoda `napraviIzvestaj` moramo da raspolažemo znanjem o svim pojedinostima i specifičnostima implementacije za svaki od potrebnih formata i (2) ako dođe do promene nekog od postojećih formata, ili se doda neki novi format, onda ćemo morati da menjamo implementaciju metoda `napraviIzvestaj`. To je posledica uspostavljenih zavisnosti celine višeg nivoa (naš metod `napraviIzvestaj`) od celina nižeg nivoa (različiti formati izveštaja).

Ideja principa inverzne zavisnosti je da se ukloni zavisnost celine višeg nivoa od celina nižeg nivoa. Obično se primenjuje tako što se uvodi dodatna apstrakcija između celine višeg nivoa i upotrebljavanih celina nižeg nivoa i zatim se sve celine (i višeg i nižeg nivoa) preoblikuju tako da zavise od te nove apstrakcije. Ta apstrakcija najčešće predstavlja interfejs celina nižeg nivoa, odnosno baznu klasu nove hijerarhije.

U konkretnom primeru bismo uveli apstraktnu klasu `Format` koja definiše interfejs koji mora da zadovolji svaki konkretan format, na primer:

```
class Format {
...
    virtual string naslov( const string& n ) = 0;
...
};
```

kao i odgovarajuće konkretne klase formata:

```
class HtmlFormat : public Format {
...
    string naslov( const string& n ) override;
    ...
    };
...
```

posle čega bismo umesto oznake formata, metodu `napraviIzvestaj` predavali objekat koji predstavlja odgovarajući format. U implementaciji metoda bismo koristili interfejs tog objekta:

```
string napraviIzvestaj( const Podaci& p, const Format& format ){
    ...
    izvestaj += format.naslov( p.naslov );
    ...
}
```

Ostaje još da proverimo da li dobijeni programski kod poštuje princip inverzne zavisnosti. Najpre prepoznajmo zavisnosti u našem primeru programa:

- metod `napraviIzvestaj` zavisi od apstraktne klase `Format`;
- klase konkretnih formata (kao `HtmlFormat`) zavise od apstraktne klase `Format`;

a zatim procenimo u kojoj su meri apstrahovani različiti elementi koji učestvuju u prepoznatim zavisnostima:

- najapstraktniji deo programa je klasa `Format`;
- metod `napraviIzvestaj` je nešto konkretniji, ali je i dalje relativno apstraktan;
- najkonkretniji elementi našeg rešenja su konkretne klase formata, poput `HtmlFormat`.

Zaista, sve zavisnosti idu u smeru od konkretnijih elemenata prema apstraktnijim elementima, što znači da naš primer sada poštuje princip inverzne zavisnosti.

Primetimo da ovakav pristup uvodi nove elemente programa (dodaje se nov apstraktni interfejs, ponašanje za različite formate se grupiše u novim klasama), što može da predstavlja faktor povećavanja i usložnjavanja programa, ali je daleko veća dobit koju ostvarujemo ugradnjom čistijih zavisnosti, a time i olakšanog razumevanja i održavanja programa.

Alternativa uvođenju novog interfejsa je primena parametarskog polimorfizma, u kom slučaju bismo metod `napraviIzvestaj` mogli da napišemo kao šablonski metod, gde je parametar šablona objekat za formatiranje. U konkretnom slučaju ovakav pristup nema neke značajne prednosti, zato što svejedno moramo da definišemo interfejs koji svi formati moraju da zadovolje. Štaviše, zbog formalnog definisanja interfejsa u vidu apstraktne klase, prvo rešenje je pouzdanije i bolje. Sličnu ulogu bi mogli da odigraju koncepti u novijim verzijama programskog jezika C++. U nekim drugim slučajevima, a posebno ako nije moguće ili poželjno sve te

konkretne objekte stavljati u istu hijerarhiju (a to je u praksi relativno čest slučaj, na primer ako oni već pripadaju različitim hijerarhijama), onda upotreba parametarskog polimorfizma može da bude bolje rešenje.

6.3 Principi dodeljivanja odgovornosti

Kreg Larman je u svojoj knjizi [*Larman 2002*][25] predstavio skup principa projektovanja, koje je oblikovao prvenstveno radi rešavanja problema raspoređivanja (dodeljivanja) odgovornosti. Predstavljen skup principa je nazvao *opštim softverskim obrascima dodeljivanja odgovornosti* (engl. *GRASP – General Responsibility Assignment Software Patterns*) i opisao ih je nalik na obrasce za projektovanje, da bi oni mogli da se lakše koriste pri učenju i svakodnevnom projektovanju softvera. Iako se prvenstveno odnose na OO metodologije, ovi principi uglavnom mogu da se primene i na druge metodologije. Većina ovih principa nam pomaže da oblikujemo strukturu programa tako da se smanji broj i složenost zavisnosti između različitih delova programa, što nam zatim omogućava da lakše pišemo i održavamo program.

Kroz čitav proces projektovanja softvera suočavamo se sa pitanjima zavisnosti i odgovornosti. Ako pogledamo predstavljene ključne principe OO dizajna, videćemo da su i svi ti principi povezani sa pitanjima zavisnosti i odgovornosti. Mogli bismo reći i da se neki principi dodeljivanja odgovornosti mogu izvesti iz principa OO dizajna, ali i obrnuto. U svakom slučaju, zbog drugačijeg fokusa, predstavićemo sve ove principe projektovanja.

Pošto su svi principi *GRASP* tesno povezani sa problemom odgovornosti, podsetićemo se da odgovornosti elemenata strukture programa određuju šta taj element programa *zna* i šta *ume da uradi*. Znanje se odnosi prvenstveno na enkapsulirani sadržaj konkretnog elementa ali i na poznavanje drugih povezanih elemenata programa (putem referenci ili pokazivača) i njihovih interfejsa, dok se *umeće* odnosi na sopstveno ponašanje i sposobnost upravljanja radom drugih elemenata programa.

Informacioni ekspert

„Odgovornost za obavljanje posla se dodeljuje klasi koja ima informacije neophodne za obavljanje tog posla.“

Princip Informacioni ekspert (engl. *Information Expert*)[26] nam sugeriše da bi bilo najbolje da ponašanje (tj. metodi koji obavljaju neki posao) i odgovarajuće znanje (tj. informacije potrebne za obavljanje tog posla) budu locirani na istom mestu. To je u

[25] Prvo izdanje je iz 1997. godine.

[26] U nekim izvorima se naziva samo kratko Ekspert.

skladu sa težnjom da u programu imamo što manje zavisnosti između različitih strukturnih elemenata programa.

Ako, na primer, jedna klasa sadrži informacije, a druga obavlja posao za koji su te informacije neophodne, onda se između ovih klasa mora uspostaviti komunikacija, a time i zavisnost – klasa koja obavlja posao će zavisiti od klase koja sadrži potrebne informacije. Sa druge strane, ako su i informacije i obavljanje posla locirani u istoj klasi, onda ne moramo da uspostavljamo ni dodatnu komunikaciju ni dodatnu zavisnost, pa je dizajn čistiji.

Ako pri rešavanju nekog problema ne poštujemo ovaj princip, vrlo je verovatno da će nam se u programu pojaviti neke vrste kohezije koje ne spadaju u poželjnije, kao i neki relativno nepoželjni nivoi spregnutosti. Na primer, ako podelimo znanje i ponašanje na različite klase jedne komponente, onda ćemo verovatno dobiti komunikacionu koheziju između delova komponente. Sa druge strane, ako posmatramo problem na nivou klasa te komponente, onda ćemo među njima verovatno imati spregnutost po sadržaju, preko zajedničkih delova, spoljašnju spregnutost ili spregnutost preko kontrole – što su sve nepoželjniji nivou spregnutosti. Dodatni problem može da nastane ako više klasa počne da koristi ili čak menja informacije iz jedne klase, zato što onda može da dođe i do mešanja odgovornosti – postavlja se pitanje ko je odgovoran za održavanje znanja, a ko za različite aspekte njegove upotrebe?

Primer narušavanja principa Informacioni ekspert je naveden u odeljku *10.4 Primer refaktorisanja*, na stranici 260. U tom primeru klasa `Pitanje` sadrži podatke, ali je kompletno ponašanje implementirano u klasi `Test`, koja intenzivno koristi objekte klase `Pitanje`. Refaktorisanjem se ponašanje postepeno premešta u klasu u kojoj se nalaze podaci i dizajn se popravljanja tako da bude u skladu sa ovim principom.

U nekim slučajevima, princip Informacioni ekspert može da se sukobi sa Principom razdvajanja interfejsa. Na primer, videli smo da dizajn klase `std::string` predstavlja vid narušavanja Principa razdvajanja interfejsa. Sa druge strane, takav dizajn je u skladu sa principom Informacioni ekspert, zato što sve informacije za obavljanje složenijih operacija nad niskama (pretraživanje i zamenjivanje podniski i slično) postoje u samim niskama. U ovom slučaju je principu Informacioni ekspert data veća težina, ali to nije uvek dobar izbor i trebalo bi da se razmatra od slučaja do slučaja.

Stvaralac

„Odgovornost za pravljenje objekta klase A se dodeljuje klasi B ako važi bar jedno od:

- **instanca klase B predstavlja kompoziciju instanci klase A;**
- **instanca klase B referiše instance klase A;**
- **instanca klase B blisko koristi instance klase A;**

- **instanca klasa B ima informacije za inicijalizaciju instanci klase A i prenosi ih pri pravljenju.“**

Kao i u slučaju principa Informacioni ekspert, i ovde je primarni motiv težnja da se smanji broj zavisnosti. Osnovna ideja principa Stvaralac (engl. *Creator*) je da se odgovornost za pravljenje instanci neke klase uspostavi na mestu na kome se ta klasa već upotrebljava i na kome već postoji izražena i jasna zavisnost od te klase. Na taj način se ne dodaje nova zavisnost, mada može doći do proširenja već postojeće zavisnosti. U opisu principa su navedena četiri takva slučaja, kada postojeća zavisnost klase B od klase A predstavlja dovoljno dobar razlog da se klasi B dodeli i odgovornost za pravljenje instanci klase A.

Ovaj princip predstavlja primenu dekomponovanja prema promenljivosti. Ako je već neophodno da izaberemo neku klasu X, koja će da pravi instance klase A, onda je potrebno da uočimo da se time uspostavlja relativno jaka zavisnost klase X od klase A. Ta zavisnost predstavlja jednu novu osu promenljivosti. Ako već postoje neke druge zavisnosti neke klase B od klase A, kao i odgovarajuće ose promenljivosti, onda je poželjno da pokušamo da grupišemo veći broj osa promenljivosti tako da idu od A prema istoj klasi. Zato je dobro da za X biramo baš klasu B.

Kao primer primene ovog principa može da nam posluži obrazac za projektovanje Apstraktna fabrika [*Gamma 1995*]. Kada je potrebno da se u zavisnosti od specifičnih okolnosti prave i koriste objekti različitih familija klasa, onda se odgovornost za pravljenje takvih objekata prepušta apstraktnoj fabrici, koja će na osnovu konteksta, argumenata i drugih raspoloživih informacija umeti da odabere odgovarajuću konkretnu klasu, čiji će objekat da napravi i vrati pozivaocu. Na primer, ako bi program podržavao više različitih vrsta korisničkih interfejsa, onda bi klasa `FabrikaInterfejsa` mogla da ima metode `napraviProzor`, `napraviDugme` i druge, a koji bi u zavisnosti od konteksta pravili objekat koji odgovara potrebnoj vrsti interfejsa. Gde god da u programu zatreba novi prozor ili novo dugme, takvi objekti se ne bi pravili neposredno već korišćenjem usluga apstraktne fabrike. Ovaj primer predstavlja ilustraciju poslednjeg od četiri slučaja previđenih principom Stvaralac, zato što apstraktna fabrika ima na raspolaganju informacije na osnovu kojih može da odlučuje koje i kakve objekte pravi. Pri tome se neke dodatne informacije prosleđuju kao argumenti metodima apstrakte fabrike.

Visoka kohezija

„Odgovornosti se dodeljuju tako da kohezija ostane visoka.“

Kohezija (engl. *cohesion*) je stepen međusobne povezanosti elemenata koji čine jednu strukturnu celinu programa (klasu, modul, komponentu, funkciju i sl.). Zajedno sa spregnutošću, kohezija predstavlja jednu od najvažnijih karakteristika softverskih celina. Princip Visoka kohezija ističe značaj težnje da pišemo programe

tako da u svakoj strukturnoj celini programa kohezija bude visoka. Drugim rečima, svaka celina bi trebalo da predstavlja skup čvrsto međusobno povezanih elemenata.

O tome je već bilo reči ranije (*4.7 – Kohezija i spregnutost*), pa nećemo da ponavljamo već napisano.

Niska spregnutost

„Odgovornosti se dodeljuju tako da spregnutost ostane niska."

Spregnutost (engl. *coupling*) je stepen međusobne povezanosti elemenata koji pripadaju različitim strukturnim celinama programa. Zajedno sa kohezijom predstavlja jednu od najvažnijih karakteristika softverskih celina. Ovaj princip nam sugeriše da bi trebalo da pišemo programe tako da spregnutost bude što je moguće niža, tj. da različiti strukturni elementi programa budu što manje međuzavisni. To praktično znači da želimo da imamo što manje veza između različitih strukturnih elemenata i da pri tome težimo da te veze budu što je moguće slabije.

Kao i koheziju, i spregnutost smo već predstavili u prethodnim poglavljima (*4.7 – Kohezija i spregnutost*).

Kontroler

„Odgovornosti za primanje ili obrađivanje poruka o sistemskim događajima se dodeljuju klasi koja:

- **predstavlja ceo sistem, uređaj ili podsistem (tzv. fasadni kontroler), ili**
- **predstavlja scenario slučaja upotrebe u kome se pojavljuju sistemski događaji (i pri tome obično nosi naziv poput `NazivSlučaja_Rukovalac`, `NazivSlučaja_Koordinator`, `NazivSlučaja_Sesija` i slično, tj. predstavlja kontroler sesije ili slučaja upotrebe)."**

U kontekstu principa Kontroler (engl. *Controller*), pod sistemskim događajima se podrazumevaju sve ulazne ili izlazne operacije koje nisu neposredan proizvod rada programa koji pišemo. Obično se tu radi o akcijama korisnika (putem korisničkog interfejsa), uređaja (putem različitih API-ja) ili drugih povezanih programa (na primer, dobijanje el.pošte i slično).

Osnovna ideja je da je reagovanje na takve akcije poželjno grupisati na mestu odakle se upravlja aktivnostima i komunikacijom u konkretnom slučaju upotrebe. Obično je dobro da se na sve događaje u jednom slučaju upotrebe (ili jednoj zaokruženoj celovitoj komunikacionoj sesiji) reaguje na istom mestu, kako bi se sa tog mesta vršila kontrola izvršavanja odgovarajućih operacija, a koje su zbog prirode oblikovanja slučaja upotrebe obično već relativno čvrsto povezane.

Na primer, ako slučajem upotrebe upravlja korisnik, onda akcija korisnika (tj. korisničko upravljanje slučajem) mora nekako da se prenese na programsko

upravljanje tokom izvršavanja implementiranog slučaja upotrebe. Obično nećemo da pravimo klasu koja predstavlja korisnika, ali ćemo da napravimo klasu koja će biti odgovorna da prihvata informacije o događajima koje korisnik proizvodi putem korisničkog interfejsa i da reaguje na njih pokretanjem odgovarajućih funkcija.

Eventualno, ako ima manje slučajeva upotrebe u kojima se obrađuju neke klase događaja, onda takva komunikacija može da bude dodatno grupisana u tzv. fasadni kontroler, koji upravlja reagovanjem na odgovarajuće klase događaja za više različitih slučajeva upotrebe ili čak za sve slučajeve upotrebe u komponenti ili programu.

Pojam kontrolera je prisutan u različitim modelima rešavanja problema korisničkog interfejsa, pa i u arhitekturama „Model-pogled-kontroler" (engl. *„Model-View-Controller" – MVC*), „Prezentacija-apstrakcija-kontroler" (engl. *„Presentation-Abstraction-Controller" – PAC*) i nekim drugim. Većina okruženja za razvoj korisničkog interfejsa počiva na nekom vidu kontrolera.

Na primer, u okruženju *Qt* [*Qt*] se prilično široko primenjuje model programiranja vođenog događajima, tako što se reagovanje na različite vrste događaja izvodi proizvođenjem *signala*, koji se zatim prosleđuju do tzv. *slotova*, koji predstavljaju mesto reagovanja na signale i mesto obrade događaja. Slotovi se uobičajeno grupišu u okviru klasa ili objekata koji imaju ulogu kontrolera. Značajno je da *Qt* omogućava dinamičko povezivanje signala i slotova, što omogućava da se povezivanje ne izvodi na nivou klasa nego na nivou konkretnih objekata, pa tako različiti objekti iste klase (na primer nekog apstrahovanog kontrolera) mogu da obavljaju poslove kontrolera za različite formulare ili za različite slučajeve upotrebe. U okruženju *Qt* sistem signala i slotova se ne koristi samo za reagovanje na događaje, koji nastaju u okviru korisničkog interfejsa, već signali mogu da se proizvode i pri promenama stanja određenih objekata u programu, a zatim na te promene može da se reaguje na proizvoljno mnogo drugih mesta, preko povezanih slotova.

Polimorfizam

„Kada se neka vrsta ponašanja menja prema tipovima, onda se odgovornosti za odgovarajuće ponašanje raspoređuju po tim tipovima, primenom polimorfizma."

Ovaj princip ističe osnovnu ulogu polimorfizma – da omogući da apstraktan kod funkcioniše na konkretnim objektima različitih tipova, tako što će oni aspekti ponašanja koji se za neke tipove razlikuju imati iste interfejse ali različite implementacije. Iako je ideja polimorfizma prvenstveno oblikovana za OO hijerarhije tipova, ona se na identičan način primenjuje i na druge vrste polimorfizma (*11 - Polimorfizam*, str. 299), tako da je ovaj princip u tom smislu opšteg karaktera i nije vezan samo za OO metodologije i programske jezike.

Osnovna motivacija za primenu ovog principa je u tome da se izbegne uvođenje dodatnih zavisnosti od različitih konkretnih tipova, na onim mestima gde se koriste

oni aspekti njihovog ponašanja koji se razlikuju od tipa do tipa. Umesto toga se uvodi samo zavisnost od njihove zajedničke (uopštene) apstrakcije, tj. od apstraktnog interfejsa. U tom pogledu princip Polimorfizam je jedna varijacija već predstavljenog Principa inverzne zavisnosti.

Tipičan primer primene ovog principa je refaktorisanje „Zamena uslova polimorfizmom" [*Fowler 1999*]. Ako na nekom mestu u programu imamo višestruko grananje (na primer naredbu `switch`), koje zavisi od nekog parametra ili neke oznake vrste slučaja, onda je poželjno uvesti hijerarhiju klasa, takvu da svakom posebnom slučaju odgovara po jedna konkretna klasa, a zatim se grananje zamenjuje pozivanjem metoda, koji će u zavisnosti od stvarne klase objekta (a ne u zavisnosti od nekog eksplicitnog parametra) da uradi odgovarajući posao.

Izmišljotina

„Tesno povezan i zaokružen skup odgovornosti se dodeljuje veštački uvedenoj klasi, koja ne predstavlja koncept iz domena problema – koja je izmišljena da bi omogućila visoku koheziju, nisku spregnutost i višestruku upotrebu."

Princip Izmišljotina (engl. *Fabrication*)[27] sugeriše da u nekim slučajevima projektant ne mora da tačno sledi strukture iz domena koji se modelira, već može da *izmišlja* nove koncepte, kako bi lakše i bolje modelirao implementaciju. Izmišljanje novih koncepata se obično preporučuje u slučajevima kada dosledno praćenje postojećih elemenata i koncepata iz prostora domena ima za rezultat problematične zavisnosti, koje se obično iskazuju u obliku niske kohezije, visoke spregnutosti ili dvosmernih ili cirkularnih zavisnosti. Može da bude od koristi i u slučajevima kada dosledno praćenje domena dovodi do pravljenja velikog broja jednostavnih klasa ili raspoređivanja višestrukih odgovornosti u jednu klasu ili komponentu.

Rezultat izmišljanja je obično nova klasa, koja ne odgovara neposredno nijednom strukturnom elementu domena, ali obuhvata aspekte ponašanja koji su relativno čvrsto međusobno povezani, a relativno slabo povezani sa drugim konceptima i strukturama.

Princip Izmišljotina stoji iza većeg broja obrazaca za projektovanje (*7 - Obrasci za projektovanje*, str. 137). Neki od jednostavnijih primera izmišljotina su obrasci za projektovanje Fasada ili Adapter. U slučaju obrasca Fasada, ideja je da se složenost nekog sistema zakloni objektom koji predstavlja fasadu i koji stoji između sistema i njegovih korisnika. Obrazac Adapter ne zaklanja neki veći sistem već jednu klasu, odnosno jedan objekat – on prilagođava interfejs neke klase specifičnom kontekstu u kome je potrebno da se ona koristi. U ovim slučajevima rezultat izmišljanja nije neki

[27] U nekim izvorima se sreće pod imenom Čista izmišljotina (engl. *Pure Fabrication*).

novi složeni koncept već samo novi vid enkapsuliranja složenog ili drugačijeg ponašanja u neke razumljivije okvire (interfejse).

Nešto složeniji vidovi izmišljanja se mogu prepoznati u obrascima kao što su Posetilac ili Dekorater. Obrazac Posetilac počiva na potpuno veštački izgrađenom konceptu posetioca, koji omogućava da se različite odgovornosti izmeste iz neke hijerarhije klasa u posebne klase – posetioce. Obrazac Dekorater omogućava da se složene hijerarhije klasa (sa različitim faktorima specijalizovanja i potencijalno višestrukim nasleđivanjem) značajno pojednostave uvođenjem potpuno novog koncepta dekoracija i zamenjivanjem izvedenih klasa objektima sa dekoracijama – dodacima koji donose novo ili menjaju postojeće ponašanje.

Indirekcija

„Odgovornosti se dodeljuju objektu koji je posrednik između drugih komponenti ili servisa, tako da oni ne moraju da budu neposredno spregnuti."

Ideja principa Indirekcija (engl. *Indirection*) je da se veći broj međusobnih zavisnosti između nekih elemenata softvera zameni uređenijim skupom zavisnosti između tih komponenti i jednog objekta (klase) koji ima ulogu posrednika. Potencijalan problem sa međusobnim zavisnostima skupa komponenti je da one često mogu da postanu dvosmerne ili cirkularne, što može da prilično poveća spregnutost među njima i zakomplikuje i pisanje programa i njegovo održavanje. Uvođenjem posrednika se teži pojednostavljenju i koncentrisanju tih odnosa, tako da nikoje dve komponente iz posmatranog skupa više ne budu neposredno međusobno zavisne, već da se svaka komunikacija između komponenti obavlja isključivo preko posrednika.

Obično se teži da se zavisnosti posrednika urede tako da druge komponente zavise samo od njegovog interfejsa, a da implementacija posrednika zavisi od interfejsa komponenti. Na taj način se postiže da se u slučaju promena interfejsa neke od komponenti te promene propagiraju najdalje na implementaciju metoda posrednika, ali da obično ne zahtevaju izmene u drugim komponentama posmatranog skupa.

Različiti obrasci za projektovanje (*7 - Obrasci za projektovanje*, str. 137) primenjuju princip Indirekcija u različitim kontekstima i na različite načine. Na primer, obrazac Apstraktna fabrika predstavlja jedan vid posrednika koji, od onoga kome je potreban nov objekat neke klase, sakriva kako se prave novi objekti, a u mnogim slučajevima čak i konkretan tip objekta koji se pravi. Sličnu ulogu imaju i drugi gradivni obrasci. Sa druge strane, strukturni obrasci poput Adaptera, Mosta i Fasade sakrivaju od korisnika specifičnosti implementacije objekata koji se koriste. Obrasci ponašanja uglavnom teže da sakriju složenost nekog ponašanja uvođenjem nekog vida indirekcije.

Izolovane promenljivosti

„Odgovornosti se dodeljuju tako da se oblikuje stabilan interfejs oko prepoznatih tačaka predvidive promenljivosti ili nestabilnosti.“

Princip Izolovanih (ili zaštićenih) promenljivosti (engl. *Protected Variations*) ukazuje nam da bi sve prepoznate elemente, za koje se zna ili se pretpostavlja da mogu da budu promenljivi u budućnosti, trebalo da izolujemo od okoline odgovarajućim apstraktnim interfejsom. U tom smislu ovaj princip ima sličnosti sa principom Indirekcija i Principom inverzne zavisnosti. I Princip zamenljivosti mu je veoma sličan po motivaciji i načinu implementacije – definišemo apstraktnu baznu klasu kao osnovu hijerarhije klasa, koja zapravo predstavlja univerzalni interfejs prema svim klasama hijerarhije, koje će zatim uvesti različite varijacije kroz različite implementacije odgovarajućih metoda.

Kao i princip Stvaralac, i ovaj princip predstavlja primenu dekomponovanja prema promenljivosti. Međutim, ovde je fokus na tačkama, a ne na osama promenljivosti. Princip nam sugeriše da bi tačke promenljivosti trebalo izolovati od ostatka softvera dovoljno uopštenim interfejsom, koji bi trebalo da eventualne promene uspešno „zadrži u lokalu“.

Ovaj princip predstavlja jedan od najvažnijih principa projektovanja softvera. Slično kao i principi Visoka kohezija i Niska spregnutost, i princip Izolovanih promenljivosti se odnosi na jedno opšte merilo dobrog dizajna. On je tesno povezan sa konceptima interfejsa i enkapsulacije, koji su među najvažnijim konceptima OO programiranja i OO metodologija. Tesno je povezan i sa funkcionalnom dekompozicijom sistema i sa apstrahovanjem (na primer generalizacija klasa). Većina obrazaca za projektovanje i refaktorisanja predstavlja vid primene ovog principa. Praktično svaki put kada pokušavamo da modeliramo neki deo strukture softvera, mi pri tome primenjujemo i ovaj princip – nekada u sklopu drugih, nešto konkretnijih principa, a nekada neposredno, kao jasno izraženo staranje o izolovanju tačaka promenljivosti.

6.4 Principi oblikovanja celina

Kao što prethodno opisani principi *GRASP* teže da opišu na koji način je potrebno da dodeljujemo odgovornosti strukturnim elementima programa (najčešće klasama), tako principi oblikovanja celina pokušavaju da ukažu na ispravne načine grupisanja strukturnih elemenata u složenije celine (na primer, kako se grupišu klase u pakete ili komponente). Principi iz ove grupe su tesno povezani sa ostvarivanjem kohezije i spregnutosti, kao i sa dekomponovanjem prema promenljivosti, pa se može reći i da predstavljaju dalje preciziranje prethodno opisanih principa *Kohezija*, *Spregnutost* i *Izolovane promenljivosti*. Mogu se naći u donekle različitim oblicima, a ovde ćemo ih predstaviti u obliku u kome ih je naveo Robert Martin [*Martin 2003*].

Principe oblikovanja celina možemo da podelimo na dve manje grupe, na *principe grupisanja* i *principe razdvajanja*. Principi grupisanja se nazivaju i *principima kohezije* komponenti, zato što se bave ostvarivanjem visoke kohezije. U njih spadaju:

- Princip ekvivalentnosti izdanja i ponovljive upotrebe;
- Princip zajedničke zatvorenosti i
- Princip zajedničke upotrebe.

Drugu grupu čine principi razdvajanja. Nazivaju se i *principima spregnutosti* komponenti, zato što je u njihovom fokusu ostvarivanje što niže spregnutosti. Principi razdvajanja obuhvataju:

- Princip stabilne zavisnosti;
- Princip stabilne apstrakcije i
- Princip acikličnih zavisnosti.

Princip ekvivalentnosti izdanja i ponovljive upotrebe (REP)

„Granula ponovljive upotrebe je granula objavljivanja."

U kontekstu ovog principa, *objavljivanje* se posmatra pre svega kao zvanično objavljivanje (bilo u okviru razvojnog tima ili šire) formalne specifikacije funkcionalnosti i interfejsa neke strukturne celine programa (komponente, paketa, klase i sl.).

Princip ekvivalentnosti izdanja i ponovljive upotrebe (engl. *REP – The Reuse-Release Equivalence Principle*) nam ukazuje na to da bi trebalo da postoji tesna veza između ponovljive upotrebe i objavljivanja i to u oba smera – (1) ako bi neka celina trebalo da se koristi na više mesta u programu, onda je neophodno da se u nekom trenutku objavi njena specifikacija (koja obuhvata detaljne opise funkcionalnosti i interfejsa), kako bi svi kojima je ona potrebna mogli da je koriste i (2) ako nešto objavljujemo, onda moramo da računamo s tim da će neko to da upotrebljava.

Zbog izvesne ili očekivane ponovljive upotrebe, svaka objavljena celina mora da ima što stabilniji interfejs i što bolju enkapsulaciju. Koliko god da je implementacija neke celine složena, ona bi morala da sakrije od korisnika sve što oni ne moraju da znaju (tj. detalje implementacije) i da im ponudi jasan i čist interfejs (tj. funkcionalnost). Princip *REP* promoviše funkcionalno grupisanje elemenata u celine, tj. *funkcionalnu koheziju*, kao jedan od najpoželjnijih vidova kohezije.

Ovaj princip implicira ne samo funkcionalnu već i *tesnu* povezanost između delova celine – ako je nešto sakriveno interfejsom celine, a pripada toj celini, onda je je to zato što celina bez toga ne može da funkcioniše. Neprihvatljiva alternativa je da smo greškom ostavili u celini nešto što se ne upotrebljava i što niko nikada neće ni moći da upotrebljava zato što nije dostupno kroz interfejs celine.

Princip *REP* i funkcionalna kohezija se primarno odnose na pravljenje celina koje predstavljaju funkcionalno zaokružene elemente strukture programa – klase i komponente. U praksi se često razmatra zajedno sa principom Izolovane promenljivosti, zato što težimo da funkcionalnu dekompoziciju kombinujemo sa dekompozicijom po promenljivosti.

Princip zajedničke upotrebe (CRP)

„Klase u paketu se koriste zajedno. Ako se koristi jedna od klasa u paketu, onda se koriste sve."

Princip zajedničke upotrebe (engl. *CRP – The Common-Reuse Principle*) se prvenstveno odnosi na logičku strukturnu dekompoziciju i pakete (za razliku od principa *REP*, koji se prvenstveno odnosi na funkcionalnu dekompoziciju i komponente).

Primetimo da imamo dva osnovna vida zajedničkog korišćenja klasa. Prvi je da su te klase međusobno funkcionalno povezane i da implementacije nekih od tih klasa zavise od interfejsa ili implementacije drugih klasa. Ako koristimo klase koje su međuzavisne, onda posredno koristimo i klase od kojih one zavise. Takva povezanost je zapravo već razmotrena u okviru principa *REP* i ovde nam nije od primarnog značaja.

Umesto toga, sada se fokusiramo na drugi vid zajedničkog korišćenja – na celine koje se uvek (ili bar vrlo često) zajedno koriste u spoljnom programskom kodu. Na primer, ako obrađujemo HTTP-zahteve, onda moramo da ih primimo, parsiramo, analiziramo, prosledimo odgovarajućim servisima, primimo odgovore, složimo odgovore u rezultate i isporučimo rezultate. Sve te celine posla su međusobno uglavnom nezavisne, ali ćemo najčešće da ih koristimo zajedno – ako nam je potrebna neka od njih, vrlo verovatno su nam potrebne i ostale. Rezultat jedne operacije predstavlja ulaz u drugu operaciju – to je tipičan primer sekvencijalne kohezije.

Kao drugi primer možemo da iskoristimo apstraktno drvo izraza. U skladu sa Principom zajedničke upotrebe bismo mogli da u jednom paketu grupišemo sve klase hijerarhije koja modelira čvorove apstraktnog drveta izraza, zato što se svaki put kada se pravi i koristi drvo izraza, potencijalno koriste sve te klase.

Ovaj princip može da se primenjuje i na funkcionalnu dekompoziciju, ali onda uglavnom ne na funkcionalnu koheziju već pre svega na komunikacionu, proceduralnu ili sekvencijalnu koheziju unutar komponente. Pomaže nam da se odlučimo da grupišemo u istu celinu klase koji se zajedno koriste. Iako se odnosi primarno na klase, može da se primeni i na druge strukturne elemente softvera.

Princip *CRP* nam pomaže i da se odlučimo da neke elemente *ne* grupišemo u istu celinu. Ako imamo elemente strukture softvera koji nisu međusobno funkcionalno

zavisni i pri tome se još i nezavisno upotrebljavaju, onda nema osnova da primenimo ni *REP* ni *CRP*, pa je verovatno bolje da ih ne stavljamo u istu celinu.

Princip zajedničke zatvorenosti (CCP)

„Delovi celine bi trebalo da budu zajedno i podjednako zatvoreni u odnosu na iste vrste promena. Ako celina mora da se menja, onda se pretpostavlja da moraju da se menjaju i drugi delovi te celine ali ne i druge celine."

Princip zajedničke zatvorenosti (engl. *CCP – The Common-Closure Principle*) se najčešće odnosi na slučajeve komunikacione kohezije. Ukazuje nam na to da može da bude dobro da delove grupišemo u celinu zato što imaju zajedničke faktore promenljivosti. Obično su to zajedničke zavisnosti od nekih elemenata, koji mogu da budu bilo u istoj ili u drugoj celini.

Princip *CCP* se obično odnosi na pakete, koji sadrže skupove takvih klasa (ili skupove komponenti), koje nisu neophodno međusobno funkcionalno povezane niti se često koriste zajedno. Zbog toga u formulaciji ovog principa umesto pojma *celina* može da se koristi i pojam *paket*. Princip *CCP* može da se odnosi i na komponente i klase, pri čemu takve komponente i klase obično imaju veoma širok interfejs, ali i neki oblik objedinjenog interfejsa (npr. obrazac *Fasada*), da bi se bar delimično zaklonila složenost implementacije. Takav interfejs često ne pruža sve funkcionalnosti koje delovi celine podržavaju, već samo one koje se najčešće koriste.

Na primer, neka imamo skup funkcija (metoda) za formatiranje delova izveštaja. One mogu da budu međusobno relativno nezavisne, ali sve zavise od specifikacije formata izveštaja. Ako se promeni format izveštaja, vrlo je verovatno da će se menjati i veći broj tih funkcija. Grupisanjem ovakvih funkcija u jednu klasu ne ostvarujemo funkcionalnu koheziju, zato što one nisu međusobno zavisne, pa ne primenjujemo *REP*, ali ostvarujemo komunikacionu ili proceduralnu koheziju i primenjujemo *CCP*. Na taj način se ostvaruje dobit time što jednu vrstu promena lokalizujemo u jednoj celini, umesto da moramo da menjamo delove različitih celina.

Princip stabilne zavisnosti (SDP)

„Smer zavisnosti bi trebalo da se poklapa sa smerom porasta stabilnosti."

Princip stabilne zavisnosti (engl. *SDP – The Stable-Dependencies Principle*), kao i naredna dva principa, pripada principima razdvajanja. Oni nam više govore o dobrim načinima uspostavljanja zavisnosti između različitih celina nego o zavisnostima unutar celina, tj. pre se odnose na spregnutost nego na koheziju. Međutim, njihova primena može da ima smisla i na nivou delova celina, zato što pitanje smera zavisnosti može da bude značajno i u tom slučaju.

Pod pojmom *stabilnost* podrazumevamo procenjenu meru verovatnoće menjanja nekog elementa programa. Kažemo da je element stabilan ako relativno mali broj faktora može da dovede do potrebe da ga menjamo, a da je nestabilan ako ima više takvih faktora.

Princip stabilne zavisnosti nam sugeriše da bi zavisnosti između elemenata programa trebalo da se uspostavljaju tako da idu od manje stabilnih elemenata prema stabilnijima (tj. da manje stabilni elementi zavise od stabilnijih elemenata), a ne obrnuto. Motivacija je sasvim jednostavna – ako je neki element programa nestabilan, to znači da će on relativno često da se menja, pa ako od njega zavise drugi elementi, onda je moguće da će usled njegovih promena i oni morati da se menjaju. To znači da je u slučaju nepoklapanja smera zavisnosti i porasta stabilnosti mnogo veći rizik da dođe do tzv. *lavine promena*, kada promene u jednom elementu indukuju promene u drugim zavisnim elementima i tako redom.

Može da izgleda da je ostvarivanje poštovanja ovog principa relativno jednostavno, zato što zavisnost jedne celine od mnogo drugih celina izaziva njenu nestabilnost, pa onda može da se stekne utisak da se poklapanje smerova zavisnosti i porasta stabilnosti ostvaruje „po definiciji". Takav utisak može da nas prevari. Jedan oblik potencijalnog narušavanja ovog principa nastaje kada neka celina zavisi od mnogo drugih i kada istovremeno od nje zavise mnoge druge celine. Takvi slučajevi su relativno česti u praksi, na primer kada pravimo fasadu ispred nekog složenog skupa celina – fasada istovremeno i koristi više drugih celina i ima više korisnika; ona mora da enkapsulira veći broj potencijalno nestabilnih klasa, a sama mora da bude stabilna. Jedno rešenje ovakvog problema je u primeni principa Izolovanih promenljivosti, tj. u definisanju što opštijeg i apstraktnijeg interfejsa fasade, čime se ostvaruje njena stabilnost i poštovanje Principa stabilne zavisnosti.

Ovaj princip se u praksi često primenjuje kao detektor lošeg dizajna, tj. služi nam prvenstveno da bi nam ukazao na postojanje problema, koji se zatim rešava primenom nekih drugih principa (na primer, Principa inverzne zavisnosti).

Princip stabilne apstrakcije (SAP)

„Celina bi trebalo da bude apstraktna onoliko koliko je stabilna."

Princip stabilne apstrakcije (engl. *SAP – The Stable-Abstractions Principle*) se bavi odnosom stabilnosti i apstraktnosti. On nam sugeriše da bi apstraktne delove programa trebalo da pišemo tako da oni ne zavise od nestabilnih delova programa.

Ključan aspekt Principa stabilne apstrakcije je u povezivanju apstraktnosti i stabilnosti. U tom smislu je ovaj princip neposredno u vezi sa nekim drugim principima, a pre svih sa Principom stabilne zavisnosti, Principom izolovane promenljivosti i Principom inverzne zavisnosti. On nam pomaže da razrešimo situacije kada neki strukturni element zavisi od jednog ili više elemenata koji su nestabilni, a istovremeno mora da bude stabilan zato što od njega zavise drugi elementi. Princip stabilne apstrakcije nam sugeriše da je rešenje za takve probleme u postizanju stabilnosti tog elementa kroz podizanje njegovog nivoa apstraktnosti.

Dobar primer primene ovog principa predstavlja hijerarhijski polimorfizam. Svaka hijerarhija klasa, ako posmatramo zbirno sve njene elemente, implementira

neko složeno ponašanje. To ponašanje obuhvata veći broj različitih slučajeva, koje opisujemo različitim klasama hijerarhije. Sa druge strane, imamo korisnike te hijerarhije za koje ne želimo da zavise od složenih aspekata ponašanja koje je ugrađeno u hijerarhiju, pa čak ni da znaju za sve te različite slučajeve. Problem rešavamo pravljenjem apstraktne bazne klase hijerarhije, koja će da predstavlja jedinstveni interfejs za sve elemente hijerarhije i sve različite slučajeve ponašanja. Apstraktni interfejs predstavlja istovremeno i primenu Principa inverzne zavisnosti, zato što sve klase hijerarhije zavise od konkretnog ustanovljenog interfejsa bazne klase, dok interfejs bazne klase zavisi samo od sveukupnog načelnog ciljnog ponašanja hijerarhije. Time što je napravljena kao apstraktna, bez zavisnosti od brojnih specifičnosti implementacije, bazna klasa je postala i stabilna.

Primer narušavanja ovog principa može da bude hijerarhija klasa kroz koju se implementira veći broj odgovornosti. Svako dodavanje ili menjanje odgovornosti hijerarhije menja i njen interfejs (baznu klasu) i utiče na njene korisnike, pa je jasno da se tako narušava Princip stabilne zavisnosti – bazna klasa hijerarhije je nestabilna a od nje zavisi veći broj korisnika. Jedno od uobičajenih rešenja ovakvog problema je primena obrasca Posetilac.

Kao drugi primer može da nam posluži fasada, pomenuta u prethodnom odeljku. Na osnovu ovog principa možemo da zaključimo da je dobro da fasada ima apstraktan interfejs, zato što se na taj način lakše ostvaruje njena stabilnost.

Princip acikličnih zavisnosti (ADP)

„Ne dopuštati ciklične zavisnosti paketa.“

Princip acikličnih zavisnosti (engl. *ADP – The Acyclic-Dependencies Principle*) se odnosi prvenstveno na pakete. Može da se primenjuje i na funkcionalne elemente programa, pre svega na komponente i klase, ali tada moramo imati u vidu da ciklične funkcionalne zavisnosti ponekad ne mogu da se potpuno eliminišu, pa se u takvim slučajevima samo trudimo da ih modifikujemo tako da imaju što manji uticaj na stabilnost delova programa.

Osnovni problem sa cikličnim zavisnostima je u tome što one značajno otežavaju lokalizovanje promena u programu. Ako imamo neke pakete koji su međusobno ciklično zavisni, pa se dogodi da moramo da promenimo neki od tih paketa, onda postoji mogućnost da ta promena mora da se propagira na naredni povezan paket, pa tako u krug. Da bismo sprečili eventualni dugačak niz promena, pri menjanju prvog paketa moramo odmah da imamo u vidu sve aspekte ostalih ciklično uvezanih paketa, čime se i rezonovanje i programiranje značajno otežavaju. Logična posledica takvih problema je upravo ovaj princip – ne želimo da imamo ciklične zavisnosti paketa.

Cikličnost se obično „razbija“ tako što se promeni smer zavisnosti između neka dva elementa, koji učestvuju u cikličnoj zavisnosti. Tu obično ima mesta za primenu

Principa inverznih zavisnosti ili Principa izmišljotina. Smer zavisnosti se obično menja uvođenjem novog apstraktnog koncepta, koji često ne izgleda prirodno u posmatranom domenu, već predstavlja vid izmišljotine.

Kao primer može da nam posluži često primenjivana arhitektura korisničkog interfejsa „Model-pogled-kontroler“. Ideja ove arhitekture je da model opisuje objekte, pogled prikazuje njihovo stanje korisniku, a kontroler upravlja izmenama. To je u osnovi ciklična arhitektura, zato što kontroler upravlja izmenama stanja modela, model promene mora da prosledi pogledu da bi se prikazale, a korisnik preko pogleda zadaje naredbe kontroleru. Jedan način da se u ovom slučaju prevaziđe cikličnost je uvođenje obrasca Posmatrač u odnos pogleda i modela, čime se taj odnos apstrahuje tako da i model i pogled zavise od apstraktnog interfejsa koji propisuje taj obrazac, a ne neposredno jedan od drugog. Analogno, odnos između pogleda i kontrolera može da se apstrahuje upotrebom principa razvoja vođenog događajima, za koji možemo da kažemo da predstavlja uopštenje i unapređenje obrasca Posmatrač.

6.5 Umesto zaključka

U ovom poglavlju su predstavljeni neki od najistaknutijih skupova principa projektovanja softvera, koji su nastali kao rezultat intenzivnog i obimnog rada istraživača i razvijalaca u oblasti OO programiranja i agilnog razvoja softvera. Principi projektovanja ne predstavljaju univerzalno rešenje za sve probleme u projektovanju softvera, ali mogu da nam budu od velike pomoći, zato što nam pomažu da izbegnemo uobičajene greške i usmeravaju nas prema boljim rešenjima. Osnovna uloga principa projektovanja softvera je da nam ukažu na značajne aspekte dizajna i da nam na taj način pomognu da uočimo potencijalne slabosti u strukturi softvera i sagledamo moguće načine njihovog prevazilaženja. Neki principi se međusobno preklapaju, neki se pomalo i sukobljavaju, a na nama je da pažljivo procenjujemo koji kada i kako treba da primenimo.

Principi projektovanja se u literaturi pominju u donekle različitim oblicima. Ovde su navedeni u skladu sa njihovim predstavljanjem u [*Martin 2003*, *Larman 2002*]. Alternativa neposrednom izučavanju principa projektovanja je izučavanje praktičnih primera njihove primene, a teško da ima boljih primera nego što su obrasci za projektovanje [*Gamma 1995*] i refaktorisanje [*Fowler 1999*]. Zbog njihovog velikog značaja obrasci za projektovanje i refaktorisanje su predstavljeni u narednim poglavljima.

7 - Obrasci za projektovanje

Najvažnija razlika između eksperta za arhitekturu softvera i početnika
je znanje o tome šta radi i šta ne radi
Andrej Aleksandresku

7.1 Ekvivalentnost problema

Programeri i projektanti softvera se u svakodnevnom radu stalno iznova susreću sa različitim vidovima *ponavljanja problema*. Ako se neki problem ponovi u nekom projektu na različitim mestima u doslovno istom obliku, onda je uobičajeno da se rešenje problema izdvoji, na odgovarajući način, i zatim tako izdvojeno ponovo upotrebljava gde god je to potrebno i moguće. Najjednostavniji način izdvajanja koda koji rešava neki problem je pravljenje potprograma. U složenijim slučajevima nisu dovoljni pojedinačni potprogrami, već je neophodno da se prave moduli, klase ili biblioteke. Dobro izdvojen programski kod može da se upotrebljava za rešavanje ponovljenih problema ne samo u okviru jednog projekta, već i mnogo šire. Uobičajeno je da se jednom oblikovane biblioteke upotrebljavaju u različitim projektima u nekom razvojnom okruženju. Mnoge dobre biblioteke su prevazišle granice okruženja u kojima su nastale i koriste se širom sveta u mnogim projektima.

Ako se problem ne ponavlja u potpuno istom obliku, već postoje određene varijacije, onda nije uvek moguća primena nepromenljivih biblioteka. Zbog toga dobre biblioteke moraju da omoguće proširivanje u skladu sa potrebama korisnika, u skladu sa principom otvorenosti i zatvorenosti. Da bi na prihvatljiv način mogle da se obuhvate što šire potencijalne varijacije, takve biblioteke se obično zasnivaju na primeni *polimorfizma*. U savremenom razvoju softvera se primenjuju svi vidovi

polimorfizma – hijerarhijski polimorfizam je najzastupljeniji, ali mnogi programski jezici podržavaju i implicitni ili parametarski polimorfizam[28]. Primena polimorfizma mora dobro da se odmeri. Ako se pretera u apstraktnosti (tj. širini) rešenja, onda lako može da dođe do suvišnog *naduvavanja* biblioteke, a time i povećane složenosti i smanjene efikasnosti pri njenom korišćenju. Sa druge strane, ako biblioteka nije dovoljno apstraktna, onda se potencijalno sužava prostor za njenu primenu.

Pri posmatranju i analiziranju nekog problema neophodno je da se prepoznaju značajni elementi, koji su po nečemu karakteristični za taj problem. Elemente nekog problema čine različiti entiteti koji figurišu u problemu, kao i različite vrste odnosa koji postoje među tim entitetima. Elementi problema obuhvataju sve zahteve i preduslove koji moraju da se uvaže da bi problem mogao da se rešava.

Naravno, jedan isti problem može da se posmatra na više različitih načina, pa zbog toga i uočeni elementi problema neće biti isti u različitim okolnostima. Mnogi aspekti posmatranja problema utiču na uočavanje njegovih elemenata. Ako se pri posmatranju problema najviše pažnje posvećuje nekim njegovim opštim karakteristikama, onda kažemo da se radi o uopštenom posmatranju, a ako se posmatraju detalji konkretnog slučaja, onda se radi o posmatranju specifičnosti.

Pri posmatranju problema, aspekti problema se modeliraju različitim sredstvima, da bi se poboljšalo njihovo razumevanje. Ponekad model problema može da bude veoma blizak samom problemu. Takav slučaj imamo kada su pojmovi koji se koriste u modelu veoma bliski pojmovima koji se koriste u samom opisu problema u njegovom domenu. Suprotno od toga, ako pojmovi modela ne mogu da se bijektivno preslikaju u pojmove domena, onda model nije sasvim blizak posmatranom problemu i predstavlja nekakvu njegovu apstrakciju. Od svih karakteristika posmatranja problema, na ovom mestu nas najviše zanima upravo nivo apstraktnosti posmatranja. Apstraktnost posmatranja je proporcionalna stepenu uopštavanja i apstraktnosti modela. Što je stepen uopštavanja viši, to je viši i nivo apstraktnosti posmatranja. Takođe, ako je viši nivo apstraktnosti posmatranja, onda je viši i nivo apstraktnosti modela.

Za probleme kažemo da su *međusobno slični* u odnosu na neke njihove modele ako možemo da napravimo preslikavanje modela jednog problema u model drugog problema. Ako može da se uspostavi bijektivno preslikavanje između modela dvaju problema, onda svaki od tih modela jednako dobro modelira oba problema. U takvom slučaju, kada za dva problema praktično imamo jedan zajednički model, onda kažemo da su ti problemi *međusobno ekvivalentni* u odnosu na taj zajednički model.

[28] Polimorfizmu i posebno parametarskom polimorfizmu ćemo posvetiti više pažnje u poglavlju *11 - Polimorfizam*, na strani 299.

Primetimo da za bilo koja dva problema možemo da pronađemo model u odnosu na koji su ekvivalentni. Potrebno je samo da nivo apstrahovanja pri posmatranju problema podignemo dovoljno visoko. Na primer, skoro svaki problem možemo da „rešimo“ sledećim algoritmom:

- prikupimo potrebne resurse;
- izvršimo izračunavanje i potrebne poslove;
- napravimo izveštaje.

Naravno, korist od takvog uopštenog „rešavanja“ problema je prilično ograničena. Da bismo od ekvivalentnosti problema u odnosu na neki model mogli da imamo neke praktične koristi, neophodno je da nivo apstrakcije modela bude u nekim razumnim granicama.

Ako su neka dva problema međusobno ekvivalentna u odnosu na model koji je izveden iz jednog od njih bez skoro ikakvog apstrahovanja, onda možemo da kažemo da se radi o *ponovljenim problemima*, kod kojih eventualno postoje tek neke manje varijacije. Takvi problemi se rešavaju na isti ili vrlo sličan način, uz moguću primenu postojećih ili pravljenje novih biblioteka. Ako se ekvivalentnost problema ostvaruje u odnosu na neki model koji je dobijen apstrahovanjem, ali je nivo apstrakcije i dalje relativno nizak, tako da se problemi i dalje mogu rešavati praktično istom implementacijom ili bibliotekom, onda ćemo reći da su takvi problemi *implementaciono ekvivalentni*. Ponovljeni i implementaciono ekvivalentni problemi predstavljaju ekstreman slučaj, kada zajednički model tih problema može da se dobije uz vrlo malo apstrahovanja.

Drugi ekstremni slučaj predstavljaju problemi koji su međusobno ekvivalentni tek na nekom veoma visokom nivou apstrakcije, koji nam ne donosi praktične koristi. Za probleme čiji su modeli ekvivalentni tek na nekom ekstremno visokom nivou apstrakcije možemo da kažemo da *praktično nisu ekvivalentni* ili da su samo *trivijalno ekvivalentni*. Primer trivijalne ekvivalentnosti je ranije naveden primer svođenja rešavanja problema na suviše uopšten algoritam, tako da nam takva ekvivalentnost skoro uopšte ne pomaže pri rešavanju problema.

Opisani ekstremni slučajevi nam nisu posebno ineresantni – trivijalna ekvivalentnost ne predstavlja praktično nikakav doprinos, a implementaciona ekvivalentnost je osnov izgradnje programa od samog nastanka računarstva i programiranja. Zato ćemo u ovom poglavlju najviše pažnje da posvetimo trećoj vrsti ekvivalentnih problema – onima koji su ekvivalentni pri nekom *umereno visokom nivou* apstrahovanja. Takve probleme, koji su netrivijalno ekvivalentni, a nisu implementaciono ekvivalentni, nazivamo *konceptualno ekvivalentnim problemima*.

7.2 Pojam obrasca za projektovanje

Ako imamo konceptualno ekvivalentne probleme, tada oni nisu dovoljno slični da bismo mogli da ih implementiramo istim programskim kodom, ali među njima ipak ima dovoljno sličnosti da možemo da uočimo neki apstraktan model u odnosu na koji su netrivijalno ekvivalentni. Ključno pitanje za razvijaoce softvera je: da li možemo da uočene sličnosti na neki način prevedemo u uopšteno zajedničko rešenje, čak i kada su te sličnosti ustanovljene na relativno visokom nivou apstraktnosti? Odgovor je, naravno, negde između – možemo da napravimo nešto poput *zajedničkog rešenja*, ali to ne može da bude zajednički programski kod. Kao što je i ekvivalentnost problema konceptualna, tako i zajedničko rešenje najčešće može da bude najviše konceptualno, a ne i implementaciono.

Konceptualno rešenje može da nam bude od velike koristi. Ako neki problem rešimo konceptualno, onda takvo rešenje možemo ponovo da primenimo pri rešavanju ekvivalentnih problema. Naravno, moraćemo ponovo da pišemo programski kod (celog rešenja ili bar dela rešenja, zato što se ne radi o implementacionoj ekvivalentnosti problema), moraćemo ponovo i da testiramo napisani kod i da tražimo eventualne greške u kodiranju, ali ćemo uštedeti vreme tako što nećemo morati da ponovo analiziramo sve detalje problema ili da pokušavamo da primenimo neka potencijalno neodgovarajuća rešenja. U takvim slučajevima najčešće nećemo morati ni da od početka smišljamo algoritam i načine njegove implementacije.

Da bi neko konceptualno rešenje bilo višestruko *konceptualno primenjivo*, tj. primenjivo u vidu ponovljene primene istog koncepta, ono mora da zadovolji neke uslove:

- mora da bude jasno prepoznato koji apstraktan problem se rešava;
- moraju da se uoče i istaknu svi preduslovi za uspešnu primenu rešenja;
- rešenje mora da bude dovoljno apstraktno da može da se primeni u većem broju netrivijalno ekvivalentnih slučajeva;
- rešenje mora da bude dovoljno konkretno da može da se razume i implementira;
- poželjno je da budu uočeni i naglašeni važni aspekti implementacije rešenja i
- moraju da budu poznate i dokumentovane posledice primene rešenja.

Ako jedno konceptualno rešenje, osim što ima sve navedene karakteristike, još i odgovara nekoj značajnoj klasi problema, onda se takvo rešenje, zajedno sa pripadajućom dokumentacijom, naziva *obrazac za projektovanje*[29].

Svaki obrazac za projektovanje ima četiri osnovna elementa:

- naziv obrasca;
- problem koji se obrascem rešava;
- rešenje problema i
- posledice rešenja.

Naziv obrasca za projektovanje se bira tako da, u svega nekoliko reči, što bolje opiše problem, njegova rešenja i posledice. Davanjem naziva obrascima se povećava rečnik projektovanja i omogućava raspoznavanje obrasca u komunikaciji i njegovo lakše referisanje; olakšava se razmena mišljenja o obrascima i diskutovanje o konceptualnim rešenjima, kao i pisanje i čitanje projektne dokumentacije; sam postupak projektovanja se podiže na viši nivo apstrakcije. Bez imenovanja obrazaca ne bismo mogli da napravimo kataloge ili rečnike obrazaca.

Svakom obrascu odgovara neka vrsta problema koji njegovom primenom može da se reši. Detaljan ali istovremeno i dovoljno apstraktan opis problema predstavlja osnovno sredstvo za prepoznavanje slučajeva u kojima neki obrazac može da se primeni. Opis problema može da obuhvata i konkretne primere, kao i opise elemenata problema (pojmova, odnosa,...) i odgovarajuće dijagrame. Može da obuhvata i spisak uslova koje je potrebno ispuniti da bi obrazac mogao da se uspešno primeni.

Rešenje problema predstavlja detaljan apstraktan opis elemenata koji čine projekat (dizajn) rešenja, njihove odgovornosti i međusobne odnose, kao i opis i tok saradnje među elementima. Rešenje ne sme da bude ograničeno na opisivanje određenog konkretnog projekta ili implementacije, zato što obrazac treba da predstavlja uputstvo koje može da se primeni u mnogim različitim slučajevima. Neka rešenja su bolje prilagođena nekim konkretnim programskim jezicima nego nekim drugim, ali osnovna ideja svakog rešenja bi trebalo da bude izložena dovoljno uopšteno da ono može da se primeni u različitim programskim jezicima.

Posledice rešenja obuhvataju različite rezultate i ocene primene obrasca. One mogu biti veoma značajne pri razmatranju pogodnosti primene nekog rešenja i odabiranju jednog od više mogućih načina rešavanja nekog problema. Sagledavanjem

[29] U srpskom jeziku su u upotrebi i termini „uzorak“ i „šablon“. Termin na engleskom jeziku je *design pattern*.

posledica omogućava se vaganje između prednosti i nedostataka nekog rešenja, odnosno obrasca.

Obrasci za projektovanje su najpre prepoznati u urbanizmu i arhitekturi [*Alexander 1997*]. Kasnije se o obrascima počelo razmišljati i u drugim domenima. Značaj obrazaca u razvoju softvera je uočen početkom 1990-ih godina. Šira stručna javnost se sa obrascima u razvoju softvera upoznala 1995. godine, kada je objavljena knjiga „Obrasci za projektovanje – Elementi više puta upotrebljivog objektno-orijentisanog softvera" (engl. *Design Patterns – Elements of Reusable Object-Oriented Software*) [*Gamma 1995*]. Ta knjiga se odlikuju izuzetno visokim kvalitetom sadržaja i izlaganja, zbog čega je postala (i ostala) nezaobilazan deo programerske lektire.

7.3 Primer – Obrazac *Unikat*

Obrazac *Unikat* (engl. *Singleton*) spada u najjednostavnije obrasce, ali se relativno često primenjuje. I pored jednostavnosti, on dobro ilustruje koncept obrazaca za projektovanje, pa je zbog toga pogodan da nam posluži kao uvodni primer. Namena obrasca Unikat je obezbeđivanje da neka klasa može da ima najviše jedan primerak.

Relativno često je potrebno da se u programu napravi i upotrebljava tačno jedan objekat neke klase. Postoji mnogo primera, a među najčešće spadaju različiti vidovi upravljačkih ili kontrolnih objekata i različite vrste registara ili kataloga objekata. Na primer, ako bismo pravili program za crtanje koji omogućava dinamičko dodavanje umetaka (engl. *plug-in*), onda bismo za upravljanje umecima mogli da napravimo klasu `UpravljačUmecima`, koja bi smela da ima tačno jednu instancu. Slično, ako bismo želeli da u program ugradimo interpretator nekakvih skriptova, imalo bi smisla da, radi podizanja efikasnosti, pamtimo prethodno parsirane skriptove, pri čemu bi implementacija takvog kataloga prevoda takođe morala da ima tačno jednu instancu.

U najjednostavnijim slučajevima je dovoljno da se definiše jedan globalni objekat i da se koristi u programu kao jedinstven primerak klase. Međutim, takav pristup ne rešava mnoge potencijalne probleme. Jedan od najvažnijih problema je to što programer i dalje može da napravi više primeraka klase, što onda može da dovede u pitanje funkcionalnost ili performanse sistema. Drugi ozbiljan problem je što ne postoji mogućnost jednostavnog upravljanja redosledom pravljenja takvih „jedinstvenih" primeraka različitih, potencijalno međuzavisnih klasa. Promena redosleda prevođenja ili povezivanja delova programa može da ima uticaja na redosled pravljenja takvih objekata, što u slučaju složenih međuzavisnosti može da bude veoma problematično.

Osnovni zahtevi koji se postavljaju pred rešenje su:

- obezbeđivanje interfejsa za pravljenja jedinstvenog objekta, uz skrivanje samog pravljenja;

- odloženo pravljenje objekta do njegove prve upotrebe (lenjo pravljenje);
- uništavanje objekta pri završetku rada programa i
- inicijalizacija bezbedna za niti.

Osnovna ideja rešenja obrasca Unikat je da se staranje o jedinstvenom primerku klase prepusti samoj klasi. U programskom jeziku C++ to može da se ostvari pisanjem statičkog metoda[30] `Primerak()`, koji vraća referencu na jedinstveni primerak klase – *unikat*. Pri tome se svi konstruktori skrivaju tako da ne postoji mogućnost da se napravi drugi objekat te klase:

```
class PrimerUnikata
{
public:
    static PrimerUnikata& Primerak()
    {
        static PrimerUnikata primerak {};
        return primerak;
    }

    // Metodi koji čine interfejs klase
    ...

private:
    PrimerUnikata()
    {...}

    PrimerUnikata( const PrimerUnikata& ) = delete;
    PrimerUnikata& operator=( const PrimerUnikata& ) = delete;
};
```

U programskom jeziku C++ se statički objekti po pitanju pravljenja ne razlikuju značajno od globalnih automatskih promenljivih, pa ni za njih nije jednostavno predvideti redosled pravljenja. Ipak, lokalne statičke promenljive (kao promenljiva `primerak` u metodu `PrimerUnikata::Primerak`) se prave, tj. inicijalizuju, tek pri

[30] U programskom jeziku C++ se *metodi* nazivaju *funkcijama članicama* ili *članskim funkcijama* (engl. *member function*), pa se i *statički metodi* nazivaju *statičkim funkcijama članicama* ili *statičkim članskim funkcijama*. Zbog toga što se u ovoj knjizi istovremeno bavimo i opštim principima objektno-orijentisanog programiranja i konkretnim tehnikama u programskom jeziku C++, zbog usklađenosti terminologije ćemo se držati termina *metod*.

prvom izvršavanju bloka u kome su definisane, pa navedeno rešenje omogućava lenjo pravljenje unikata. Uz to, ono je i bezbedno za niti[31].

Svi konstruktori unikatne klase moraju da budu privatni, da bi pravljenje objekata klase bilo moguće samo u okviru njene implementacije, konkretno u okviru implementacije metoda `Primerak`. U primeru je naveden konstruktor bez argumenata, ali umesto njega može lako da se napravi i zatim iskoristi i neki drugi konstruktor sa argumentima, ako je potreban.

Konstruktor kopije mora eksplicitno da se zabrani (tj. obriše), da bi se sprečilo da neko namerno ili slučajno napravi drugi objekat kopiranjem dobijenog. Radi kompletnosti je uobičajeno da se isto uradi i sa operatorom dodeljivanja, iako do njegove primene ne bi trebalo da može da dođe ako imamo najviše jedan objekat[32].

Na svakom mestu u programu gde je potrebno da se koristi unikatni objekat, njemu mora da se pristupa putem metoda `Primerak`, na primer:

```
... PrimerUnikata::Primerak().metod(...) ...
```

Potencijalan problem sa navedenim rešenjem može da nastupi ako implementacije više različitih unikata koriste jedna drugu. Redosled pravljenja će biti ispravan, zato što će se objekti unikata praviti onda kada se po prvi put koriste. Međutim, problem je u tome što programer ne može da utiče na redosled deinicijalizovanja i brisanja statičkih objekata, pa može da se dogodi da primenjeni redosled[33] proizvede određene probleme, poput deinicijalizovanja objekata koji se još koriste. U takvom slučaju može da se primeni dinamička alokacija[34]:

---

[31] Za potprogram ili deo koda kažemo da je *bezbedan za niti* (engl. *threadsafe*) ako može bez ikakvog dodatnog staranja da se istovremeno koristi iz više niti, bez potrebe da se dodatno obezbeđuje, na primer muteksima ili katancima.

[32] Navedena sintaksa je u upotrebi od verzije C++11. U starijim verzijama su se ovi metodi deklarisali kao privatni, a nisu se obezbeđivale njihove implementacije, pa bi eventualni pokušaj upotrebe dovodio do greške u fazi prevođenja ili povezivanja.

[33] Redosled deinicijalizovanja statičkih objekata bi trebalo da bude obrnut od redosleda njihovog pravljenja, ali takav redosled u nekim slučajevima nije idealan. Programer ima vrlo ograničen uticaj na ovaj redosled, a i to zavisi od konkretne implementacije alata za prevođenje.

[34] Primetimo da ovde nema mnogo smisla upotreblavati pametne pokazivače, zato što bi uz njihovu primenu rezultat semantički bio po svemu isti kao u prvom rešenju, bez primene dinamičke alokacije.

```
static PrimerUnikata& Primerak()
{
    static PrimerUnikata* primerak = new PrimerUnikata {};
    return *primerak;
}
```

Slabost ovog rešenja je da ono ne omogućava automatsko brisanje unikata. Dinamički alociran unikat mora da se obriše eksplicitno, ili nikada neće biti obrisan, pa će doći do curenja memorije. Međutim, eksplicitno brisanje unikata takođe može da dovede do neispravnosti na sličan način kao i prethodno rešenje sa automatskim brisanjem. U takvim slučajevima se unikati ponekad namerno ne brišu. Iako zbog toga dolazi do curenja memorije, u ovom kontekstu to nekada može da bude prihvatljivo. Unikati se prave samo po jedanput i postoje tokom čitavog toka izvršavanja programa. Nakon završetka rada programa operativni sistem će osloboditi svu lokalnu memoriju procesa, pa i onu koja je bila zauzeta pravljenjem dinamičkih objekata koji nisu obrisani. Ipak, ako unikat alocira i koristi globalnu memoriju ili neke druge globalne resurse operativnog sistema (otvorene datoteke, mrežne resurse, veze sa bazama podataka i drugim servisima i sl.), koji se ne oslobađaju automatski pri završetku programa, onda je njihovo eksplicitno brisanje neophodno i moramo mu posvetiti dodatnu pažnju. Takođe, problem sa curenjem memorije dobija na značaju ako su unikati jedinstveni za svaku nit posebno, ali ne i za ceo program. Tada pravljenje većeg broja niti ima za posledicu i pravljenje većeg broja unikata, pa ako se oni ne uništavaju ispravno, onda to može da dovede do značajnog curenja memorije.

Pri dinamičkoj alokaciji mogu da se koriste i deljeni pokazivači, kao u narednom primeru. Upotreba deljenih pokazivača kombinuje dobre karakteristike rešenja sa statičkim objektima i rešenja sa dinamičkom alokacijom, a uspešno onemogućava da se objekat obriše ako se još uvek negde upotrebljava (tj. ako se još uvek čuva neki deljivi pokazivač na taj objekat). Ako se objekti prave pomoću šablonske funkcije `make_shared`, onda performanse ovog rešenja ne zaostaju za performansama rešenja sa običnim pokazivačima:

```
static PrimerUnikata& Primerak()
{
    static shared_ptr<PrimerUnikata> primerak =
        make_shared< PrimerUnikata>();
    return *primerak;
}
```

7.4 Primer – Obrazac *Sastav*

Pre razmatranja obrasca *Sastav* (engl. *Composite*), najpre ćemo da pretpostavimo da imamo napisanu jednostavnu hijerarhiju ravnih likova, čiju osnovu predstavlja klasa `Lik`:

- svaka klasa hijerarhije odgovara nekom liku u ravni;
- svaki lik ima položaj, implementiran u klasi `Lik`;
- ne obraćamo posebnu pažnju na orijentaciju likova – pretpostavljamo da su stranice kvadrata i pravougaonika paralelne koordinatnim osama i
- jedini „važan“ metod izračunava površinu: `double Povrsina() const`.

Metod `Povrsina()` izračunava površinu lika. Implementiran je na način koji je uobičajen za hijerarhijski polimorfizam, sa potpuno virtualnom (tj. apstraktnom) deklaracijom u baznoj klasi `Lik` i sa odgovarajućim implementacijama u konkretnim klasama hijerarhije.

```
#include <iostream>
#include <vector>
#include <math.h>

using namespace std;

//-----------------------------------------------------------------
class Tacka
{
public:
    Tacka( double x, double y )
        : X_(x), Y_(y)
        {}

private:
    double X_;
    double Y_;

    friend ostream& operator<<( ostream&, const Tacka& );
    friend istream& operator>>( istream&, Tacka& );
};

ostream& operator<<( ostream& ostr, const Tacka& t )
{
    ostr << "(" << t.X_ << "," << t.Y_ << ")";
    return ostr;
}
```

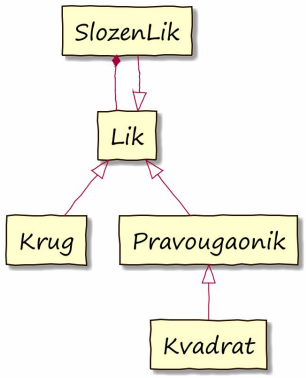
```
istream& operator>>( istream& istr, Tacka& a )
{
    char c1,c2,c3;
    istr >> c1 >> a.X_ >> c2 >> a.Y_ >> c3;
    if( c1!='(' || c2!=',' || c3!=')' )
        istr.setstate( ios::failbit );
    return istr;
}

//-----------------------------------------------------------------
class Lik
{
public:
    Lik( double x, double y )
        : Polozaj_(x,y)
        {}

    virtual ~Lik()
        {}

    const Tacka& Polozaj() const
        { return Polozaj_; }

    void PromeniPolozaj( const Polozaj& p )
        { Polozaj_ = p; }

    virtual double Povrsina() const = 0;

private:
    Tacka Polozaj_;
};

ostream& operator<<( ostream& ostr, const Lik& l )
{
    ostr << l.Polozaj() << " P=" << l.Povrsina();
    return ostr;
}

//-----------------------------------------------------------------
class Pravougaonik : public Lik
{
public:
    Pravougaonik( double x, double y, double s, double v )
        : Lik(x,y), Sirina_(s), Visina_(v)
        {}

    double Sirina() const
        { return Sirina_; }

    double Visina() const
        { return Visina_; }
```

```
    double Povrsina() const override
        { return Sirina() * Visina(); }

private:
    double Sirina_;
    double Visina_;
};

//---------------------------------------------------------------------
class Kvadrat : public Pravougaonik
{
public:
    Kvadrat( int x, int y, int a )
        : Pravougaonik( x, y, a, a )
        {}
};

//---------------------------------------------------------------------
class Krug : public Lik
{
public:
    Krug( double x, double y, double r )
        : Lik(x,y), R_(r)
        {}

    double R() const
        { return R_; }

    double Povrsina() const override
        { return R() * R() * M_PI; }

private:
    double R_;
};

//---------------------------------------------------------------------
int main()
{
    vector<Lik*> likovi;
    likovi.push_back( new Pravougaonik(1,2,3,4) );
    likovi.push_back( new Kvadrat(5,6,7) );
    likovi.push_back( new Krug(8,9,10) );
    for( Lik* lik: likovi ){
        cout << *lik << endl;
        delete lik;
    }

    return 0;
}
```

Pretpostavimo sada da je za našu aplikaciju neophodno da omogućimo da se sa nekim skupom likova radi kao da se radi o *jednom složenom liku*. Takav složen lik ima dvojaku prirodu – sa jedne strane on bi trebalo da bude u hijerarhiji likova, zato što složen lik *jeste* lik, a sa druge strane bi trebalo da predstavlja kolekciju jednostavnijih likova. Sličan problem se ponavlja veoma često. Na primer, ako modeliramo računske izraze, *suma* nekog niza podizraza je složen izraz i odnosi se prema hijerarhiji operacija i izraza na sličan način kao što se složen lik odnosi prema likovima.

Rešenje ovog tipa problema predstavlja obrazac *Sastav* (engl. *Composite*). Ovaj obrazac opisuje kako se prave elementi hijerarhije koji predstavljaju kolekcije drugih elemenata iste hijerarhije. Spada u strukturne obrasce. Ukratko ćemo predstaviti ideju, na uopšten način, nezavisno od programskog jezika:

- *sastav* se pravi kao klasa hijerarhije, tj. kao naslednik bazne klase hijerarhije;
- ima privatnu kolekciju objekata iste hijerarhije i javne metode za osnovne operacije sa tom kolekcijom, kao što je dodavanje novih ili brisanje postojećih elemenata;
- sastav je *vlasnik* svojih elemenata, tj. kada se briše sastav, brišu se i njegovi elementi;
- implementiraju se svi potrebni metodi hijerarhije.



Slika 22 – Obrazac *Sastav*, dijagram klasa

Napravićemo klasu `SlozenLik`, u okviru koje ćemo kolekciju objekata `Likovi_` da predstavimo kao `vector<SlozenLik*>`, a za dodavanje novih elemenata ćemo da napišemo metod `void Dodaj( Lik* l )`:

```
class SlozenLik : public Lik
{
public:
    SlozenLik( double x, double y )
        : Lik(x,y)
        {}

    ~SlozenLik()
    {
        for( Lik* lik: Likovi_ )
            delete lik;
    }

    void Dodaj( Lik* l )
        { Likovi_.push_back(l); }

    double Povrsina() const override
    {
        double p = 0;
        for( const Lik* lik: Likovi_ )
            p += lik->Povrsina();
        return p;
    }

private:
    vector<Lik*> Likovi_;
};
```

Implementacija metoda `Povrsina` je napisana tako da se ne proverava da li se neki od likova možda preklapaju. U ovom kontekstu je prihvatljivo da pretpostavimo da nema preklapanja, zato što nam je cilj da predstavimo obrasce, a ne da se bavimo algoritmima za izračunavanje preseka likova.

Primetimo da ova klasa nije dovršena, zato što nismo napisali operator dodeljivanja i konstruktor kopije, a oni su *neophodni* zbog načina na koji se čuvaju i brišu elementi kolekcije likova. Pisanje ovih metoda nije sasvim jednostavno. Problem je u tome što iz ugla složenog lika mi ne možemo da znamo kojim klasama pripadaju pojedinačni elementi, pa stoga ne možemo ni da ih neposredno iskopiramo. Ideja rešavanja ovog problema počiva na tome da, iako složeni lik to ne zna, *svaki pojedinačni element* složenog lika *zna* kojoj klasi on pripada. To *znanje* se ispoljava kroz ispravan odabir metoda koji se pozivaju pri dinamičkom vezivanju metoda. Znači, umesto da se složen lik brine o tome kojoj klasi pripada koji lik, on može da poruči svakom pojedinačnom liku da napravi svoju kopiju i da očekuje da će primenom dinamičkog vezivanja metoda biti upotrebljeni ispravni metodi za kopiranje i napravljene ispravne kopije elemenata kolekcije.

Prethodnom uopštenom opisu koncepta rešavanja dodajemo još neke elemente koji se odnose na slučaj upotrebe programskog jezika C++:

- u baznoj klasi hijerarhije se deklariše apstraktan metod `Kopija`, koji pravi kopiju objekta:
  ```
  virtual Lik* Kopija() const = 0;
  ```
- u svim konkretnim klasama hijerarhije ovaj metod se implementira na odgovarajući način – uobičajeno je da se to radi relativno jednostavno, pozivanjem konstruktora kopije ili pozivanjem nekog drugog konstruktora;
- u klasi koja predstavlja *sastav* se konstruktor kopije i operator dodeljivanja implementiraju tako da se svi elementi kolekcije kopiraju korišćenjem metoda `Kopija()`;
- pri implementiranju operatora „=“ primenjen je uobičajen idiom kopiraj-razmeni (engl. *copy-swap*); zbog toga su u klase `Lik` i `SlozenLik` dodate odgovarajuće prijateljske funkcije `swap`.

Sve skupa, to bi trebalo da izgleda ovako:

```
class Lik { ...
public:
    virtual Lik* Kopija() const = 0;

    friend void swap( Lik& lik1, Lik& lik2 ) {
        std::swap( lik1.Polozaj_, lik2.Polozaj_ );
    }
};

class Pravougaonik : public Lik { ...
public:
    Lik* Kopija() const override
        { return new Pravougaonik(*this); }
};

class Kvadrat : public Pravougaonik { ...
public:
    Lik* Kopija() const override
        { return new Kvadrat(*this); }
};

class Krug : public Lik { ...
public:
    Lik* Kopija() const override
        { return new Krug(*this); }
};
```

```
class SlozenLik : public Lik { ...
public:
    SlozenLik( const SlozenLik& sl )
        : Lik(sl)
    {
        for( const Lik* lik: sl.Likovi_ )
            Dodaj( lik->Kopija() );
    }
    SlozenLik& operator=( SlozenLik sl )
    {
        swap( *this, sl );
        return *this;
    }

    friend void swap( SlozenLik& lik1, SlozenLik& lik2 )
    {
        using std::swap;
        swap( static_cast<Lik&>(lik1), static_cast<Lik&>(lik2) );
        swap( lik1.Likovi_, lik2.Likovi_ );
    }

    Lik* Kopija() const override
        { return new SlozenLik(*this); }

private:
    void deinit()
    {
        for( Lik* lik: Likovi_ )
            delete lik;
        Likovi_.clear();
    }

    void init( const SlozenLik& sl )
    {
        for( const Lik* lik: sl.Likovi_ )
            Dodaj( lik->Kopija() );
    }
};
```

Upotrebu složenih likova možemo da ilustrujemo jednostavnim primerom:

```
int main()
{
    SlozenLik sl(0,0);
    sl.Dodaj( new Pravougaonik(1,2,3,4) );
    sl.Dodaj( new Kvadrat(5,6,7) );
    sl.Dodaj( new Krug(8,9,10) );
    SlozenLik* sl2 = new SlozenLik(1,1);
    sl2->Dodaj( new Krug(2,3,4));
    sl2->Dodaj( new Kvadrat(2,4,5));
    sl.Dodaj(sl2);
```

```
    cout << sl << endl;
    return 0;
}
```

Detaljan opis obrasca *Sastav*, uz razmatranje preduslova za upotrebu i različitih složenih okolnosti koje mogu da utiču na rešenje, može da se pročita iz knjige „Obrasci za projektovanje“ [*Gamma1995*].

7.5 Primer – Obrazac *Posetilac*

Kao treći primer obrazaca razmotrićemo obrazac *Posetilac* (engl. *Visitor*). To je obrazac ponašanja, koji opisuje kako možemo da na relativno jednostavan način apstrahujemo operacije koje se odvijaju uz obilaženje složenih struktura objekata, kao što su grafovi ili stabla. Nastavićemo na mestu na kome smo stali pri opisivanju obrasca *Sastav*, sa jednostavnom hijerarhijom likova. Za dalje razmatranje ovog primera najvažnije nam je da obratimo pažnju na sledeće:

- hijerarhija sadrži sastav `SlozenLik` i
- sve klase implementiraju metod `Povrsina()`.

Prethodni primer programskog koda je relativno jednostavan i intuitivan. Svi metodi su tamo gde im je mesto i rade jasno razdvojene poslove. Ali šta bi bilo kada bismo osim metoda `Povrsina()` imali još nekoliko metoda, kao na primer, `Obim()`, `BrojLikova()` i druge? Sve nove metode bismo implementirali kroz celu hijerarhiju likova: deklarisali bismo virtualne metode u klasi `Lik`, a zatim napisali odgovarajuće implementacije u klasama hijerarhije. I dalje bi takvo rešenje bilo relativno jednostavno i intuitivno, ali možemo da primetimo i dve nezanemarljive slabosti:

- u svim novim metodima u klasi `SlozenLik` imali bismo ponavljanje koda koji implementira obilazak kolekcije;
- sa porastom broja klasa i metoda, sagledavanje ukupne implementacije jedne vrste izračunavanja bi postalo relativno teško – da bismo videli kako se računa površina morali bismo da pronađemo i proučimo sve odgovarajuće metode u različitim klasama hijerarhije; takođe, i eventualno menjanje načina izračunavanja bi zahtevalo unošenje izmena u veliki broj klasa.

Tu stupa na scenu obrazac *Posetilac*. On uvodi važna unapređenja dizajna i prevazilazi obe navedene slabosti; eliminiše ponavljanje delova koda koji se odnose na obilazak složene strukture i locira sve aspekte izvršavanja jedne operacije u jednoj klasi. Sredstvo rešavanja je preusmeravanje odgovornosti, što donekle umanjuje performanse rešenja, ali to obično ne predstavlja problem.

Koncept rešenja je relativno jednostavan. Logika koja odgovara poslu zbog koga se struktura obilazi se implementira u novoj klasi `Posetilac`. Klasa `Posetilac` ima po jedan metod za svaku klasu hijerarhije koja se obilazi. Logika obilaska strukture i prepoznavanja konkretnih klasa hijerarhije se implementira u okviru hijerarhije koja se obilazi, pomoću dodatnog metoda `PrihvatiPosetu(Posetilac&)`. Ideja je slična ideji implementacije metoda `Kopija` opisanoj u okviru opisa obrasca *Sastav*: iako posetilac ne može da zna kojoj klasi pripada objekat koji se posećuje, kao ni da li on sadrži neke podobjekte, svaki konkretan objekat koji se posećuje to naravno vrlo dobro *zna*.

Metod `PrihvatiPosetu` se implementira u svakoj od klasa hijerarhije na odgovarajući način. U slučaju klasa *listova*, koje ne sadrže druge objekte, implementacija se po pravilu svodi na pozivanje metoda posetioca koji odgovara klasi posećenog objekta. Na primer, u klasi `Pravougaonik` ovaj metod može da se implementira ovako:

```
void PrihvatiPosetu( PosetilacLikova& p ) override
    { p.PosetiPravougaonik( *this ); }
```

Na taj način svaka implementacija metoda `PrihvatiPosetu` zapravo „javlja“ posetiocu kojoj klasi pripada posećeni objekat i koji konkretan metod posećivanja je potrebno da se primeni.

U slučaju klasa koje mogu da sadrže i druge objekte, pored pozivanja metoda posetioca koji odgovara konkretnoj klasi, metod `PrihvatiPosetu` mora da omogući posetiocu i da obiđe i sve sadržane objekte. To se radi pozivanjem metoda `PrihvatiPosetu` za svaki od sadržanih objekata. Na primer, u klasi `SlozenLik`, implementacija prihvatanja posete može da se napiše ovako:

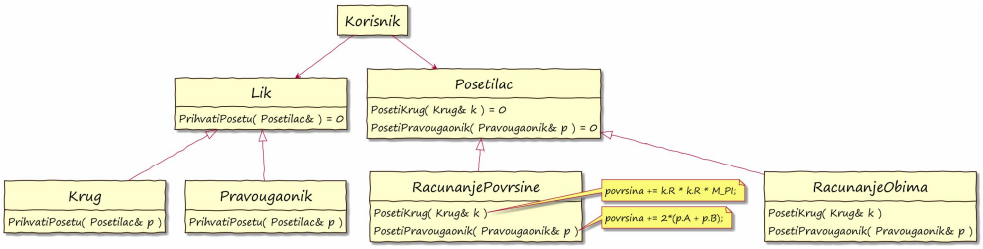
```
void PrihvatiPosetu( PosetilacLikova& p ) override
{
    p.PosetiSlozeniLik( *this );
    for( Lik* lik: Likovi_ )
        lik->PrihvatiPosetu(p);
}
```

U zavisnosti od prirode hijerarhije i poslova radi kojih se vrši obilazak, posećivanje sadržanih objekata može da se izvodi pre ili posle pozivanja metoda `PosetiSlozeniLik`, tj. posećivanje može da se radi prefiksno ili postfiksno. U nekim slučajevima čak može da se implementira i u više koraka, kao na primer:

```
void PrihvatiPosetu( PosetilacLikova& p ) override
{
    p.ZapocniPosetuSlozenogLika( *this );
    for( Lik* lik: Likovi_ )
        lik->PrihvatiPosetu(p);
```

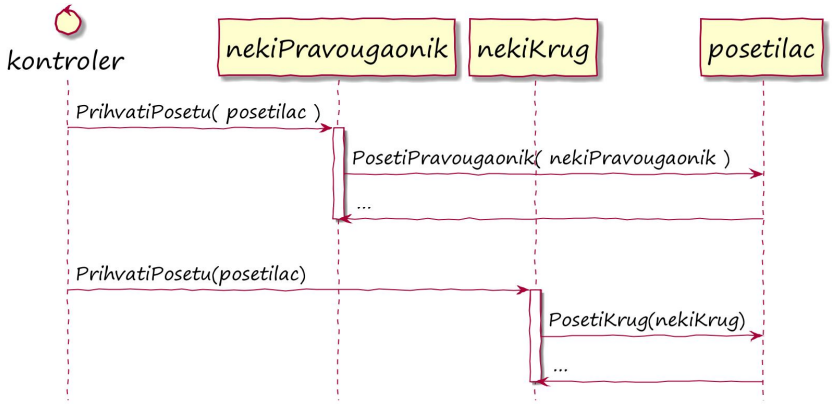
```
        p.ZavrsiPosetuSlozenogLika( *this );
    }
```

U konkretnom primeru, tj. u slučaju računanja površine, dodatno posećivanje složenog lika je suvišno (zato što je dovoljno posetiti sve njegove elemente), pa možemo da napravimo grešku i izostavimo ga. Međutim, u nekom drugom slučaju, na primer pri serijalizaciji objekata (npr. zapisivanju sadržaja u nisku ili u datoteku) te dodatne posete će biti neophodne da bi mogao da se uradi kompletan posao. Za one posetioce za koje je to suvišno (na primer, pri računanju površine), možemo da implementiramo prazan metod `PosetiSlozenLik`.



Slika 23 – Obrazac *Posetilac*, dijagram klasa

Obilaženje neke strukture objekata započinje tako što se na osnovnom objektu strukture pozove metod `PrihvatiPosetu(posetilac)`. U zavisnosti od konkretne klase i njene unutrašnje strukture, metod `PrihvatiPosetu` poziva jedan ili više odgovarajućih metoda posetioca. Postupak obilaska ilustruje dijagram sekvence (*Slika 24*).



Slika 24 – Obrazac *Posetilac*, dijagram sekvence

Klasa `Posetilac` u potpunosti obuhvata i implementira sve što se odnosi na obavljanje konkretnog posla. Na primer, ako posetilac izračunava površinu likova, onda bi metod `PosetiPravougaonik` izračunavao površinu pravougaonika. Pri tome

posetilac komunicira sa objektom koji posećuje iz samo dva razloga – posećeni objekat mora da „javi“ kojoj klasi pripada i da omogući obilaženje podobjekata. Posetilac sam obezbeđuje sve dodatne potrebne podatke i algoritme, kao što je na primer izračunavanje površine i čuvanje rezultata u našem primeru:

```
void PosetiPravougaonik( Pravougaonik& p ) override
    { povrsina += p.Sirina() * p.Visina(); }
```

Primetimo da predstavljena implementacija koristi rekurziju. To može da predstavlja problem u slučaju veoma složenih struktura podataka, zbog potencijalno velike dubine rekurzije. U tom slučaju posetilac može da se implementira tako da sadrži listu objekata koje mora da poseti, a metod `SlozenLik::PrihvatiPosetu` bi umesto rekurzivnog pozivanja metoda `PrihvatiPosetu` samo dodavao sve sadržane objekte u tu listu.

Obrazac *Posetilac* se obično implementira samo ako je potrebno da se podrži više različitih posetilaca, tj. ako imamo više različitih razloga ili načina posećivanja (tj. obilaženja) elemenata jedne hijerarhije. Svi potrebni posetioci se organizuju u posebnu hijerarhiju. Interfejs hijerarhije posetilaca mora da odgovara hijerarhiji klasa koja će se obilaziti. Bazna klasa samo propisuje interfejs (apstraktne metode), a konkretni posetioci ih implementiraju. U našem slučaju bazna klasa hijerarhije posetilaca može da bude:

```
class PosetilacLikova
{
public:
    virtual void PosetiPravougaonik( Pravougaonik& ) = 0;
    virtual void PosetiKvadrat( Kvadrat& ) = 0;
    virtual void PosetiKrug( Krug& ) = 0;
    virtual void PosetiSlozeniLik( SlozenLik& ) = 0;
};
```

Konkretan posetilac bi mogao da bude:

```
class RacunanjePovrsine : public PosetilacLikova
{
public:
    RacunanjePovrsine()
        : Povrsina_(0)
        {}

    double Povrsina() const
        { return Povrsina_; }

    void PosetiKrug( Krug& k ) override
        { Povrsina_ += k.R() * k.R() * M_PI; }
```

```
    void PosetiKvadrat( Kvadrat& k ) override
        { Povrsina_ += k.Sirina() * k.Sirina(); }

    void PosetiPravougaonik( Pravougaonik& p ) override
        { Povrsina_ += p.Sirina() * p.Visina(); }

    void PosetiSlozeniLik( SlozenLik& ) override
        {}

private:
    double Povrsina_;
};
```

Radi ilustracije, u narednom primeru predstavljamo primer koda, uz implementaciju dva posetioca, za računanje površine i obima lika. Radi uštede prostora navodimo samo delove koda koji su izmenjeni u odnosu na prethodni primer obrasca *Sastav*:

```
class Pravougaonik;
class Kvadrat;
class Krug;
class SlozenLik;

class PosetilacLikova
{
public:
    virtual ~PosetilacLikova() {}
    virtual void PosetiPravougaonik( Pravougaonik& ) = 0;
    virtual void PosetiKvadrat( Kvadrat& ) = 0;
    virtual void PosetiKrug( Krug& ) = 0;
    virtual void PosetiSlozeniLik( SlozenLik& ) = 0;
};

class Lik
{...
    virtual void PrihvatiPosetu( PosetilacLikova& p ) = 0;
};

class Pravougaonik : public Lik
{...
    void PrihvatiPosetu( PosetilacLikova& p ) override
        { p.PosetiPravougaonik( *this ); }
};

class Kvadrat : public Pravougaonik
{...
    void PrihvatiPosetu( PosetilacLikova& p ) override
        { p.PosetiKvadrat( *this ); }
};
```

```
class Krug : public Lik
{...
    void PrihvatiPosetu( PosetilacLikova& p ) override
        { p.PosetiKrug( *this ); }
};

class SlozenLik : public Lik
{...
    void PrihvatiPosetu( PosetilacLikova& p ) override
    {
        p.PosetiSlozeniLik( *this );
        for( Lik* lik: Likovi_ )
            lik->PrihvatiPosetu(p);
    }
};

// posetilac koji računa površinu
class RacunanjePovrsine : public PosetilacLikova
{
public:
    RacunanjePovrsine()
        : Povrsina_(0)
        {}

    double Povrsina() const
        { return Povrsina_; }

    void PosetiPravougaonik( Pravougaonik& p ) override
        { Povrsina_ += p.Sirina() * p.Visina(); }

    void PosetiKvadrat( Kvadrat& k ) override
        { Povrsina_ += k.Sirina() * k.Sirina(); }

    void PosetiKrug( Krug& k ) override
        { Povrsina_ += k.R() * k.R() * M_PI; }

    // složen lik nema dodatnu površinu u odnosu na komponente
    // a one će biti posebno posećene
    void PosetiSlozeniLik( SlozenLik& ) override
        {}

private:
    double Povrsina_;
};

// posetilac koji računa obim
class RacunanjeObima : public PosetilacLikova
{
public:
    RacunanjeObima()
        : Obim_(0)
        {}
```

```
    double Obim() const
        { return Obim_; }

    void PosetiPravougaonik( Pravougaonik& p ) override
        { Obim_ += 2* ( p.Sirina() + p.Visina()); }
    void PosetiKvadrat( Kvadrat& k ) override
        { Obim_ += 4 * k.Sirina(); }

    void PosetiKrug( Krug& k ) override
        { Obim_ += 2 * k.R() * M_PI; }

    // složen lik nema dodatni obim u odnosu na komponente
    // a one će biti posebno posećene
    void PosetiSlozeniLik( SlozenLik& ) override
        {}

private:
    double Obim_;
};

int main()
{...
    RacunanjePovrsine rp;
    sl.PrihvatiPosetu(rp);
    cout << "POVRSINA: " << rp.Povrsina() << endl;

    RacunanjeObima ro;
    sl.PrihvatiPosetu(ro);
    cout << "OBIM: " << ro.Obim() << endl;

    return 0;
}
```

Već smo istakli kvalitete obrasca *Posetilac*. Međutim, moramo da istaknemo i neke slabosti. Prva slabost je da primena obrasca *Posetilac* može da oteža dodavanje novih klasa hijerarhiji klasa koje se obilaze – svaki put kada se dodaje nova klasa u hijerarhiju, mora da se doda i odgovarajući nov metod u sve postojeće klase posetilaca. Druga slabost je da za neka izračunavanja može da bude neophodno da posetilac naruši enkapsulaciju nekih klasa hijerarhije.

Implementacija posetilaca u programskom jeziku *C++* pruža i neke dodatne pogodnosti. Na primer, ako u okviru klasa posetilaca za računanje površine i obima dodamo konstruktorima kao argument objekat od koga se započinje obilaženje, pa još implementiramo i konvertor posetioca u realan broj, onda klase posetilaca možemo da koristimo i kao funkcije:

```
class RacunanjePovrsine : public PosetilacLikova
{
public:
```

```
    RacunanjePovrsine( Lik& l )
        : Povrsina_(0)
        { l.PrihvatiPosetu(*this); }

    operator double() const
        { return Povrsina(); }
...
};

class RacunanjeObima : public PosetilacLikova
{
public:
    RacunanjeObima( Lik& l )
        : Obim_(0)
        { l.PrihvatiPosetu(*this); }

    operator double() const
        { return Obim(); }
...
};

int main()
{...
    cout << "POVRSINA: " << RacunanjePovrsine(sl) << endl;
    cout << "OBIM: " << RacunanjeObima(sl) << endl;

    return 0;
}
```

7.6 Katalog obrazaca

Svaki put kada u razvoju softvera naiđemo na neki problem, koji je sličan nekom poznatom obrascu, onda možemo da razmotrimo primenu tog prepoznatog obrasca na način koji je prilagođen konkretnom slučaju. Da bi pronalaženje odgovarajućeg obrasca bilo lakše, prave se *katalozi obrazaca*. Da bi obrasci u katalozima mogli da se lako pronađu i prepoznaju, daju im se prepoznatljiva imena i klasifikuju se.

Ne postoje formalno ustanovljeni kriterijumi za odlučivanje o tome da li neko konceptualno rešenje zaslužuje da bude dokumentovano i predstavljeno u katalogu obrazaca. Kriterijumi u velikoj meri zavise od vrste kataloga, a prvenstveno od njegove namene i ciljane korisničke grupe. Katalog obrazaca može biti opšte namene, ali i vrlo specifične namene (na primer, katalog obrazaca za razvoj korisničkih interfejsa). U skladu sa tim razlikuju se i nivoi apstrakcije obrazaca, kao i njihov značaj. Neki obrazac može da bude značajan u nekom katalogu specifične namene, a da istovremeno nema dovoljno značajnu ulogu da bi se pojavio u nekom katalogu šireg domena, ali i obratno.

U skladu sa ranije navedenim važnim elementima obrazaca, svaki obrazac u katalogu se detaljno dokumentuje. Dokumentacija obrazaca može da obuhvata njihove različite karakteristike, zavisno od vrste i namene kataloga. U opisima obrazaca u katalogu se obično navode sledeće stavke:

- **ime obrasca**, koje sažeto izlaže suštinu obrasca. Veoma je važno da ime bude dobro odabrano, zato što ono ulazi u rečnik projektovanja;
- **alternativna imena** su druga imena pod kojima je poznat isti obrazac;
- **klasifikacija** omogućava lakše snalaženje među obrascima u katalogu. Obično se izvodi prema kontekstu primene, prema tehnici koja je dominantna u obrascu ili oboje;
- **namena** ukratko opisuje čemu obrazac služi, koji problem rešava, šta pojednostavljuje, kada se koristi i slično;
- **motivacija** za primenu opisuje doprinose obrasca; može da sažeto ukazuje na osnovne odnose elemenata obrazaca i da predstavlja uvod u detaljnije opise;
- **primenjivost** opisuje gde i kada se obrazac može da se upotrebi i koje probleme može da reši;
- **preduslovi za primenu** moraju da se navedu u okviru opisa obrasca, da bi se predupredile potencijalne greške u vidu neodgovarajuće primene obrasca;
- **logička struktura rešenja** se obično predstavlja nekom od dijagramskih tehnika i pratećim tekstom; uobičajeno je da se koriste dijagrami *UML*-a;
- **entiteti rešenja** su klase ili objekti koji imaju važnu ulogu u obrascu za projektovanje; među entitetima su i specifični pomoćni entiteti koje uvodi obrazac, ali i oni koji postoje nezavisno od primene obrasca, a na koje se obrazac odnosi;
- **saradnja** između učesnika u rešenju predstavlja opis tokova poruka između objekata obrasca; uobičajeno je da se opisuje dijagramom sekvenci *UML*-a;
- **posledice primene rešenja** su neophodne kao i preduslovi, zato što mogu da utiču na odlučivanje o primeni obrasca ali i da ukažu na potencijalne probleme ili dodatne koristi od primene obrasca;
- **implementacija** predstavlja načelno uputstvo za primenu obrasca, pri čemu u složenijim slučajevima može da se navede i više uputstava za različite okolnosti ili za različite programske jezike;
- **poznati primeri korišćenja** predstavljaju najneposredniji način da se potencijalnom korisniku obrasca predstave njegovo ponašanje i struktura, ali i da se prektično ilustruje njegov značaj;

- **primeri koda** predstavljaju konkretne primere implementacije obrasca;
- **pregled srodnih obrazaca** ukazuje na srodne obrasce u istom ili drugom katalogu obrazaca.

Ukratko ćemo da predstavimo katalog obrazaca izložen u knjizi „Obrasci za projektovanje“. Ovaj katalog je opšte namene i obuhvata najvažnije obrasce za projektovanje koji su autorima bili poznati u vreme pisanja knjige. Obrasci su klasifikovani prema nameni u tri grupe:

- gradivni obrasci (engl. *Creational Patterns*);
- strukturni obrasci (engl. *Structural Patterns*) i
- obrasci ponašanja (*Behavioral Patterns*).

U svakoj od ovih grupa obrasci su dalje podeljeni prema domenu primene, na one koji rešavaju problem uspostavljanjem ili iskorišćavanjem odnosa između klasa (tzv. *obrasci sa domenom klase*) i one koji u rešenju počivaju na odnosima između objekata (tzv. *obrasci sa domenom objekata*).

Gradivni obrasci predstavljaju rešenja za različite načine pravljenja novih objekata. Oni apstrahuju postupak pravljenja novih objekata i omogućavaju da se na uopšten način pristupi kako samom pravljenju tako i međusobnom povezivanju napravljenih objekata. Iako neki problemi mogu da se reše primenom različitih gradivnih obrazaca, oni se međusobno veoma razlikuju – ako im se u nekom slučaju poklope uslovi za primenu, onda im se značajno razlikuju posledice. Zajedničko za sve ove obrasce je da oni pokušavaju da od korisnika sakriju veći deo složenosti postupaka pravljenja i povezivanja napravljenih objekata. Od korisnika se sakrivaju međusobni odnosi objekata i klasa, a često i sami konkretni tipovi napravljenih objekata. Umesto toga, korisniku se stavljaju na raspolaganje samo odgovarajući (najčešće apstraktni) interfejsi za pravljenje, povezivanje i korišćenje objekata.

Ovi obrasci omogućavaju da se različite klase i objekti sa složenim ponašanjem grade na relativno jednostavan način sastavljanjem manjeg skupa osnovnih ponašanja. Oni tako enkapsuliraju složene aspekte pravljenja objekata.

Gradivni obrasci mogu da imaju domen klase ili domen objekata. Gradivni obrasci sa domenom klase uglavnom koriste nasleđivanje kao osnovni odnos među objektima koji se prave. Gradivni obrasci sa domenom objekta prepuštaju (delegiraju) pravljenje nekom drugom objektu.

U gradivne obrasce spadaju:

- Proizvodni metod (engl. *Factory Method*);
- Apstraktna fabrika (engl. *Abstract Factory*);

- Graditelj (engl. *Builder*);
- Prototip (engl. *Prototype*) i
- Unikat (engl. *Singleton*).

Strukturni obrasci rešavaju probleme organizovanja objekata i klasa u veće funkcionalne celine. Strukturni obrasci sa domenom klasa koriste nasleđivanje klasa i hijerarhije za povezivanje objekata i klasa. Strukturni obrasci sa domenom objekata neposredno povezuju objekte u veće celine. Zbog toga što su odnosi među klasama statički, a odnosi među objektima su dinamički i mogu da se menjaju tokom rada programa, strukturni obrasci sa domenom objekata se upotrebljavaju za pravljenje fleksibilnijih složenih struktura.

U strukturne obrasce spadaju:

- Adapter (engl. *Adapter*);
- Most (engl. *Bridge*);
- Sastav (engl. *Composite*);
- Proksi[35] (engl. *Proxy*).
- Dekorater (engl. *Decorator*);
- Fasada (engl. *Facade*) i
- Muva (engl. *Flyweight*).

Obrasci ponašanja se odnose na implementiranje algoritama ili raspodelu odgovornosti između objekata. Oni rešavaju različite probleme pomoću uspostavljanja odgovarajućih odnosa među klasama i objektima i kroz odgovarajuće načine razmenjivanja informacija (poruka) između objekata. Jedna od osnovnih namena im je da enkapsuliraju složenu prirodu kontrole i prebace složenost upotrebe na povezivanje objekata. Klasni obrasci ponašanja kao osnovno sredstvo za distribuiranje ponašanja među klasama koriste nasleđivanje. Sa druge strane, u objektnim obrascima ponašanja značajniju ulogu igra povezivanje pojedinačnih objekata.

U obrasce ponašanja spadaju:

- Lanac odgovornosti (engl. *Chain of Responsibility*);
- Komanda (engl. *Command*);

[35] Uobičajen prevod za termin *proxy* je *posrednik*, ali ovde to nije prihvatljivo, zato što postoji uzorak ponašanja koji se zove *Posrednik* (engl. *Mediator*). Da bi se izbegli nesporazumi, uobičajeno da se ovaj uzorak i na srpskom jeziku naziva *Proksi*.

- Interpretator (engl. *Interpreter*);
- Iterator (engl. *Iterator*);
- Posrednik (engl. *Mediator*) ;
- Podsetnik (engl. *Memento*);
- Posmatrač (engl. *Observer*);
- Stanje (engl. *State*);
- Strategija (engl. *Strategy*);
- Šablonski metod (engl. *Template Method*) i
- Posetilac (engl. *Visitor*).

U prethodnim odeljcima smo predstavili po jedan primer iz svake od tri grupe obrazaca: Unikat je gradivni obrazac, Sastav je strukturni obrazac i Posetilac je obrazac ponašanja.

7.7 Umesto zaključka

Imajući u vidu značaj obrazaca za projektovanje, čitaocima se preporučuje da pročitaju knjigu „Obrasci za projektovanje“ [*Gamma 1995*] i prouče obuhvaćene obrasce. Pored toga što nosi mnogo informacija o obrascima za projektovanje, čitaoci imaju priliku da se iz nje upoznaju i sa mnogo dobrih primera objektno-orijentisanog projektovanja i programiranja.

Napisano je još mnogo drugih knjiga o obrascima za projektovanje, čiji su autori pokušali da ovu temu predstave na još bolji ili razumljiviji način, ili da prepoznaju i ponude čitaocima nove obrasce. Na primer, Martin Fauler je napisao odličnu knjigu o obrascima za projektovanje u domenu arhitektura poslovnih aplikacija [*Fowler 2002*]. Nekoliko primera upotrebe obrazaca za projektovanje u programskom jeziku C++ je predstavljeno u [*Malkov 2007*].

Na internetu postoji mnogo izvora informacija o obrascima za projektovanje. Među njima može da se izdvoji veb lokacija Vinsa Hjustona [*Huston veb*].

8 - Agilni razvoj softvera

Primena nekog agilnog metoda ne znači da će ulagači dobiti šta žele.
To jednostavno znači da će moći da kontrolišu tim
da bi dobili najveću poslovnu vrednost uz najniže troškove.
Robet Martin

8.1 Agilne metodologije

Agilni razvoj softvera je familija metodologija razvoja softvera koja je nastala krajem poslednje decenije XX veka. Naziv je oblikovan 2001. godine, prilikom osnivanja *Agilnog saveza* (engl. *Agile Alliance*) [*Agile Alliance*] i formulisanja *Manifesta agilnog razvoja softvera*. Upotrebljavaju se i termini *agilne metodologije*, *agilni razvojni proces* i *agilni proces*.

Agilni razvoj softvera ima mnogo sličnosti sa OO metodologijama, ali ima značajno drugačiji pristup planiranju, u odnosu na klasične OO metodologije. Pri posmatranju odnosa agilnih i klasičnih OO metodologija moramo imati u vidu da većina agilnih metodologija pretpostavlja upotrebu osnovnih tehnika OO projektovanja i programiranja, ali da istovremeno većina elemenata agilnih metodologija može da se primenjuje i na razvojne projekte koji nisu striktno objektno-orijentisani.

Termini *agilni razvoj*, *rapidni razvojni ciklus*, *ekstremno programiranje* i mnogi drugi termini u vezi sa agilnim metodologijama su do danas postali veoma popularni. Posledica je da se veoma često upotrebljavaju i u sasvim pogrešnom kontekstu, za označavanje nečega što tek po nekim aspektima *liči* na agilni razvoj softvera. Zbog toga je dobro da najpre pokušamo da odgovorimo na pitanje: *Šta nije agilni razvoj softvera*?

Agilni razvoj softvera nije:

- odsustvo sistematičnosti i strukturnog pristupa;
- anarhično ili svojevoljno ponašanje članova tima;
- potpuno odsustvo dokumentacije;
- selektivna primena samo nekih od principa agilnog razvoja softvera[36];
- najlakši metod razvoja softvera;
- univerzalno rešenje za sve probleme;
- pravi način rada za neiskusne ili nedovoljno stručne programere i
- još mnogo toga što bi neki voleli da bude.

Iako neki principi agilnog razvoja softvera mogu da izgledaju nesistematično, pa čak i da stvore utisak da podstiču neuredan rad, to nije ni izbliza tako. Nekada je bilo mišljenja da su agilne metodologije površne, ali ni to ne stoji. Možda je najveći problem u tome što mnogi od principa agilnog razvoja izgledaju jednostavno za primenu, pa neki ove metodologije zbog toga smatraju za univerzalne i lako primenjive, što jednostavno nije tačno.

8.2 Manifest agilnog razvoja softvera

Osnivači Agilnog saveza su osnovnu ideju, koja stoji iza agilnih metodologija, izrazili objavljivanjem Manifesta agilnog razvoja softvera [*Agile Manifesto*]:

Otkrivamo bolje načine razvoja softvera
razvijajući ga i pomažući drugima u tome.
Kroz taj rad smo naučili da više vrednujemo:

Ljude i odnose među njima – od procesa i alata;
Softver koji radi – od iscrpne dokumentacije;
Saradnju sa klijentima – od pregovaranja oko ugovora i
Reagovanje na promene – od pridržavanja plana.

Odnosno, iako cenimo vrednosti na desnoj strani,
smatramo za vrednije one koje su na levoj.

[36] Moramo da primetimo da takva selektivna primena polako postaje uobičajena, koliko god da su se autori agilnih metodologija trudili da ukažu na njene slabosti.

Kada se ističe da su *ljudi i odnosi među njima* ispred *procesa i alata*, time se naglašava da su ljudi, kao članovi razvojnog tima, najvažniji i presudan činilac uspešnog razvoja. Ako je tim sastavljen od pojedinaca koji nisu dovoljno sposobni, onda projekat teško može da bude uspešan. Jednako je teško doći do cilja i ako su u timu sposobni pojedinci, ali su odnosi među njima loši. Naravno, procesi i alati su veoma važni i to ne sme da se previdi, zato što loš razvojni proces može i najbolje ljude da učini manje produktivnim, a dobar ih može učiniti još boljima. Slično važi i za dobre i loše alate. Međutim, ako se previše pažnje posvećuje procesima i alatima, onda to može da bude gore od potpunog odsustva alata, zato što može da stavi ljude u drugi plan. Posvećivanje pažnje ljudima i izgradnja kvalitetnog tima moraju uvek da budu važniji od izgradnje okruženja.

Pretpostavka da je *softver koji radi* važniji od *iscrpne dokumentacije* podseća nas na to da je cilj razvoja softver, a ne dokumentacija ili lepe želje. Mnogo je projekata imalo izvrsnu dokumentaciju, a da razvijeni softver nije radio – takva dokumentacija na kraju priče nije imala nikakvu vrednost. Bila bi velika greška ako bi neko ovu pretpostavku razumeo tako da ne mora da pravi dokumentaciju. Softver koji nije praćen odgovarajućom dokumentacijom je potpuna propast – kako za korisnike i vlasnike, tako i za one koji ga budu održavali. Kada bi, kojim slučajem, programski kod bio lak za čitanje i mogao da predstavlja jedino sredstvo komunikacije, onda niko ne bi ni izmišljao ni pravio druge vidove dokumentacije. Ali stvari ne stoje tako i zato nam je dobra i razumljiva dokumentacija neophodna. Suština je u tome da se odredi prava mera količine i preciznosti dokumentacije, kao i pravo vreme za njeno pisanje. Moramo da imamo u vidu da prerano napisana dokumentacija obično nije stabilna i da kasnije često mora da se temeljno prerađuje. Potrebno je da se pažljivo odmeri i obim dokumentacije, zato što se obimna dokumentacija veoma teško sinhronizuje sa naknadnim izmenama programa.

Treća pretpostavka agilnog razvoja softvera ističe da je *saradnja sa klijentom* mnogo važnija od *pregovaranja oko ugovora*. Jedan od najvećih problema pri izradi velikog softvera je blagovremeno i tačno procenjivanje troškova izrade. Zbog toga se početak softverskih projekata često odlikuje napornim „natezanjima“ izvođača i klijenata oko procenjene vrednosti i predviđenih rokova, kao i oko preciziranja i ugovaranja pojedinosti specifikacije softvera koji se razvija. Koliko god da se brzo razvija softverska industrija, složenost problema koji se rešavaju raste mnogo brže. Posledica toga je da je u mnogim slučajevima veoma teško da se pre započinjanja razvoja naprave tačne procene obima poslova, troškova i rokova isporuke.

Naravno, pregovaranje i ugovaranje su neophodni – to niko ne dovodi u pitanje. Međutim, kod svih agilnih metodologija se teži da se pri ugovaranju posla prvenstveno jasno i precizno odrede međusobni odnosi, a ne kompletan obim poslova. Ideja je da se tokom projekta, kroz međusobnu saradnju razvijalaca i klijenta, postepeno formulišu i zatim dogovaraju, projektuju i implementiraju manji delovi softverskog sistema, umesto da se ceo softver ugovara na samom početku.

Praktično sve agilne metodologije počivaju na nekom vidu iterativnog razvoja. Veličina iteracija se razlikuje između metodologija, ali se obično planira da iteracija traje nekoliko nedelja. Samo prva naredna iteracija se planira sasvim detaljno, dok se sve ostale, koje će doći kasnije na red, na početku sagledavaju samo okvirno.

Agilni razvoj softvera prihvata da je i poslovno okruženje klijenta vrlo verovatno agilno, što za posledicu ima veliku verovatnoću da se tokom razvoja softvera promene neki već iskazani stavovi klijenta. Mogu da se promene okolnosti poslovanja, da se pojave neki novi zadaci, pa i da se potpuno promene neki ranije oblikovani zadaci. Jedna od osnovnih pretpostavki je da je *reagovanje na promene* važnije od doslednog *praćenja plana*. Pretpostavka je da će promene *sigurno nastupiti*, samo je pitanje kada i kakve. Ako je zbog nekih promena potrebno da se menjaju planovi, agilne metodologije nam kažu da to onda neupitno i neodložno moramo da uradimo.

Planovi jesu važni i moraju da se prave i slede, ali je reagovanje na promene još važnije. U mnogim slučajevima upravo sposobnost reagovanja na promene može da presudno utiče na uspešnost projekta. Zato planiranje ne bi trebalo da ide daleko u budućnost, što je još jedan motiv za iterativni razvoj. Detaljno planiranje je neophodno, ali samo za kratak period, zato što detaljni planovi ne mogu da ostanu stabilni tokom dužeg perioda. U praksi se to obično svodi na detaljno planiranje samo jedne naredne iteracije. Jedino što bi trebalo da postoji zacrtano na duge staze jeste *vizija* (tj. prepoznati glavni ciljevi razvoja), mada čak i ona može da se promeni.

Veoma je važno da se ne previdi napomena sa kraja manifesta – tradicionalne vrednosti starijih metodologija razvoja softvera se *ne odbacuju*, već opstaju i dalje, ali se prepoznaju neke nove vrednosti kao značajnije. Agilne metodologije su izgrađene na svim navedenim vrednostima, uz stavljanje akcenta na one *sa leve strane*.

8.3 Principi agilnog razvoja softvera

Agilni razvoj softvera počiva na dvanaest principa, koji su izvedeni iz manifesta:

Najviši prioritet je zadovoljiti klijenta kroz brzo i neprekidno isporučivanje vrednog softvera. Razvojni projekat je započet zato što klijent ima neke potrebe i ciljeve. Ispunjavanje tih potreba i ciljeva je osnovni smisao postojanja projekta, pa zato ima i najviši prioritet. Suština agilnog razvoja je u iterativnosti, tj. u razvijanju i isporučivanju jednog po jednog dela projekta, tako da svaki deo projekta zadovolji neke ciljeve i potrebe klijenta.

Uvek otvoreno prihvatati promene, čak i u kasnim fazama razvoja. Agilni razvoj softvera uvažava promene kao sredstvo ostvarivanja kvaliteta. Ako je nešto promenjeno u zahtevima ili u domenu, onda softver to mora da isprati, zato što u suprotnom projekat neće moći da ispuni svoje ciljeve.

Isporučivati softver koji radi, što češće, sa intervalom od par nedelja do par meseci, pri čemu se kraći intervali više cene. Već smo istakli da je iterativnost u suštini agilnog razvoja. Što su intervali kraći, to su isporuke i kontakti sa klijentom redovniji, čime se postiže da se međusobni odnos i saradnja bolje razvijaju. Sa druge strane, svaka isporuka zahteva dodatan utrošak vremena, pa bi zato trebalo pronaći dobar balans po pitanju veličine pojedinačnih iteracija.

Poslovni ljudi i razvijaoci moraju svakodnevno da sarađuju na projektu. Ovaj princip je u duhu prihvatanja promena i kratkih iteracija. Nema boljeg načina da se na vreme primete greške u projektu nego što je tesna saradnja i stalna komunikacija sa klijentom.

Zasnivati projekat na motivisanim pojedincima. Pružiti im okruženje i potrebnu podršku i imati poverenja u njih da će obaviti posao. Pri predstavljanju manifesta smo istakli da ni najbolja metodologija i najnoviji alati ne mogu da reše probleme ako im članovi tima nisu dorasli. Agilne metodologije obično razmatraju propisana pravila više kao preporuke nego kao propise – ako članovi tima smatraju da je nešto bolje da se uradi drugačije, onda se tako i radi.

Najefikasniji način za prenošenje informacija timu i unutar tima je razgovor licem u lice. Neformalna komunikacija je često brža i sadržajnija od pisane komunikacije. Videćemo da se jedan broj praksi ekstremnog programiranja odnosi na omogućavanje i promovisanje neposredne komunikacije.

Softver koji radi je osnovno merilo napretka. Svi planovi i lepe želje su beznačajni ako ne dobijemo kao rezultat funkcionalan softver. Napredak može da se meri i brojem modela, planova i lepih želja, ali broj dovršenih i isporučenih celina je ipak najvažniji.

Agilni razvoj softvera promoviše uzdržani razvoj. Sponzori, razvijaoci i korisnici bi trebalo da budu u stanju da neograničeno dugo održavaju ujednačen ritam. Prekovremeni rad narušava raspoloženje članova tima i odnose u timu, a istovremeno stvara uslove za nastajanje povećanog broja grešaka, pa bi trebalo da se izbegava.

Neprekidno posvećivanje pažnje tehničkoj doteranosti i dobrom dizajnu podiže agilnost. Agilnost ne podrazumeva žurbu, naprotiv. Svaki korak je potrebno da bude dobro promišljen, a svaka promena da bude dobro uklopljena u postojeću strukturu. Dobar dizajn olakšava i ubrzava proširivanje i menjanje programa.

Jednostavnost, kao umetnost maksimizovanja količine posla koji se ne obavlja, je od suštinskog značaja. U agilnom razvoju *jednostavnost* podrazumeva da se pri planiranju i implementiranju ne posvećuje pažnja elementima i aspektima koji će *možda* biti potrebni. Tako se štedi na vremenu pri planiranju, razvoju ali i održavanju.

Najbolje arhitekture, zahtevi i projekti potiču iz samoorganizovanih timova. Timovi moraju da imaju visok nivo samostalnosti. Zbog toga moraju da imaju zastupljene sve specijalnosti razvijalaca koje su potrebne na projektu.

Tim mora da u redovnim intervalima sagledava svoj rad i mogućnosti unapređivanja svog ponašanja i efikasnosti. Tim uvek mora da traži načine za sopstveno unapređivanje. Jedan od osnovnih preduslova za napredovanje je uočavanje slabosti u ostvarenim rezultatima i u postojećoj organizaciji rada.

Navedni principi agilnog razvoja softvera predstavljaju vid uputstva za konkretizaciju manifesta i uglavnom neposredno slede iz pretpostavki manifesta. Svaka konkretna agilna metodologija može da uvodi nove pretpostavke i dodaje nove specifične principe. Pored toga, svaka konkretna metodologija propisuje metode i tehnike koje služe za praktično ostvarivanje tih principa. Zbog toga nećemo da se na ovom mestu mnogo zadržavamo na principima. Odgovarajućim pitanjima ćemo da posvetimo nešto više pažnje pri razmatranju konkretnih praksi ekstremnog programiranja, koje su oblikovane radi ostvarivanja navedenih principa.

8.4 Primeri agilnih metodologija

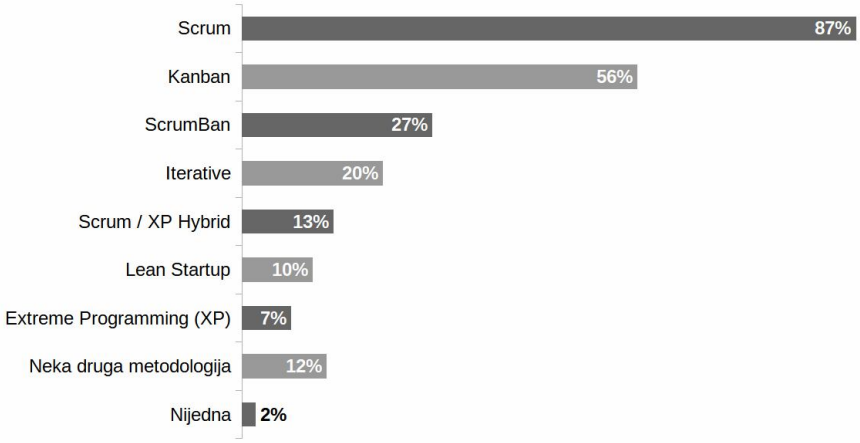
Agilni razvoj softvera je osnova za više različitih metodologija. Mnoge od njih se koriste u razvojnim timovima širom sveta. Neke od poznatijih su:

- Ekstremno programiranje (engl. *XP – Extreme Programming*);
- Skram (engl. *Scrum*);
- Agilno modeliranje (engl. *AM – Agile Modeling*);
- Agilan objedinjen proces (engl. *AUP – Agile Unified Process*) i
- Otvoren objedinjen proces (engl. *OpenUP – Open Unified Process*).

Predstavićemo detaljnije osnovne elemente i karakteristike metodologija *Ekstremno programiranje* i *Skram*. Ekstremno programiranje je jedna od starijih agilnih metodologija. Predstavlja „relativno kompletnu“ razvojnu metodologiju. Sa druge strane, Skram bi pre mogao da se nazove metodološkim okvirom, koji je otvoren za primene tehnika iz drugih metodologija. U praksi se veoma često kombinuju Skram i tehnike Ekstremnog programiranja.

Zanimljivo je da kada se Ekstremno programiranje poredi sa klasičnim razvojnim metodologijama, pa i savremenim kompleksnim metodologijama (na primer, RUP – *Rational Unified Process*), onda ne izgleda baš kompletno, već više liči na koncept koji nije do kraja razrađen. Može se naići i na stav da je XP „samo metodološki okvir“. Razlog je u tome što se mnoga pitanja ostavljaju razvijaocima na relativno slobodno tumačenje i odlučivanje, dok su neke druge metodologije (a posebno klasične) daleko

striktnije i manje toga ostavljaju nedorečeno. Međutim, kada se Ekstremno programiranje uporedi sa nekim drugim agilnim metodologijama, na primer sa Skramom, onda se vidi da je ipak u pitanju relativno kompletna metodologija, koja ima i više elemenata i daleko razrađenije elemente (ideje, principe, metode i drugo) nego Skram. Može se reći da je u pogledu složenosti i kompletnosti mnogo manja razlika između klasičnih metodologija i Ekstremnog programiranja, nego između Ekstremnog programiranja i savremenih metodoloških okvira kao što je Skram.



Slika 25 – Pregled zastupljenosti agilnih metodologija

Savremena praksa naginje povećanoj upotrebi jednostavnijih metodoloških okvira, kao što su Skram ili Kanban. Istraživanje [*State of agile 2022*] (*Slika 25*) pokazuje da velika većina anketiranih razvijalaca koristi Skram ili kombinaciju Skrama i neke druge metodologije. Prema istom istraživanju, tek manji broj razvijalaca je naglasio da koristi Ekstremno progamiranje. Međutim, kada se ti rezultati uporede sa rezultatima istraživanja o upotrebi agilnih razvojnih praksi [*State of agile 2020*] (*Slika 26*), onda dolazi do izražaja njihova neusaglašenost – iako relativno mali broj razvijalaca upotrebljava Ekstremno programiranje, ipak relativno veliki broj njih koristi prakse Ekstremnog programiranja. Ovaj problem ćemo podrobnije da razmotrimo u narednim odeljcima.

Ekstremno programiranje i Skram su prvenstveno namenjeni pojedinačnim timovima. Slično važi i za sve ostale agilne metodologije koje se pominju u ovom poglavlju. Ako su razvojni projekti dovoljno obimni i složeni da zahtevaju angažovanje većeg broja timova, onda je neophodno da se u širem razvojnom okruženju obezbede uslovi za međusobnu komunikaciju i saradnju timova, kao i složenija infrastruktura za njihov rad. Neki od problema koji tada moraju da se reše obuhvataju i geografsku dislociranost timova, veliki ukupan broj razvijalaca (obično se računa od 50 pa do više stotina), usklađivanje ciljeva razvoja i razvojnih ciklusa po timovima i drugo. Radi toga se oblikuju posebni razvojni okviri, tzv. *okviri za velike*

projekte (engl. *large scale framework*). Ako počivaju na osnovnim agilnim principima, onda su to *agilni okviri za velike projekte* (engl. *large scale agile frameworks*).

Agilni okviri za velike projekte su prvenstveno namenjeni koordinaciji i saradnji više timova. Oni se uglavnom ne bave radom u pojedinačnim timovima, već se pretpostavlja da se za to koristi neka druga metodologija. Kao primere agilnih okvira za velike projekte možemo da izdvojimo *Disciplined Agile Delivery* (*DAD*) [*Ambler 2012*], *Large-Scale Scrum* (*LeSS*) [*Larman 2016*] i *Scale Agile Framework* (*SAFe*) [*Knaster 2020*]. *DAD* je relativno fleksibilan okvir koji dopušta da svaki tim bira svoju metodologiju, sve dok se aktivnosti odvijaju u zadatim okvirima. Nasuprot tome, *LeSS* i *SAFe* propisuju da pojedinačni timovi moraju da primenjuju Skram.

8.5 Ekstremno programiranje

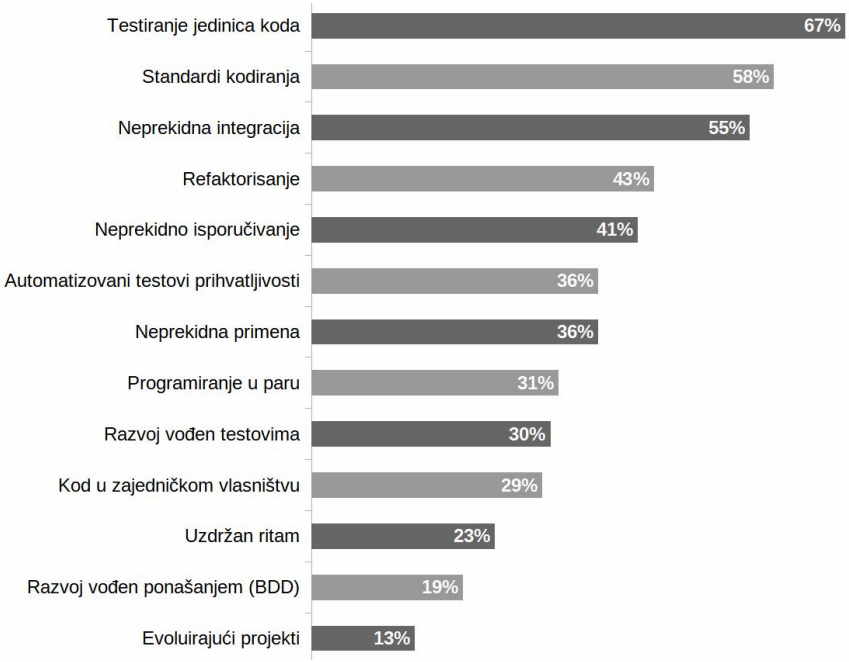
Ekstremno programiranje (EP) je predstavio Kent Bek 1999. godine u nekoliko članaka i izlaganja na konferencijama [*Back 1999*]. Motivacija i ciljevi Ekstremnog programiranja se praktično poklapaju sa onim što je kasnije formulisano kao Manifest agilnog razvoja.

Priroda Ekstremnog programiranja se najneposrednije sagledava razmatranjem osnovnih pojmova sa kojima se susrećemo pri razmatranju razvojnog ciklusa. Oni se nazivaju i *strukturnim elementima* Ekstremnog programiranja. Drugu celinu metodologije predstavljaju *prakse* Ekstremnog programiranja. Prakse uobličavaju i predstavljaju principe, tehnike i metode na kojima počiva metodologija. Predstavićemo ih u narednom odeljku. Strukturne elemente i prakse ćemo da predstavimo u narednim odeljcima.

Različita istraživanja uglavnom pokazuju da je metodologija Ekstremno programiranje danas relativno slabo zastupljena u praksi. Međutim, njen stvarni značaj je realno daleko veći nego što bi se na osnovu površnih istraživanja moglo zaključiti. Rezultati istraživanja o primeni metodologija su često neprecizni i diskutabilni. Njihovom površnom analizom bi moglo da se ustanovi da razvojni timovi danas najčešće ne primenjuju dosledno *nijednu* metodologiju (iako tvrde drugačije), već da samo primenjuju neke izabrane tehnike, za koje je u okviru tima procenjeno da najviše odgovaraju konkretnom projektu i konkretnom organizacionom modelu po kome tim funkcioniše. Ali kada se pogleda koje su to tehnike, vidi se da je najveći broj njih potekao iz EP ili je bar imao značajno mesto u okviru te metodologije.

Kao i svaka druga metodologija i Ekstremno programiranje ima i zagovornike i protivnike. Protivnici EP obično smatraju da jedna ili više praksi nisu u skladu sa ciljevima uvođenja metodologije, tj. sa efikasnom i kvalitetnom izradom softvera. Zagovornici EP, sa druge strane, često ističu da se nijedna od praksi ne sme posmatrati, primenjivati pa ni ocenjivati sama za sebe, već da sve one ostvaruju svoj planirani cilj tek ako se primenjuju sve skupa, zato što se neke njihove slabosti prevazilaze primenom drugih praksi. U skladu sa tim se preporučuje da se ne vrši

odabir poželjnih praksi, već da se uvek primenjuju *sve zajedno*. Ipak, kao što smo već videli, istraživanja pokazuju da se u realnim primenama najčešće ne radi tako i da razvojni timovi biraju i primenjuju samo neke od praksi, za koje sami procene da bi mogle da najviše doprinesu njihovom specifičnom razvojnom procesu.



Slika 26 – Pregled zastupljenosti agilnih praksi

Imajući u vidu prethodno navedeno, važno je da naglasimo da predstavljanje Ekstremnog programiranja u ovoj knjizi ne predstavlja istovremeno i otvorenu preporuku za primenu ove metodologije, već je pre sugestija da je izučavanje agilnih metodologija najbolje započeti upravo od upoznavanja Ekstremnog programiranja. Prvi razlog za to je što je EP jedna od najkompletnijih i najuticajnijih agilnih metodologija. Drugi razlog je što će nam poznavanje tehnika EP biti od velike pomoći pri upoznavanju i primenjivanju bilo koje druge agilne metodologije, zato što praktično sve tehnike EP nalaze svoje mesto u savremenim razvojnim timovima, iako u možda donekle izmenjenom ili prilagođenom obliku.

8.5.1 Strukturni elementi Ekstremnog programiranja

Struktura Ekstremnog programiranja (EP) može da se opiše kroz osnovne elemente metodologije, koji se tiču načina organizovanja posla i uređivanja razvojnog ciklusa. U osnovne strukturne elemente EP se obično ubrajaju:

- Razvojni ciklus;
- Korisničke celine;
- Izdanja;
- Iteracije;
- Zadaci i
- Testovi.

Razvojni ciklus

Jedan od osnovnih doprinosa EP je drastično skraćivanje razvojnog ciklusa. Već smo videli da je to zajedničko za praktično sve agilne metodologije. Ono što je ovde specifično je podela razvojnih ciklusa na dve osnovne vrste: iteracije i izdanja.

Razvojni ciklus projekta započinje planiranjem korisničkih celina, nakon čega se ulazi u prvo *izdanje*. Jedno izdanje može da se sastoji od više *iteracija*, a po njegovom završetku se započinje novo izdanje ili se projekat završava.

O planiranju korisničkih celina, izdanjima i iteracijama će biti više reči u narednim odeljcima, kao i u okviru opisa prakse *Igra planiranja*.

Korisničke celine

Korisničke celine (ili *korisničke priče*, engl. *user stories* ili često samo *stories*) predstavljaju okvirnu specifikaciju zahteva. Korisničke celine imaju sličnu ulogu kao *slučajevi upotrebe* u OO metodologijama i *UML*-u, ali se u mnogo čemu od njih razlikuju. Piše ih klijent, pa njihovi opisi obično nemaju tehničku preciznost kao slučajevi upotrebe. Naprotiv, opisi korisničkih celina su često vrlo kratki.

Korisničke celine služe kao relativno površne reference na stvarne zahteve, koji će se detaljnije razmatrati tek kada dođu na red za implementaciju. Osnovna namena korisničkih celina je da pomognu pri pravljenju okvirnog plana i praćenju toka projekta. One predstavljaju osnovne elemente za pravljenje planova iteracija. Zbog toga im se obično dodeljuju prioriteti i grube procene resursa potrebnih za implementaciju. Pošto one nisu dovoljno detaljne, a uz to mogu i da se menjaju tokom vremena, na osnovu njih ne mogu da se prave tačne procene.

Teži se da se korisničke celine definišu relativno brzo na početku projekta, pa se njihovom inicijalnom oblikovanju ne posvećuje mnogo vremena. Na primer, Bek procenjuje da je za projekat obima 10 čovek-godina dovoljno da se jedan mesec posveti inicijalnom ustanovljavanju korisničkih celina. Naravno, one se kasnije u toku rada na projektu mogu i menjati i dodavati ili uklanjati.

Iako se ustanovljavaju u relativno kratkom roku, ipak se očekuje da svaka korisnička celina ima neke bitne elemente. Pre svega, svaka korisnička celina mora da bude jasna i proverljiva – tj. mora da obuhvata razumljivo i nedvosmisleno iskazan poslovni cilj koji bi korisničkom celinom trebalo da se ostvari, kao i

kriterijume kojima će moći da se ustanovi da li je ona dovršena i da li je cilj ostvaren ispravno i dovoljno kvalitetno.

U idealnom slučaju korisnička celina bi trebalo da može da se implementira u relativno kratkom vremenskom periodu, odnosno u okviru jedne iteracije. Ako to nije slučaj, onda se korisnička celina (prilikom planiranja iteracija) prevodi u skup manjih *zadataka* tako da svaki zadatak može da stane u jednu iteraciju.

Izdanja

Izdanje je osnovni deo razvojnog ciklusa, čiji je cilj da se izabrani skup korisničkih celina implementira i ako je moguće pusti u produkciju. Izdanje može da se planira na dva osnovna načina:

- najpre klijent izabere prioritetne korisničke celine i zatim razvojni tim proceni potrebno angažovanje i okvirno trajanje izdanja, ili
- najpre razvojni tim u dogovoru sa klijentom definiše trajanje izdanja i proceni raspoložive razvojne resurse, a zatim klijent (uz konsultovanje sa razvojnim timom) izabere korisničke celine koje mogu da se implementiraju u zadatim okvirima.

Prvi pristup više liči na klasičan pristup razvoju softvera, dok je drugi karakterističan za agilni pristup i poželjniji u okviru Ekstremnog programiranja. O drugom pristupu će biti više reči u predstavljanju prakse *Igra planiranja*.

U svakom slučaju, za jedno izdanje se skup korisničkih celina obično bira tako da predstavlja jednu veću zaokruženu celinu, koja može klijentu da pruži veoma jasnu i prepoznatljivu korist. Kada je god moguće, teži se da izdanje bude ne samo jedinica razvoja nego i jedinica puštanja u produkciju, tj. da korisničke celine koje se dovrše u okviru izdanja mogu da uđu u produkciju, bilo odmah posle završetka izdanja bilo nakon što prođe period testiranja isporučenog izdanja od strane klijenta.

Jedno izdanje ne bi trebalo da traje više od nekoliko meseci. Ono se sastoji od više iteracija. Sastavni deo planiranja izdanja je planiranje iteracija, pri čemu se okvirno planira koliko će biti iteracija i šta će od izabranih celina da ide u koju iteraciju, ali se na početku izdanja detaljno planira samo prva iteracija. U toku jednog izdanja mogu da se menjaju planirane iteracije. Iako nije zabranjeno da se dodaju nove ili da se odustaje od planiranih iteracija, obično se teži da se promene planova tokom izdanja ograničavaju na sadržaj planiranih iteracija, tako da se njihov broj ne menja.

Iteracije

Iteracija je manji razvojni ciklus. Kao što se ceo projekat deli na izdanja, tako se svako izdanje deli na iteracije. Cilj svake iteracije bi trebalo da bude implementacija jednog dela skupa korisničkih celina koje su izabrane za tekuće izdanje.

Izbor korisničkih celina za iteraciju se odvija na sličan način kao i u slučaju izdanja, ali razvojni tim često ima nešto veći uticaj na njihov izbor nego u slučaju izdanja. Po izboru celina, razvojni tim ih prevodi u zadatke, a ako je neophodno (zbog neslaganja obima celine i obima iteracije) vrši i odabir zadataka koji će ući u iteraciju. Zatim se programeri raspoređuju po zadacima za čiju su implementaciju zaduženi. Nakon što završe svoje zadatke, programerima se dodeljuju novi zadaci, predviđeni za tu iteraciju.

Uporedo sa tokom razvoja, klijent se bavi pravljenjem testova prihvatljivosti. Isporučeni testovi se ugrađuju u projekat i izvode na implementiranim zadacima i celinama.

Jedna iteracija bi trebalo da traje najviše nekoliko nedelja. Na njenom početku se detaljno analiziraju izabrane celine i zadaci. Rezultat rada iteracije u načelu može da ide i u produkciju, ali se to radi ređe nego u slučaju izdanja.

Zadaci

Zadaci predstavljaju elementarne delove korisničkih celina. Teži se da se zadaci oblikuju tako da jedan zadatak može da se završi za nekoliko sati. Naravno, to nije uvek moguće, pa neki zadaci mogu da budu i značajno većeg obima.

Rad na zadatku počinje formiranjem para (praksa *Programiranje u paru*) i kratkim sastankom (oko 15 minuta) sa klijentom i programerima koji rade na neposredno povezanim zadacima. Taj sastanak bi trebalo da ima za rezultat početni kratak ali sadržajan spisak testova koji bi morali da uspešno prolaze da bi se zadatak smatrao završenim (videti prakse *Testovi prihvatljivosti* i *Razvoj vođen testovima*). Nakon podele korisničke celine na zadatke, može da se desi da neki zadaci imaju strogo tehnički karakter i da nemaju neposrednog dodira sa klijentom. U takvom slučaju nema potrebe da sastanku prisustvuje klijent, već se umesto njega obično uključuje vođa tima.

Cilj rada na jednom zadatku je ispunjavanje definisanih testova. Tokom rada se obično prave i proveravaju i novi detaljniji testovi (praksa *Razvoj vođen testovima*).

Testovi

Pored specifičnog oblika razvojnog ciklusa, moglo bi se reći da su testovi ključni element EP-a. Testiranje softvera je uvek postojalo, ali je praktično tek sa EP postalo uobičajena praksa na svim nivoima razvoja i puštanja softvera u rad. Danas se raspoznaje veliki broj različitih vrsta testova i oni se sistematski projektuju i sprovode da bi se obezbedio visok kvalitet završnog proizvoda.

Testovi se obično dele na one koje pravi klijent i one koje prave programeri. Testovi koje oblikuje klijent[37] služe da se proveri da li je proizvod (ili deo proizvoda) završen u skladu sa očekivanjima. Takve testove obično nazivamo testovima prihvatljivosti i oni služe da provere ispravnost i kvalitet proizvoda ili neke njegove komponente. Razmotrićemo ih kroz predstavljanje prakse *Testovi prihvatljivosti*.

Drugu grupu testova prave programeri i njihov osnovni cilj je da testiraju pojedinačne strukturne elemente softvera. Na najnižem nivou imamo testove jedinica koda, koji služe da testiraju rad pojedinačnih funkcija, metoda, klasa ili paketa. Iznad toga imamo testove integracije ili testove sistema, koji na tehničkom nivou proveravaju ispravnost rada većih funkcionalnih celina softvera, kao što su moduli, komponente, servisi i slično. Sve ove testove prave i izvršavaju programeri i obično ih objedinjeno razmatramo u okviru prakse *Razvoj vođen testovima*.

8.5.2 Prakse Ekstremnog programiranja

Specifičnosti Ekstremnog programiranja (EP) su iskazane kroz tzv. *prakse* Ekstremnog programiranja. Prakse EP predstavljaju tehnike, metode i principe ove metodologije i opisuju kako se u njoj ostvaruju opštiji principi agilnog razvoja. Može se reći da prakse EP predstavljaju *sredstvo* ostvarivanja principa agilnog razvoja. Većina praksi nije potpuno nova, niti su osmišljene samo zbog EP, ali sve skupa predstavljaju uniju onoga što je u tom periodu već počelo da se prepoznaje kao kvalitativna promena u pristupu razvoju softvera i što je Kent Bek pokušao da oblikuje kao celovitu metodologiju.

> *Prakse su oblikovane da funkcionišu zajedno*
> *i pokušaj da se upozna jedna od njih ubrzo vodi do ostalih.*
>
> *Kent Bek*

Spisak praksi Ekstremnog programiranja se vremenom menjao, pa se u različitim radovima i knjigama može naići na različita prilagođavanja. Uglavnom se svi autori slažu oko 12 osnovnih praksi, ali većina uvodi i neke dodatne prakse. Razlike nastaju usled toga što neki autori u prakse svrstavaju i strukturne elemente metodologije (neke ili sve), dok se kod nekih autora neke prakse spuštaju na nivo *preporuke*.

Prakse se obično klasifikuju u tri grupe, prema aspektu razvojnog procesa na koji se pretežno odnose. Tu ponekad ima i preklapanja. Na primer, Vejk [*Wake 2000*] predlaže sledeću klasifikaciju praksi (sa ponavljanjem):

[37] Primetimo da klijent relativno retko pravi testove na tehničkom nivou, ali je zato obično zadužen za njihovo konceptualno definisanje.

- Procesne prakse:
  - Klijent je član tima;
  - Testiranje;
  - Kratki ciklusi;
  - Igra planiranja;
- Timske prakse:
  - Kolektivno vlasništvo;
  - Neprekidna integracija;
  - Metafora;
  - Standardi kodiranja;
  - Uzdržan ritam;
  - Programiranje u paru;
  - Kratki ciklusi;
- Programerske prakse:
  - Jednostavan dizajn;
  - Testiranje;
  - Refaktorisanje i
  - Standardi kodiranja.

U nastavku ćemo predstaviti prakse koje nismo opisali u obliku strukturnih elemenata. U odnosu na predstavljenu klasifikaciju uvešćemo par manjih izmena: praksu Testiranje ćemo da predstavimo kao dve odvojene prakse – procesnu praksu Testiranje i programersku praksu Razvoj vođen testovima; predstavićemo i praksu Otvoren radni prostor, koju je Bek izvorno uveo; predstavićemo kao posebnu praksu i Praćenje toka razvoja[38].

Procesne prakse

Klijent je član tima

Ekstremno programiranje zastupa stav da je za kvalitetnu saradnju klijenta i razvijalaca neophodno da klijenti i razvijaoci rade zajedno. Da bi to bilo što efikasnije, zahteva se da bar jedan predstavnik klijenta bude stalno prisutan, kao

[38] Praćenje toka razvoja se u većini knjiga detaljno razmatra kao jedna od tehnika koje čine rad na razvojnim ciklusima. Kako ovde nemamo prostora da detaljnije izlažemo tok rada na iteracijama i izdanjima, a ova tehnika zaslužuje da bude pomenuta, predstavićemo je u vidu prakse.

praktično ravnopravan član razvojnog tima. Među najvažnije ciljeve, koji bi time trebalo da se postignu, spadaju:

- ubrzano dobijanje odgovora na pitanja koja se pojavljuju tokom razvoja;
- neformalno saopštavanje potreba za izmenama;
- podizanje poverenja klijenta u razvojni tim i
- intenzivno uključivanje klijenta u definisanje testova prihvatljivosti.

Oko ove prakse se često lome koplja. Njeni protivnici smatraju da ona može nepovoljno da utiče na efikasnost rada. Najpre, stalna raspoloživost predstavnika klijenta može da podstakne tim da pitanja postavlja neplanski, bez mnogo prethodnog razmišljanja, što može da vodi tome da se raspoložive informacije sagledavaju samo parcijalno, ni dovoljno duboko ni dovoljno široko. Iako to na prvi pogled može da izgleda čudno, na veliki broj pitanja u razvoju precizniji i kvalitetniji odgovor može da da razvojni tim nego klijent. Za klijenta treba ostavljati prvenstveno ona pitanja na koja *samo on* može da pruži odgovor.

Drugi potencijalno problematičan aspekt uključivanja klijenta u tim je što mu se tako možda preterano olakšava da saopštava želje za izmenama. To može da proizvede neumereno veliki broj zahteva za izmenama, tako da nastupi lavina izmena koja može da uspori, pa čak i da zaustavi razvoj. Postoji više metoda da se izađe na kraj sa ovim problem. Na primer, jedan način je da se u završnom delu perioda razvoja (na primer, tokom poslednjih nekoliko dana ili nedelja, a prema dužini konkretnog perioda razvoja) zabrani menjanje specifikacija i zadataka od strane klijenta.

Testovi prihvatljivosti

Testovi prihvatljivosti predstavljaju primarni oblik dokumentovanja merila ispravnosti i kvaliteta implementacije korisničke celine. Umesto da se posebno prave specifikacije ponašanja i testovi za njihovo proveravanje, pojedinosti o korisničkim celinama se izražavaju upravo u obliku opisa test-slučajeva i očekivanog ponašanja softvera. Testove prihvatljivosti određuje, a po mogućnosti i tehnički definiše klijent. Pišu se ili neposredno pre implementacije (tj. pri planiranju odgovarajuće iteracije) ili tokom implementacije korisničke celine. U svakom slučaju, moraju da budu spremni pre kraja iteracije, da bi mogli da se izvršavaju u vreme kada se implementacija privodi kraju.

Ako razvojno i izvršno okruženje to dopuštaju, testovi prihvatljivosti se pišu na nekom skript-jeziku, koji omogućava da se testovi kasnije automatizovano ponavljaju. Jednom kada test uspešno prođe, on se dodaje u kolekciju položenih testova. Položeni testovi prihvatljivosti bi trebalo da se ponavljaju pri završetku svake iteracije ili izdanja. Obično nema razloga da se ponavljaju toliko često kao i testovi jedinica koda (tj. svaki put pri izgradnji sistema, po nekoliko puta dnevno). Na taj

način se obezbeđuje da kada se zahtev jedanput zadovolji, on više ne bude doveden u pitanje kasnijim razvojem.

Kratki ciklusi

Za Ekstremno programiranje je uobičajena redovna isporuka softvera. Obično se nova verzija softvera koji radi isporučuje na svake 2 nedelje. Pri tome se ne misli na razne prototipove i demonstracione verzije, već na trenutno raspoložive produkcione verzije.

Učestala isporuka ne mora da znači da se svaki put zamenjuje verzija softvera na produkcionim serverima i radnim stanicama klijenta. Uobičajeno je da se isporučene verzije kroz nekoliko iteracija samo isprobavaju u okruženjima za testiranje, pre nego što softver dostigne dovoljnu zrelost da se prenese i u produkciono okruženje. Razvojni ciklus EP razlikuje *izdanja* i *iteracije*.

Iteracija je kratak ciklus (obično oko 2 nedelje) tokom koga se radi na izabranom skupu korisničkih celina. Primarni cilj iteracije je dovršavanje korisničkih celina i omogućavanje klijentu da redovno dobija na testiranje nove verzije koda, kako bi mogao da ima neposredan uvid u dinamiku razvoja i da prilagođava postojeće zahteve novim okolnostima. Iako to nije osnovna namena iteracije, verzija softvera koja predstavlja rezultat iteracije može i da se postavi u produkciono okruženje.

Izdanje je duži ciklus razvoja. Obično obuhvata nekoliko iteracija, recimo 4 do 6. Izdanja ne moraju da budu iste veličine i mogu da se planiraju i prema nekim datumima koji su značajni za poslovanje klijenta. Jedan od osnovnih ciljeva izdanja je zaokruživanje većih celina i dovršavanje verzije softvera koja bi trebalo da ide u produkciju. Za razliku od iteracija, koje mogu ali se uglavnom ne postavljaju u produkciono okruženje, izdanja ne moraju, ali najčešće idu u produkciju.

Kratke iteracije povoljno utiču na stvaranje poverenja između razvojnog tima i klijenta. Ustaljen i dobar ritam razvoja pruža osećaj sigurnosti obema stranama. Klijent može da računa na ritam u kome će softver biti unapređivan i uspeva da vidi realan napredak. Razvojni tim redovno dobija informacije o tekućim zahtevima, ali i povratne informacije od klijenta, što mu omogućava da ima blagovremeni uvid u eventualno mimoilaženje toka razvoja i planova klijenta, kao i u obim i složenost poslova koji tek predstoje.

Igra planiranja

Iako naziv prakse može da sugeriše da se ovde radi o igri ili nekom neozbiljnom pristupu, naravno da stvari stoje drugačije. Termin *igra* ukazuje na široku uključenost razvojnog tima i predstavnika klijenta u proces planiranja i na intenzivnu komunikaciju koja je pri tome neophodna.

EP jasno ističe podelu odgovornosti u procesu planiranja između klijenta i razvojnog tima. Klijenti odlučuju o značajnosti različitih karakteristika razvijanog softvera. Oni određuju šta je potrebno da se napravi, a šta nije, kao i o prioritetima

karakteristika. Potrebne karakteristike se evidentiraju u obliku korisničkih celina (engl. *user stories*). Korisničke celine se najpre evidentiraju sasvim sažeto, a tek kasnije, kada se približi vreme implementacije, razmatraju se do pojedinosti. Na primer, jedna korisnička celina može da se opiše rečenicom „Korisnik može da vrati kupljeni proizvod u određenom roku." Takav opis je dovoljan da svi članovi tima imaju u vidu da će takva funkcionalnost biti implementirana (u nekom trenutku), ali ne odvlači pažnju sa drugih delova softvera, koji su u tom trenutku prioritetniji. U detaljniju analizu problema se ulazi tek kada se približi trenutak implementacije korisničke celine, tj. kada je već implementirana većina prioritetnijih celina.

Odmah po određivanju korisničke celine, pre detaljne analize, razvojni tim daje grubu procenu troškova njene implementacije. Na osnovu takvih grubih procena se kasnije bira koje celine mogu da uđu u sastav planiranih iteracija.

Planiranje iteracija započinje određivanjem vremenskih i budžetskih okvira iteracije, gde „budžet" podrazumeva broj radnih sati koje razvojni tim u datom periodu može da posveti projektu. Okvire iteracije određuje razvojni tim, pri čemu na termin završetka eventualno može da utiče neka značajna okolnost poslovanja klijenta. Na taj način razvijaoci saopštavaju klijentu koliko posla mogu da urade u narednom koraku. Zatim klijent bira celine koje će se implementirati u okviru planirane iteracije, na osnovu sopstvene procene o njihovom prioritetu. Tom prilikom se izabrane korisničke celine detaljnije razmatraju, kako bi se napravila tačnija procena potrebnog obima poslova. Na osnovu novih procena se prema potrebi menja plan iteracije – ako se ispostavi da ne mogu sve izabrane celine da se implementiraju u predviđenim okvirima iteracije, onda neka od celina može da se odbaci, a ako se ispostavi da ima još slobodnog prostora, onda može da se doda i neka nova celina.

Jednom kada iteracija započne, klijent više ne menja karakteristike i detalje korisničkih celina koje su ušle u sastav iteracije. Razvojni tim deli celine na *poslove* (*zadatke*) i razvija ih onim redom koji je najpovoljniji iz tehničkog i poslovnog ugla, na šta klijent ne može da utiče. Obično se na pola iteracije vrši provera stanja. Sagledava se da li se planovi ostvaruju ili ne i po potrebi se, u dogovoru sa klijentom, donose potrebne odluke. Po završetku svake iteracije, klijent je dužan da pruži odgovarajuću povratnu informaciju.

Izdanja se planiraju na sličan način kao i iteracije, ali uz nešto manje zalaženja u pojedinosti. Razvojni tim određuje budžetske okvire izdanja. Uobičajeno je da razvojni tim određuje i trajanje ciklusa izdanja, ali neka poslovna ograničenja na strani klijenta mogu da značajno (pa i presudno) utiču na izbor datuma dovršavanja izdanja. Klijent zatim određuje prioritete celina i na osnovu grubih procena pokušava da u izdanje rasporedi najvažnije celine. Za razliku od planova iteracija, planovi izdanja su podložni naknadnim promenama. Za razliku od iteracija, u nekim slučajevima, a u zavisnosti od specifičnosti projekta i značajnosti određenih korisničkih celina, planiranje izdanja može da se odvija i na klasičan način, bez

primene igre planiranja, tj. da se prvo izaberu korisničke celine, pa da se onda procenjuju trajanje, broj iteracija i angažovanje razvijalaca.

U odnosu na klasične objektno-orijentisane metodologije, korisničke celine su obično negde između *poslovnih slučajeva* (engl. *business case*) i *slučajeva upotrebe* (engl. *use case*), a poslovi (zadaci) na koje se one dele obično odgovaraju manjim slučajevima upotrebe ili delovima većih slučajeva upotrebe. U skladu sa time mogu da se koriste i slične tehnike opisivanja korisničkih celina i poslova, kao što su *UML* dijagrami aktivnosti, dijagrami stanja, dijagrami sekvenci, *UML* način dokumentovanja slučajeva upotrebe, *BPMN* dijagrami i druge tehnike.

Praćenje toka razvoja

Tokom razvoja je neophodno da se učestalo (ako je moguće i neprekidno) na određen način *meri* i prati napredak razvojnog procesa. Uobičajeno je da se uvode različite numeričke mere, čije se vrednosti redovno utvrđuju i stavljaju na uvid svim članovima tima. Neke od mera koje se najčešće koriste, a koje zajedno mogu da prilično dobro predstave tok rada na projektu su:

- broj prepoznatih zadataka;
- broj zadataka čija je implementacija u toku;
- broj zadataka koji su u fazi testiranja i
- broj dovršenih zadataka.

Iste mere mogu da se uvode i za praćenje broja uočenih i ispravljenih grešaka (bagova). Mogu da se prate i mere i drugi elementi projekta ili njihove karakteristike, kao na primer:

- broj programskih datoteka;
- broj klasa;
- broj linija koda;
- broj testova
- i druge.

Problem merenja i metrike u razvoju softvera smo već pominjali na samom kraju poglavlja *4 - Uvod u projektovanje softvera*. Tamo je bilo reči o tzv. metrikama dizajna softvera, dok se ovde radi o merama i metrikama za praćenje razvoja softvera. Više informacija o metrikama i merenju softvera može da se pročita, na primer, u [*Fenton 2014*].

Timske prakse

Kolektivno vlasništvo

Ekstremno programiranje propagira *kolektivno vlasništvo* nad napisanim kodom. Pod kolektivnim vlasništvom se podrazumeva da je svaki deo napisanog koda *odgovornost* svih članova tima.

Svi članovi tima (smenjujući se u parovima) rade na svim delovima i nivoima softvera, od baze podataka, preko srednjeg sloja, pa sve do korisničkog interfejsa. Samim tim, nijedan programer nije pojedinačno odgovoran za bilo koji konkretan napisan modul ili primenjenu ili razvijenu tehnologiju. Svaki član tima i svaki par imaju pravo da provere i unaprede bilo koji deo koda, a ne samo onaj koji su sami razvijali.

Neprekidna integracija

Kolektivno vlasništvo nad kodom ima za posledicu da se relativno često dešava da više parova radi na istim delovima programskog koda. Zbog toga razvojni timovi moraju da koriste sisteme za kontrolu verzija, koji omogućavaju takav rad. Posledica je da može doći do konflikata pri integraciji.

Da bi se smanjila verovatnoća nastajanja konflikata, kao i da bi se smanjila njihova složenost i olakšalo njihovo razrešavanje, Ekstremno programiranje zastupa *neprekidnu integraciju*. Neprekidna integracija (engl. *Continuous Integration - CI*) podrazumeva da se integracija koda (tj. postavljanje izmenjenog koda na server za kontrolu verzija koda) ne obavlja tek kada se dovrši neka velika složena celina, već mnogo češće, praktično posle svake iteracije zadovoljavanja napisanih testova. Na taj način se dobija kod koji možda nije dovršen (ne postoje sve funkcije), ali može da se prevede i da zadovolji sve napisane testove. Istovremeno se značajno smanjuje mogućnost nastajanja konflikata, a i onda kada nastanu oni se odnose se na manje delove napisanog koda i lakše se rešavaju.

Postupak programiranja jednog para se deli na manje cikluse. Tokom jednog ili dva sata rada par radi na jednom poslu. Piše testove i odgovarajući produkcioni kod. U nekom pogodnom trenutku, mnogo pre završetka posla, kod se integriše. Pre svake integracije mora da se proveri da li se kod ispravno prevodi i da li prolaze svi testovi. Ako je potrebno, obavlja se uklapanje koda sa ranije podnesenim izmenama na istom kodu (razrešavanje konflikata).

U tome značajnu ulogu imaju alati za neprekidnu integraciju, čiji je posao da što je moguće češće, a poželjno više puta dnevno, preuzimaju sa sistema za upravljanje verzijama izvornog koda kompletan izvorni kod projekta, prevode ga i izgrađuju od početka i to sa različitim opcijama i za sve ciljne platforme, zatim automatski izvršavaju sve raspoložive testove i pripremaju izveštaje o obavljenom testiranju, izgrađuju distribucione i instalacione pakete, pa čak po potrebi i pokreću instaliranje

izgrađenog softvera na platformi za testiranje i izvršavaju odgovarajuće testove prihvatljivosti.

Osim pojednostavljivanja razrešavanja konflikata, glavni doprinos neprekidne integracije je i stalna raspoloživost relativno sveže verzije kompletnog izgrađenog softvera, što omogućava tačan uvid u trenutno stanje razvoja. Sekundarni značaj je u olakšavanju debagovanja.

Kao nadgradnja neprekidne integracije nastale su još dve prakse koje su veoma zastupljene u savremenom razvoju – neprekidno isporučivanje i neprekidna primena. Neprekidno isporučivanje (engl. *Continuous Delivery - CD*) podrazumeva automatizaciju pripremanja za isporuku i isporučivanja svih delova softvera koji su dovršeni i uspešno testirani. Neprekidna primena ide još jedan korak dalje i odnosi se na automatizaciju postavljanja novih verzija softvera u produkcionom okruženju. Neprekidno isporučivanje i neprekidna primena se često integrišu u jednu celinu.

Metafora

Metafora je idealizovana velika slika softvera koji se razvija. Ako imamo u vidu da EP praktično usmerava (a delimično čak i ograničava) programere da gledaju najviše jednu iteraciju daleko, onda ova praksa lako pada u oči kao nešto potpuno suprotno. Upravo zbog toga je značajna. Njena primarna uloga je održavanje fokusa tokom razvoja i sprečavanje *skretanja sa puta*.

Metafora odgovara konceptu *vizije* u nekim drugim metodologijama. Čitav ciljni sistem se sagledava na vrlo visokom nivou apstrakcije. Svi konkretni zahtevi i korisničke celine bi trebalo da posredno ili neposredno potiču iz metafore. Ona se često formalizuje u vidu rečnika pojmova, koji identifikuje najvažnije koncepte softvera i probleme koji bi on trebalo da reši. Taj rečnik pojmova je često simboličnog ili apstraktnog karaktera i predstavlja apstraktan model sistema u nekom sasvim drugom, dovoljno ilustrativnom domenu.

Uzdržan ritam

U razvoju softvera veoma često dolazi do kašnjenja i prekoračenja rokova. Uobičajena posledica je učestali prekovremeni rad. Nasuprot tome, agilni razvoj softvera promoviše ustaljen ritam rada i izbegavanje prekovremenog rada. Ekstremno programiranje ide i korak dalje i *zabranjuje* prekovremeni rad. Zato se ova praksa često naziva i *Četrdesetočasovna radna nedelja*.

Jedini izuzetak je poslednja nedelja izdanja, kada se očekuje da tim bude u potpunosti posvećen kvalitetu i popravljanju uočenih nedostataka, pa čak i po cenu prekovremenog rada.

Razvoj softvera nije sprint, nego maraton.
Sprint je dopušten samo pred ciljem.

Robert Martin

Primarna motivacija za uzdržavanje od preteranog rada je rasterećivanje članova tima i održavanje kvalitetnih odnosa u timu. Posledica rasterećenosti je veća motivisanost članova tima, manji broj grešaka, veća iskorišćenost radnog vremena, pa i ukupno veća produktivnost.

Programiranje u paru

Jedna od najprepoznatljivijih karakterističnih praksi Ekstremnog programiranja je programiranje u paru. Osnovna ideja je da se sav produkcioni kod piše u parovima od po dva programera, koji rade zajedno na istoj radnoj stanici. Dok jedan član tima piše kod, drugi član tima u hodu kritički posmatra napisan kod, proverava da li je sve u redu i daje predloge za njegovo unapređivanje. Programiranje u paru podrazumeva stalnu i intenzivnu saradnju članova para. Ne bi smelo da se dogodi da jedan član para piše kod, a drugi kuva kafu, koristi mobilni telefon ili radi neke druge poslove koji se ne dotiču neposredno koda koji se razvija.

Parovi moraju biti dinamični. Poželjno je da se uloge članova para menjaju i po više puta u toku jednog sata. Time se održavaju svežina i efikasnost u radu. Sa druge strane, i sastav parova mora da se redovno menja. Preporuka je da se sastav parova menja bar jedanput dnevno, tako da svaki član tima u toku dana učestvuje u bar dva para. Motivacija za menjanje sastava parova je višestruka, od boljeg međusobnog upoznavanja članova tima, preko obučavanja članova tima, pa sve do širenja informacija o projektu u okviru tima. Tokom jedne iteracije svaki član tima bi trebalo da radi u parovima sa svim ostalim članovima tima, kao i da radi na svim ili skoro svim delovima iteracije. Na taj način se postiže da svaki član tima ima uvid u sve delove iteracije.

Programiranje u paru se prevashodno odnosi na *produkcioni* kod. Tokom razvoja softvera prave se mnogi pomoćni alati, koji nemaju mesto u produkcionoj verziji softvera. Podeljena su mišljenja oko toga da li i takav kod mora da se piše u paru.

Programiranje u paru je jedna od praksi EP čija je zastupljenost u razvojnim timovima u poslednje vreme relativno umerena. Istraživanja pokazuju da se koristi u oko 30% timova koji *tvrde* da primenjuju agilni razvoj. Stavovi o programiranju u paru su oprečni. Protivnici ove prakse (kao i protivnici EP) zameraju da se dva programera nepotrebno angažuju da obave posao koji može da uradi i samo jedan. Druga zamerka se odnosi na nepotrebno univerzalno učešće članova tima u svim elementima iteracije, umesto da se poštuju specijalnosti.

Sa druge strane, zagovornici ističu studije koje pokazuju da je efikasnost para veća nego što bi bila efikasnost pojedinaca, kao i da je broj grešaka u napisanom kodu daleko manji [*Cockburn 2001*]. Ističu da su posledice ovakvog rada veoma brzo širenje informacija o projektu među članovima tima. Specijalnosti i dalje ostaju na pojedincima, ali se i drugi članovi tima upoznaju sa njihovim rezultatima, odlukama i razlozima za njihovo donošenje, što im omogućava da, po potrebi, nastave posao koji je započeo „specijalista“.

Otvoren radni prostor

Praksa *Otvoren radni prostor* propisuje da bi razvojni tim trebalo da radi u jednoj dovoljno velikoj prostoriji. Za to ima nekoliko motiva, a među najvažnijima su da se tako olakšava i podstiče funkcionalna komunikacija među članovima tima. Otvoren radni prostor je obično neophodan za uspešno ostvarivanje programiranja u paru.

Ako imamo u vidu da članovi tima često menjaju saradnike u parovima, jasno je bi implementacija programiranja u paru bila daleko složenija ako bi programeri (ili parovi) radili u odvojenim prostorijama. Naravno, mnogo je nezgodnije često menjati radni prostor nego samo preći za drugi radni sto u istom prostoru, ili čak drugu radnu stanicu na istom velikom radnom stolu. Pretpostavlja se da u dovoljno velikoj prostoriji postoji dovoljan broj dovoljno velikih radnih stolova, tako da na svakom stoje dve ili tri radne stanice, a za svakom radnom stanicom po dve stolice. Poželjno je da stolovi ne budu razdvojeni pregradnim zidovima, kako bi svi parovi mogli da se vide i da komuniciraju tokom rada. Uobičajeno je da su na zidovima postavljene table i panoi.

U tako organizovanom radnom prostoru svako može po potrebi da se obrati saradniku za pomoć, na primer kolegi sa kojim je prethodnog dana radio u paru na nekom delu softvera. Uobičajeno je da komunikacija bude tiha, u odmerenom tonu koji ne ometa druge. Međutim, i kada nije tako, dobro je što svi članovi tima mogu da imaju neposredan uvid u eventualne teškoće u koje je zapao neki deo tima, pa po potrebni mogu da se priključe i pomognu.

Ovo je jedna od praksi EP koje u poslednje vreme imaju najviše protivnika. Neka od novijih istraživanja su pokazala da doprinos otvorenog radnog prostora u praksi nije značajan, kao i da ne prija nekim razvijaocima (umanjena privatnost, osetljivost na gužvu, i sl.), pa se predlažu kompromisi u vidu nešto manjih prostorija sa po tri do pet radnih stolova. U poslednje vreme je često prisutan i argument da otvoren radni prostor pruža veći potencijal za širenje respiratornih infekcija.

Programerske prakse

Jednostavan dizajn

Jednostavan dizajn je jedna od praksi Ekstremnog programiranja koja je u potpunosti u skladu sa agilnim razvojem, ali se oko nje ipak relativno često lome koplja zastupnika i protivnika ove metodologije. Osnovni cilj ove prakse je da dizajn softvera bude što jednostavniji i što izražajniji. Pri dizajniranju i pisanju nekog dela koda vodi se računa samo o delovima softvera koji su već napisani i o delovima koji će biti napisani u tekućoj iteraciji. Sve ono što će (možda) kasnije doći na red se uopšte ne razmatra.

Neposredna posledica takvog pristupa je da se dizajn softvera menja od iteracije do iteracije. To ima i dobre i loše posledice. Dobro je što je dizajn uvek optimalno prilagođen aktuelnom stanju softvera. Loše je što učestalo menjanje može da pokvari

dizajn, ali i da poveća ukupan obim poslova. U cilju olakšavanja učestalog menjanja primenjuje se refaktorisanje.

U klasičnim metodologijama je uobičajeno da se nakon temeljnog planiranja najpre razvija infrastruktura, koja uključuje bazu podataka, mehanizme komunikacije među slojevima i servisima i drugo. Upravo suprotno od toga, EP zastupa pristup da se infrastruktura ne razvija unapred, već postepeno, onako kako bude postajala potrebna, i samo u meri u kojoj je potrebna za tekuću iteraciju. U skladu sa time, osnovni cilj je da grupa korisničkih celina koja čini iteraciju *proradi* i to na *najjednostavniji mogući način*. U EP nije cilj infrastruktura, nego softver koji radi.

Jednostavan dizajn se ostvaruje kroz primenu tri osnovna pravila za odlučivanje o pitanjima dizajna softvera:

- Razmotriti prvo najjednostavnije rešenje koje bi moglo da uradi posao;
- Neće biti potrebno i
- Jedanput i samo jedanput.

Prvo pravilo je potpuno u duhu EP i ove prakse. Na primer, ako infrastruktura još ne obuhvata bazu podataka, a nešto može da se reši primenom datoteka, onda ne treba dodavati bazu podataka, nego je bolje da se upotrebe datoteke. Ili, ako nešto može da se reši bez paralelizacije, onda to tako treba i da se uradi. Osnovno merilo vrednosti rešenja je *vreme potrebno da se rešenje razvije*. Naravno, pod rešenjem se podrazumeva rešenje koje zadovoljava sve funkcionalne i nefunkcionalne zahteve koji su definisani za korisničku celinu, uključujući fleksibilnost, performanse i drugo.

Pravilo „Neće biti potrebno“ sugeriše da za sve one potencijalne elemente softvera, koji će biti potrebni tek *možda* ili *za neko vreme*, valja pretpostaviti da *neće biti potrebni*. Osnovna motivacija potiče iz relativno čestog iskustva da se softver razvija primenom nekog složenog dizajna u cilju omogućavanja neke planirane opcije, a da onda ta opcija nikada ne bude implementirana, ili čak da bude implementirana i zatim nikada upotrebljena. Primena ovog pravila doprinosi jednostavnom dizajnu onih elemenata softvera koji se razvijaju u tekućoj iteraciji. Iz ugla programera, sve do završetka tekuće iteracije, ta iteracija se posmatra kao *jedini* cilj razvoja – kao da naredne iteracije nikada neće nastupiti.

Ekstremno programiranje ne dopušta ponavljanja u kodu. Kada god nastane neko ponavljanje, ono mora da se otkloni primenom odgovarajuće tehnike refaktorisanja, obično pravljenjem novog metoda ili klase, podizanjem ponašanja uz hijerarhiju ili na neke druge načine[39]. Čak i kada su delovi koda samo *veoma slični*

[39] Vidi poglavlje *10 - Refaktorisanje*.

preporučuje se njihovo apstrahovanje. Redundantnost u kodu mora da se izbegava zato što značajno otežava menjanje i održavanje koda i povećava verovatnoću nastajanja grešaka pri menjanju i održavanju koda. Kao što smo već videli iz drugih praksi, a posebno iz prethodnih pravila ove prakse, primena EP počiva na *stalnom* i *intenzivnom* menjanju koda, pa je izbegavanje redundantnosti tim važnije. Najbolji način za izbegavanje redundantnosti u kodu je redovna i opšta primena apstrahovanja. Pored otklanjanja redundantnosti, apstrahovanje smanjuje i međusobnu spregnutost delova koda i doprinosi čistijoj i fleksibilnijoj arhitekturi softvera.

Priča o jednostavnom dizajnu ne sme da prođe bez veoma važne napomene – razmatranje najjednostavnijih rešenja i jednostavnog dizajna *nikako* ne znači da programski kod sme da bude loše dizajniran. „*Jednostavno*“ ne znači ni *na brzinu* ni *nepažljivo*. Upravo suprotno tome, jednostavna rešenja su vrlo često ona koja nisu očigledna i do kojih se stiže posle vrlo ozbiljnog rada, koji obuhvata dobro upoznavanje domena, pažljivu analizu problema i projektovanje uz primenu principa i obrazaca za projektovanje.

Objektivno loša strana ove prakse je povećani rizik od kasnog ustanovljavanja i učestalog menjanja arhitekture softvera. O arhitekturi softvera i značaju njene stabilnosti i relativno ranog ustanovljavanja bilo je već reči u poglavlju *4 - Uvod u projektovanje softvera*, a dodatnu pažnju tom problemu ćemo posvetiti i u odeljku *8.7 - Agilne metodologije i projektovanje softvera*.

Razvoj vođen testovima

Razvoj vođen testovima ima veoma važnu ulogu u svim agilnim metodologijama, pa i u Ekstremnom programiranju. Osnovna ideja je da se produkcioni kod piše sa ciljem da zadovolji testove jedinica koda. Umesto da se prvo piše kod, pa da se onda proverava njegova ispravnost, radi se upravo obrnuto – počinje se od pisanja testova, koji ne prolaze zato što ne postoji odgovarajuća funkcionalnost, a zatim se piše kod koji omogućava da testovi prođu.

Iteracije pisanja testova i koda moraju da se smenjuju veoma brzo, često na svaki minut. Umesto da se prvo napiše veliki broj testova, pa zatim mnogo koda, preporučuje se da se pišu jedan do dva testa, pa odgovarajući kod. Testovi i kod bi trebalo da zajedno evoluiraju, tako da testovi budu malo ispred koda.

Najvažnija posledica ovakvog razvoja je da se zajedno sa kodom dobija i kolekcija testova, koja omogućava programeru da proverava da li jedinica koda radi ispravno ili ne. Pored toga, sprečava se naknadno nastajanje grešaka u kodu prilikom eventualnih izmena, pomaže se *refaktorisanje* i vrši se dodatni pritisak da se razdvajaju jedinice koda, čime se dobija bolji dizajn projekta.

Razvoj vođen testovima predstavlja jednu od najšire zastupljenih praksi Ekstremnog programiranja. Iscrpnije je predstavljena u poglavlju *9 - Razvoj vođen testovima*, na strani 211.

Refaktorisanje

Kao što smo u više navrata videli u prethodnim odeljcima, Ekstremno programiranje počiva na stalnom i intenzivnom menjanju koda. Ako se kod dograđuje ili menja, čak i kada je idealno dizajniran, postepeno može da mu opadne kvalitet. Sredstvo za borbu protiv opadanja kvaliteta koda je *refaktorisanje*.

Refaktorisanje je preduzimanje niza malih transformacija koda, kojima se unapređuje struktura koda bez vidljive promene ponašanja sistema. Pojedinačne transformacije koda su uglavnom sasvim jednostavne, skoro trivijalne. Posle svake primenjene transformacije se, testiranjem jedinica koda, proverava da nije slučajno izmenjeno ponašanje. Transformacije se ponavljaju sve dok se ponovo ne dobije čist i dobro dizajniran programski kod.

Refaktorisanje bi trebalo da se primenjuje redovno, svaki put kada se uoči da neka izmena narušava dizajn koda. Ne bi trebalo da se odlaže *za kasnije*, već da se odvija naizmenično sa razvojem testova i produkcionog koda.

Refaktorisanje, kao jedna od najvažnijih savremenih tehnika razvoja softvera, je detaljnije predstavljeno u poglavlju *10 - Refaktorisanje*, na stranici 241.

Standardi kodiranja

Svaki tim bi trebalo da odredi i poštuje neka pravila u vezi sa načinom pisanja programskog koda. Pravila se odnose na imenovanje elemenata koda i samih programskih modula, načine pisanja koda i komentara, način i obim pisanja dokumentacije u kodu i van koda, organizaciju datoteka po direktorijumima i sve druge aspekte pisanja koda.

Među najvažnija pravila spadaju ona koja se odnose na imena u programskom kodu i vizualno oblikovanje programskog koda. Postoji više stilova pisanja koda i teško je reći da li je i zašto jedan bolji od drugog, ali bi tim trebalo da se pridržava ujednačenog stila. Primarni motiv je podizanje nivoa čitljivosti koda. Agilni razvoj podrazumeva učestalo menjanje napisanog koda, pa je čitljivost koda mnogo važnija nego kod klasičnih metodologija.

8.6 Skram

Skram je agilna metodologija koju su razvili Ken Švaber i Džef Saterlend u prvoj polovini 1990-ih, a prvi put je javno predstavljen 1995. godine. Kao što je već naglašeno, za razliku od Ekstremnog programiranja, koje predstavlja relativno kompletnu metodologiju, Skram je zamišljen kao relativno lagan razvojni okvir, koji je veoma fleksibilan i prilagodljiv.

Skram ne definiše tehnike i alate, kao ni vrste rezultata rada koji nastaju tokom njegove primene. Umesto toga, fokusira se na organizovanje toka i strukture razvojnog procesa, a tehnike i alate pozajmljuje iz drugih metodologija. Imajući u

vidu agilnu prirodu Skrama, njegova primena često počiva na primeni nekog izabranog podskupa praksi Ekstremnog programiranja i drugih agilnih metodologija.

Čak i pri propisivanju toka i strukture razvojnog procesa Skram je relativno fleksibilan i, za razliku od većine drugih metodologija, ne donosi gomilu strogih pravila. Umesto toga, pretpostavlja se da će Skram da primenjuje tim sastavljen od sposobnih ljudi, koji su u stanju da razumeju postojeća načelna pravila (tako da im nisu potrebna mnogo preciznija uputstva) i koji pri tome mogu samostalno da procene šta je i kako potrebno da se prilagođava konkretnom razvojnom projektu i okolnostima razvojnog procesa u kome učestvuju.

Skram je relativno dobro obrađen u literaturi. Ovde ćemo se pri njegovom predstavljanju najviše osloniti na priručnik koji su napisali i održavaju autori metodologije [*Schwaber 2020*] i na knjigu Keneta Rubina [*Rubin 2013*].

8.6.1 Tim

Osnovna organizaciona jedinica koja primenjuje Skram se naziva *tim*. Tim bi trebalo da bude dovoljno mali da bude agilan i fleksibilan, a sa druge strane dovoljno veliki da može da iznese značajniji obim posla. Uobičajeno je da tim ima 10 ili manje članova. Jedan isti tim je zadužen za sve aspekte posla u vezi sa zadacima koji mu se dodeljuju – od saradnje i komunikacije sa klijentom, istraživanja domena i analize problema, preko planiranja i projektovanja, pa sve do implementacije, testiranja, dokumentovanja, izveštavanja i svih ostalih aktivnosti potrebnih za ispunjavanje dodeljenih zadataka.

Svaki tim ima tri vrste članova: jednog stručnjaka za Skram (engl. *Scrum Master*), jednog vlasnika proizvoda (engl. *Product Owner*) i više razvijalaca.

Stručnjak za Skram je zadužen za primenu pravila Skrama. On je zadužen da upoznaje sve ostale članove tima sa teorijskim i praktičnim aspektima metodologije, ali i da im pomaže u svakodnevnom radu. Stručnjak za Skram je dužan da pažljivo prati napredak svih članova tima, da im ukazuje na greške i moguće načine da unaprede svoj rad. Jedna od osnovnih njegovih odgovornosti je da se svi sastanci odvijaju u odgovarajuće vreme i na odgovarajući način. Posebno, on pomaže vlasniku proizvoda da definiše ciljeve i poslove, kao i da se stara o spisku neobavljenih poslova. Stručnjak za Skram je odgovoran i za ostvarivanje saradnje i komunikacije sa klijentom.

Vlasnik proizvoda je zadužen da krajnji rezultat rada bude što bolji. Njegov zadatak je da usmerava rad tima kako bi se na odgovarajući način oblikovali ciljevi i poslovi. Posebno je važno da se stara o razumljivosti opisa svih neobavljenih poslova i definiše kriterijume koje implementacija mora da zadovolji (praktično pravi ili bar okvirno definiše testove prihvatljivosti). Vlasnik proizvoda može da deo svog posla delegira drugima, ali je uvek on nosilac pune odgovornosti za kvalitet. Samo on može da dodaje nove poslove na spisak neobavljenih poslova.

Svi ostali članovi tima su *razvijaoci*. Oni mogu da imaju različite specijalnosti, u zavisnosti od vrste projekta na kome se radi. Odgovorni su za sve poslove koje obavlja tim, od planiranja sprinta i spiska poslova, preuzimanja i obavljanja poslova do kritičkog razmatranja toka rada i učestvovanja u svakodnevnom usavršavanju tima.

Skram u potpunosti počiva na pretpostavci o kvalitetnim kadrovima, koji su istovremeno i veoma sposobni i dovoljno dobro obučeni da mogu da funkcionišu zajedno kao tim. Neophodno je da u timu postoji visok nivo međusobnog poverenja i saradnje i da međusobni odnosi članova tima budu visoko profesionalni. Očekuje se da svi članovi tima budu posvećeni svom poslu, fokusirani i na celovit posao i na preuzete zadatke, otvoreni u komunikaciji i pri iznalaženju dobrih rešenja, da poštuju svoje kolege i da budu odvažni u svom radu.

8.6.2 Razvojni proces

Skram propisuje iterativni razvojni proces i po tome je sličan ostalim agilnim metodologijama. Ono što je specifično za Skram su pojedinosti organizacije rada na iteracijama. Timske aktivnosti se nazivaju *događajima* (engl. *events*). Glavni događaj je sprint. Sprint predstavlja jednu razvojnu iteraciju i obuhvata više manjih događaja, kao što su planiranje sprinta, dnevni skram, sagledavanje sprinta, i retrospektiva.

Sprint predstavlja jednu iteraciju posla. Trajanje sprinta je ograničeno i unapred određeno. Uobičajeno trajanje sprinta je tri do četiri nedelje, ali može da bude i kraći. Svaka aktivnost u razvojnom procesu se odvija unutar nekog sprinta. Odmah po završetku jednog sprinta započinje drugi sprint (osim, naravno, u slučaju završetka projekta).

Sprint mora da bude zaokružen u smislu celovitosti i kompletnosti. Teži se da poslovi koji se odvijaju u okviru jednog sprinta budu međusobno povezani. Sprint mora da ima jasno definisan cilj (engl. *Sprint Goal*). Sve što se radi u okviru sprinta je usmereno prema ostvarivanju tog cilja. Iako su u okviru sprinta dopuštene promene spiska predviđenih neobavljenih poslova, nije dopušteno da se prave promene koje dovode u pitanje ispunjavanje cilja sprinta. Ako se u toku sprinta ispostavi da je cilj nedostižan ili da je iz nekog razloga postao beznačajan, sprint može da se prekine pre isteka. Sprint sme da prekine samo vlasnik proizvoda.

Planiranje sprinta je aktivnost kojom započinje sprint. Vremenski je ograničeno, obično na 8 sati ili manje. Cilj planiranja sprinta je da se svi članovi tima upoznaju sa ciljevima i poslovima. U planiranju učestvuju svi članovi tima i očekuje se da prethodno budu upoznati sa najvažnijim stavkama iz spiska neobavljenih poslova. Planiranju mogu da prisustvuju i predstavnici klijenata ili druge kolege, ako se proceni da njihovo prisustvo može da bude od pomoći.

Planiranje sprinta mora da odgovori na nekoliko osnovnih pitanja:

- *U čemu je osnovna vrednost ovog sprinta?*
  Vlasnik proizvoda je zadužen da predloži kako ukupna vrednost proizvoda može da se uveća tekućim sprintom. Članovi tima zatim zajednički definišu cilj sprinta tako da svima bude jasno zašto je on važan klijentu.
- *Šta može da se dovrši u ovom sprintu?*
  U saradnji vlasnika proizvoda i članova tima biraju se poslovi iz spiska neobavljenih poslova koji će ući u tekući sprint. Može biti više kriterijuma izbora, a među najvažnijim su povezanost posla sa ciljem sprinta, povezanost posla sa drugim već izabranim poslovima, složenost posla, procena da li posao može da se završi u toku tog sprinta i drugo.
- *Kako će izabrani poslovi da se urade?*
  Za svaki izabrani posao se okvirno planira kako će da se uradi, a da se pri tome zadovolje kriterijumi dovršavanja. To ne znači da se do detalja planira način implementacije, ali se obično već u ovoj fazi razmatraju elementi povezivanja strukturnih elemenata, odnosno neki od važnijih elemenata strukturnog dizajna.

Nakon završetka planiranja sprinta započinje njegova realizacija. U toku realizacije sprinta se svakodnevno organizuje *dnevni skram*. Dnevni skram je kratak sastanak, obično ograničen na do 15 minuta, koji se po pravilu organizuje u isto vreme i na istom mestu. Dnevni skram uključuje sve članove tima. Osnovni cilj je sagledavanje trenutnog toka rada na sprintu, ali može da obuhvati i manja savetovanja pa i promene planova. U zavisnosti od fizičke organizacije tima, dnevni skram može da se organizuje na početku, na kraju ili negde usred radnog dana. U poslednje vreme se obično izbegava da termin dnevnog skrama bude na početku ili kraju radnog vremena, da bi se olakšala primena fleksibilnog ili kliznog radnog vremena.

Na samom kraju sprinta se organizuje *razmatranje sprinta* (engl. *Review*). U razmatranju sprinta učestvuju i tim i ulagači. Cilj je da se tokom sastanka, koji obično traje do 4 sata, razmotri da li je i u kojoj meri sprint ispunio očekivanja i zacrtane planove. Razmatraju se i eventualne promene okolnosti ili šireg stanja projekta i planiraju se predstojeći koraci. Rezultati razmatranja sprinta često utiču na planiranje narednog sprinta.

Jedina aktivnost koja se odvija posle razmatranja sprinta je *retrospektiva*. Za razliku od razmatranja sprinta, u retrospektivi učestvuju samo članovi tima. Uobičajeno je da retrospektiva traje do tri sata. Ona služi da tim sagleda kvalitete i slabosti u svom radu, koji su ispoljeni u toku sprinta. To je prilika da se izvrši samokritični uvid u tok sprinta, da se istaknu dobro urađene stvari i eventualne

greške, da se prepozna da li je potrebno da se radi na popravljanju međusobnih odnosa, na dodatnom usavršavanju članova tima, na zamenjivanju ili unapređivanju nekih alata i drugo. Osnovni cilj retrospektive je da tim potraži načine da dodatno unapredi svoj rad.

8.6.3 Artefakti

Klasične metodologije navode veliki broj različitih artefakata, od izveštaja o istraživanju domena, modela domena, pa sve do planova, projekata, izvornog koda i dokumentacije. Skram ne precizira ništa od toga. Kao što ne propisuje tehnike i alate, tako ne propisuje ni različite vrste rezultata rada na projektu. Umesto svega toga, Skram propisuje samo tri osnovne vrste artefakata.

Spisak nedovršenih poslova (engl. *Product Backlog*) predstavlja uređenu listu neobavljenih poslova, koje bi trebalo uraditi tokom rada na projektu. Svi poslovi na listi moraju da budu usmereni prema ostvarivanju krajnjeg cilja projekta, tj. dostizanja ciljnog proizvoda (engl. *Product Goal*). Štaviše i sam završni cilj bi trebalo da bude na spisku, kao poslednji posao koji će se dovršiti. Ovaj spisak održava vlasnik proizvoda u saradnji sa drugim članovima tima i drugim zainteresovanim licima (ulagačima), a prvenstveno sa predstavnicima klijenta i svojim nadređenima.

Spisak nedovršenih poslova je uređena lista. Uređenje predstavlja okvirni redosled po kome se očekuje izvođenje poslova, a može da obuhvata i okvirni plan raspoređivanja poslova po iteracijama, kao i oznake međuzavisnosti poslova. Ipak, to uređenje nije striktno, već je relativno fleksibilno. Cilj tog uređenja je pre svega olakšavanje sagledavanja trenutnog stanja projekta i olakšavanje izbora poslova pri planiranju sprinta. Stvarni redosled realizacije poslova se ustanovljava za svaki sprint zasebno, tokom planiranja sprinta.

Ako je neki posao na spisku prevelik da bi mogao da se dovrši u okviru jednog sprinta, onda mora da se razloži na više manjih poslova, tako da svaki posao bude ostvariv u toku jednog sprinta. To razlaganje poslova se obavlja u saradnji vlasnika proizvoda i razvijalaca.

Spisak poslova sprinta (engl. *Sprint Backlog*) obuhvata ciljeve jednog sprinta i poslove koji su izabrani da se dovrše u okviru tog sprinta. Obuhvata i što detaljniji plan toka sprinta. Za razliku od spiska neobavljenih poslova, spisak poslova sprinta prvenstveno prave i održavaju razvijaoci. Spisak poslova sprinta se često ažurira tokom odvijanja sprinta. Neki poslovi se razlažu na manje, neki se označavaju kao urađeni, neki se možda označavaju kao problematični i slično. Dobro je da ovaj spisak bude što sadržajniji, da bi mogao da se koristi za procenjivanje uspešnosti praćenja plana sprinta. Često se koriste i detaljnije tehnike praćenja, kao što su Kanban-table i slično.

Doprinos (engl. *Increment*) je artefakt koji predstavlja prepoznatljiv i uočljiv korak na putu do krajnjeg ciljnog proizvoda. Svaki doprinos se dodaje na prethodne

doprinose i povećava ukupnu vrednost do tada realizovanog dela projekta. Ostvareni doprinosi se sagledavaju u okviru razmatranja sprinta, ali mogu da se isporuče klijentu i ranije, već u toku sprinta.

Doprinos može da obuhvati samo poslove koji su zadovoljili kriterijume dovršenosti. U tom smislu su kriterijumi dovršenosti čvrsto povezani sa poslovima i doprinosima sprinta. Kriterijumi dovršenosti su konceptualno slični testovima prihvatljivosti. Definišu se formalno a proveravaju se bilo automatski ili manuelno, zavisno od konkretnog slučaja.

U okviru jednog sprinta mora da bude ostvaren bar jedan doprinos, a često se ostvaruje i više njih. Jednom poslu na listi poslova može da odgovara jedan ili više doprinosa, ali neki poslovi mogu da budu i bez doprinosa, a pre svega zbog pomenutog kriterijuma dovršenosti. Na primer, jedan od poslova može biti detaljnija analiza nekog drugog posla i planiranje podele tog posla na manje poslove, da bi u narednim sprintovima oni mogli da se lakše realizuju. Ili, posao može da predstavlja definisanje interfejsa neke komponente koja će tek kasnije da se implementira. U smislu razvojnog projekta, rezultati ovakvih poslova jesu značajni, ali se oni često ne smatraju doprinosima u kontekstu puta prema ciljnom proizvodu.

8.6.4 Skram i druge metodologije

Skram ne propisuje sve uobičajene metodološke elemente, pa je zato uobičajeno da se kombinuje sa drugim metodologijama. Skram nije striktno vezan za OO tehnike i metodologije, ali ni za bilo koje druge. U načelu je ortogonalan u odnosu na sve raspoložive tehnike i može da se koristi u praktično svim razvojnim projektima.

Kao što smo već istakli, Skram propisuje organizacione elemente i tehnike upravljanja razvojnim ciklusom, a konkretne razvojne tehnike se preuzimaju iz drugih metodologija. Iz predstavljenih istraživanja se vidi da su to često tehnike koje potiču iz Ekstremnog programiranja i drugih agilnih metodologija.

Jedan od problema koje donosi takvo povezivanje je u često proizvoljnom odabiru tehnika i praksi koje će se upotrebljavati. Kao što smo već istakli kada smo predstavljali Ekstremno programiranje, tehnike i prakse koje čine neku metodologiju nisu slučajno oblikovane i izabrane, već su grupisane na odgovarajući način zato što se pokazuje da one više i bolje ispoljavaju svoje pozitivne karakteristike kroz svoje međusobno povezivanje i kombinovanje. Proizvoljnim i delimičnim odabirom tehnika dolazimo u priliku da ostvarimo neke benefite koje nam pruža njihova primena ali i da propustimo da ostvarimo neke dodatne benefite zato što ih ne primenjujemo u kombinaciji sa tehnikama sa kojima se one prirodno nadopunjuju. O tome bi trebalo voditi računa pri izboru tehnika i alata.

8.6.5 Ograničenja Skrama

Skoro sve raspoložive analize pokazuju da je danas Skram najzastupljenija agilna metodologija. To bi moglo da bude dobro i za timove i za razvijaoce – timovi će uspešno raditi svoj posao, a razvijaoci će biti u prilici da upoznaju metodologiju i da se zatim pri eventualnoj promeni sredine relativno lako uklapaju u novi tim. Međutim, u praksi često nije baš sve idealno.

Videli smo da se Skram obično kombinuje sa tehnikama drugih metodologija, kao i da se te tehnike obično ne primenjuju dosledno već prema nekom izboru. Posledica takvog pristupa je da se onda često čak i Skram ne primenjuje dosledno, već se i on prilagođava specifičnostima projekta ili sklonostima rukovodilaca i drugih članova tima. Ali ako se metodologija, koja je već sasvim jednostavna, dodatno modifikuje i pojednostavljuje, to onda može da ima vrlo neugodne posledice.

Procenjujem da 75% organizacija koje koriste Skram
neće uspeti da od njega dobiju koristi kojima se nadaju.

Skram je veoma jednostavan okvir
unutar kojeg se odvija „igra“ složenog razvoja proizvoda.
Skram razotkriva svaku neadekvatnost ili disfunkciju u okviru postupaka
koji se primenjuju u organizaciji pri razvoju proizvoda ili sistema.

Namena Skrama je da te slabosti učini transparentnim
kako bi organizacija mogla da ih popravi.
Nažalost, mnoge organizacije menjaju Skram
da bi ga prilagodile svojim slabostima, umesto da ih otklone.

Ken Švaber, 2010.

Ako procenu Kena Švabera primenimo na procenat timova koji koriste (ili kažu da koriste) Skram, onda možemo da procenimo da 43% svih anketiranih timova kažu da primenjuju Skram, ali to rade neispravno i neće imati od njega skoro nikakve koristi. Štaviše, ako se projekat ipak odvija relativno uspešno, onda možemo da se zapitamo da li im više koristi donosi primena izabranih praksi Ekstremnog programiranja ili tehnika preuzetih iz neke druge metodologije, ili pogrešna primena Skrama?

Nedosledna primena Skrama može da se prepozna kao problem na strani razvojnih timova, ali ako je takva primena vrlo česta, onda moramo da se zapitamo da li je problem možda, bar delimično, i na strani metodologije? Na primer, kada bi Skram bio čvrsto vezan za neke konkretno određene tehnike i alate, onda bi možda imao manje korisnika, ali bi oni verovatno bili manje skloni krivudanju u njegovoj primeni?

8.7 Agilne metodologije i projektovanje softvera

Agilne razvojne metodologije su nam donele princip: „ako nešto nije potrebno sada i odmah, onda to nije potrebno“. Taj princip nam pomaže da pri pisanju i planiranju strukture programa očuvamo najveću moguću jednostavnost, bez povećavanja složenosti zbog nekih stvari koje će *možda* i *nekada* biti potrebne. Tome smo već posvetili pažnju u ovom poglavlju i videli da primena ovog principa može da bude veoma dobra, ali i da moramo da budemo oprezni zbog toga što odlaganje nekih vrsta promena može kasnije skupo da nas košta. Ranije smo već pomenuli neke elemente odnosa agilnog razvoja i projektovanja (*4.3 Pristupi projektovanju softvera*, str. 51), a ovde ćemo malo temeljnije da se posvetimo primeni principa agilnog razvoja u oblasti projektovanja softvera.

Dovođenjem prihvatanja promena u prvi plan, agilni razvoj softvera nam je u velikoj meri zakomplikovao bavljenje projektovanjem. Dok je u klasičnim metodologijama projektovanje softvera predstavljalo jedan celovit i zaokružen posao, koji bi se uradio jednokratno u odgovarajućoj fazi razvoja, a nakon čega bi se njegovi rezultati skoro bespogovorno koristili, sada nam agilni razvoj propisuje da u svakoj fazi razvoja softvera možemo i moramo da prihvatamo izmene, što povlači modifikovanje ne samo programskog koda, već i strukture programa a time i projekta. Zbog toga je u agilnom procesu uobičajeno da se projekat menja tokom implementiranja programskog koda, ali i kasnije, tokom testiranja, debagovanja ili održavanja – kada god se ustanovi da nam je potrebna neka promena. Praktično svaki aspekt razvoja softvera ima potencijal da menja projekat i da utiče na njegove elemente i karakteristike. Možemo da kažemo da u agilnom razvoju projektovanje u velikoj meri prestaje da bude izdvojena aktivnost, već se prepliće sa svim drugim aktivnostima, ali i obrnuto, da sve druge aktivnosti postaju deo projektovanja.

U takvim uslovima je sasvim očekivano da se dovedu u pitanje uloga i značaj unapred pripremljenog projekta softvera. Ako izrađujemo projekat unapred i ulažemo u projektovanje veliku energiju i značajno vreme, a zatim sve to kasnije relativno često menjamo, onda tu možda nešto nije u redu? Da li je dobro da projektujemo softver pre implementacije? Da li je dobro da menjamo projekat u toku implementacije? Da li bi trebalo da ukinemo projektovanje pre implementacije? Odgovori na ova pitanja uglavnom slede neposredno iz izloženih materijala o agilnom razvoju i projektovanju softvera, ali ćemo ovde pokušati da ih malo eksplicitnije predstavimo.

8.7.1 Zašto je sve projektovanje?

Videli smo da se u agilnom razvoju projektovanje izvodi u sklopu praktično svih razvojnih aktivnosti. Svaki put kada se u okviru pisanja programskog koda prave nove klase i njihovi interfejsi ili kada se menjaju interfejsi ili strukture klasa, to predstavlja menjanje projekta. Kao posledica toga se prepoznaje da svaka aktivnost u

okviru koje može da bude potrebno da se preduzima menjanje programskog koda takođe može da dovede i do promena u projektu. Na primer, ako se u fazama testiranja ili debagovanja ustanovi da je potrebno da se prave izmene u programskom kodu, sasvim je moguće da te izmene predstavljaju i promenu u projektu. Promene u fazi održavanja softvera su uobičajeno još češće i temeljnije, pa samim tim mogu da imaju i veliki uticaj na projekat.

Mogli bismo da primetimo da se navedenim izmenama prvenstveno menja dizajn a ne i arhitektura, ali to nije opšte pravilo – veliki broj izmena dizajna (pa i manji broj velikih izmena dizajna) može na kraju da dovede i do određenih promena na nivou arhitekture.

Dobre strane projektovanja kroz implementaciju

Preduzimanje projektovanja paralelno sa pisanjem koda ima i prednosti i mane. Pokušaćemo da sagledamo najvažnije posledice takvog pristupa.

Dobro obučeni i iskusniji programeri često mogu da bez posebnih teškoća istovremeno projektuju i kodiraju. Na taj način se sa razvojem koda postepeno pravi i izgrađuje projekat softvera, ali možemo reći i da se odvija obratan posao – da se prvenstveno pravi projekat, koji se zapisuje na programskom jeziku umesto da se predstavlja dijagramima. Čest je slučaj da problem može da se modelira i implementira na više različitih načina. Izbor adekvatnog načina se obično određuje pri oblikovanju modela, tj. pri projektovanju, a onda programer mora da nametnut način rešavanja implementira. Prepuštanjem projektovanja programeru, mi mu dajemo veću slobodu i mogućnost da, u slučajevima kada postoji izbor, on sam pravi izbor u skladu sa sopstvenim mogućnostima i sklonostima.

> *Verovatno je bolje pustiti originalne projektante da napišu originalni kod, nego da neko drugi prevodi njihov dizajn na programski jezik.*
>
> *Džek Rivs*

Ako je potrebno da neko razume problem i zatim ga modelira i isplanira implementaciju, a zatim da neko drugi razume taj model pa ga implementira, pri čemu će često i da ga menja, onda je prilično očigledno da tu može da bude prostora za uštede. Ukidanjem projektovanja pre implementacije se potencijalno smanjuju i broj ljudi koji učestvuje u razvoju i trajanje razvoja. Uz to, brže se započinje implementiranje, što u agilnom razvoju često može da bude veoma važno.

Loše strane projektovanja kroz implementaciju

Čim se malo odvojimo od pojedinačnih klasa ili grupa klasa i počnemo da posmatramo veće strukturne elemente projekta, neminovno počinjemo da uočavamo neke probleme koje nam donosi projektovanje kroz implementaciju.

Tokom razvoja se zadaci i poslovi obično dele na različite timove. Ako se svaki tim zatim bavi i projektovanjem i implementiranjem svog dela softvera, postavlja se pitanje kako će ti delovi softvera da se povezuju? Ako tim *A* razvija komponentu *A*, a tim *B* komponentu *B*, koja koristi komponentu *A*, postavlja se pitanje kada će i kako tim *A* da saopšti timu *B* kako izgleda interfejs komponente *A*? Da li će to moći da uradi dovoljno rano da tim *B* ne bi kasnio u razvoju? Da li će moći da garantuje da taj interfejs neće da se menja? Da li tim *A* uopšte može da oblikuje dobar interfejs komponente koju razvija ako ne zna šta radi komponenta *B* i kakve su njene potrebe? Da li ovi timovi znaju ko će još da koristi komponente *A* i *B* i kakve će oni imati zahteve u odnosu na njih?

Do odgovori na ova pitanja često nije lako doći, ali se može steći utisak da bi se navedeni problemi lakše rešili ako bi komponente *A* i *B* razvijao isti tim. Naravno, u razvoj uključujemo više timova onda kada je potrebno da se skrati vreme razvoja, tako da ne smemo da računamo da će sve komponente (pa ni neke izabrane *A* i *B*) razvijati isti tim. Ipak, takav utisak je važan zato što ukazuje na problematične elemente ovakvog pristupa projektovanju. Problem nije u broju timova, nego u činjenici da je za projektovanje interfejsa neke komponente potrebno znati mnogo toga, ne samo o toj komponenti nego i o svim komponentama koje bi trebalo da nju koriste. Ako bi svaki tim morao da upoznaje sve komponente da bi mogao da razvija svoju, onda takav razvoj postaje višestruko neefikasan. Prvo se gubi vreme na tome da svi timovi moraju da razmatraju sve informacije, a onda se još postavlja i pitanje da li će eventualne izmene na nekoj komponenti zahtevati da se sve to ponovi?

Da bi tim bio efikasan, potrebno je da bude u prilici da uradi svoj deo posla bez detaljnog razmatranja čitavog projekta. Ali ako tim mora da definiše interfejse komponenti, onda to nije do kraja moguće. Znači, ili će tim pokušati da radi efikasno i imati za rezultat potencijalno problematično povezivanje komponenti, ili će pokušati da razmatra ceo projekat i izgubiti na efikasnosti. U oba slučaja gubimo i energiju razvijalaca i vreme.

Poseban problem je što pojedinačnim projektovanjem velikog broja delova lako možemo da dobijemo rezultat koji nije dobro usklađen. Projektovanje kroz implementaciju često odlaže oblikovanje arhitekture i distribuira ga po timovima, što može da ima za posledicu prilično kasno uspostavljanje arhitekture i relativno nestabilnu arhitekturu, koja kasnije trpi veći broj izmena. Kao što smo već ranije istakli, arhitekturu je veoma teško i skupo menjati, pa zato takve ishode želimo da predupredimo.

8.7.2 Zašto je važno imati projekat pre kodiranja

Jedan od osnovnih kvaliteta koje nam donosi projektovanje pre implementiranja je jasnija slika projekta, koja se stiče na osnovu velike količine informacija o različitim aspektima problema koji se rešava. Širina sagledavanja problema omogućava da se

dobije i dobra vizija krajnjeg rezultata, koja često može da izostane kada se projektovanje ostavlja da teče uz implementaciju. Vizija je važna zato što može da posluži razvojnim timovima kao čvrst oslonac, posebno u slučajevima kada pri oblikovanju i implementiranju pojedinačnih delova projekata dođu u dilemu kojim kriterijumima kvaliteta ili aspektima krajnjeg cilja bi trebalo dati veću težinu.

Pored celovite slike i vizije, ovakav način projektovanja nam donosi i bolje sagledavanje pojedinačnih komponenti (koje čine tu celinu) i njihovih odnosa. Kao rezultat projektovanja pre implementiranja obično se dobijaju bolji interfejsi komponenti, zato što su oblikovani uz razmatranje mnogo širih informacija nego što su one koje pojedinačni razvojni timovi imaju na raspolaganju tokom implementacije.

Poseban kvalitet koji nam donosi projektovanje pre implementacije je iscrpna dokumentacija koja opisuje projekat. Projektna dokumentacija mora da bude dovoljno obimna i jasna da bi implementatori na osnovu nje mogli da implementiraju softver. Projektna dokumentacija je vid poruke koju projektanti šalju implementatorima. Sa druge strane, ako neki tim projektuje deo softvera tokom implementacije, onda on nema potrebu da nekome „šalje poruku“, pa samim tim nema ni potrebu da iscrpno dokumentuje napravljeni dizajn, već je programski kod obično sasvim dovoljan.

Slabosti ovog pristupa smo uglavnom već istakli – ogledaju se primarno kroz dodatni utrošak vremena da bi se napravilo nešto što će se kasnije vrlo često menjati. Naglasili smo i da je posledica ovakvog pristupa odlaganje započinjanja implementacije, zato što najpre mora da se sačeka na dovršavanje projektovanja. Posebna slabost se odnosi na dokumentaciju – koliko god da je dokumentacija iscrpna, ona često postaje nepouzdana i često neupotrebljiva zbog neažurnosti. Ako naknadno menjanje projekta nije dosledno propraćeno menjanjem projektne dokumentacije (a ispostavlja se da najčešće nije), onda korisnici te dokumentacije više ne mogu da se na nju oslanjaju, zato što ne znaju šta je ažurno (tj. tačno) a šta nije. Zbog svega toga, jasno je da detaljno projektovanje softvera pre započinjanja implementacije uglavnom ne predstavlja dobar izbor.

8.7.3 Projektovanje do nivoa komponenti

Kada sagledamo dobre i loše strane opisanih pristupa projektovanju, onda možemo da vidimo da nijedan od njih nije savršen, već da oba imaju i izrazito dobre i izrazito loše strane. Projektovanje kroz implementiranje ispoljava svoje kvalitete kada se bavimo projektovanjem manjih delova softvera, a slabosti se ispoljavaju kada se podignemo na nivo većih programskih celina. Sa projektovanjem pre implementiranja je upravo suprotan slučaj – ono donosi relativno malo koristi kada se odnosi na manje delove softvera i njihovo detaljno projektovanje, zato što će se oni tokom implementiranja (i kasnije) često značajno menjati u odnosu na originalni

projekat, dok je veća korist u slučaju većih celina, čije je elemente bolje sagledati ranije i koji se relativno ređe menjaju tokom implementiranja.

Agilni razvoj softvera načelno podstiče projektovanje kroz implementaciju, ali je uobičajeno da se u iole većim projektima koristi kombinovani pristup. Ideja je da se koriste oba pristupa projektovanju, svaki na način i sa ciljevima kojima je primereniji. U poglavlju o projektovanju softvera smo naveli citat Grejdija Buča koji kao kriterijum razlikovanja arhitekture i dizajna prepoznaje cenu pravljenja izmena. Sada prepoznajemo da isti kriterijum može da se primeni i na definisanje granice između onoga što je potrebno projektovati pre implementacije i onoga što će se projektovati tokom implementacije.

Odatle možemo da zaključimo da je u agilnom razvoju najbolje da arhitekturu oblikujemo pre implementiranja, a da ostale elemente dizajna oblikujemo tokom implementiranja. U skladu sa ranije prepoznatim granicama arhitekture i dizajna, to bi značilo da je najčešće ispravan pristup da pre početka implementiranja izvedemo što detaljniju funkcionalnu dekompoziciju i prepoznamo komponente i njihove odnose i interfejse, kao i da prepoznamo i oblikujemo strukturne elemente koji imaju širok uticaj na različite delove softvera. Većinu ostalih stvari možemo da projektujemo kasnije, tokom implementacije. Takav pristup se primenjuje u većem broju savremenih metodologija, pa i u jednoj od najkompleksnijih i najznačajnijih – *Rational Unified Process* (*RUP*) [*Kruchten 2003*].

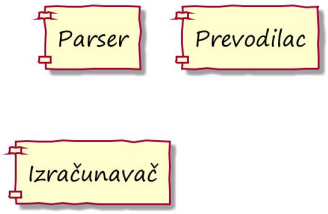
Agilni principi razvoja su u osnovi ortogonalni u odnosu na paradigmu metodologije – mogu da se primenjuju i na OO metodologije ali i u kombinaciji (ili kao njihova modifikacija i dopuna) sa nekim klasičnim metodologijama. Ipak, savremeni razvoj najčešće počiva na primeni (nekih izabranih) principa OO metodologija i agilnog razvoja. U skladu sa tim, obično se podrazumeva da komponenta predstavlja funkcionalnu celinu, koja se sastoji od jedne ili više klasa, koje definišu i implementiraju njen interfejs, i od više pomoćnih klasa, koje implementiraju njenu funkcionalnost. U tom kontekstu, planiranje arhitekture predstavlja definisanje glavnih komponenti i klasa koje čine njihove interfejse, a za fazu implementacije se ostavlja projektovanje ostalih klasa koje čine i implementiraju komponente.

8.7.4 Primer

Radi ilustracije, recimo da je potrebno da se napravi interpretator koji čita i izvršava skriptove na nekom programskom jeziku. U prvom koraku određujemo cilj razvoja – naravno, to je *Interpretator* (*Slika 27*).

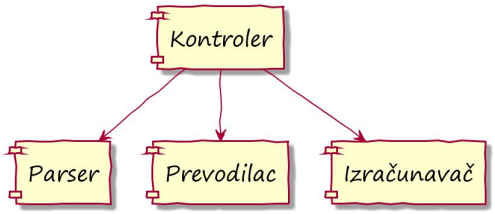
Slika 27 – Početak projektovanja – ceo softver je jedna komponenta

Pretpostavimo da je za naše skriptove najbolje da se obrađuju u tri koraka, koje poveravamo različitim komponentama: najpre će *Parser* da pročita i raščlani skript, zatim će *Prevodilac* da napravi graf izračunavanja koji odgovara pročitanom skriptu i na kraju će *Izračunavač* da izračuna takav grafom predstavljen program (*Slika 28*).



Slika 28 – Prepoznate su glavne komponente

Primetimo da su komponente potpuno razdvojene. Da bismo ih povezali, dodajemo komponentu *Kontroler*, koja će upravljati celim poslom koristeći preostale komponente (*Slika 29*).

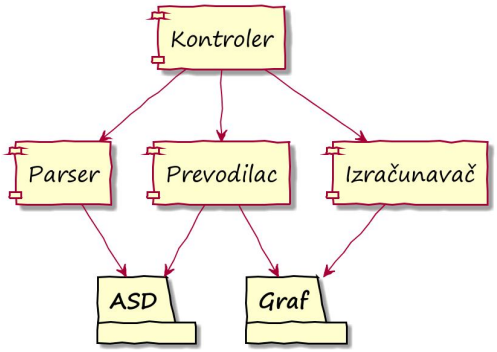


Slika 29 – Prepoznata je potreba da kontroler upravlja poslom

Sada bismo mogli da pređemo na detaljniju analizu pojedinačnih komponenti. Komponente *Parser* i *Prevodilac* moraju biti u stanju da rade sa apstraktnim sintaksnim drvetom (ASD) – *Parser* čita skript i pravi sintaksno drvo, a *Prevodilac* ga koristi da bi napravio graf izračunavanja. Slično tome, i *Prevodilac* i *Izračunavač* moraju da budu u stanju da rade sa grafom izračunavanja. Zbog toga su nam potrebne celine *ASD* i *Graf*. Međutim, ove dve celine nemaju prepoznatu funkciju i odgovarajući interfejs, već se pre radi o vrsti složenih struktura sa kojima želimo da radimo. To znači da, za razliku od prepoznavanja komponenti, ovde imamo logičku i strukturnu a ne funkcionalnu dekompoziciju – želimo da odgovornosti za internu strukturu apstraktnog sintaksnog drveta i grafa izračunavanja izmestimo iz prepoznatih komponenti i grupišemo u posebne celine. Logička dekompozicija nam pomaže da oblikujemo pakete *ASD* i *Graf* (*Slika 30*).

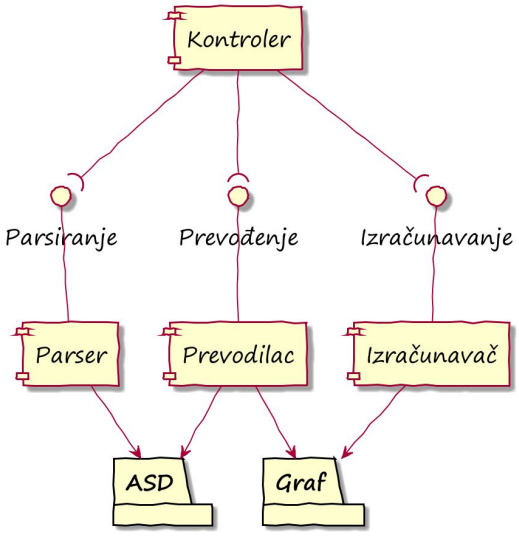
Svaka komponenta mora da ima jasno prepoznat interfejs. On može, ali ne mora da se predstavlja na dijagramu. Ako komponenta ima više različitih interfejsa, onda oni moraju da se navedu. Ako komponenta ima samo jedan interfejs onda nije

neophodno da se on eksplicitno navodi, ali je dobro da bi se komponente bolje razlikovale od paketa (*Slika 31*).



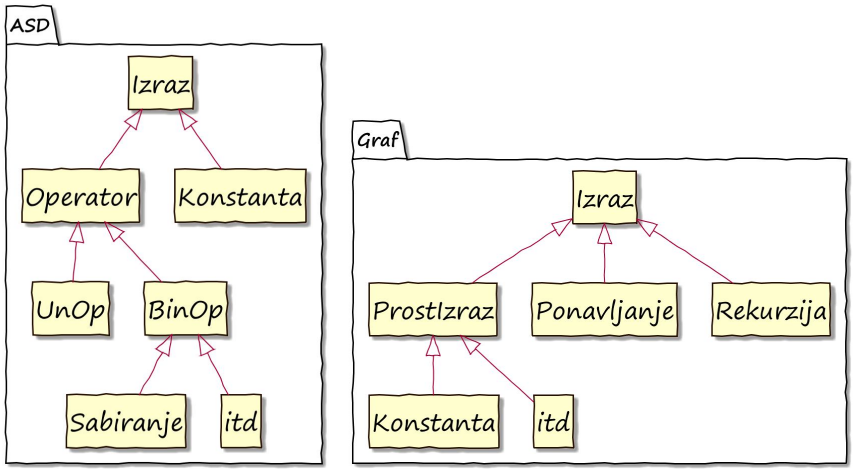
Slika 30 – Dodajemo pakete *ASD* i *Graf*

Paketi *ASD* i *Graf* bi trebalo da sadrže klase koje opisuju sintaksno drvo i graf izračunavanja. Klase koje čine *ASD* se koriste i pri parsiranju i pri prevođenju, pa je važno da se osnovni elementi njihove strukture što preciznije i što ranije odrede, kako bi zatim razvoj komponenti *Parser* i *Prevodilac* mogao da se odvija paralelno, potencijalno od strane različitih timova, umesto da, na primer, započinjanje razvoja komponente *Prevodilac* mora da sačeka dok se pri razvoju komponente *Parser* ne definiše struktura *ASD*. Počeli bismo od prepoznavanja ključnih klasa (*Slika 32*) a zatim bismo nastavili sa definisanjem njihovog interfejsa. (U primeru ćemo preskočiti definisanje interfejsa, zato što je to već nešto specifičniji posao, koji zahteva bolje poznavanje problema, a samim tim i značajno više prostora.)



Slika 31 – Eksplicitno predstavljanje interfejsa

Komponente *Parser*, *Prevodilac* i *Izračunavač* se takođe dalje razrađuju. U slučaju jednostavnijih interpretatora, tu bismo mogli da prođemo bez novih komponenti i da pređemo na analizu algoritama i unutrašnje strukturne organizacije ovih komponenti. U nekom složenijem slučaju, verovatno bismo dodavali još i komponente zadužene za optimizaciju, keširanje međurezultata, koordinaciju paralelizacije i slično.



Slika 32 – Početak modeliranja paketa *ASD* i *Graf*

Drugi aspekt projektovanja je prepoznavanje razvojnih zadataka. Na primer, relativno lako se uočavaju celine poput komponenti ili paketa. Jasno je i da bi bilo dobro da prvo razvijemo pakete *ASD* i *Graf*, pa zatim komponente koje ih koriste i na kraju komponentu *Kontroler*. Štaviše, ako su poznati interfejsi različitih komponenti i strukture paketa, onda njihov razvoj može da ide i paralelno. Naravno, detaljnija analiza komponenti i paketa bi mogla da nam omogući da prepoznamo i neke manje poslove od kojih se sastoje ovi veći, pa da tako izdelimo projekat na još manje celine koje bismo zatim lakše implementirali i čiju bismo realizaciju mogli mnogo preciznije da pratimo.

Umesto da idemo dalje u projektovanje pojedinosti svake od komponenti i paketa, ovde bismo mogli da stanemo i da kažemo da smo se približili okvirnoj arhitekturi našeg projekta. Do formalne arhitekture nedostaju nam još specifikacije interfejsa komponenti i hijerarhija klasa koje čine pakete *ASD* i *Graf*.

8.8 Veličina tima

Većina savremenih preporuka podstiče formiranje agilnih timova koji imaju „3 do 5“, „5 do 7“ ili „oko 7“ članova, dok se kao gornje granice za uspešne timove obično navode 7 ili najviše 9 članova [*Cohn 2009, Rodrigez 2012*]. Razlozi za takve

preporuke nisu slučajni i u najvećoj meri se odnose na različite aspekte efikasnosti ali i ljudskost članova tima. Izdvojićemo neke od njih:

- U velikim timovima specifične sposobnosti pojedinaca teže dolaze do izražaja, pa oni mogu da se osete zapostavljeno i da budu manje zainteresovani i ambiciozni.
- Za uspešnu saradnju članova tima je potrebno da se oni dobro upoznaju. U manjim timovima je za to potrebno manje vremena, pa se brže dolazi do kvalitetne radne atmosfere.
- U manjem timu je lakše sagledati ko šta radi, pa svaki član tima može da bude neprestano informisan o tekućim aktvnostima, što olakšava međusobno savetovanje i pomaganje.
- Komunikacija članova većih timova zahteva više vremena i napora. Samim tim pada i efikasnost, zato što ostaje manje vremena za konkretan rad na projektu.
- Dogovori se lakše postižu u timovima koji nemaju mnogo članova. U većim timovima lakše dolazi do podela na manje grupe, među kojima je onda otežana komunikacija i saradnja.
- U slučaju većih timova, dešava se da neki članovi učestvuju u radu više timova, čime se delimično gubi fokus i otežava upravljanje radnim vremenom i tokom razvoja. To se najviše odnosi na eksperte za neke uže oblasti, čija su iskustva potrebna u više timova, ali najčešće ne tokom čitavog razvojnog ciklusa.

Savremena praksa pokazuje da su takve preporuke uglavnom ispravne. Ipak, postoje okolnosti koje mogu da podstaknu pravljenje nešto većih timova. Različita optimalna veličina tima može da bude posledica većeg broja faktora, među kojima su i priroda problema koji se rešava, životni period projekta (na primer, da li se pravi novi proizvod ili održava postojeći), skup tehnologija koje se koriste, visoka složenost projekta (razvoj novih algoritama i sl.) i drugi faktori.

Pri razvijanju sasvim novog proizvoda ili čak nove tehnologije, važno je da tim raspolaže dovoljno bogatim iskustvom i znanjem da bi mogao da uspešno dođe do rešenja. Sličan problem imamo i ako se tokom razvoja zahteva kombinovanje različitih alata ili postupaka. Mali timovi raspolažu manjim znanjem nego veći i u stanju su da kvalitetno pokriju manje različitih tehnologija i specijalnosti, pa se zato često teži da se povećavanjem timova podignu obim i nivo njihovih sposobnosti.

Neki problemi ne mogu jednostavno da se podele na manje celine. Ostavljanje takvih problema u celovitom obliku ima za posledicu velike zadatke. Ako se veliki zadatak poveri malom timu, onda potrebno vreme angažovanja tima na tom zadatku može da pređe neke prihvatljive granice, bilo zato što može da dovede u

pitanje rokove, ili zato što produženo angažovanje na jednom istom problemu može da dovede do zamora i pada efikasnosti u timu. U takvim slučajevima ima smisla da se napravi veći tim, a da se potencijalne negativne posledice rešavaju učestalim internim raspoređivanjem na manje grupe od po 2-3 člana, nalik na programiranje u paru, kao i prilagođavanjem dinamike redovnih sastanaka.

U slučaju pravljenja većih timova obično može da se postavi pitanje da li je to uopšte jedan veliki tim ili samo skup malih timova koji međusobno sarađuju? Odgovor zavisi od prirode posla i dinamike pravljenja i menjanja manjih grupa (tj. delova tima). Ako manji delovi posla ne predstavljaju samostalne celine, već samo male neodvojive delove velikog problema, koji se pri tome implementiraju relativno brzo, tako da se male radne grupe prave svakodnevno, pa možda i više puta u toku dana, onda to svakako predstavlja jedan tim. Primetimo da je takva praksa uglavnom prihvatljiva za Ekstremno programiranje, a da se u Skramu teži da se izbegava.

8.9 Kritički pogled na agilni razvoj

Ništa nije savršeno, pa ni agilni razvoj softvera i predstavljene agilne metodologije. Dok neki autori i timovi ističu kvalitete ovakvog pristupa problemu razvoja softvera, neki drugi, naprotiv, izražavaju velike sumnje u njegovu efikasnost. Primedbe su različite i odnose se kako na principe agilnog razvoja, tako i na konkretne tehnike ili prakse koje se koriste za njihovo ostvarivanje.

Svaka metodologija u osnovi počiva na ljudima koji je primenjuju. Nijedna metodologija ne može da u potpunosti prevaziđe potencijalne ljudske slabosti u razvojnom timu. Specifičnost agilnih metodologija je u tome da to uglavnom i *ne pokušavaju*. Naprotiv, one pretpostavljaju i ističu u prvi plan ljudske i stručne kvalitete članova tima. Neki principi i prakse ovih metodologija imaju za cilj održavanje visokog nivoa stručnosti i dobrih odnosa u timu, ali nijedan princip ni praksa ne nude konkretna praktična rešenja za slučaj kada se odnosi u timu ipak poremete. Ova slabost metodologije je prilično nezgodna, ali moramo da primetimo da se ni najveći broj drugih metodologija ne razlikuje mnogo od njih po tom pitanju.

Značajan problem može da predstavlja pretpostavljen relativno visok nivo stručnosti članova tima. Zbog toga se početnici često relativno sporo uklapaju u agilne timove i pri tome predstavljaju slabe karike u razvoju. Agilni razvoj podrazumeva da je svaki programer u stanju ne samo da piše programski kod već i da ga *dobro* dizajnira, ali u praksi je teško da se od početnika očekuje dobro projektovanje.

Iako zbog dinamike razvojnog procesa i programiranja u parovima može da izgleda da agilni razvojni timovi lako trpe izmene u sastavu, stvari ipak stoje malo drugačije. Odlazak pojedinačnog člana tima se relativno lako podnosi, zbog opšte uključenosti tima u sve delove projekta. Sa druge strane, zbog relativno inertnog odnosa prema pisanju dokumentacije, istovremeni odlazak više članova tima može

da ostavi mnogo teže posledice nego u slučaju primene neke od metodologija sa klasičnim pristupom pisanju dokumentacije. Dolazak nekog novog člana tima obično ne predstavlja problem ako se radi o iskusnom programeru, ali dolazak novog početnika može da bude veoma stresan i za pridošlicu i za starije članove tima. Jedan od osnovnih uzroka je način organizacije rada u timu, tj. praktično istovremeno uključivanje novog člana tima u sve aktivnosti tima. Dodatni problem je u tome što svaki agilni tim ima donekle specifičnu radnu kulturu (pre svega u kontekstu načina komunikacije i organizacije rada), pa je novim članovima obično neophodan period privikavanja na uslove rada.

Jedna od potencijalno problematičnih posledica primene agilnih metodologija je i specifičan vid ugovaranja poslova. Ne ugovaraju se dovršeni projekti, već se umesto toga ugovaraju međusobni odnosi klijenta i razvijalaca, poput obima mesečnog angažovanja i nekog prognoziranog napretka. Međutim, to jednom broju potencijalnih klijenata može da predstavlja veliki problem, zato što su teže sagledivi ukupan obim radova, rokovi i cena. Nekim klijentima više odgovara da ugovore fiksne rokove i troškove, čak i ako su u njih ugrađene i velike rezerve (za slučajeve nastupanja rizika) i potencijalno nerealno visok profit razvijalaca. To ponekad može da značajno oteža ugovaranje poslova. Neki autori ističu kao veliki problem to što ugovaranje bez definisanog cilja može da se zloupotrebi za izvlačenje novca od klijenta, a da se nikada ne dođe do zadovoljavajućeg zaokruženog rezultata, iako je alternativan klasičan pristup, koji je praktično jednako rizičan a i skoro izvesno mnogo skuplji.

Negativna posledica iterativnog razvoja može da bude zapostavljanje korisničkih celina koje ne donose neposredan profit, pa zato iz ugla klijenta imaju nizak prioritet, a mogu da suštinski utiču na ukupnu upotrebljivost razvijanog softvera. Takve celine mogu da se postepeno odlažu za kasnije iteracije, a na kraju neke od njih mogu da ostanu i trajno neimplementirane. To je često posledica prekoračenja ukupno obezbeđenih rokova ili troškova. Pri tome se obično ne uzima u obzir da su ti isti rokovi i sredstva obuhvatili i neke celine koje inicijalno nisu ni bile planirane, pa je zbog povećavanja obima radova kao sasvim prirodna posledica došlo i do prekoračenja rokova ili troškova. Jedino sredstvo za sprečavanje zapostavljanja inicijalno zacrtanog velikog cilja je neki vid *vizije* ili *metafore*. U vezi sa načinom planiranja postoje i određene teškoće pri definisanju nefunkcionalnih kvalitativnih zahteva u vidu korisničkih celina. I njihovo ispunjavanje može na kraju da bude dovedeno u pitanje.

Dosledna težnja jednostavnim rešenjima i detaljnom planiranju samo tekuće iteracije može u nekim slučajevima da ima neugodne posledice. Postoje sistemi koji su po svojoj prirodi veoma složeni i *ne mogu* da se opišu jednostavnim modelima. Iterativni razvoj predstavlja pritisak da se neki problemi *veštački* dele na manje celine, kako bi mogli da se upakuju u iteracije. Ako se i u takvim slučajevima dosledno slede principi i prakse agilnih metodologija, onda se pri implementaciji jedne celine

svesno ne uzima u obzir da će naredna celina *morati* da se implementira, pa zato može da se iskoristi neprimereno jednostavan dizajn, koji će zatim u više narednih iteracija morati da se *značajno menja*. Temeljnost izmena koje su u takvim slučajevima neophodne može da ima za posledicu drastično smanjivanje efikasnosti razvojnog procesa. U takvim slučajevima je preporučljivo da se odustane od striktne primene agilnih principa i da se primeni planiranje i apstrahovanje unapred.

Jednako su problematični slučajevi kada neki deo softvera *ne može* da se podeli na manje celine koje imaju smisla. Neke probleme je bolje rešavati kao celinu, a ne po delovima. U tom slučaju može da se vanredno produži iteracija, ali to onda odstupa od propisanog načina planiranja iteracija i ustaljenog ritma razvoja i može da napravi određene probleme kako unutar tima tako i u odnosu sa klijentom. Iako najveći broj problema može da se podeli na manje celine i prilagodi agilnom razvoju, u nekim slučajevima je za uspešan i efikasan razvoj ipak neophodan širi plan. Slično je i u slučaju sistema za koje je od samog početka jasno da će im na kraju biti neophodna složena infrastruktura (na primer informacioni sistemi), zato što prakse agilnih metodologija izradu infrastrukture često odlažu i nepotrebno cepkaju na parčiće.

Često je predmet zamerki odsustvo sistematično uređene dokumentacije. Agilni timovi razvijaju dokumentaciju prema potrebi, što može da ima za posledicu da je ona parcijalna i rascepkana, tj. da pokriva pojedinačne delove softvera i sastoji se od skupova dokumenata koji ne predstavljaju objedinjenu celinu. Takva dokumentacija je dovoljna članovima tima dok rade na razvoju, ali često ne odgovara nekome ko će kasnije morati da održava ili dograđuje razvijen softver. Dosledna primena agilnih principa bi bila da se po završetku razvoja napravi dovoljno temeljna objedinjena dokumentacija (*tek tada*, zato što je tek tada potrebna, radi budućeg održavanja), ali iterativni pristup planiranju i razvoju ima za posledicu da u trenutku raskidanja ugovora i deklarativnog završavanja posla, obično više nema vremena za dodatno staranje o dokumentaciji. Možemo da primetimo da bi ovu zamerku trebalo uputiti ne samoj metodologiji, nego onima koji je nedosledno primenjuju.

Primedbe koje se odnose na stalno učešće predstavnika klijenta u razvojnom timu odnose se pretežno na dva aspekta problema. Prvi je cena, zato što klijent i razvijalac često nisu u istom gradu, pa se za stalnu saradnju mora obezbediti ili trajni boravak predstavnika klijenta u razvojnom timu, ili se moraju pokriti redovni troškovi putovanja. Dodatni uticaj na troškove ima i iterativni razvoj, zato što su neophodni širi sastanci klijenata i razvojnog tima radi analize urađenog i sagledavanja eventualnih primedbi na rezultate iteracija i izdanja. Deo tih troškova može da se prevaziđe primenom savremenih telekomunikacionih tehnologija, ali su sastanci uživo uvek poželjniji vid komunikacije.

Drugi aspekt problema je što stalna prisutnost predstavnika klijenta može da proizvede lavinu izmena zahteva. Iako je agilni razvoj naklonjen redovnim

izmenama zahteva, neodmerena količina takvih izmena može da napravi velike probleme i zastoje u napredovanju razvoja. Iako ovakve primedbe nisu sasvim opravdane, zato što praktično sve agilne metodologije imaju neki mehanizam sprečavanja lavine izmena zahteva, one ipak ukazuju na realnu opasnost. Neka od uobičajenih rešenja su, na primer, ograničenja da zahtevi i izmene mogu da se podnose samo tokom planiranja iteracije, ili samo u nekom unapred definisanom početnom periodu razvojnog ciklusa.

Relativno često se nailazi na primedbu da postoji gornja granica složenosti softvera koji može efikasno da se razvija agilnim metodologijama. Jedan od pristupa, koji se koriste u složenijim razvojnim okruženjima, je da se veliki problemi (na primer informacioni sistemi) temeljno analiziraju i da se zatim isplanira opšta arhitektura rešenja. Tek onda, kada su prepoznate i razdvojene osnovne komponente sistema, od kojih svaka ima sagledivu složenost, svaka od komponenti se prepušta agilnim timovima na detaljno projektovanje i razvijanje [*Cummings 2008*]. Sa druge strane, postoje i ozbiljne analize načina prilagođavanja agilnih metodologija velikim timovima i složenim problemima. Na primer, postoji mišljenje da je metodologija Skram posebno dobra za veoma velike projekte [*Schiel 2009*].

Neke od kritika na račun agilnih metodologija su opravdane, ali su neke druge posledica nedovoljnog razumevanja principa agilnog razvoja i tehnika konkretnih metodologija. Posebno je teško vrednovati kritike koje se odnose na pojedinačne prakse, u slučajevima kada se njihove slabosti ispoljavaju, između ostalog, zato što nisu primenjivane neke druge prakse, iako metodologija zahteva da se uvek primenjuju zajedno.

8.10 Umesto zaključka

Kao što se OO programiranje izdvojilo kao vodeća paradigma programiranja, tako su se agilne metodologije [*Agile Alliance*; *Agile Manifesto*] izdvojile i našle svoje mesto u brojnim razvojnim projektima, bez obzira na veličinu projekata i u praktično svim domenima. Agilne metodologije su do sada već mnogo puta potvrđene u praksi, tako da su postale praktično najzastupljenija vrsta metodologija u savremenom razvoju softvera. Odlično se kombinuju sa OO metodologijama, pa možemo da primetimo i da su mnoge knjige o agilnom razvoju jednako posvećene i elementima OO metodologija [*Martin 2003*].

Danas je najzastupljenija agilna metodologija Skram [*Schwaber 2020*, *Rubin 2013*]. Iako jasno i precizno definiše pravila odvijanja razvojnog procesa, Skram je fleksibilan u odnosu na izbor i primenu konkretnih razvojnih tehnika i praksi, pa se obično koristi u kombinaciji sa praksama Ekstremnog programiranja [*Back 1999*, *2004*, *Wake 2000*].

Kada je reč o primeni agilnih metodologija, važno je da istaknemo da se pokazuje da najveći broj timova, koji tvrde da primenjuju agilne metodologije, zapravo ne

primenjuje nijednu konkretnu metodologiju već samo neke izabrane skupove agilnih tehnika i praksi. Primena agilnih metodologija nosi sa sobom određene rizike o kojima je potrebno voditi računa. To je posebno značajno u slučajevima kada se ne primenjuje neka celovita metodologija već samo izabrane tehnike i prakse.

9 - Razvoj vođen testovima

Problem sa programerima je
što nikada ne možete da budete sigurni šta rade,
sve dok ne bude prekasno.
Sejmur Krej

9.1 Testiranje softvera

Testiranje softvera je postupak proveravanja da li se razvijeni softver ponaša na ispravan način, u nekim izabranim reprezentativnim slučajevima. Testiranje ima veoma važnu ulogu u okviru obezbeđivanja kvaliteta proizvoda, pa predstavlja jedan od nezaobilaznih poslova u okviru razvojnog procesa. Obično je najzastupljenije u završnim fazama razvoja softvera ili njegovih pojedinačnih delova, ali neke vrste testova imaju svoje mesto i u drugim fazama razvoja.

Važno je da istaknemo da sprovedeno testiranje ne može da predstavlja *dokaz* ispravnosti softvera, već samo može da potvrdi da softver radi ispravno za jedan unapred određen skup test primera. Eventualno dokazivanje korektnosti je predmet formalne verifikacije softvera i ne može da se ostvari testiranjem. Ipak, dobro izabran skup testova može da značajno umanji verovatnoću postojanja neprimećenih grešaka u kodu, pa zbog toga dobro isplanirano i dovoljno temeljno i široko testiranje ima veliku težinu u okviru procenjivanja kvaliteta softvera. Zbog toga se testiranje često naziva i *neformalnom* verifikacijom.

Pri razvoju softvera se u kontekstu provere ispravnosti upotrebljavaju (i često mešaju) dva termina: *verifikacija softvera* je proveravanje da li je softver razvijen u skladu sa ustanovljenim formalnim i neformalnim specifikacijama i kriterijumima kvaliteta, dok je *validacija softvera* proveravanje da li softver zadovoljava stvarne potrebe naručioca. Često kažemo da je verifikacija proveravanje da li se softver *dobro*

pravi, a da je validacija proveravanje da li pravimo *dobar softver*. Većina testova se odnosi na verifikaciju.

Da bi testiranje softvera imalo značaja, ono mora da bude obavljano sistematično i sa punim razumevanjem softvera koji se testira. Pri planiranju i projektovanju testova neophodno je sveobuhvatno i temeljno poznavanje ciljeva i projektnih zahteva, kao i svih elemenata arhitekture i implementacije softvera.

U razvojnom procesu svoje mesto pronalazi veći broj različitih vrsta testova, koji se primenjuju u različitim periodima razvoja i sa različitim ciljevima. Testovi mogu da se razlikuju i klasifikuju u odnosu na neka od svojstava, kao što su, na primer, cilj testiranja, predmet testiranja, obim testiranja, vreme testiranja i drugo. Najpre ćemo predstaviti vrste testova prema cilju i obimu, a zatim ćemo se temeljnije posvetiti testovima jedinica koda.

Vrste testova prema cilju

U odnosu na cilj testiranja prepoznaju se tri velike kategorije testova:

- *funkcionalni testovi*;
- *nefunkcionalni testovi* i
- *holistički testovi*.

Funkcionalni testovi obuhvataju sve vrste testova koji imaju za cilj proveravanje da li jedinice koda, komponente ili softver kao celina rade u skladu sa zahtevima, specifikacijama i dokumentacijom:

- *testovi ispravnosti* proveravaju da li funkcije i metodi izračunavaju ispravne rezultate i proizvode ispravne promene stanja sistema, kao i da li upotreba aplikativnog interfejsa softvera proizvodi planirane posledice;
- *testovi kompatibilnosti* služe da se proveri da li se softver povezuje sa drugim softverom na način na koji je to predviđeno.

Nefunkcionalni testovi obuhvataju sve one vrste testova kojima nije primarni cilj proveravanje ispravnosti rezultata rada softvera i njegovih komponenti:

- *testovi upotrebljivosti* imaju za osnovni cilj proveravanje da li je korisnički interfejs dovoljno upotrebljiv za ciljne korisnike; kao specijalni slučajevi testova upotrebljivosti prepoznaju se još:
  - *testovi upotrebljivosti za korisnike sa posebnim potrebama*, koji služe da se provere specifični aspekti korisničkog interfejsa, i
  - *regionalni testovi*, koji služe da se proveri prilagođenost softvera različitim jezicima ili lokalnim okolnostima upotrebe;

- *testovi performansi* podrazumevaju višestruko izvršavanje značajnih operacija u različitim okolnostima, radi procenjivanja efikasnosti sistema i njegovih delova u realnim uslovima[40]; prema vrsti testiranih performansi ovi mogu da obuhvate:
  - *testovi efikasnosti*, koji mere brzinu rada ili vreme odziva softvera u uslovima predviđenog radnog opterećenja;
  - *testovi opterećenja*, koji mere opterećenje razliitih komponenti i prate njihovo ponašanje pod povećanim opterećenjem[41];
  - *testovi stabilnosti* procenjuju kako se softver ponaša u uslovima ekstremnog opterećenja, koje prevazilazi okvire za koje je softver razvijen; fokus je na ustanovljavanju da li dolazi do značajnih padova performansi ili čak i potpunog otkaza rada softvera;
  - *testovi istrajnosti* proveravaju kako se softver ponaša u uslovima produžene neprekidne upotrebe;
  - *testovi skalabilnosti*, koji proveravaju da softver može da se skalira i koliko se to lako ostvaruje u slučaju povećanog opterećenja i
  - *testovi uskih grla* su fokusirani na najopterećenije delove softvera i hardvera; slični su testovima opterećenja i testovima stabilnosti
  - i drugo;
- *testovi bezbednosti* proveravaju da li su svi bezbednosni aspekti sistema ispravno implementirani – na primer, da li neidentifikovan korisnik može da pristupi zaštićenim elementima softvera, da li identifikovan korisnik može da pristupi elementima za koje nema autorizaciju i drugo;
- *testovi instalacije* obuhvataju instaliranje softvera u različitim uslovima i proveravanje da li su na kraju instalacije svi delovi sistema ispravno konfigurisani i pripremljeni za rad;
- *destruktivni testovi* proveravaju koliko je sistem otporan na (namerne i slučajne) pokušaje da se dovede u neaktivno stanje;

Holistički testovi su celoviti testovi softvera od strane korisnika. Prema trenutku u kome se testiranje odvija i vrsti korisnika koji vrše testiranje, prepoznajemo nekoliko tipova holističkih testova:

[40] Iako slabe performanse mogu da dovedu u pitanje praktičnu funkcionalnost softvera, ipak je uobičajeno da se pitanja vezana za performanse softvera svrstavaju u nefunkcionalne zahteve.

[41] Merenje vremena odziva i propusnosti softvera ili njegovih delova se svrstava i u testove opterećenja i u testove efikasnost.

- *razvojni testovi* predstavljaju redovno celovito testiranje upotrebe softvera tokom njegovog razvoja od strane članova razvojnog tima, koji su specijalizovani za poslove testiranja;
- *testovi prihvatljivosti* se odvijaju u odnosu na celovit softverski proizvod i proveravaju da li on ispunjava uslove i zadovoljava zahteve određene projektnim zadatkom; predmet testova prihvatljivosti može biti konačan proizvod, rezultat pojedinačne iteracije ili čak samo deo sistema koji predstavlja ostvarenje pojedinačnog slučaja upotrebe ili korisničke celine;
- *alfa testiranje* je celovito testiranje od strane uže grupe korisnika, obično relativno dobro tehnički obrazovanih, čija je namena da se u relativno ranoj fazi proizvodnje (kada su uglavnom sve mogućnosti softvera bar delimično implementirane, ali još ne i do kraja doterane) prikupe kritička mišljenja; obično primarni cilj nije uočavanje bagova u implementaciji već grešaka i slabosti u konceptima, planovima i korisničkom interfejsu;
- *beta testiranje* je celovito testiranje koje se sprovodi od strane šire grupe korisnika, koji se biraju tako da što približnije odgovaraju ciljnoj grupi korisnika gotovog proizvoda; primarni cilj je blagovremeno uočavanje bagova, ali se testeri podstiču da komentarišu i konceptualne greške.

Alfa i beta testiranje se po pravilu primenjuju u slučaju softvera koji je namenjen širokom krugu korisnika. Mogu da se primenjuju i u slučaju softvera koji se radi namenski (npr. informacioni sistem nekog preduzeća) ali onda imaju nešto drugačiju prirodu. Nasuprot tome, testovi prihvatljivosti imaju veći značaj u slučaju namenskog softvera, dok se kod softvera za širi krug korisnika umesto njih često koriste nešto širi razvojni testovi.

Testovi prihvatljivosti su konceptualno drugačiji od ostalih vrsta testova, zato što se primarno bave načinom upotrebe i rezultatima upotrebe softvera iz ugla korisnika, a ne samo tehničkim aspektima realizacije. Oni se najviše bave testiranjem ponašanja sistema kao celine. Implementiraju se drugačijim alatima nego druge vrste testova. Često se definišu specifičnim skript jezikom, koji prateći alati mogu da izvršavaju simulirajući realnu upotrebu softvera (kroz simuliranje događaja koji nastaju upotrebom tastature ili miša i slično).

Vrste testova prema obimu

Prema obimu ili nivou testiranja obično se prepoznaju:

- *testovi jedinica koda*;
- *testovi komponenti;*
- *testovi integracije* i
- *testovi sistema*.

Testovi jedinica koda služe za proveravanje ispravnosti najmanjih funkcionalnih jedinica koda. Najpre se pojedinačni testovi upotrebljavaju za testiranje ispravnosti funkcija i metoda. Svaki pojedinačan test bi trebalo da proverava tačno jedan aspekt ponašanja testirane jedinice. Ispravno izgrađen skup testova bi trebalo da proverava sve aspekte ponašanja testirane funkcije ili metoda, uključujući ponašanje za različite ulazne vrednosti, kao i u različitim stanjima sistema. Neophodno je da se posebno proverava ponašanje za sve granične slučajeve, kao i za neispravne ulazne vrednosti.

Kada se više jedinica koda povezuje u jednu celinu, onda način testiranja ispravnosti te celine zavisi od načina povezivanja delova. Ako se delovi povezuju korišćenjem osnovnog načina povezivanja jedinica koda u konkretnom programskom jeziku (npr. pozivanje funkcija ili razmenjivanje poruka između objekata), onda ispravnost celine može da se proverava testovima jedinica koda. Sa druge strane, ako se delovi povezuju putem nekog drugog protokola (na primer, putem protokola HTTP), onda se testiranje ispravnosti takvog interfejsa obično ne obavlja testovima jedinica koda, već moraju da se koriste testovi komponenti ili testovi integracije[42].

Testovi komponenti se nalaze negde između testova jedinica koda i testova integracije. Često se u literaturi ne navode kao posebna vrsta testova, već se svrstavaju u testove integracije. Primarni predmet testiranja komponente je testiranje ispravnosti implementacije njenog interfejsa. Interno funkcionisanje delova komponente, pa čak i njenog celovitog funkcionisanja, sve do nivoa javnog interfejsa ali bez njega, se obično proverava već u okviru testiranja jedinica koda.

Interfejs komponente se najčešće definiše pomoću jedne klase, kojom se zaklanja implementacija. To je tipična primena obrasca za projektovanje *Fasada*. U tom slučaju funkcionalnost komponente i njenog interfejsa u suštini može da se svede na testiranje interfejsa fasadne klase, što može da se radi odgovarajućim testovima jedinica koda, ako fasadna klasa ima i interni osnovni interfejs.

Međutim, spoljašnji interfejs komponente (onaj interfejs koji vide i koriste druge komponente, tj. interfejs koji se koristi u realnom okruženju) može da koristi neke protokole povezivanja koji se razlikuju od osnovnih načina povezivanja jedinica koda u konkretnom programskom jeziku. Tada njegovo testiranje ne može da se obavi testovima jedinica koda. Tehnika testiranja interfejsa pojedinačne komponente se obično ne razlikuje značajno od tehnike sprovođenja testova integracije, pa se zato

[42] Naravno, čak i takvo testiranje može da se sprovede pomoću testiranja jedinica koda, ali je problem u tome što upotreba dodatnih protokola obično zahteva širu pripremu (na primer instaliranje, konfigurisanje i pokretanje modula koji se povezuju) i njegova efikasnost je za red veličine niža od većine „običnih“ testova jedinica koda, pa se zato to obično radi malo drugačije.

testiranje komponenti često posmatra kao donekle specifičan oblik testova integracije.

Testovi integracije (ili *integralni testovi*) služe za proveravanje da li se komponente ispravno povezuju u veće celine. Neophodan uslov za testiranje integracije je da je prethodno uspešno sprovedeno testiranje svih pojedinačnih komponenti koje se integrišu u celinu.

Za testove integracije nije dovoljno pisanje jednostavnih testova, kao za testove jedinica koda. Obično se pišu posebni programi koji simuliraju povezivanje komponenti u ciljnom okruženju i njihovu upotrebu u relativno složenom kontekstu. Ako se komponente povezuju i upotrebljavaju putem nekog od standardizovanih protokola (na primer, Veb servisi ili REST), onda mogu da se upotrebljavaju i komercijalni alati za testiranje, koji obično omogućavaju pisanje skriptova za testiranje.

Testovi sistema se odnose na testiranje celog sistema. Mogu da imaju odlike testova integracije (ako se sistem posmatra iz tehničkog ugla) ili odlike testova prihvatljivosti (ako se sistem posmatra iz ugla korisnika), a najčešće predstavljaju njihovu kombinaciju. U svakom slučaju, testovi sistema podrazumevaju testiranje čitavog proizvoda i svih njegovih karakteristika.

Za razliku od ostalih navedenih vrsta testova, testovi sistema mogu da se bave i aspektima validacije, a ne samo verifikacije softvera. Tokom čitavog razvoja validacija može samo da se bavi pitanjem da li su specifikacije i kriterijumi kvaliteta ustanovljeni u skladu sa stvarnim potrebama klijenta ili ne, a tek kada je proizvod gotov, onda može da obuhvati i neke praktične provere. Dok se ostali testovi u suštini bave samo proveravanjem da li su zadovoljene specifikacije i formalni kriterijumi kvaliteta (tj. bavi se verifikacijom), testovi sistema mogu da obuhvate i neformalne kriterijume i da proveravaju da li je zaista razvijen proizvod kakav je bio potreban, ili je iz nekog razloga razvijen pogrešan proizvod.

9.2 Testovi jedinica koda

Kao što je već navedeno, osnovna namena testova jedinica koda je proveravanje ispravnosti najmanjih funkcionalnih jedinica koda. Suština testiranja jedinice koda je u tome da se na praktičan način, proveravanjem ponašanja u nekim uobičajenim, specifičnim ili graničnim uslovima proveri da li se pri upotrebi programskog interfejsa jedinice koda dobijaju rezultati koji odgovaraju specifikaciji.

Najčešći predmet testiranja su funkcionalni zahtevi. Oni se testiraju kroz proveravanje da li se pri upotrebi interfejsa jedinice koda potvrđuju očekivane zavisnosti postuslova od preduslova:

- Da li rezultat funkcije ili metoda odgovara datim ulaznim vrednostima i datom početnom stanju sistema?
- Da li se na ispravan način menja stanje sistema?

Pored funkcionalnih zahteva može da se testira i robusnost jedinice koda:

- Da li se funkcije i metodi ponašaju na odgovarajući način u slučaju neispravnih ulaznih vrednosti?
- Da li se ispravno izveštava o problemima?
- Da li se ispravno izbacuju izuzeci?

Jedinica koda koja se testira može da bude jedna operacija, funkcija ili metod, neka struktura podataka ili klasa. Jedinica koda može da predstavlja i skup više manjih integrisanih jedinica koda. Ako ima odgovarajući programski interfejs, koji može da posluži za testiranje, onda kao jedinica koda može da se testira čak i softverski paket ili neki interni ili eksterni podsistem.

Pojedinačni testovi testiraju pojedinačne aspekte ponašanja pojedinačnih funkcija ili metoda. Sistematično izgrađene kolekcije testova se upotrebljavaju za testiranje ponašanja čitavih klasa ili paketa. Testiranju složenog ponašanja klase, tj. ispravnosti složenije upotrebe objekata i metoda klase, može da se pristupi tek nakon što se prethodno testira elementarno ponašanje svakog pojedinačnog metoda klase. Ako više klasa sarađuju međusobno u većem podsistemu, onda i ispravnost takvog podsistema može da se proverava testovima jedinica koda.

Testovi jedinica koda se pišu kao tzv. *neprodukcioni* kod za testiranje, tj. programski kod koji se piše radi testiranja neće biti sastavni deo proizvoda, već mu je isključiva namena testiranje. Obično se pri pisanju testova jedinica koda koriste odgovarajuće biblioteke za testiranje, a u specifičnim slučajevima može da bude potrebno čak i razvijanje sopstvenih delova koda koji služe samo za olakšavanje implementiranja testova jedinica koda.

Uobičajeno je da se testovima jedinica koda testira samo javni interfejs klasa i paketa. Ipak, u nekim kompleksnijim slučajevima je dobro da se omogući i testiranje privatnih delova, u kom slučaju se dodaju posebni metodi ili klase čija isključiva namena je da pomognu pri testiranju. Takve delove nekada nije lako izbaciti iz produkcionog koda, pa se dešava da ostanu u njemu. Poželjno je da se ipak uloži dodatni napor da se kod organizuje tako da oni mogu da se izbace[43].

[43] Na primer, jedna tehnika za testiranje privatnih elemenata klase `A` je da se deklariše prijateljska klasa (npr. `friend class TestHelperA`) i da se u njoj napišu odgovarajući

Testovi jedinica koda nisu dovoljni da dokažu ispravnost softvera. Dobro oblikovani testovi će dovesti do ispoljavanja većine bagova, ali oni praktično nikada ne mogu da obuhvate sve moguće uslove u kojima softver može da se nađe. Ako je potrebno da se dokaže korektnost softvera, onda je neophodno da se upotrebljavaju neke formalne metode verifikacije, bilo manuelne ili automatske. Sa druge strane, dobro oblikovana kolekcija testova će moći da nam pomogne da prepoznamo i otklonimo većinu bagova.

9.3 Biblioteka Catch2

Postoji veliki broj biblioteka koje pružaju podršku testiranju jedinica koda na programskom jeziku *C++*. Kao primer ćemo da ukratko predstavimo biblioteku *Catch2* [*Catch2*], koja tokom poslednjih nekoliko godina postaje sve zastupljenija[44].

Kod većine starijih biblioteka postoji problem netrivijalnog pisanja i izvođenja testova. Najčešće svaki test ili grupa testova mora da se na neki način registruje da bi bila izvedena. Pri razvoju biblioteke *Catch2* je osnovna motivacija bila prevazilaženje tih uobičajenih problema. Zaista, jedan od osnovnih razloga što je biblioteka *Catch2* postala popularna, jeste jednostavnost pisanja, prevođenja i izvođenja testova. Ona ne zahteva nikakvo dodatno povezivanje ili registrovanje testova – sve se odvija automatski. Kao što ćemo videti, programer je dužan samo da napiše testove i ne mora da se zatim stara o njihovoj tehničkoj implementaciji. Da bismo preveli i povezali testove dovoljno je da prevedemo i povežemo sve module koji su potrebni za testiranje (modul programa za testiranje i module sa testovima) i ostale module koje koristimo pri testiranju.

Najveća mana biblioteke *Catch2* je u tome što se distribuira i prevodi kao biblioteka u jednom zaglavlju, koja intenzivno koristi parametarski polimorfizam. Iako to čini upotrebu jednostavnijom, za posledicu ima relativno sporo prevođenje testova i programa za testiranje. Iz tog razloga se od verzije 3 prelazi na upotrebu statičke biblioteke.

U primerima ćemo da koristimo aktuelnu stabilnu verziju 2 (u trenutku pisanja ovog teksta to je verzija 2.13.10), koja se distribuira kao biblioteka u jednom za-

metodi. Ne predstavlja problem ako takva deklaracija ostane u produkcionom kodu, ali ne bi bilo dobro da u njemu ostane i definicija te klase (`TestHelperA`).

[44] Broj 2 u imenu biblioteke nije oznaka verzije nego deo imena. Ova biblioteka se originalno zvala *Catch* ali je autor primetio da takvo ime pravi probleme kada se traži na vebu, pa je sa prelaskom na verziju 2 biblioteci promenjeno ime u *Catch2*. U planu je da nove verzije zadrže isto ime, pa je tako u vreme pisanja ove knjige u toku dovršavanje verzije 3, koja se i dalje zove *Catch2*.

glavlju. U slučaju verzije 3 jedina razlika je u tome što je potrebno da se uključe druga zaglavlja i da se poveže statički prevedena biblioteka.

Pisanje programa za testiranje

Biblioteka *Catch2* sadrži makroe za testiranje i program koji izvršava testove. Da bismo u naš program ugradili komandni program za testiranje, tj. funkciju `main` iz biblioteke *Catch2*, koja izvodi testiranje, potrebno je da napišemo samo:

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"
```

Prvi red definiše konfiguracioni makro koji sugeriše da je potrebno da se ugradi funkcija `main` programa za testiranje. Drugi red uključuje zaglavlje, koje ako je definisan prethodni konfiguracioni makro, uključuje sva druga potrebna zaglavlja i odgovarajuću definiciju funkcije `main`.

Program za testiranje možemo da pokrećemo uz upotrebu velikog broja konfiguracionih opcija. U najvažnije opcije spadaju opcije za odabir testova koje je potrebno izvesti, što nam omogućava da se u slučaju većih kolekcija testova fokusiramo samo na određene podskupove.

Svaki test ima ime. Kao parametar pokretanja programa za testiranje možemo da navedemo ime testa koji je potrebno da se izvrši, ali možemo da koristimo i džoker znake. Takođe, možemo da odaberemo i testove koje ne želimo da izvodimo.

Pored imena, svaki test može da ima i jednu ili više *oznaka* (engl. *tag*) koje nam služe za grupisanje ili klasifikovanje testova. Na primer, oznaka može da predstavlja ime klase čije metode testiramo, ime skupa operacija ili bilo šta drugo. Ako među oznakama nekog testa navedemo i neku koja počinje tačkom (ili upravo oznaku tačka „`[.]`“), onda je test podrazumevano *skriven*, tj. izvodiće se samo ako to eksplicitno zatražimo odgovarajućim opcijama.

Na primer, sledeći argumenti određuju da je potrebno da se izvrše svi testovi sa oznakom `Razlomak`, osim onih čije ime počinje sa `Kons`:

```
... ~Kons* [Razlomak]
```

Pisanje testova

Moduli sa testovima se pišu sasvim jednostavno. Dovoljno je da se uključe zaglavlje biblioteke i odgovarajuća zaglavlja jedinice koda koja se testira i da se napišu testovi. Na primer, jedan ispravan modul za testiranje bi mogao da izgleda ovako:

```
#include "catch2/catch.hpp"
```

```
TEST_CASE( "Provera sabiranja ", "[Celobrojni]" ) {
    CHECK( 1 + 2 = 3 );
}
```

Osnovna jedinica organizacije je *test* (engl. *test case*). Jedan test može da obuhvata veći broj *provera*, koje mogu dodatno da se organizuju po *sekcijama*. Testovi mogu da se grupišu upotrebom *oznaka* (tagova). Test se definiše u bloku koji počinje makroom:

```
TEST_CASE( naziv_testa [, oznake] )
```

Prvi parametar je naziv testa i uvek mora da se navede. Drugi parametar je opciona niska sa oznakama. Oznake se koriste za grupisanje testova, radi omogućavanja da se kasnije biraju grupe testova koje se izvršavaju. Svaka oznaka se navodi unutar uglastih zagrada, bez dodatnih separatora. U prethodnom primeru je testu „`Provera sabiranja`“ dodeljena oznaka `Celobrojni`.

U okviru jednog testa može da se navede veći broj *provera* (ili *pretpostavki*, engl. *assertion*). Bibilioteka *Catch2* podržava veći broj različitih provera ali se najčešće koriste dve najjednostavnije: `REQUIRE` i `CHECK`.

Provera `REQUIRE( uslov )` proverava da li je dati uslov ispunjen i u slučaju neuspeha prekida izvođenje testa u kome se nalazi. Provera `CHECK( uslov )` u osnovi radi isto, ali u slučaju neuspeha ne prekida izvođenje testa. Provera `CHECK` je praktičnija, zato što omogućava da uočimo više neispravnosti u istom testu. Međutim, ako mora da bude ispunjen neki uslov da bi mogle da se izvedu naredne provere, onda je neophodno da se koristi `REQUIRE` za proveravanje tog uslova. Na primer, ako želimo prvo da proverimo da li je pokazivač ispravan, pa da zatim proverimo da li objekat na koji pokazuje ima neka svojstva, onda pišemo provere poput:

```
REQUIRE( pRazlomak );
CHECK( pRazlomak->Imenilac() == 10 );
```

Pored ovih osnovnih provera postoje i mnoge druge, na primer:

- `REQUIRE_FALSE( uslov )`, `CHECK_FALSE( uslov )` – proveravaju da li je uslov netačan (slično kao kada se koristi negacija uslova u proverama `REQUIRE` i `CHECK`, ali negacija može da ometa automatsko razlaganje izraza, pa se preporučuje ovaj oblik);
- `REQUIRE_NOTHROW( izraz )`, `CHECK_NOTHROW( izraz )` – proveravaju da li izraz (ne) izbacuje izuzetak;
- `REQUIRE_THROWS( izraz )`, `CHECK_THROWS( izraz )` – proveravaju da li izraz (ne) izbacuje izuzetak;

- `REQUIRE_THROWS_AS( izraz, tip_izuzetka )`, `CHECK_THROWS_AS( izraz, tip_izuzetka )` – proveravaju da li izraz izbacuje izuzetak datog tipa;
- provere poklapanja (engl. *matcher*), pre svega za niske i kolekcije; na primer, može da se proveri da li se niska poklapa sa nekim šablonom
- i druge.

Jedan test može da sadrži više sekcija. Svaka sekcija predstavlja blok koji počinje makroom:

```
SECTION( naziv_sekcije [, opis_sekcije ] )
```

Programski kod koji se nalazi pre prve sekcije jednog testa predstavlja *inicijalizacioni kod*, koji se izvršava pre svake pojednačne sekcije (u engl. terminologiji testiranja se naziva *setup*). Slično, programski kod koji se nalazi posle poslednje sekcije predstavlja *deinicijalizacioni* kod, koji se izvršava posle svake pojedinačne sekcije (u engl. terminologiji testiranja se naziva *teardown*). Između sekcija ne bi trebalo da se nalazi nikakav programski kod.

Na primer, sledeći test sadrži dve sekcije, koje koriste istu početnu vrednost vektora `v`:

```
TEST_CASE( "Vektori", "[Primeri]" ) {
    std::vector<int> v { 0, 1, 2 };
    SECTION( "Promena vrednosti elementa" ) {
        REQUIRE( v.size() == 3 );
        CHECK( v[1] == 1 );
        v[1] = 101;
        CHECK( v[1] == 101 );
        v.pop_back();
        CHECK( v.size() == 2 );
    }
    SECTION( "Konstruktor" ) {
        REQUIRE( v.size() == 3 );
        CHECK( v[0] == 0 );
        CHECK( v[1] == 1 );
        CHECK( v[2] == 2 );
    }
}
```

Semantika sekcija je definisana tako da svaka sekcija predstavlja jedan alternativni tok izvršavanja testa. Zbog toga sekcije mogu da se nalaze i u okviru drugih sekcija, pri čemu važe ista pravila inicijalizacije i deinicijalizacije.

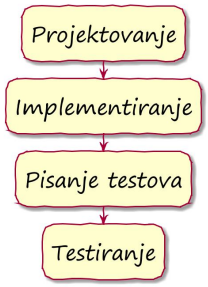
Napredne mogućnosti

Biblioteka *Catch2* ima još mnogo toga što ovde nećemo detaljnije opisivati. Neke od naprednijih mogućnosti biblioteke su:

- iako sekcije omogućavaju izvođenje relativno složenih testova, podržane su i alternativne organizacije testova kao što su *testiranje vođeno ponašanjem* (engl. *Behavior Driven Development – BDD*), ili pravljenje posebnih celina (kompleta) testova (engl. *test fixtures*);
- testovi mogu da se pišu i u obliku šablona, tako da se jednom napisan test izvršava za različite konkretne tipove;
- postoje tzv. *generatori podataka* (engl. *data generators*), koji mogu da budu od pomoći pri pravljenju kolekcija podataka na kojima se izvodi testiranje, ali i da omoguće da se iste provere izvode na većem broju različitih primera;
- možemo da napišemo i svoju funkciju `main`, a da pri tome iskoristimo neke elemente izvođenja testova koje nudi biblioteka;
- možemo da koristimo više različitih generatora izveštaja (engl. *reporters*)
- i drugo.

9.4 Razvoj vođen testovima

Klasičan redosled aktivnosti pri razvoju softvera podrazumeva da se softver najpre projektuje, pa zatim implementira i na kraju testira. Takav redosled može da se primeni na svim nivoima razvoja, uključujući i projekat kao celinu, ali i svaku pojedinačnu razvijanu jedinicu koda.



Slika 33 – Klasičan redosled aktivnosti pri razvoju softvera

Da bi testiranje moglo da se sprovede, neophodno je da se najpre napravi odgovarajući skup test primera. Da bi skup test primera bio dovoljno reprezentativan, potrebno je da test primeri budu usklađeni sa konkretnom imple-

mentacijom. Ali, da bi to bilo moguće, zaključujemo da test primere mora da pravi neko ko je dobro upoznat sa predmetom testiranja i potencijalnim slabim tačkama koje se testovima posebno ciljaju. Zato testove obično pravi neko ko je učestvovao u projektovanju ili implementiranju odgovarajućeg dela koda, ili specijalizovani stručnjak za testiranje, koji ima punu saradnju ostalih članova tima.

Kao što vidimo, testiranje je složen postupak koji obuhvata nekoliko različitih poslova. U prethodnom pasusu smo naveli neke od poslova koji čine testiranje. Potpuniji spisak poslova obuhvata:

- određivanje ciljeva testiranja;
- prepoznavanje predmeta testiranja;
- analiziranje predmeta testiranja;
- određivanje nivoa detaljnosti testiranja;
- oblikovanje testova;
- ustanovljavanje očekivanih rezultata;
- implementiranje testova;
- izvršavanje testova;
- analiziranje neuspešnih testova
- i druge poslove.

Pravljenje testova na samom kraju razvoja nije nimalo jednostavan posao. Da li bi možda bilo jednostavnije i bolje da se testovi prave u nekom drugom trenutku?

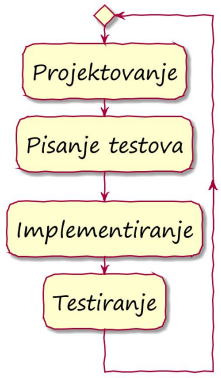
Pogledajmo sada isečak iz jednog primera testa:

```
...
CHECK( Razlomak(1,2).Brojilac() == 1 );
CHECK( Razlomak(1,2).Imenilac() == 2 );
CHECK( Razlomak(2,1).Brojilac() == 2 );
CHECK( Razlomak(2,1).Imenilac() == 1 );
...
```

Čak i bez poznavanja deklaracije ili definicije klase `Razlomak` i makroa biblioteke *Catch2*, iz navedenog zapisa testa možemo da uočimo da konstruktor klase `Razlomak` ima za argumente vrednosti brojioca i imenioca, kao i da naši testovi proveravaju da li su međusobno usklađeni konstruktor i metodi koji izdvajaju vrednosti brojioca i imenioca. Pri izvođenju tog zaključka smo napisane testove upotrebili kao vid dokumentacije. Zaista, testovi mogu da veoma dobro ilustruju ponašanje testiranih jedinica koda. Ako su testovi dobro napravljeni, onda oni mogu da opišu način upotrebe testiranih jedinica jednako dobro i precizno kao tehnička specifikacija.

Ako imamo u vidu da je uvek poželjno (pa i neophodno) da se pre implementacije funkcija i metoda napravi njihova specifikacija, onda bismo mogli da na samom početku, pre implementiranja programskog koda, najpre napišemo testove kao vid specifikacije, pa da tek onda pišemo implementaciju, a da na kraju izvedemo testiranje pomoću tako napisanih testova. Takva izmena redosleda aktivnosti može da ima više pozitivnih posledica:

- napisani testovi mogu da predstavljaju vid specifikacije jedinica koda koje ćemo tek da napišemo; štaviše, ako dovoljno jasno napišemo testove, onda obično ne moramo da žurimo da napišemo i odgovarajuću tekstualnu specifikaciju;
- nakon što napišemo testove kao vid specifikacije, program ili neće moći da se prevede ili testovi neće proći, zato što odgovarajuća jedinica koda nije uopšte napisana ili joj ponašanje nije implementirano u skladu sa novom „specifikacijom“, pa zato
- u razvoju vođenom testovima *cilj pisanja programskog koda* postaje upravo *zadovoljavanje napisanih testova*; svaka pojedinačna naredba programskog koda se piše da bi se zadovoljili testovi.



Slika 34 – Redosled aktivnosti pri razvoju vođenom testovima

Takvom izmenom redosleda aktivnosti smo na praktičan način predstavili osnovnu ideju *razvoja vođenog testovima*. Osnovni principi razvoja vođenog testovima su da (1) testovi prethode kodu i da je pri tome (2) neophodan visok nivo sistematičnosti.

Za razliku od klasičnog pristupa razvoju softvera, koji promoviše testiranje softvera nakon razvoja, u slučaju razvoja vođenog testovima zahteva se da *testovi prethode kodu*, tj. da pre pisanja koda moramo:

- da isplaniramo kako je potrebno da se kod ponaša;
- da te planove pretočimo u vid specifikacije – u obliku testova;
- da testovima obuhvatimo sve uobičajene i granične slučajeve i
- da uvek kritički razmotrimo da li je potrebno da se testovima obuhvate i još neki specijalni slučajevi.

Na taj način se već pre započinjanja pisanja ili menjanja nekog dela koda dobija odgovarajuća kolekcija testova. Napisani testovi predstavljaju ciljne kriterijume kvaliteta koda. Posle toga, cilj pisanja produkcionog koda nije ništa drugo do zadovoljavanje novih postavljenih kriterijuma, a bez narušavanja već ranije dostignutih kriterijuma kvaliteta softvera.

Posle pisanja programskog koda, testovi se izvode (izvršavaju). Ako je testiranje uspelo (*testovi su prošli*), onda to znači da je programski kod zadovoljio postavljene kriterijume kvaliteta. Ako testiranje nije uspelo, onda programski kod nije dovoljno dobar i mora da se popravlja. Popravljanje koda se nastavlja sve dok se ne omogući uspešno testiranje.

Razvoj vođen testovima promoviše visok nivo sistematičnosti, do te mere da se nijedna funkcija programa ne razvija sve dok ne postoji test koji ne uspeva zbog njenog odsustva. Štaviše, nijedna linija koda se ne dodaje niti menja sve dok ne postoji test koji zbog nje ne uspeva.

Razvoj vođen testovima (engl. *Test Driven Development*) se odnosi primarno na testove jedinica koda, ali može da se primenjuje i šire, na primer na korisničke celine ili slučajeve upotrebe, pa čak i na određene testove integracije ili testove sistema.

Redosled koraka

1. Analiza i projektovanje

Svaka iteracija razvoja započinje analizom i projektovanjem (tj. planiranjem) novih funkcija, metoda ili klasa, ili potrebnih izmena postojećih funkcija, metoda ili klasa.

Imajući u vidu da će u narednom koraku da se naprave testovi kao vid formalne specifikacije, rezultat projektovanja i planiranja može da bude i neformalna specifikacija. Alternativno, ovaj korak može da se integriše sa narednim.

2. Pisanje testova kao formalne specifikacije zadatka

Kada znamo šta je potrebno da se razvija, onda možemo to i formalno da zapišemo – u obliku testova jedinica koda koje ćemo razvijati ili menjati. Pisanjem testova definišemo kako će izgledati interfejs novih jedinica koda i kako će one da se ponašaju.

Testovi koje napišemo u ovoj fazi vrlo često ne mogu ni da se prevedu, zato što još ne postoje odgovarajući metodi ili klase koji se u testovima upotrebljavaju.

3. Pisanje kostura koda

Nakon što su napisani testovi, piše se kostur programskog koda, koji omogućava da se testovi prevedu. To odgovara pisanju interfejsa odgovarajućih jedinica koda. Dodaju se nove funkcije, metodi i klase ili se menjaju postojeći interfejsi, radi usklađivanja sa novom specifikacijom. Ako je potrebno da novi metodi vraćaju neki rezultat, obično se u ovoj fazi privremeno implementiraju tako da vraćaju neku konstantnu vrednost.

Ovaj korak se završava proveravanjem da li su napravljeni svi neophodni metodi i klase. To se radi prevođenjem programa za testiranje – ako uspe da se prevede, onda je ovaj deo posla završen. Naravno, još uvek ne očekujemo da testovi uspešno prolaze, zato što još nije implementirano odgovarajuće ponašanje.

4. Pisanje programskog koda koji omogućava da testovi prođu

Posle pisanja kostura koda, potrebno je da se novi metodi, jedan po jedan, implementiraju tako da zadovolje „specifikaciju", tj. testove. Tokom implementiranja mogu (i preporučuje se) da se dodaju novi testovi, koji odgovaraju različitim posebnim slučajevima. To je važno zato što se pri implementiranju koda najbolje sagledavaju potencijalne osetljive tačke. Dok su u prethodnim koracima mogli da se predvide neki granični slučajevi samo na osnovu interfejsa, sada mogu da se uoče i neki novi granični slučajevi, koji su posledica konkretne implementacije, pa zato i oni moraju da se testiraju.

Male iteracije

Preporučuje se da opisane iteracije razvoja budu što manje. Nije dobro da se prvo napiše veliki broj testova pa da se zatim implementira mnogo koda. Takav pristup ima više potencijalnih slabosti. Pre svega, testovi će moći da se izvrše tek kada budu napisani svi odgovarajući delovi koda, a ako to zahteva mnogo posla, onda je lako moguće da se usput napravi veći broj grešaka. Što je više grešaka koje moraju da se rešavaju, to je njihovo pronalaženje i otklanjanje složenije i zahteva više vremena.

Zato je mnogo bolje da se piše u malim iteracijama. Na početku iteracije se napiše mali broj novih testova, koji se odnose na samo jednu jedinicu koda. Zatim se piše ili menja ta jedinica koda. Kada testiranje pokaže da je jedinica koda dostigla traženi kvalitet, iteracija se završava i počinje nova.

Iteracije pisanja testova i koda bi trebalo da se se veoma brzo smenjuju, često na nekoliko minuta. Na taj način testovi i kod zajedno evoluiraju, tako da su testovi tek malo ispred koda.

Sistematičnost

Za uspešno sprovođenje razvoja vođenog testovima je od presudnog značaja da se testovi pišu sistematično:

- svaka karakteristika softvera mora da se pokrije testovima;
- svi granični slučajevi moraju da budu obuhvaćeni testovima;
- svaka linija koda mora da bude testirana;
- svaka naredba izbora i svaka petlja moraju da budu obuhvaćeni testovima; svaka grana mora da bude testirana;
- svaka operacija, funkcija i metod moraju da se testiraju;
- svaka polimorfna upotreba objekata mora da se testira na različitim klasama
- i drugo.

Kada se naknadno uoče bagovi u gotovom proizvodu, bilo pri holističkom testiranju ili pri upotrebi, to najčešće može da se objasni kao posledica nedovoljno sistematičnih testova – da su testovi bili dovoljno dobro definisani, oni bi na vreme ukazali na problem, umesto da se on kasnije ispoljava kao bag.

Testiranje jedinica koda se odnosi prvenstveno na javne metode klasa. Ne postoji opšta saglasnost oko toga da li je dobro da se pišu testovi jedinica koda i za privatne metode. Za razliku od javnog interfejsa koji bi trebalo da bude relativno stabilan, privatni elementi mogu da trpe određene značajnije promene ponašanja tokom razvoja, na primer zbog optimizacije, pa onda to povlači i češće održavanje testova. Drugi problem je tehničkog karaktera, zato što elementi biblioteka za testiranje obično nemaju neposredan pristup privatnim metodima naših klasa. Jedan način da se ovaj problem prevaziđe je da se obezbede prijateljske klase, koje mogu da pristupe privatnim elementima i koriste se samo radi testiranja, ali to predstavlja vid narušavanja enkapsulacije.

Neki privatni metodi su jednostavni i od propuštanja njihovog testiranja nas verovatno neće boleti glava. U nekim drugim slučajevima se privatni metodi koriste prilično neposredno u javnim metodima tako da se dobrim testiranjem javnih metoda praktično u potpunosti (iako implicitno) testiraju i privatni metodi, pa i tada nemamo problem. Ipak, u nekim slučajevima je privatni metod tako definisan da ne možemo da testiramo sve različite slučajeve njegove upotrebe samo testiranjem javnih metoda koji ga koriste. Tada moramo da proverimo (1) da li su možda ti slučajevi zabranjeni (i ako jesu, onda možemo da popravimo robusnost uvođenjem pretpostavki, a ne testiranjem) i (2) da li bi možda takav metod trebalo da bude javan (zato što ima veće mogućnosti primene nego postojeći javni interfejs)? Tek ako na

oba pitanja dobijemo negativan odgovor, onda bi trebalo da razmislimo o testiranju (i načinu testiranja) privatnog metoda.

Smer razvoja

Softver se obično razvija u jednom od dva smera – ili od vrha prema dnu, ili od dna prema vrhu. Ako se softver razvija od vrha prema dnu, onda se najpre razvijaju glavni algoritmi i koncepti, dok se implementacija detalja ostavlja za kasnije. Da bi programi mogli da se prevode i testiraju, umesto svih onih delova koji još nisu razvijeni prave se tzv. *umeci* (engl. *stubs*). Umetak je privremeni deo koda koji definiše interfejs ili druge metode, ali ih implementira samo *praznim* kodom. U slučaju potprograma koji ne izračunavaju rezultat, umetak najčešće ne radi doslovno ništa. U slučaju funkcija ili metoda koji moraju da vrate neki rezultat, umetak obično vraća neku konstantnu vrednost. Jedina namena umetaka je da omoguće da se klase, komponente i program prevedu. Naravno, tako prevedene klase, komponente i programi neće raditi ispravno, ali će bar biti moguće da se testiraju pojedini delovi implementacija, kao i da se proveri da li na visokom nivou algoritimi rade kako je predviđeno.

Ako se razvoj softvera odvija u suprotnom smeru, od dna prema vrhu, onda se za testiranje upotrebljavaju tzv. *izvođači* (engl. *drivers*). Izvođači su privremeni delovi koda koji povezuju manje funkcionalne celine isključivo radi testiranja. U odsustvu većih funkcionalnih celina produkcionog koda, čiji razvoj tek sledi, pravljenje izvođača je vro pristupačan način da se neki delovi povežu u celinu i testiraju.

9.4.1 Primer

Ilustrovaćemo koncepte razvoja vođenog testovima predstavljanjem nekoliko koraka razvoja klase `Razlomak`. Za implementiranje testova ćemo da koristimo biblioteku *Catch2*, verziju 2. Pretpostavićemo da je zaglavlje biblioteke u datoteci: `catch2/catch.hpp`.

Započećemo od prazne klase, koju ćemo implementirati u zaglavlju `razlomak.h`[45]:

```
class Razlomak
{};
```

[45] Ako bi klasa `Razlomak` postala složenija, onda bi trebalo da se deo implementacije prebaci iz zaglavlja u datoteku `razlomak.cpp` i da se doda `razlomak.cpp` u naredbu za prevođenje. U našem primeru to nije neophodno.

Testove naše klase ćemo da pišemo u datoteci `razlomak.test.cpp`:

```
#include "catch2/catch.hpp"
#include "razlomak.h"
```

Glavni program za testiranje implementiramo u datoteci `main.test.cpp`:

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"
```

Prevođenje i testiranje možemo da izvodimo korišćenjem prevodioca `g++` na `Linuxu`, pomoću sledećeg skripta zapisanog u datoteci `tst.sh`:

```
#!/bin/bash
g++ main.test.cpp razlomak.test.cpp -o test
if [ $? -eq 0 ]; then
    ./test
fi
```

ili korišćenjem paketa *Visual Studio* na operativnom sistemu *Windows*, pomoću skripta zapisanog u datoteci `tst.cmd`:

```
cl /EHsc main.test.cpp razlomak.test.cpp /Fetest.exe
@if not errorlevel 1 test.exe
```

Korak 1 – Konstruktor

Počinjemo od pravljenja konstruktora, koji kao argumente ima brojilac i imenilac. Najpre pišemo odgovarajuće testove:

razlomak.test.cpp

```
#include "catch2/catch.hpp"
#include "razlomak.h"

TEST_CASE( "Konstruktori klase Razlomak", "[Razlomak]" ) {
    Razlomak r(2,5);
    CHECK( r.Imenilac() == 5 );
    CHECK( r.Brojilac() == 2 );
}
```

Zatim pišemo konstruktor i dodajemo podatke `Brojilac_` i `Imenilac_`. Da bismo mogli da proverimo uspešnost konstruktora, potrebno je da napravimo i pristupne metode, kojima dohvatamo vrednosti brojioca i imenioca.

razlomak.h

```
class Razlomak
{
public:
    Razlomak( int b, int i )
        : Imenilac_(i),
          Brojilac_(b)
    {}

    int Imenilac() const
        { return Imenilac_; }

    int Brojilac() const
        { return Brojilac_; }

private:
    int Imenilac_;
    int Brojilac_;
};
```

Prevodimo i pokrećemo testove:

```
===============================================================================
All tests passed (2 assertions in 1 test case)
```

Korak 2 – Pozitivan imenilac

Dodajemo pretpostavku da imenilac nikada ne sme da bude ni nula ni negativan. Ako je negativan, promenićemo znak i imenioca i brojioca, a ako je nula, onda ćemo da izbacimo izuzetak.

Prvo dodajemo odgovarajuće testove. Dodajemo sekcije da bismo razdvojili različite aspekte testiranja konstruktora.

razlomak.test.cpp

```
#include "catch2/catch.hpp"
#include "razlomak.h"

TEST_CASE( "Konstruktori", "[Razlomak]" ) {
    SECTION( "Jednostavan konstruktor"){
        Razlomak r(2,5);
        CHECK( r.Imenilac() == 5 );
        CHECK( r.Brojilac() == 2 );
    }
    SECTION( "Konstruktor sa imeniocem nula"){
        CHECK_THROWS_AS( Razlomak( 3, 0 ), std::domain_error );
    }
```

```
    SECTION( "Konstruktor sa negativnim imeniocem"){
        Razlomak r(2,-5);
        CHECK( r.Imenilac() == 5 );
        CHECK( r.Brojilac() == -2 );
    }
}
```

Prevodimo testove i pokrećemo ih:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
test.exe is a Catch v2.0.1 host application.
Run with -? for options

-------------------------------------------------------------------------
Konstruktori
  Konstruktor sa imeniocem nula
-------------------------------------------------------------------------
razlomak.test.cpp(10)
.....................................................................

razlomak.test.cpp(11): FAILED:
  CHECK_THROWS_AS( Razlomak( 3, 0 ), std::domain_error )
because no exception was thrown where one was expected:

-------------------------------------------------------------------------
Konstruktori
  Konstruktor sa negativnim imeniocem
-------------------------------------------------------------------------
razlomak.test.cpp(13)
.....................................................................

razlomak.test.cpp(15): FAILED:
  CHECK( r.Imenilac() == 5 )
with expansion:
  -5 == 5

razlomak.test.cpp(16): FAILED:
  CHECK( r.Brojilac() == -2 )
with expansion:
  2 == -2

=========================================================================
test cases: 1 | 1 failed
assertions: 5 | 2 passed | 3 failed
```

Zatim popravljamo konstruktor tako da pretpostavka o pozitivnosti imenioca bude uvek zadovoljena.

razlomak.h

```
class Razlomak {
...
    Razlomak( int b, int i )
        : Imenilac_(i),
          Brojilac_(b)
    {
        if( !Imenilac_ )
            throw std::domain_error( "Imenilac je nula!" );
        else if( Imenilac_ < 0 ){
            Imenilac_ = - Imenilac_;
            Brojilac_ = - Brojilac_;
        }
    }
...
};
```

Prevodimo testove i pokrećemo ih:

```
===============================================================================
All tests passed (4 assertions in 1 test case)
```

Korak 3 – Poređenje jednakosti

Dodajemo operator ==. Najpre pišemo testove, pa onda i operator. Zatim prevodimo i testiramo.

razlomak.test.cpp

```
...
TEST_CASE( "Poredjenje", "[Razlomak]" ) {
    CHECK( Razlomak(2,5) == Razlomak(2,5) );
    CHECK( Razlomak(2,5) == Razlomak(-2,-5) );
    CHECK( Razlomak(2,5) == Razlomak(4,10) );
}
```

razlomak.h

```
class Razlomak {
...
    bool operator==( const Razlomak& r ) const
    {
        return Imenilac() == r.Imenilac()
            && Brojilac() == r.Brojilac();
    }
...
};
```

Primećujemo da poslednji test ne prolazi. Dva razlomka koja poredimo su suštinski ista, ali se njihovi brojioci i imenioci razlikuju. Da bi prošao i taj test, moramo da popravimo poređenje[46].

```
class Razlomak {
...
   bool operator==( const Razlomak& r ) const
   {
      return Imenilac() * r.Brojilac() == r.Imenilac() * Brojilac();
   }
...
};
```

Korak 4 – Pisanje i čitanje

Pri implementaciji pisanja i čitanja trebalo bi da vodimo računa da ove dve operacije budu usaglašene, tako da zapis koji pravimo pri ispisivanju može kasnije da se pročita. Počinjemo od pisanja. Pri testiranju koristimo `std::ostringstream`.

razlomak.test.cpp

```
...
#include <sstream>
...
TEST_CASE( "Pisanje i citanje", "[Razlomak]" ) {
    std::ostringstream str;
    str << Razlomak(2,5);
    CHECK( str.str() == "2/5" );

    ostr.str("");
    ostr.clear();
    ostr << Razlomak(2,-5);
    CHECK( ostr.str() == "-2/5" );
}
```

razlomak.h

```
class Razlomak { ... };

inline std::ostream&
operator<<( std::ostream& ostr, const Razlomak& r )
{
    ostr << r.Brojilac() << '/' << r.Imenilac();
    return ostr;
}
```

Zatim prelazimo na čitanje.

[46] Alternativno rešenje je da pri konstrukciji uvek uprostimo razlomak.

razlomak.test.cpp

```
...
#include <sstream>
...
TEST CASE( "Pisanje i citanje", "[Razlomak]" ) {
    ...
    std::istringstream istr( ostr.str() );
    Razlomak r(3,4);
    istr >> r;
    CHECK( r == Razlomak(-2,5) );
}
```

razlomak.h

```
class Razlomak { ... };

inline std::istream& operator>>( std::istream& istr, Razlomak& r )
{
    int i,b;
    char c;
    istr >> b >> c >> i;
    if( istr ){
        if( c == '/' )
            r = Razlomak( b, i );
        else
            istr.setstate( std::ios::failbit );
    }
    return istr;
}
```

Sada znamo kako je implementirano čitanje, pa dodajemo još odgovarajućih testova i zatim prevodimo i testiramo.

razlomak.test.cpp

```
...
TEST_CASE( "Pisanje i citanje", "[Razlomak]" ) {
    ...
    istr.str( "7;8" );
    istr.clear();
    istr >> r;
    CHECK( !istr );
    CHECK( r == Razlomak(-2,5) );

    istr.str( "7/a8" );
    istr.clear();
    istr >> r;
    CHECK( !istr );
    CHECK( r == Razlomak(-2,5) );

    istr.str( "7/8" );
```

```
    istr.clear();
    istr >> r;
    CHECK( istr );
    CHECK( r == Razlomak(7,8) );
}
```

Korak 5 – Poređenje „manji od“

Pišemo testove za operator poređenja „<“. Zatim pišemo operator, prevodimo i testiramo.

razlomak.test.cpp

```
...
TEST_CASE( "Poredjenje", "[Razlomak]" ) {
    ...
    CHECK( Razlomak(2,5) < Razlomak(3,5) );
    CHECK( Razlomak(2,5) < Razlomak(2,4) );
    CHECK( Razlomak(-3,5) < Razlomak(-2,5) );
    CHECK( Razlomak(-2,5) < Razlomak(2,5) );
}
```

razlomak.h

```
class Razlomak {
...
    bool operator<( const Razlomak& r ) const
    {
        return Brojilac()*r.Imenilac() < r.Brojilac()*Imenilac();
    }
...
};
```

Korak 6 – Skraćivanje razlomka

Dodajemo metod `Skrati`, koji skraćuje razlomak. Prvo pišemo testove, pa implementiramo metod.

razlomak.test.cpp

```
...
TEST_CASE( "Skracivanje", "[Razlomak]" ) {
    CHECK( Razlomak(2,5).Skrati().Imenilac() == 5 );
    CHECK( Razlomak(2,5).Skrati().Brojilac() == 2 );
    CHECK( Razlomak(25,15).Skrati().Imenilac() == 3 );
    CHECK( Razlomak(25,15).Skrati().Brojilac() == 5 );
    CHECK( Razlomak(15,25).Skrati().Imenilac() == 5 );
    CHECK( Razlomak(15,25).Skrati().Brojilac() == 3 );
    CHECK( Razlomak(15,-25).Skrati().Imenilac() == 5 );
    CHECK( Razlomak(15,-25).Skrati().Brojilac() == -3 );
}
```

razlomak.h

```
class Razlomak {
...
    Razlomak& Skrati()
    {
        int n = nzd( abs(Brojilac_), Imenilac_ );
        Imenilac_ /= n;
        Brojilac_ /= n;
        return *this;
    }

private:
    static unsigned nzd( unsigned n1, unsigned n2 )
    {
        if( n1 < n2 )
            swap( n1, n2 );
        else if( !n1 )
            return 1;
        while( n2 ){
            unsigned r = n1 % n2;
            n1 = n2;
            n2 = r;
        }
        return n1;
    }
...
};
```

Korak 7 – Sabiranje i oduzimanje

Kao završni deo ovog primera, dodajemo operatore sabiranja i oduzimanja. I u ovom slučaju, prvo pišemo testove, pa implementiramo operatore, pa onda prevodimo i testiramo.

razlomak.test.cpp

```
...
TEST_CASE( "Sabiranje i oduzimanje", "[Razlomak]" ) {
    CHECK( Razlomak(2,5) + Razlomak(-1,5) == Razlomak(1,5) );
    CHECK( Razlomak(2,5) - Razlomak(-1,5) == Razlomak(3,5) );
    CHECK( Razlomak(2,5) + Razlomak(1,2) == Razlomak(9,10) );
    CHECK( Razlomak(2,5) - Razlomak(1,2) == Razlomak(-1,10) );

    CHECK( Razlomak(3,10) + Razlomak(4,15) == Razlomak(17,30) );
    CHECK( Razlomak(3,10) - Razlomak(4,15) == Razlomak(1,30) );
    CHECK( Razlomak(3,10) + Razlomak(-4,15) == Razlomak(1,30) );
    CHECK( Razlomak(3,10) - Razlomak(-4,15) == Razlomak(17,30) );
}
```

razlomak.h

```
class Razlomak {
...
    Razlomak operator+( const Razlomak& r ) const
    {
        return Razlomak(
                    Brojilac() * r.Imenilac()
                        + Imenilac() * r.Brojilac(),
                    Imenilac() * r.Imenilac()
                ).Skrati();
    }

    Razlomak operator-( const Razlomak& r ) const
    {
        return Razlomak(
                    Brojilac() * r.Imenilac()
                        - Imenilac() * r.Brojilac(),
                    Imenilac() * r.Imenilac()
                ).Skrati();
    }
...
};
```

Rezultat testiranja:

```
===============================================================================
All tests passed (36 assertions in 5 test cases)
```

U prethodnih nekoliko koraka smo predstavili kako bi trebalo da teče razvoj vođen testovima. Primer je sasvim jednostavan, ali smo imali priliku da vidimo i specijalne i granične slučajeve, sa imeniocem koji je manji od nule ili jednak nuli. Zadržali smo se na elementarnim mogućnostima biblioteke *Catch2*, zato da bi pažnja bila usmerena na postupak, a ne na alat. Primetimo da smo testirali sve javne elemente klase `Razlomak`, ali ne i privatni statički metod `nzd`.

9.5 Uloga testova jedinica koda

Osnovna uloga testova jedinica koda je proveravanje da li se napisani delovi programskog koda ponašaju u skladu sa odgovarajućom tehničkom specifikacijom. Oni predstavljaju važno sredstvo za obezbeđivanja kvaliteta softvera. Međutim, testovi jedinica koda mogu da imaju i neke dodatne uloge u procesu razvoja softvera.

Testovi predstavljaju vid parcijalne verifikacije softvera. Kolekcija testova omogućava programeru da proverava da li jedinica koda radi ispravno u nekim unapred definisanim okolnostima. U opštem slučaju testovi ne mogu da obuhvate

sve moguće okolnosti upotrebe, pa zato ne mogu da predstavljaju potpunu verifikaciju softvera. Ipak, i parcijalna verifikacija je veoma važna za obezbeđivanje željenog kvaliteta softvera.

> *Testiranje programa može da bude veoma efikasno sredstvo*
> *da se pokaže postojanje bagova,*
> *ali je beznadežno neadekvatno za pokazivanje da ih nema*
>
> *Edsher Daikstra*

Testovi jedinica koda bi trebalo da proveravaju ispravnost svake pojedinačne jedinice koda: funkcije, metoda, klase i drugih vrsta jedinica koda. Mnogi zameraju da se pisanjem testova usporava razvoj produkcionog koda, ali to usporavanje ima i pozitivan uticaj na razvoj – sprečava se srljanje. Testovi jedinica koda, posebno ako se pišu na način koji propisuje razvoj vođen testovima, omogućavaju programeru da pre pisanja svake pojedinačne jedinice koda zastane i zapita se „Šta je potrebno da napravim?“ i „Kako je potrebno da se ponaša?“. Na taj način se sprečava lako-misleno zaletanje u pisanje niza možda nepotrebnih operacija.

Razvoj vođen testovima omogućava da se blagovremeno uoče eventualne nepotpunosti ili neispravnosti implementacije, ali i dizajna pa i konceptualnog projekta. Tako veliki doprinos testova potiče otud što dok piše testove programer mora da razmišlja i kao korisnik jedinice koda koju testira, a ne samo kao njen implementator. Takva promena ugla posmatranja može značajno da pomogne pri uočavanju eventualnih slabosti projektovanog interfejsa i drugih grešaka na nivou koncepta ili projekta. Pri pisanju testova u prvom planu su interfejs testirane jedinice koda i apstrakcija njenog ponašanja. Sa druge strane, pri implementiranju se može nehotice izgubiti iz vida značaj interfejsa, pa i apstrakcije ponašanja, zato što je programer prinuđen da se bavi sasvim konkretnim problemima i pojedinostima u kodu. Na ovaj način testovi doprinose pisanju *lako upotrebljivog* koda.

Da bi jedinice koda mogle da se dobro testiraju, one moraju da budu jasno uobličene i uokvirene. Veoma je teško napraviti dobar skup testova za jedinicu koda u kojoj postoji slaba kohezija elemenata, pa njeno ponašanje nije jasno i dobro uobličeno. Blagovremeno testiranje i posebno razvoj vođen testovima, teraju programera da sve jedinice koda što bolje oblikuje i međusobno razgraniči, tako da svaka ima jasno određeno ponašanje i smisao postojanja. Na taj način se vrši dodatni pritisak na programera da pravi bolje dizajniran kod, jer samo *dobro dizajniran kod* može da bude i *lako proverljiv*.

Testovi jedinica koda su posebno važni za primenu refaktorisanja. Refaktorisanje je postupak unapređivanja dizajna koda ali tako da se ne menja njegovo ponašanje (vidi poglavlje *10 - Refaktorisanje*). Ključni problem je: kako da budemo sigurni da menjanjem dizajna nismo promenili i ponašanje? Ako postoje dobro implementirani

testovi, onda je odgovor jednostavan – posle svake promene koda izvedemo testiranje i tako proverimo da li smo slučajno izmenili ponašanje.

Ako je u nekoj kasnijoj fazi razvoja potrebno da se menja ranije napisan kod, onda jedan od otežavajućih faktora može da bude otežana proverljivost – kako napraviti neophodne promene ponašanja, a da se ne promene i još neki aspekti ponašanja? Kao i u slučaju refaktorisanja, i za ovaj problem rešenje su nam testovi jedinica koda. Oni omogućavaju da se lako uoče eventualne greške u kodu nastale prilikom naknadnih izmena.

Jedna od najvažnijih „sporednih" uloga testova je da oni predstavljaju vid dokumentacije. Testovi jedinica koda uvek imaju oblik primera ispravne upotrebe interfejsa, a ako su dobro sistematizovani, onda mogu da predstavljaju i oblik formalne specifikacije testirane jedinice koda. Oni opisuju uslove funkcionisanja i različite načine upotrebe interfejsa jedinice koda. Posebno, testovi obično dobro ilustruju različite granične slučajeve. Jedna od najznačajnijih karakteristika ovog vida dokumentacije je *stalna ažurnost*. Jedan od osnovnih problema sa skoro svim drugim vidovima dokumentacije je neažurnost. Ali testovi jedinica koda ne mogu da budu neažurni, jer inače ne bi uspešno prolazili. Ako se kolekcija testova redovno održava i raste uporedo sa programskim kodom, onda ona pored vida neformalne verifikacije predstavlja i *najkompletniji* i *najažurniji* vid dokumentacije.

Testovi jedinica koda predstavljaju veliku pomoć i u procesu debagovanja. Svaki put kada se naknadno uoči neka neispravnost, koja nije prepoznata postojećim testovima, to može da bude signal da postojeći testovi nisu dovoljno dobri. Onda se obično pravi novi test, ili više testova, koji ne prolaze zbog postojanja baga. Slično kao u slučaju razvoja vođenog testovima, na ovaj način i debagovanje može da se svede na menjanje (tj. na popravljanje) koda radi zadovoljavanja testova. Štaviše, za razliku od privremenih provera, koje se često koriste pri debagovanju, testovi jedinica koda mogu da ostanu kao trajne provere i potvrde ostvarene ispravnosti.

9.6 Umesto zaključka

Danas se smatra da je programski kod *zastareo*, ako ne obuhvata odgovarajuće testove jedinica koda. Danas testovi jedinica koda nisu samo osnovno sredstvo za proveru ispravnosti napisanog programskog koda, već i veoma važan vid dokumentacije. Ako nemamo odgovarajuće testove jedinica koda, veliko je pitanje da li zaista znamo kako se neki programski kod ponaša i u kojoj meri odgovara onome što piše u pratećoj tekstualnoj dokumentaciji, koja nam dolazi uz taj programski kod. Da li je implementirano sve što je bilo predviđeno? Da li je možda program naknadno izmenjen tako da više nije u skladu sa dokumentacijom?

Tehnike za implementiranje i izvršavanje testova su postale neizostavni deo ozbiljnih razvojnih alata, a sami testovi obavezno gradivo na iole naprednijim programerskim obukama. Jedna od knjiga koje su ostvarile veliki uticaj u tom smeru

je knjiga Kenta Beka iz 2002. godine [*Back 2002*]. Dobar pregled različitih biblioteka za testiranje jedinica koda za programski jezik *C++* napisao je Noel Lopis [*Llopis 2010*]. Iako je to relativno star pregled (iz 2010. godine), iz njega se mogu videti osnovni pristupi razvoju biblioteka za testiranje.

10 - Refaktorisanje

Kod je kvarljiv!
Robert Martin

10.1 Pojam refaktorisanja

Refaktorisanje programskog koda je preduzimanje niza uzastopnih malih transformacija koda, kojima se unapređuje njegova struktura, a bez spolja vidljive promene ponašanja.

Navedena definicija ukazuje na tri suštinske karakteristike refaktorisanja:

- unapređivanje strukture koda;
- zadržavanje istog ponašanja i
- ostvarivanje putem niza malih transformacija.

Osnovni cilj refaktorisanja je unapređivanje strukture koda, tj. dizajna softvera. Nakon refaktorisanja, struktura koda bi trebalo da zadovoljava neke uspostavljene kriterijume kvaliteta, koji nisu bili ispunjeni pre početka refaktorisanja. Najvažnija posledica primene refaktorisanja je da se tako izmenjen programski kod mnogo lakše održava i unapređuje.

Preduslovi za refaktorisanje

Najvažnija pretpostavka za razmišljanje o refaktorisanju je *ispravnost* postojećeg programskog koda. U ovom kontekstu ćemo smatrati da je programski kod ispravan ako se program ponaša na odgovarajući način u svim poznatim okolnostima i ako se odgovarajući testovi jedinica koda uspešno izračunavaju. Jedna od najvažnijih odlika neispravnog koda je da mi, zapravo, *ne znamo* kako se on ponaša. Refaktorisanjem

neispravnog koda ulazimo u potencijalno veoma ozbiljne probleme – cilj je da izmenimo programski kod tako da se ne promeni njegovo ponašanje, ali mi ne možemo da proverimo da li smo u tome uspeli, ako ne znamo kako se program ponaša. Refaktorisanje neispravnog koda može usput da reši problem i ispravi grešku, pre slučajno nego namerno, ali isto tako može i da je zamaskira ili da proizvede neke sasvim nove greške. Zbog toga *nije preporučljivo* da se refaktoriše neispravan kod.

Obično smatramo da je kod ispravan ako nema poznatih bagova. Ako ih ima, onda bi to trebalo da bude dovoljan razlog da ne pristupamo refaktorisanju. Međutim, u nekim težim slučajevima debagovanja, može da se ispostavi da je potrebno da se preduzme refaktorisanje da bi programski kod uopšte mogao da se analizira ili da bi greška mogla da se locira. Takve slučajeve je potrebno rešavati izuzetno pažljivo i strpljivo.

Druga pretpostavka je da je uočena neka slabost u strukturi koda. Ako je programski kod ispravan, a uz to se u kodu ne uočava nikakva slabost, ili ona ne može da se potpuno jasno i nedvosmisleno opiše, onda takav kod ne treba da se menja. Besciljno refaktorisanje je često lošiji izbor nego zadržavanje postojećeg stanja. Nijednu izmenu ni dopunu programskog koda ne treba da pravimo ako ne postoji jasno prepoznat cilj. Ako nema slabosti u dizajnu softvera, ili bar ne možemo da ih jasno prepoznamo i istaknemo, onda ne možemo da prepoznamo ni šta bismo refaktorisanjem popravili, pa zbog toga nema ni stvarne potrebe za refaktorisanjem.

Postupak refaktorisanja

Kada smo ustanovili da je programski kod ispravan i da postoji neka uočena slabost u strukturi programskog koda, onda možemo da započnemo refaktorisanje.

Prvi korak pri refaktorisanju je određivanje cilja. Cilj može da se postavi kao *lako dostižan*, kada jasno vidimo kako možemo da dođemo do njega putem malog broja transformacija, ili kao *udaljeni cilj*, kada na samom početku postupka ne možemo da potpuno tačno sagledamo ukupan obim neophodnih transformacija. Oba načina određivanja cilja imaju uporište u praksi. Za početnike u refaktorisanju lakše je i bolje da se ciljevi postavljaju i ispunjavaju u malim i sagledivim koracima. Iskusniji programeri mogu sebi da postavljaju i udaljenije ciljeve.

Posle određivanja cilja, planira se skup transformacija kojima će se taj cilj ispuniti. U slučaju jednostavnijih ciljeva, plan se pravi relativno lako. U slučaju težih ciljeva, plan transformacija ponekad ne može da se unapred sagleda sve do samog kraja. U takvim slučajevima se najpre planira prvih nekoliko koraka, pa se po njihovom dovršavanju nastavljaju analiziranje i planiranje. U svakom slučaju, svaka pojedinačna transformacija bi trebalo da bude sasvim jednostavna, skoro trivijalna.

Pre svake pojedinačne transformacije potrebno je da se proveri da li pri njenoj primeni može da dođe do nekakve promene ponašanja na koju postojeći testovi

jedinica koda možda ne bi ukazali. Kandidati se traže na osnovu razmatranja delova koda koji će biti obuhvaćeni planiranim transformacijama. Zatim se po potrebi pišu novi testovi, koji će da nam pomognu da proveravamo da li je ponašanje ostalo neizmenjeno.

Planirana transformacija može da se primeni tek kada smo sigurni da imamo odgovarajuću kolekciju testova. Nakon transformisanja koda potrebno je da se izmenjeni kod prevede i da se izvrše svi testovi. Ako je sve u redu i svi testovi prolaze, to znači da nije promenjeno poznato ponašanje i onda može da se pređe na sledeću transformaciju.

Opisani postupak se ponavlja sve dok se ne dostigne postavljeni cilj, tj. dok posmatrani deo koda ne postane dobro strukturiran.

Mesto refaktorisanja u programiranju

Programiranje bismo mogli da posmatramo kao proces koji se sastoji od naizmeničnog pisanja novog koda i refaktorisanja. Pisanje novog koda i refaktorisanje bi trebalo da u tom procesu budu potpuno i jasno razdvojeni, ali i da se redovno smenjuju i to u što manjim koracima.

Pisanje novog koda i refaktorisanje postojećeg koda se značajno razlikuju. Pisanje novog koda, u idealnom slučaju, ne menja mnogo postojeći kod, već pretežno dodaje novo ponašanje. Sa druge strane, refaktorisanje ne menja vidljivo ponašanje, već samo postojeći programski kod i njegovu strukturu.

Refaktorisanje nije aktivnost koju bi trebalo da primenjujemo povremeno ili u određenim vremenskim intervalima. Svako dodavanje novog programskog koda, ili menjanje postojećeg koda radi promene ponašanja, ima potencijal da pokvari dobru strukturu koda. Zbog toga programski kod mora da se refaktoriše *redovno*. Svaki put pre dodavanja novog koda potrebno je da se analizira postojeći kod i da se proveri da li postoji neki problem u njegovoj strukturi, koji može da napravi probleme prilikom pisanja novog koda. Ako uvidimo da postoje takvi delovi koda, to znači da smo uočili *slabost strukture* koda i da je potrebno da radimo na popravljanju tih slabosti, tj. da je vreme za refaktorisanje. Slično tome, svaki put posle dodavanja novog koda potrebno je da analiziramo da li je time pokvarena struktura koda i da je po potrebi popravimo primenom refaktorisanja.

Motivacija

Refaktorisanje podiže nivo kvaliteta dizajna softvera. To je ujedno cilj refaktorisanja i glavni motiv za njegovu primenu. Najvažnije prednosti dobro dizajniranog programskog koda u odnosu na loše strukturiran kod su:

- bolja razumljivost;
- lakše debagovanje;
- lakše održavanje i nadograđivanje.

U savremenom razvoju softvera se teži da životni vek svakog pojedinačnog dela programskog koda bude što duži. Ako je dizajn nekog programskog koda dobar, onda će on moći da se lakše i pouzdanije održava, menja, prilagođava novim potrebama i koristi u drugim projektima. Zbog toga ulaganje u dobar dizajn, tj. u dobru strukturu programskog koda, predstavlja trajnu vrednost.

Kada smo kao jedan od važnih ciljeva prepoznali produženje životnog veka našeg programskog koda, a uz to imamo u vidu da održavanje složenog softvera može da bude veoma skupo, onda je jasno koliko je važno da uradimo sve što možemo da bi cena održavanja koda bila što niža. Refaktorisanje je alat koji nam pomaže da spustimo cenu održavanja programskog koda kroz podizanje nivoa njegove uređenosti.

U klasičnom razvoju softvera se pod terminom *optimizacija* po pravilu podrazumevala optimizacija u odnosu na performanse izvršavanja. Danas to možemo da posmatramo malo drugačije i da primetimo da je i refaktorisanje vid optimizacije koda, ali ne u odnosu na performanse izvršavanja već *u odnosu na performanse održavanja*.

10.2 Kandidati za refaktorisanje

Ne postoji uopšten i nedvosmislen spisak uslova koje bi neki programski kod trebalo da zadovoljava da bismo ga refaktorisali. Umesto toga, možemo da pokušamo da prepoznamo *kandidate*, tj. slučajeve u kojima je potrebno razmotriti da li se refaktorisanjem može dobiti pomak u kvalitetu strukture koda.

Najopštije pravilo bi moglo da bude „ako kod zaudara, potrebno ga je izmeniti“ [*Fowler 1999*]. Važna karakteristika ovako oblikovanog pravila je isticanje subjektivnosti u procenjivanju strukture koda. Zaista, neće svi programeri biti jednako osetljivi za različite vrste *zaudaranja* koda. Neki programer bi mogao da odlaže neke transformacije koda, zato što mu izgledaju trivijalno i smatra da su manje značajne. Neko drugi bi mogao da istim transformacijama prida veći značaj. Na prepoznavanje potencijalnih slabosti u kodu može veoma značajno da utiče i prethodno iskustvo programera. Po pravilu „koga zmija ujede i guštera se plaši“, programeri pokazuju tendenciju da uočenim poznatim slabostima, koje su im ranije već zadavale muke, pridaju veći značaj nego nekim drugim, koje su možda i opasnije, ali sa kojima se ranije nisu susretali.

U knjizi [*Fowler 1999*] su predstavljene 22 vrste zaudaranja koda, odnosno kandidata za refaktorisanja:

- Ponavljanje koda;
- Dugačak metod;
- Velika klasa;
- Dugačka lista argumenata;
- Divergentne izmene;
- Distribuirana apstrakcija (Operisanje sačmaricom);
- Zavist prema karakteristikama;
- Skupovi podataka;
- Opsednutost primitivnim;
- Naredba *switch*;
- Paralelne hijerarhije nasleđivanja;
- Lenja klasa;
- Spekulativno uopštavanje;
- Privremeno polje;
- Lanci poruka;
- Posrednik;
- Nepoželjna bliskost;
- Alternativne klase sa različitim interfejsima;
- Nepotpune bibliotečke klase;
- Klasa-podatak;
- Odbačeno nasleđe i
- Komentari.

Ovde ćemo, radi ilustracije, da detaljnije opišemo samo nekoliko kandidata za refaktorisanje. Pri razmatranju kandidata za refaktorisanje potrebno je da se obrati pažnja na odgovarajuće slabosti u dizajnu – one su obično tesno povezane sa suprotstavljanjem nekim od ranije navedenih principa projektovanja (*6 - Principi projektovanja softvera*, na strani 109).

Navešćemo i uobičajene tehnike za refaktorisanje, kojima se odgovarajući problemi prevazilaze. Tehnikama refaktorisanja ćemo da posvetimo pažnju nešto kasnije (*10.3 Katalog refaktorisanja*, na strani 250).

Ponavljanje koda

Ponavljanje koda je jedan od najčešćih problema sa kojim se svakodnevno suočavamo. Problem sa ponovljenim kodom je, pre svega, u otežanom održavanju – ako iz nekog razloga promenimo neki programski kod, a ne promenimo na isti način i sve ostale ponovljene instance tog koda, onda tako najčešće pravimo grešku u

kodu, koja u nekim slučajevima može biti veoma teška za pronalaženje. Zbog toga je potrebno da se ponavljanje koda što doslednije izbegava.

Ponavljanje koda se refaktoriše nekim vidom apstrahovanja. Uobičajena refaktorisanja su:

- Izdvajanje metoda;
- Povlačenje metoda uz hijerarhiju;
- Pravljenje šablonskih metoda;
- Zamena algoritma i
- Izdvajanje klase.

Dugačak metod

Ako je neki metod (ili funkcija) suviše dugačak, verovatno bi trebalo da ga razložimo na više manjih. Problem sa velikim metodom je što on obično radi više stvari, pa se otežava njegovo razumevanje i održavanje, a time se nepotrebno povećava verovatnoća da u njemu postoji neka greška. Više jednostavnijih metoda se lakše održava – kod se lakše i bolje razume, lakše i brže debaguje i jednostavnije menja i proširuje.

Velike metode obično razlažemo na manje primenom jednog ili više refaktorisanja:

- Izdvajanje metoda;
- Zamena privremenih vrednosti upitima;
- Uvođenje parametarskog objekta;
- Zamena metoda objektom i
- Dekompozicija uslova.

Velika klasa

Tokom evolucije programa često nastaju nepotrebno velike klase. Na primer, jedna mala i jasno definisana klasa može kroz više iteracija da postepeno dobija nove metode i podatke, tako da domen njene upotrebe preraste inicijalne planove. Pod velikom klasom podrazumevamo klasu koja ima *mnogo* podataka ili metoda.

Naravno, *mnogo* je relativna mera. Neka klasa može da bude velika iz opravdanih razloga. U tom slučaju, naravno, nema potrebe da se refaktoriše. Ipak, velike klase često nemaju opravdanje. Dobar pokazatelj da je klasa neopravdano velika jeste ako se u programu pravi mnogo objekata te klase i to za različite namene. Ako se u različitim kontekstima koriste različiti skupovi metoda klase, to obično znači da klasa ima više odgovornosti, što je vrlo ozbiljna slabost dizajna. U tom slučaju je potrebno da se razmotre mogući načini da se kod popravi, što obično ima za

posledicu pravljenje novih klasa ili premeštanje dela ponašanja u neke druge već postojeće klase.

Uobičajena refaktorisanja za razlaganje velikih klasa su:

- Izdvajanje klase;
- Izdvajanje potklase;
- Izdvajanje interfejsa;
- Premeštanje metoda;
- Premeštanje podataka i
- Ponavljanje praćenih podataka.

Dugačka lista argumenata

U slučaju složenijih problema često se dešava da se pri pozivanju nekih metoda (ili funkcija) koristi veoma veliki broj argumenata. Ako metod ima mnogo argumenata, to može da znači da ima suviše faktora koji utiču na njegov rad. Čak i kada to nije slučaj, kod takvih metoda je teže razumeti namenu i međusoban odnos argumenata, pa je poželjno da se njihov broj smanji, ako je to moguće.

Smanjivanje broja argumenata ne radimo po svaku cenu, zato što u nekim slučajevima to ne doprinosi lakšem razumevanju i upotrebi, već nasuprot tome može i da ih dodatno oteža.

Postoji nekoliko refaktorisanja kojima može da se smanji broj argumenata metoda:

- Zamenjivanje argumenta metodom;
- Zadržavanje celog objekta i
- Uvođenje parametarskog objekta.

Distribuirana apstrakcija

Ako planiramo da dopunimo ili izmenimo ponašanje programa, a struktura koda je takva da nam predstoji menjanje velikog broja različitih klasa, to je najčešće posledica *distribuirane apstrakcije*. Jedan koncept bi trebalo da se apstrahuje jednom klasom, kada god je to moguće. Ako je umesto toga jedna ista apstrakcija podeljena u više klasa, ili čak više hijerarhija, onda je skoro izvesno da ćemo pri menjanju te apstrakcije morati da menjamo veći broj klasa. Zbog toga što jedna promena ponašanja zahteva promene na mnogo mesta u kodu, ova vrsta zaudaranja se naziva i *Operisanje sačmaricom*.

U takvim slučajevima je bolje da najpre preuredimo kod tako da se iz njega otkloni problematično distribuiranje apstrakcije, pa da tek onda preduzmemo planirane izmene ponašanja.

Za te potrebe najčešće se koriste refaktorisanja:

- Premeštanje metoda;
- Premeštanje podataka i
- Umetanje klase.

Naredba switch

Ponovljena upotreba naredbe *switch* sa istim ili sličnim kriterijumom grananja, obično je znak da je potrebno da se apstrahuje taj kriterijum grananja. Naredba *switch* je problematična zbog toga što eventualno dodavanje novih slučajeva, ili menjanje ponašanja postojećih slučajeva na mestu jedne takve naredbe obično mora da bude praćeno odgovarajućim izmenama na svim ostalim mestima gde se ponavlja isti ili sličan uslov grananja.

Ima više načina da se ovaj problem reši, a praktično svi se svode na apstrahovanje kriterijuma grananja nekim vidom polimorfizma, obično hijerarhijom klasa. Različite vrednosti kriterijuma se apstrahuju različitim klasama, a upotrebe naredbe *switch* se zamenjuju polimorfnim pozivima metoda te hijerarhije. Na raspolaganju su nam refaktorisanja:

- Izdvajanje metoda;
- Premeštanje metoda;
- Zamena kodiranog podatka potklasom;
- Zamena uslova polimorfizmom;
- Zamena parametra eksplicitnim metodom i
- Uvođenje praznih objekata.

Spekulativno uopštavanje

Spekulativno uopštavanje se odnosi na slučajeve kada se neki deo koda dodatno uopštava zbog pretpostavke da je potrebno da se podrži neki opšti slučaj, čak i kada u praksi imamo na delu samo jedan specifičan slučaj. Uobičajena posledica spekulativnog uopštavanja su neke suvišne apstrakcije u programskom kodu, koje nepotrebno uvode dodatne strukture i odnose u programski kod.

Takve apstrakcije su potencijalno korisne u trenutku kada se pojave drugačiji specifični slučajevi, ali sve do tada predstavljaju opterećenje koda i otežavaju njegovo održavanje, a ponekad i smanjuju efikasnost. Ključno pitanje, koje moramo da postavimo u ovakvim slučajevima, je *kada će to uopštavanje zatrebati*? Ako nije sasvim sigurno da će to biti u nekoj relativno bliskoj budućnosti, onda je preporučljivo da se nepotrebne apstrakcije sklone iz programskog koda. Za to se obično koristi neko od refaktorisanja:

- Sažimanje hijerarhije;
- Umetanje klase;
- Uklanjanje argumenta i
- Preimenovanje metoda.

Posrednik

Jedna od osnovnih karakteristika OO programiranja je tendencija da se detalji implementacije sakriju i da se korisniku pruži samo jasno oblikovan interfejs. Međutim, u tome može da se ode i predaleko. U nekim slučajevima, ako čitava klasa ima samo ulogu posrednika, koji sakriva neko složenije ponašanje, ali ne donosi značajne novine u ponašanju, onda je možda bolje da razmotrimo njeno uklanjanje.

Naravno, posrednici su nekada neophodni. Ali posrednici koji su neophodni obično donose sasvim jasno novo ponašanje, na primer ujednačavanje interfejsa više klasa, ili objedinjeni interfejs prema više klasa i sl. Ako se ispostavi da posrednik ne donosi prepoznatljivu dodatnu korist, pa zbog toga nije neophodan, onda možemo da ga uklonimo primenom transformacija:

- Uklanjanje posrednika;
- Umetanje metoda i
- Zamena delegiranja nasleđivanjem.

Komentari

Komentari su neophodan sadržaj implementacije programskog koda. Sami po sebi, komentari nisu slabost i nipošto ih ne treba izbegavati. Naprotiv, poželjno je komentarisati svaki značajan element programskog koda. Ipak, postoje komentari koji mogu da ukažu na neke slabosti komentarisanog koda.

Na primer, ako je neki deo programskog koda propraćen velikim brojem komentara, ili ako imamo neke komentare koji su dugački i objašnjavaju neke složene koncepte ili odnose u vezi sa odgovarajućim elementima programskog koda, onda to *može* da bude znak da u tom delu programa ima slabosti. Na takvim mestima se komentari često upotrebljavaju da zamaskiraju problematičan programski kod.

Obimni komentari su ozbiljni kandidati za refaktorisanje. Nakon uočavanja obimnih komentara moramo da se zapitamo da li je algoritam zaista toliko složen da zaslužuje toliko komentara? Ako jeste, onda su komentari opravdani, ali ako nije, onda smo možda nepotrebno zakomplikovali strukturu koda. Preporučljivo je da promenimo strukturu rešenja tako da većina komentara više ne budu potrebni. Neka od refaktorisanja, koja se u takvim prilikama koriste za pojednostavljivanje strukture koda, su:

- Izdvajanje metoda;
- Preimenovanje metoda i
- Uvođenje pretpostavke.

10.3 Katalog refaktorisanja

Da bi refaktorisanje moglo da se lakše primenjuje u praksi, kao i da bi se znanja o tehnikama refaktorisanja lakše prenosila drugima, potrebno je da se uvede određena sistematičnost u skup elementarnih transformacija. Slično kao u slučaju obrazaca za projektovanje i u slučaju refaktorisanja se prave odgovarajući katalozi. Katalozi predstavljaju istovremeno rečnik, zbirku i klasifikaciju refaktorisanja.

Sadržaj kataloga refaktorisanja čine izabrana elementarna refaktorisanja, tj. „atomične" transformacije koje se primenjuju između dva testiranja koda[47]. Refaktorisanja se u katalogu grupišu prema nameni. Katalog u knjizi [*Fowler 1999*] sadrži 72 tehnike refaktorisanja, koje su organizovane u 7 grupa:

- Metodi komponovanja koda;
- Premeštanje koda između objekata;
- Organizovanje podataka;
- Pojednostavljivanje uslovnih izraza;
- Pojednostavljivanje pozivanja metoda;
- Razrešavanje uopštavanja i
- Velika refaktorisanja.

Opisi refaktorisanja su obično relativno sažeti, ali dovoljno ilustrativni. Opis svake elementarne tehnike refaktorisanja obuhvata:

- *ime* refaktorisanja, koje nam olakšava referisanje i pravljenje rečnika refaktorisanja;
- *rezime*, tj. sažeti opis slučaja u kome se primenjuje i tehnike izvođenja;
- *klasifikaciju*, tj. pripadnost određenoj grupi refaktorisanja;
- *objašnjenje motivacije* za primenu, koje obično ukazuje na prednosti popravljene strukture koda;

[47] Neke tehnike imaju više koraka između kojih može da se izvodi testiranje, ali se očekuje da pri takvom testiranju ne prolaze sve pojedinačne provere. Na kraju primene tehnike, naravno, svi testovi moraju da prolaze.

- *mehanizam*, tj. jasan i detaljan opis tehnike izvođenja refaktorisanja i
- *primer*.

Radi ilustracije ćemo opisati nekoliko tehnika refaktorisanja. Još neke tehnike ćemo pomenuti u okviru većeg primera refaktorisanja (odeljak 10.4 , na strani 260).

Izdvajanje metoda

Rezime

Deo implementacije nekog metoda može da se posmatra kao celina ili je objašnjen komentarom. Taj deo koda može da se izdvoji u poseban metod.

Klasifikacija

Metodi komponovanja koda.

Motivacija

Izdvajanje metoda je jedno od najčešće upotrebljavanih refaktorisanja. Predstavlja uobičajen odgovor na zaudaranja kao što su *Dugačak metod* ili *Komentari*. Proizvodi čitljiviji i jasniji programski kod. Često se upotrebljava kao deo većih zahvata.

Kratka ilustracija

Pre refaktorisanja:

```
... velikiMetod( ... ) {
    ...
    ...deo koda koji želimo da izdvojimo...
    ...
}
```

Posle refaktorisanja:

```
... noviMetod( ... ) {
    ...deo koda koji želimo da izdvojimo...
}

... velikiMetod( ... ) {
    ...
    ... noviMetod(...);
    ...
}
```

Mehanizam

- Počinjemo od prepoznavanja dela koda koji će biti izdvojen.
- Prvi korak je pravljenje novog metoda u istoj klasi. Biramo kratko i jasno ime koje ga dobro opisuje.

  - Ako ne možemo da pronađemo dobro ime, koje kratko i jasno opisuje *šta* taj deo koda radi, a ne *kako* to radi, onda je možda bolje da ne izdvajamo metod, nego da najpre probamo sa drugim tehnikama refaktorisanja.
- Zatim kopiramo odgovarajući deo koda u novi metod.
- Program u ovom trenutku obično još uvek ne može da se prevede. Potrebno je da obezbedimo razmenu podataka metoda sa pozivaocem:
  - Ako se u tom delu koda koriste neke promenljive koje dobijaju vrednost u delu koda koji mu prethodi, onda je potrebno da obezbedimo da se one prenesu u nov metod kao parametri. Dodajemo po parametar za svaku takvu promenljivu. Obično takvi parametri zadržavaju isti naziv kao promenljiva. Trudimo se da sve parametre prenosimo po vrednosti ili kao konstantne reference.
  - Ako se u tom delu koda menjaju postojeće ili definišu nove promenljive, tako da su njihove vrednosti važne i nakon njegovog izvršavanja, onda one određuju rezultat novog metoda. Ako je potrebno da se menja ili vrati stanje više promenljivih, onda mogu da se koriste parametri koji se prenose po imenu (reference).
- Sada bi trebalo da program može da se prevede. To je pravi trenutak za pisanje novih testova koji proveravaju rad novog metoda. Dodajemo testove, prevodimo i testiramo, sve dok ne budemo zadovoljni rezultatom.
- Zamenjujemo polazni deo koda pozivanjem novog metoda.
- Prevodimo i testiramo.
- Po potrebi, popravljamo novi i stari metod. U svakom od njih možda možemo da uprostimo rad sa nekim promenljivim ili čak da ih izbacimo.

Primer

Metod `Test::PostaviPitanja()` je tipičan *Dugačak metod*, koji radi mnogo toga u okviru testiranja znanja korisnika. Jedan njegov deo postavlja jedno pitanje i potrebno je da se izdvoji u poseban metod:

```
void PostaviPitanja()
{
    string odgovor;
    ...
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        Pitanje& pitanje = Pitanja_[i];
        cout << endl
             << "*** Pitanje br. " << (i+1) << " ***"
             << endl << endl
             << pitanje.TekstPitanja() << endl
```

```
                << "----------------------------------------"
                << endl;
            for( unsigned j=0;
                 j<pitanje.PonudjeniOdgovori().size();
                 j++
               )
               cout << pitanje.PonudjeniOdgovori()[j] << endl;
            cout << "----------------------------------------"
                 << endl
                 << "Upisite slovo koje stoji "
                    "ispred tacnog odgovora " << endl
                 << "ili znak @ ako ne znate: ";
            cin >> odgovor;
            ...
        }
        ...
    }
```

Pravimo novi privatni metod `Test::PostaviPitanje`:

```
void PostaviPitanje() const {}
```

Zatim iskopiramo odgovarajući deo koda:

```
void PostaviPitanje() const
{
    cout << endl
         << "*** Pitanje br. " << (i+1) << " ***"
         << endl << endl
         << pitanje.TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0;
         j<pitanje.PonudjeniOdgovori().size();
         j++
       )
       cout << pitanje.PonudjeniOdgovori()[j] << endl;
    cout << "----------------------------------------"
         << endl
         << "Upisite slovo koje stoji "
            "ispred tacnog odgovora " << endl
         << "ili znak @ ako ne znate: ";
    cin >> odgovor;
}
```

Primećujemo da se u tom delu koda upotrebljavaju promenljive `i`, `pitanje` i `odgovor`. Promeniljve `i` i `pitanje` se samo čitaju, pa ćemo da dodamo odgovarajuće parametre:

```
void PostaviPitanje( int i, const Pitanje& pitanje ) const
```

Promenljiva `odgovor` dobija svoju vrednost u tom delu koda i kasnije se koristi. Ona će da predstavlja rezultat rada metoda:

```
string PostaviPitanje( int i, const Pitanje& pitanje ) const
{
    ...
    string odgovor;
    cin >> odgovor;
    return odgovor;
}
```

Sada je vreme za pravljenje testova i testiranje novog metoda.

Na kraju izmenimo početni metod tako da koristi novi metod:

```
void PostaviPitanja()
{
    ...
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        Pitanje& pitanje = Pitanja_[i];
        string odgovor = PostaviPitanje( i, pitanje );
        ...
    }
    ...
}
```

Prevodimo i testiramo.

Premeštanje metoda

Rezime

Metod koristi više elemenata druge klase nego klase u kojoj je definisan. Taj metod može da se premesti u tu drugu klasu.

Klasifikacija

Premeštanje koda između objekata.

Motivacija

Ako metod više koristi neku drugu klasu (tj. njene metode i podatke) nego sopstvenu, to obično znači da taj metod ima odgovornost da nešto radi sa objektom druge klase, ali to onda često znači da bi metod trebalo da pripada skupu odgovornosti (i ponašanja) te klase, a ne one u kojoj se nalazi. Ovim refaktorisanjem se smanjuje nivo spregnutosti između metoda različitih klasa, pa i između klasa.

Kratka ilustracija

Pre refaktorisanja:

```
class KlasaA {
    ... metod(...)...
```

```
}
class KlasaB {
    ...
}
```

Posle refaktorisanja:

```
class KlasaA {
    ...
}
class KlasaB {
    ... metod(...)...
}
```

Mehanizam

- Prvo mora da se proveri šta sve metod koristi iz klase u kojoj se nalazi:
  - Za svaki podatak i metod, koji se koriste u metodu, proveriti da li je možda potrebno da se i oni premeste u drugu klasu. Ako je potrebno, onda prvo njih premestimo ili čak možemo da se odlučimo da ih premestimo sve zajedno.
  - Možda je potrebno da se prvo izdvoji a zatim i premesti samo deo polaznog metoda. U tom slučaju počinjemo od *Izdvajanja metoda*.
- Proveriti da li se taj metod redefiniše u nekoj izvedenoj klasi, ili je možda virtualan? U takvim slučajevima premeštanje ili nije moguće ili može da zahteva složenije izmene koda.
- Napravimo novi metod u ciljnoj klasi. Po potrebi možemo da izaberemo drugačije ime, tako da više odgovara kontekstu.
- Zatim iskopiramo telo metoda i prilagodimo ga novoj sredini. To obuhvata i uređivanje parametara i rezultata.
- Napišemo testove novog metoda, prevodimo i testiramo ciljnu klasu.
- U polaznom metodu promenimo kod tako da se telo metoda praktično svodi na pozivanje novog metoda iz druge klase.
- Prevodimo i testiramo.
- Svuda u programu zamenimo pozivanje polaznog metoda pozivanjem novog metoda. Obrišemo polazni metod.
- Prevodimo i testiramo.

Primer

Metod `Test::PostaviPitanje` iz prethodnog primera praktično u potpunosti počiva na upotrebi elemenata objekta `pitanje`, pa je očigledno da je njegovo pravo mesto u klasi `Pitanje`, a ne u klasi `Test`. Ipak, prvi deo metoda, koji ispisuje redni broj pitanja, bi trebalo da ostane u polaznoj klasi:

```
string PostaviPitanje( int i, const Pitanje& pitanje ) const
{
    cout << endl
         << "*** Pitanje br. " << (i+1) << " ***"
         << endl << endl
         << pitanje.TekstPitanja() << endl
         << "-----------------------------------------"
         << endl;
    for( unsigned j=0;
         j<pitanje.PonudjeniOdgovori().size();
         j++
        )
        cout << pitanje.PonudjeniOdgovori()[j] << endl;
    cout << "-----------------------------------------"
         << endl
         << "Upisite slovo koje stoji "
            "ispred tacnog odgovora " << endl
         << "ili znak @ ako ne znate: ";
    string odgovor;
    cin >> odgovor;
    return odgovor;
}
```

Zato ćemo prvo da primenimo refaktorisanje *Izdvajanje metoda*. Izdvajamo sve osim prva tri reda u novi metod, kome ne treba parametar `i`:

```
string PostaviPitanje( const Pitanje& pitanje ) const
{
    cout << pitanje.TekstPitanja() << endl
         << "-----------------------------------------"
         << endl;
    for( unsigned j=0;
         j<pitanje.PonudjeniOdgovori().size();
         j++
        )
        cout << pitanje.PonudjeniOdgovori()[j] << endl;
    cout << "-----------------------------------------"
         << endl
         << "Upisite slovo koje stoji "
            "ispred tacnog odgovora " << endl
         << "ili znak @ ako ne znate: ";
    string odgovor;
    cin >> odgovor;
    return odgovor;
}
```

Zatim menjamo polazni metod:

```
string PostaviPitanje( int i, const Pitanje& pitanje ) const
{
    cout << endl
```

```
            << "*** Pitanje br. " << (i+1) << " ***"
            << endl << endl;
        return PostaviPitanje( pitanje );
    }
```

Sada smo spremni da novoizdvojeni metod premestimo u klasu `Pitanje`. Prvo pravimo novi metod. Nije nam potreban parametar `pitanje`, zato što će tu ulogu igrati pokazivač `this`. Iskopiramo telo metoda i prilagodimo novom kontekstu:

```
// u klasi Pitanje...
string PostaviPitanje() const
{
    cout << TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0;
         j < PonudjeniOdgovori().size();
         j++
        )
        cout << PonudjeniOdgovori()[j] << endl;
    cout << "----------------------------------------"
         << endl
         << "Upisite slovo koje stoji "
            "ispred tacnog odgovora " << endl
         << "ili znak @ ako ne znate: ";
    string odgovor;
    cin >> odgovor;
    return odgovor;
}
```

Napišemo testove, prevodimo i testiramo.

U polaznoj klasi na svim mestima gde je pozivan stari metod, sada ćemo da pozivamo novi, a stari metod obrišemo:

```
string PostaviPitanje( int i, const Pitanje& pitanje ) const
{
    cout << endl
         << "*** Pitanje br. " << (i+1) << " ***"
         << endl << endl;
    return pitanje.PostaviPitanje();
}
```

Prevodimo i testiramo.

Razdvajanje upita od modifikatora

Rezime

Imamo metod koji vraća rezultat, ali istovremeno i menja stanje objekta. Umesto njega pravimo dva metoda, od kojih jedan samo vraća rezultat, a drugi samo menja stanje.

Klasifikacija

Pojednostavljivanje poziva metoda.

Motivacija

Ako metod vraća rezultat, a pri tome ne menja stanje, onda možemo da ga bezbedno pozivamo više puta i sa raznih mesta u programu. Sa druge strane, ako pored toga metod i menja stanje, onda moramo veoma pažljivo da ga koristimo i često smo prinuđeni da njegov rezultat čuvamo u privremenoj promenljivoj. Štaviše, takav metod nije upotrebljiv na konstantnim objektima. Zato obično težimo da razdvojimo operacije čitanja i pisanja.

Kratka ilustracija

Pre refaktorisanja:

```
class Klasa { ...
    ... metodKojiČitaIPiše ...
};
```

Posle refaktorisanja:

```
class Klasa { ...
    ... metodKojiČita ...
    void metodKojiPiše ...
};
```

Mehanizam

- Proučimo polazni metod i na osnovu njega napravimo novi metod koji samo računa rezultat i ne menja stanje objekta.
  - Često to može da se uradi tako što se iskopira ceo metod, pa se samo obrišu delovi koji menjaju stanje. Drugi način je da se promene stanja prevedu u promene lokalnih promenljivih.
- Pišemo testove novog metoda, prevodimo i testiramo.
- Promenimo polazni metod tako da vraća rezultat poziva novog metoda.
  - Obrišemo sve preostale viškove.
  - Alternativno, umesto da menjamo polazni metod, možemo da napravimo novi, koji samo menja stanje.

- Na svim mestima gde se poziva polazni metod, dodamo ispred toga poziv novog metoda i taj rezultat sačuvamo, a rezultat poziva originalnog metoda odbacimo, tj. ignorišemo.
- Izmenimo originalni metod da ne vraća rezultat.
- Prevodimo i testiramo.

Primer

Klasa `Stack` ima sledeći interfejs:

```
class Stack {
public:
    void push( const T& x ){...};
    void pop( T& x ){...};
    ...
private:
    vector<T> Elementi_;
};
```

Metod `pop` čita i briše poslednji element. Potrebno je da se podeli na dva metoda, tako da jedan samo čita a drugi samo menja stanje i briše poslednji element. Najpre pravimo metod koji samo čita poslednji element:

```
class Stack {
public:
    const T& top() const {
        return Elementi_.back();
    }
    ...
};
```

Pravimo testove za nov metod, prevodimo i testiramo.

Zatim dodajemo metod koji samo briše poslednji element:

```
class Stack {
public:
    void pop() {
        Elementi_.pop_back();
    }
    ...
};
```

Pravimo testove za nov metod, prevodimo i testiramo.

Na svim mestima u kodu gde je korišćen stari metod, zamenimo njegovu upotrebu pozivanjem para novih metoda, tako da se prvo poziva `top`, a zatim i `pop`. Prevodimo i testiramo.

Obrišemo stari metod.

10.4 Primer refaktorisanja

Pravi smisao refaktorisanja se najbolje vidi na nešto složenijim primerima. Ovde ćemo da pokušamo da sagledamo primenu nekih od osnovnih tehnika refaktorisanja na primeru programa, koji nije mnogo složen i relativno je jednostavan za razumevanje, ali ipak dopušta da se naprave neke greške u dizajnu, koje kasnije možemo da popravljamo refaktorisanjem. Još nekoliko manjih primera refaktorisanja u programskom jeziku C++ može da se pronađe u [*Malkov 2007*].

Radi se o programu za testiranje znanja korisnika. Program sprovodi testiranje postavljanjem pitanja sa ponuđenim odgovorima. Korisnik bira jedan od ponuđenih odgovora upisivanjem slova koje stoji ispred odgovora. Tačan odgovor nosi 5 poena, pogrešan odgovor nosi -3 poena, a odgovor „Ne znam“ nosi -1 poen. Jedan primer izvršavanja programa bi mogao da bude:

```
*** Pitanje br. 1 ***

Sta je to 'std::map'?
-----------------------------------------
a - Putokaz do gusarskog blaga.
b - Asocijativni niz u standardnoj biblioteci p.j. C++.
c - Mocvarni teren opasan po zivot.
-----------------------------------------
Upisite slovo koje stoji ispred tacnog odgovora
ili znak @ ako ne znate: b


*** Pitanje br. 2 ***

Koliko tacnih odgovora ima ovo pitanje?
-----------------------------------------
a - Tri.
b - Dva.
c - Jedan.
-----------------------------------------
Upisite slovo koje stoji ispred tacnog odgovora
ili znak @ ako ne znate: b


Rezultat testiranja:
-----------------------------------------
1.  ( 5 ) Tacan odgovor.
2.  (-3 ) Netacan odgovor.
-----------------------------------------
Rezultat: 2 poena
```

U planu je da se program proširuje dodavanjem novih vrsta pitanja i zapisivanjem izveštaja o testiranju u datoteci. Program nije dovoljno dobro dizajniran, pa dodavanje novih mogućnosti nije jednostavno. Zbog toga ćemo da

probamo da nizom refaktorisanja popravimo dizajn programa. Nakon nekoliko refaktorisanja uslediće dodavanje nekih novih mogućnosti, a zatim ćemo nastaviti sa još nekoliko refaktorisanja.

Podsetićemo čitaoce da je u praksi neophodno da se refaktorisanje radi uz doslednu upotrebu testova jedinica koda. U ovom primeru se ne koriste testovi jedinica koda, pre svega zbog ograničenog prostora, ali i zbog usmeravanja pažnje čitalaca na same probleme i tehnike refaktorisanja, a ne na prateće testove.

Program pre refaktorisanja

Polazni program čine dve klase: `Test` i `Pitanje`. Klasa `Pitanje` predstavlja jedno pitanje i ponuđene odgovore. Ima samo konstruktor i pristupne metode `TekstPitanja`, `TacanOdgovor` i `PonudjeniOdgovori`. Klasa `Test` predstavlja model jednog testa i može da obuhvati veći broj pitanja. Ima metode:

- podrazumevani konstruktor – pravi prazan test (bez pitanja);
- `Dodaj` – dodaje jedno pitanje testu;
- `PostaviPitanja` – izvodi testiranje i ispisuje izveštaj o rezultatima testiranja.

Sledi početni programski kod:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

//-----------------------------------------------------------------
class Pitanje
{
public:
    Pitanje( const string& tekstPitanja,
             const string& tacanOdgovor,
             const string& odgovor1,
             const string& odgovor2 = "",
             const string& odgovor3 = "",
             const string& odgovor4 = "",
             const string& odgovor5 = ""
        )
        : TekstPitanja_( tekstPitanja ),
          TacanOdgovor_( tacanOdgovor )
    {
        PonudjeniOdgovori_.push_back( odgovor1 );
        if( odgovor2 != "" )
            PonudjeniOdgovori_.push_back( odgovor2 );
```

```
        if( odgovor3 != "" )
            PonudjeniOdgovori_.push_back( odgovor3 );
        if( odgovor4 != "" )
            PonudjeniOdgovori_.push_back( odgovor4 );
        if( odgovor5 != "" )
            PonudjeniOdgovori_.push_back( odgovor5 );
    }

    const string& TekstPitanja() const
        { return TekstPitanja_; }
    const string& TacanOdgovor() const
        { return TacanOdgovor_; }
    const vector<string>& PonudjeniOdgovori() const
        { return PonudjeniOdgovori_; }

private:
    string TekstPitanja_;
    string TacanOdgovor_;
    vector<string> PonudjeniOdgovori_;
};

//---------------------------------------------------------------
class Test
{
public:
    void Dodaj( const Pitanje& p )
        { Pitanja_.push_back(p); }

    void PostaviPitanja()
    {
        string odgovor;
        vector<string> rezultati;
        int zbirPoena = 0;
        for( unsigned i=0; i<Pitanja_.size(); i++ ) {
            Pitanje& pitanje = Pitanja_[i];
        cout << endl
                 << "*** Pitanje br. " << (i+1) << " ***"
                 << endl << endl
                 << pitanje.TekstPitanja() << endl
                 << "----------------------------------------"
                 << endl;
            for( unsigned j=0;
                 j<pitanje.PonudjeniOdgovori().size();
                 j++
                )
                cout << pitanje.PonudjeniOdgovori()[j] << endl;
            cout << "----------------------------------------"
                 << endl
                 << "Upisite slovo koje stoji "
                    "ispred tacnog odgovora " << endl
                 << "ili znak @ ako ne znate: ";
            cin >> odgovor;
```

```
            if( odgovor == pitanje.TacanOdgovor() ) {
                rezultati.push_back("( 5 ) Tacan odgovor.");
                zbirPoena += 5;
            } else if( odgovor == "@" ) {
                rezultati.push_back("(-1 ) Nije odgovoreno.");
                zbirPoena -= 1;
            } else {
                rezultati.push_back("(-3 ) Netacan odgovor.");
                zbirPoena -= 3;
            }
            cout << endl;
        }

        cout << endl << "Rezultat testiranja:" << endl
             << "-------------------------------------------"
             << endl;
        for( unsigned i=0; i<Pitanja_.size(); i++ )
        cout << (i+1) << ".  "
                 << rezultati[i] << endl;
        cout << "-------------------------------------------"
             << endl
             << "Rezultat: " << zbirPoena << " poena" << endl;
    }

private:
    vector<Pitanje> Pitanja_;
};

//---------------------------------------------------------------------
int main( int argc, char** argv )
{
    Test test;
    test.Dodaj( Pitanje(
        "Sta je to 'std::map'?",
        "b",
        "a - Putokaz do gusarskog blaga.",
        "b - Asocijativni niz u standardnoj biblioteci p.j. C++.",
        "c - Mocvarni teren opasan po zivot."
    ));
    test.Dodaj( Pitanje(
        "Koliko tacnih odgovora ima ovo pitanje?",
        "c",
        "a - Tri.",
        "b - Dva.",
        "c - Jedan."
    ));

    test.PostaviPitanja();
}
```

Korak 1 – Izdvajanje metoda PostaviPitanje

Predstavljen program radi u skladu sa opisom, ali već i površno pregledanje koda je dovoljno da se uoči da je praktično sva složenost programa smeštena u samo jednom metodu: `Test::PostaviPitanja`. Ovaj metod je obuhvatio previše različitih stvari, od postavljanja pitanja, proveravanja i bodovanja odgovora, pa sve do ispisivanja izveštaja o obavljenom testiranju.

U pitanju je zaudaranje *Dugačak metod*, koje može da se popravi refaktorisanjem *Izdvajanje metoda*. Štaviše, iz našeg dugačkog metoda možemo da izdvojimo nekoliko novih metoda.

Prvi kandidat za izdvajanje je deo metoda koji postavlja jedno pitanje:

```
void PostaviPitanja()
{
    string odgovor;
    ...
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        Pitanje& pitanje = Pitanja_[i];
        cout << endl
             << "*** Pitanje br. " << (i+1) << " ***"
             << endl << endl
             << pitanje.TekstPitanja() << endl
             << "----------------------------------------"
             << endl;
        for( unsigned j=0;
             j<pitanje.PonudjeniOdgovori().size();
             j++
            )
            cout << pitanje.PonudjeniOdgovori()[j] << endl;
        cout << "----------------------------------------"
             << endl
             << "Upisite slovo koje stoji "
                "ispred tacnog odgovora " << endl
             << "ili znak @ ako ne znate: ";
        cin >> odgovor;
        ...
    }
    ...
}
```

Označeni deo koda izdvajamo u novi privatni metod `Test::PostaviPitanje`. Mehanizam refaktorisanja *Izdvajanje metoda* nalaže da prvo napravimo novi metod. Navodimo neophodne argumente i dodajemo lokalnu promenljivu `odgovor`:

```
string PostaviPitanje( int i, const Pitanje& pitanje ) const
{
    cout << endl
         << "*** Pitanje br. " << (i+1) << " ***"
```

```
             << endl << endl
             << pitanje.TekstPitanja() << endl
             << "----------------------------------------"
             << endl;
        for( unsigned j=0;
             j<pitanje.PonudjeniOdgovori().size();
             j++
            )
            cout << pitanje.PonudjeniOdgovori()[j] << endl;
        cout << "----------------------------------------"
             << endl
             << "Upisite slovo koje stoji "
                "ispred tacnog odgovora " << endl
             << "ili znak @ ako ne znate: ";
        string odgovor;
        cin >> odgovor;
        return odgovor;
    }
```

U sledećem koraku ovog refaktorisanja popravljamo postojeći dugačak metod `Test::PostaviPitanja` tako da koristi novi metod. Zamenjujemo deo koda koji je premešten u novi metod njegovim pozivanjem. Takođe, premeštamo definiciju lokalne promenljive `odgovor` sve do mesta na kome ona dobija vrednost:

```
void PostaviPitanja()
{
    ...
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        Pitanje& pitanje = Pitanja_[i];
        string odgovor = PostaviPitanje( i, pitanje );
        ...
    }
    ...
}
```

Korak 2 – Izdvajanje metoda ProveriOdgovor

Nastavljamo sa popravljanjem pretrpanog metoda `Test::PostaviPitanja`. Sada ćemo da u njemu uočimo deo koji se odnosi na proveravanje ispravnosti odgovora:

```
void PostaviPitanja()
{   ...
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        Pitanje& pitanje = Pitanja_[i];
        string odgovor = PostaviPitanje( i, pitanje );
        if( odgovor == pitanje.TacanOdgovor() ) {
            rezultati.push_back("( 5 ) Tacan odgovor.");
            zbirPoena += 5;
```

```
            } else if( odgovor == "@" ) {
                rezultati.push_back("(-1 ) Nije odgovoreno.");
                zbirPoena -= 1;
            } else {
                rezultati.push_back("(-3 ) Netacan odgovor.");
                zbirPoena -= 3;
            }
            cout << endl;
        }
        ...
    }
```

Taj deo implementacije metoda `PostaviPitanja` ćemo da izdvojimo u nov privatan metod `Test::ProveriOdgovor`.

```
int ProveriOdgovor( const Pitanje& pitanje,
                    const string& odgovor,
                    vector<string>& rezultati ) const
{
    int zbirPoena = 0;
    if( odgovor == pitanje.TacanOdgovor() ){
        rezultati.push_back("( 5 ) Tacan odgovor.");
        zbirPoena += 5;
    }else if( odgovor == "@" ){
        rezultati.push_back("(-1 ) Nije odgovoreno.");
        zbirPoena -= 1;
    }else{
        rezultati.push_back("(-3 ) Netacan odgovor.");
        zbirPoena -= 3;
    }
    return zbirPoena;
}
```

U prvom koraku izdvajanja radimo tako da što manje menjamo izdvojeni deo koda. Samo smo dodali novu lokalnu promenljivu `zbirPoena`, koju na početku inicijalizujemo nulom i na kraju vraćamo kao rezultat.

Popravljamo metod `Test::PostaviPitanja` tako da koristi nov metod za proveravanje ispravnosti odgovora:

```
void PostaviPitanja()
{
    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ){
        Pitanje& pitanje = Pitanja_[i];
        string odgovor = PostaviPitanje( i, pitanje );
        zbirPoena +=
            ProveriOdgovor( pitanje, odgovor, rezultati );
        cout << endl;
```

```
    }
    ...
}
```

Nakon što smo završili ovo refaktorisanje i testirali ponašanje programa, sada možemo da dodatno popravimo izdvojeni metod. Vidimo da se promenljiva `zbirPoena` u metodu `Test::ProveriOdgovor` koristi samo jedanput, pa ispada da ona samo komplikuje implementaciju. Rešenje je da je izbacimo:

```
int ProveriOdgovor( const Pitanje& pitanje,
                    const string& odgovor,
                    vector<string>& rezultati ) const
{
    if( odgovor == pitanje.TacanOdgovor() ){
        rezultati.push_back("( 5 ) Tacan odgovor.");
        return 5;
    }else if( odgovor == "@" ){
        rezultati.push_back("(-1 ) Nije odgovoreno.");
        return -1;
    }else{
        rezultati.push_back("(-3 ) Netacan odgovor.");
        return -3;
    }
}
```

Korak 3 – Izdvajanje i premeštanje metoda PostaviPitanje

Metod `Test::PostaviPitanje` ima dve celine koje nemaju isti domen primene. Prvi deo, koji ispisuje redni broj pitanja, predstavlja pripremu za postavljanje pitanja i zavisi od konteksta u kome se izvršava, ali ne i od pitanja koje se postavlja. Njemu je mesto u klasi `Test`. Međutim, drugi deo, koji zaista postavlja pitanje, pre pripada klasi `Pitanje` nego klasi `Test`.

Ovde imamo na delu kombinaciju dva zaudaranja: najpre *Dugačak metod*, zato što je naš metod `Test::PostaviPitanje` ispao duži nego što je poželjno, a zatim i *Klasa-podatak*, zato što klasa `Pitanje` u našem programu nema nikakvu dodeljenu odgovornost osim čuvanja sadržaja. Oba zaudaranja ćemo rešiti primenom dva refaktorisanja. Najpre ćemo da primenimo *Izdvajanje metoda*, a zatim i *Premeštanje metoda*.

Iz metoda `Test::PostaviPitanje( int i, const Pitanje& pitanje )` izdvajamo metod `Test::PostaviPitanje( const Pitanje& pitanje )`:

```
string PostaviPitanje( int i, const Pitanje& pitanje ) const
{
    cout << endl
         << "*** Pitanje br. " << (i+1) << " ***";
```

```
         << endl << endl;
    return PostaviPitanje( pitanje );
}

string PostaviPitanje( const Pitanje& pitanje ) const
{
    cout << pitanje.TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0;
         j<pitanje.PonudjeniOdgovori().size();
         j++
        )
        cout << pitanje.PonudjeniOdgovori()[j] << endl;
    cout << "----------------------------------------"
         << endl
         << "Upisite slovo koje stoji "
            "ispred tacnog odgovora " << endl
         << "ili znak @ ako ne znate: ";
    string odgovor;
    cin >> odgovor;
    return odgovor;
}
```

Zatim novi metod premeštamo u klasu `Pitanje`. Više nam nije neophodan eksplicitan argument `pitanje`, zato što će ga zameniti implicitan argument `this`:

```
string PostaviPitanje() const
{
    cout << TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0; j<PonudjeniOdgovori().size(); j++ )
        cout << PonudjeniOdgovori()[j] << endl;
    cout << "----------------------------------------"
         << endl
         << "Upisite slovo koje stoji "
            "ispred tacnog odgovora " << endl
         << "ili znak @ ako ne znate: ";
    string odgovor;
    cin >> odgovor;
    return odgovor;
}
```

Preostaje nam da popravimo metod `Test::PostaviPitanje`:

```
string PostaviPitanje( int i, const Pitanje& pitanje ) const
{
    cout << endl
         << "*** Pitanje br. " << (i+1) << " ***"
         << endl << endl;
```

```
        return pitanje.PostaviPitanje();
    }
```

Korak 4 – Premeštanje metoda ProveriOdgovor

U prethodnom koraku smo premestili metod `PostaviPitanje` (tj. deo prvobitne verzije tog metoda) iz klase `Test` u klasu `Pitanje`. I metod `ProveriOdgovor` zaslužuje isti tretman, zato što se odnosi isključivo na pojedinačno pitanje, a ne na test kao celinu.

Primetimo da je pre premeštanja ovog metoda potrebno da blago izmenimo njegovu semantiku – u klasi `Test` ovaj metod dodaje rezultat u niz rezultata, ali nije dobro da zahtevamo da metod klase `Pitanje` *zna* za niz rezultata. Zbog toga ćemo prvo da promenimo drugi argument metoda, tako da umesto niza rezultata to bude samo jedna niska prenesena po referenci.

Počinjemo popravljanjem metoda `Test::ProveriOdgovor`:

```
int ProveriOdgovor( const Pitanje& pitanje,
                    const string& odgovor,
                    string& rezultat ) const
{
    if( odgovor == pitanje.TacanOdgovor() ){
        rezultat = "( 5 ) Tacan odgovor.";
        return 5;
    }else if( odgovor == "@" ){
        rezultat = "(-1 ) Nije odgovoreno.";
        return -1;
    }else{
        rezultat = "(-3 ) Netacan odgovor.";
        return -3;
    }
}
```

Zatim popravljamo i metod `Test::PostaviPitanja`:

```
void PostaviPitanja()
{
    ...
        string rezultat;
        zbirPoena +=
            ProveriOdgovor( pitanje, odgovor, rezultat );
        rezultati.push_back(rezultat);
    ...
}
```

Sada metod `ProveriOdgovor` možemo da premestimo u klasu `Pitanje`:

```
int ProveriOdgovor( const string& odgovor,
                    string& rezultat ) const
{
    if( odgovor == TacanOdgovor() ){
        rezultat = "( 5 ) Tacan odgovor.";
        return 5;
    }else if( odgovor == "@" ){
        rezultat = "(-1 ) Nije odgovoreno.";
        return -1;
    }else{
        rezultat = "(-3 ) Netacan odgovor.";
        return -3;
    }
}
```

Ostaje nam još da popravimo metod `Test::PostaviPitanja`:

```
void PostaviPitanja()
{
    ...
        zbirPoena +=
            pitanje.ProveriOdgovor( odgovor, rezultat );
    ...
}
```

Korak 5 – Izdvajanje metoda Izvestaj iz PostaviPitanja

Prethodnim koracima smo malo pojednostavili metod `Test::PostaviPitanja`, ali on i dalje radi nekoliko različitih stvari. Da bismo ga dodatno rasteretili, potrebno je da se ispisivanje izveštaja izdvoji u poseban metod `Test::Izvestaj`. Iz postojećeg metoda `PostaviPitanja`:

```
void PostaviPitanja()
{
    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ){
        ...
    }

    cout << endl << "Rezultat testiranja:" << endl
         << "-----------------------------------------"
         << endl;
    for( unsigned i=0; i<Pitanja_.size(); i++ )
    cout << (i+1) << ".  "
         << rezultati[i] << endl;
    cout << "-----------------------------------------"
         << endl
         << "Rezultat: " << zbirPoena << " poena" << endl;
}
```

sada izdvajamo metod `Izvestaj`:

```
void PostaviPitanja()
{
    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ){
        ...
    }
    Izvestaj( zbirPoena, rezultati );
}

void Izvestaj( int zbirPoena,
               const vector<string>& rezultati ) const
{
    cout << endl << "Rezultat testiranja:" << endl
         << "----------------------------------------"
         << endl;
    for( unsigned i=0; i<Pitanja_.size(); i++ )
    cout << (i+1) << ".  "
             << rezultati[i] << endl;
    cout << "----------------------------------------"
         << endl
         << "Rezultat: " << zbirPoena << " poena" << endl;
}
```

Korak 6 – Pamćenje odgovora umesto rezultata

Dobijeni kod je nešto bolji nego što je bio, ali još uvek ima mnogo slabosti. Jedan od problema predstavlja prenošenje podataka o zbiru poena i rezultatima između delova za postavljanje pitanja i ispisivanje izveštaja. Problem nije samo u prenošenju podataka već i u izabranom mestu njihovog pravljenja. Pravo mesto za analizu odgovora nije deo u kome se postavljaju pitanja, već je to pre deo u kome se ispisuje izveštaj. Da bismo mogli da odvojimo ta dva aspekta ponašanja, najpre je neophodno da se pri postavljanju pitanja zapamte svi odgovori.

Za pamćenje odgovora, klasi `Test` dodajemo privatni podatak `Odgovori_`:

```
vector<string>  Odgovori_;
```

i menjamo metod `Test::PostaviPitanja` tako da pamti odgovore:

```
void PostaviPitanja()
{
    Odgovori_.clear();
    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ){
        Pitanje& pitanje = Pitanja_[i];
```

```
            odgovor = PostaviPitanje( i, pitanje );
            Odgovori_.push_back( odgovor );
            string rezultat;
            zbirPoena +=
                pitanje.ProveriOdgovor( odgovor, rezultat );
            rezultati.push_back(rezultat);
            cout << endl;
        }

        Izvestaj( zbirPoena, rezultati );
    }
```

Zatim izbacujemo pomoćnu promenljivu `pitanje`, koja se upotrebljava samo dva puta, i zamenjujemo je odgovarajućim izrazom:

```
void PostaviPitanja()
{
    Odgovori_.clear();
    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        string odgovor = PostaviPitanje( i, Pitanja_[i] );
        Odgovori_.push_back( odgovor );
        string rezultat;
        zbirPoena +=
            Pitanja_[i].ProveriOdgovor( odgovor, rezultat );
        rezultati.push_back( rezultat );
        cout << endl;
    }
    Izvestaj( zbirPoena, rezultati );
}
```

Na sličan način izbacujemo i pomoćnu promenljivu `odgovor`:

```
void PostaviPitanja()
{
    Odgovori_.clear();
    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        Odgovori_.push_back( PostaviPitanje( i, Pitanja_[i] ));
        string rezultat;
        zbirPoena +=
            Pitanja_[i].ProveriOdgovor( Odgovori_[i], rezultat);
        rezultati.push_back( rezultat );
        cout << endl;
    }
    Izvestaj( zbirPoena, rezultati );
}
```

Sada naša petlja `for` u svakoj iteraciji obavlja dva potpuno odvojena posla. Da bismo učinili kod jasnijim, podelićemo petlju na dve petlje. U prvoj petlji samo prikupljamo odgovore, a u drugoj samo obrađujemo rezultate:

```
void PostaviPitanja()
{
    Odgovori_.clear();
    for( unsigned i=0; i<Pitanja_.size(); i++ ){
        Odgovori_.push_back( PostaviPitanje( i, Pitanja_[i] ) );
        cout << endl;
    }

    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        string rezultat;
        zbirPoena +=
            Pitanja_[i].ProveriOdgovor( Odgovori_[i], rezultat);
        rezultati.push_back( rezultat );
    }

    Izvestaj( zbirPoena, rezultati );
}
```

Sada računanje i analizu rezultata možemo da premestimo iz metoda za postavljanje pitanja u metod za ispisivanja izveštaja. Posledica će biti da više nema potrebe da se metodu `Izvestaj` prenose podaci o poenima i rezultatima za pojedinačna pitanja, zato što će se sve računati na osnovu sačuvanih odgovora:

```
void PostaviPitanja()
{
    Odgovori_.clear();
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        Odgovori_.push_back( PostaviPitanje( i, Pitanja_[i] ) );
        cout << endl;
    }
    Izvestaj();
}
void Izvestaj() const
{
    vector<string> rezultati;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        string rezultat;
        zbirPoena +=
            Pitanja_[i].ProveriOdgovor( Odgovori_[i], rezultat);
        rezultati.push_back( rezultat );
    }
    ...
}
```

Nasuprot tome što smo malopre jednu petlju podelili na dve, zato što je radila dva potpuno odvojena posla, sada u metodu `Izveštaj` možemo da integrišemo dve petlje u jednu, zato što se one odnose na istu vrstu posla i na iste podatke. Štaviše, zbog toga prestaje potreba za čuvanjem niza `rezultati`:

```
void Izvestaj() const
{
    cout << endl << "Rezultat testiranja:" << endl
         << "-----------------------------------------"
         << endl;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        string rezultat;
        zbirPoena +=
            Pitanja_[i].ProveriOdgovor( Odgovori_[i], rezultat);
        cout << (i+1) << ".  "
             << rezultat << endl;
    }
    cout << "-----------------------------------------"
         << endl
         << "Rezultat: " << zbirPoena << " poena" << endl;
}
```

Primenili smo *niz malih transformacija* i nakon svake od njih je program mogao da se prevede i izvršava bez ikakvih promena ponašanja. Refaktorisanje se u praksi često preduzima na takav način, pri čemu neke trasformacije mogu da budu primeri iz kataloga refaktorisanja a neke mogu da budu popravke koje su specifične za koknretan slučaj.

Korak 7 – Više manjih popravki

Naš metod se zove `Test::PostaviPitanja`, a u stvari on izvršava (pozivanjem drugog metoda) i izračunavanje broja poena i ispisivanje izveštaja o testiranju. Bilo bi mnogo ispravnije da napravimo novi metod `Test::Testiranje`, koji bi pozivao metode `PostaviPitanja` i `Izvestaj`, a da iz metoda `Test::PostaviPitanja` obrišemo pozivanje metoda `Izvestaj`. Istu promenu bismo mogli da objasnimo i na drugi način: najpre menjamo naziv metoda `PostaviPitanja` u `Testiranje`, a zatim iz njega izdvajamo metod `PostaviPitanja`. Kojim god redom da to radimo, nakon ovih izmena dobijamo:

```
void Testiranje()
{
    PostaviPitanja();
    Izvestaj();
}
```

```
void PostaviPitanja()
{
    Odgovori_.clear();
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        Odgovori_.push_back( PostaviPitanje( i, Pitanja_[i] ) );
        cout << endl;
    }
}
```

Popravljamo glavnu funkciju programa tako da koristi metod `Test::Testiranje`:

```
int main( int argc, char** argv )
{
    ...
    test.Testiranje();
}
```

U nekoliko navrata smo već izdvajali kod iz složenog metoda i pravili nove metode, da bismo tako dobili jednostavnije i jasnije metode. Sada imamo pred sobom upravo suprotan slučaj – metod `Test::PostaviPitanje` ne radi dovoljno toga što bi moglo da opravdava njegovo postojanje. Zbog toga ćemo da obrišemo taj metod i da prebacimo njegovo telo u metod `Test::PostaviPitanja`, na jedino mesto u našem programu na kome se taj metod do sada koristio:

```
void PostaviPitanja()
{
    Odgovori_.clear();
    for( unsigned i=0; i<Pitanja_.size(); i++ ){
        cout << endl
             << "*** Pitanje br. " << (i+1) << " ***"
             << endl << endl;
        Odgovori_.push_back( Pitanja_[i].PostaviPitanje() );
        cout << endl;
    }
}
```

Korak 8 – Još nekoliko manjih izmena

Pri ocenjivanju pojedinačnih odgovora, zasluženi broj poena se navodi dva puta, najpre kao ceo broj a zatim i u okviru tekstualnog opisa rezultata. Tu redundantnost ne bi trebalo da ostavimo u kodu. Zbog toga je potrebno da se izmeni sadržaj poruka tako da one više ne obuhvataju broj poena. Da bi ponašanje softvera ostalo isto, potrebno je da se promeni i način ispisivanja poruka.

Najpre menjamo način izračunavanja rezultata[48]:

```
int ProveriOdgovor( const string& odgovor,
                    string& rezultat ) const
{
    if( odgovor == TacanOdgovor() ){
        rezultat = "Tacan odgovor.";
        return 5;
    }else if( odgovor == "@" ){
        rezultat = "Nije odgovoreno.";
        return -1;
    }else{
        rezultat = "Netacan odgovor.";
        return -3;
    }
}
```

Zatim menjamo i način ispisivanja rezultata u metodu `Test::Izvestaj` tako da najpre ispišemo broj poena (u istom formatu kao ranije) pa tek onda tekstualnu poruku[49]:

```
#include <iomanip>
...

    void Izvestaj() const
    {
        ...
        for( unsigned i=0; i<Pitanja_.size(); i++ ) {
            string rezultat;
            int brojPoena =
                Pitanja_[i].ProveriOdgovor( Odgovori_[i], rezultat);
            zbirPoena += brojPoena;
            cout << (i+1) << ".  ("
                 << setw(2) << brojPoena
                 << " ) " << rezultat << endl;
        }
```

[48] Na ovom mestu bi trebalo razmisliti i o eventualnoj upotrebi jednostavne strukture za prenošenje broja poena i tekstualnog opisa ocene pojedinačnih odgovora.

[49] U primeru smo upotrebili objekat `setw(2)`, koji predstavlja tzv. *manipulator izlaznog toka*. Manipulatori služe za upravljanje radom izlaznog toka. Implementiraju se tako što posebna instanca operatora prosleđivanja na osnovu dobijenog manipulatora menja način rada toka. Manipulator `setw` postavlja minimalnu širinu koju će zauzimati naredni podatak koji se pošalje u tok, što je u konkretnom slučaju ceo broj `brojPoena`. Da bismo mogli da upotrebljavamo manipulatore moramo da uključimo standardnu biblioteku `iomanip`.

```
            cout << "----------------------------------------"
                 << endl
                 << "Rezultat: " << zbirPoena << " poena" << endl;
        }
    ...
```

U uvodu smo naveli da imamo u planu da omogućimo da se izveštaji zapisuju u datotekama. Najjednostavniji način uopštavanja ispisivanja izveštaja je da kao dodatni argument odgovarajućih metoda navedemo izlazni tok. Pri tome možemo da navedemo i da je podrazumevan izlazni tok upravo konzolni izlaz, pa nećemo morati da menjamo glavnu funkciju programa:

```
void Testiranje( ostream& ostr = std::cout )
{
    PostaviPitanja();
    Izvestaj( ostr );
}

void Izvestaj( ostream& ostr ) const
{
    ostr << endl << "Rezultat testiranja:" << endl
         << "----------------------------------------"
         << endl;
    int zbirPoena = 0;
    for( unsigned i=0; i<Pitanja_.size(); i++ ) {
        string rezultat;
        int brojPoena =
            Pitanja_[i].ProveriOdgovor( Odgovori_[i], rezultat);
        zbirPoena += brojPoena;
        ostr << (i+1) << ".  ("
             << setw(2) << brojPoena
             << " ) " << rezultat << endl;
    }
    ostr << "----------------------------------------"
         << endl
         << "Rezultat: " << zbirPoena << " poena" << endl;
}
```

Korak 9 – Nova vrsta pitanja

Sada je vreme da dodamo novu vrstu pitanja. To nije refaktorisanje, već dodavanje novog ponašanja. Pri tome ćemo namerno novi kod da dodamo na prilično loš način, kako bismo posle toga nastavili sa refaktorisanjem. U praksi je mnogo bolje da se stvari rade malo drugačije, tako da se prvo preduzmu potrebna refaktorisanja, pa da se tek onda dodaje novi kod. Na taj način se obezbeđuje da se dodavanjem novog koda ne narušava kvalitet programa.

Nova vrsta pitanja liči na prethodnu – i dalje imamo jedno pitanje i više ponuđenih odgovora. Razlika je u tome što se od korisnika sada ne očekuje da

navede tačno jedan od ponuđenih odgovora, već da ih navede sve, ali u odgovarajućem tačnom redosledu. Potpuno tačan odgovor nosi onoliko poena koliko ima ponuđenih odgovora, ili 5 poena ako je broj ponuđenih odgovora manji od 5. Netačan odgovor i dalje nosi -3 poena, a izostavljen odgovor -1 poen. Novina je da odgovor može biti i delimično tačan – ako su bar dva ponuđena odgovora na tačnim mestima, onda odgovor nosi onoliko poena koliko ima odgovora na tačnim mestima.

Najpre ćemo da u glavnoj funkciji programa dopunimo naš test novim pitanjem:

```
...
test.Dodaj( Pitanje(
    "Poredjati tipove od najmanjeg do najveceg "
        "po broju bajtova.",
    "badc",
    "a - short",
    "b - char",
    "c - long double",
    "d - long"
));
...
```

Zatim menjamo tekst uputstva koje se ispisuje za novu vrstu pitanja u metodu `Pitanje::PostaviPitanje`:

```
string PostaviPitanje() const
{
    cout << TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0; j<PonudjeniOdgovori().size(); j++ )
        cout << PonudjeniOdgovori()[j] << endl;
    cout << "----------------------------------------"
         << endl;
    if( TacanOdgovor().size() == 1 )
        cout << "Upisite slovo koje stoji "
                "ispred tacnog odgovora " << endl
             << "ili znak @ ako ne znate: ";
    else
        cout << "Navedite tacan redosled odgovora "
                "upisivanjem slova" << endl
             << "navedenih ispred odgovora (npr: acbd)"
             << endl
             << "ili znak @ ako ne znate: ";
    string odgovor;
    cin >> odgovor;
    return odgovor;
}
```

Na kraju moramo da promenimo i način ocenjivanja pitanja u metodu `Pitanje::ProveriOdgovor`. Za izračunavanje broja poena u slučaju tačnog odgovora koristimo šablon `max`[50]. Kao i ranije, netačan odgovor nosi -3, a izostavljanje odgovora -1 poen. Međutim, sada moramo da procenimo i vrednost delimično tačnih odgovora:

```
int ProveriOdgovor( const string& odgovor,
                    string& rezultat ) const
{
    if( odgovor == TacanOdgovor() ){
        rezultat = "Tacan odgovor.";
        return max<unsigned>(5,TacanOdgovor().size());
    }else if( odgovor == "@" ){
        rezultat = "Nije odgovoreno.";
        return -1;
    }else if( TacanOdgovor().size() > 1 ){
        int poena = 0;
        for( unsigned i=0;
             i<TacanOdgovor().size() && i<odgovor.size();
             i++
            )
            if( TacanOdgovor()[i] == odgovor[i] )
                poena++;
        if( poena>1 ){
            rezultat = "Delimicno tacno.";
            return poena;
        }
    }
    rezultat = "Netacan odgovor.";
    return -3;
}
```

Primetimo da u slučaju nekih prevodilaca (tj. verzija biblioteka) može da bude potrebno da eksplicitno dodamo biblioteku `algorithm`, u kojoj je definisan šablon `max`:

```
#include <algorithm>
```

Korak 10 – Priprema za uvođenje hijerarhije klasa pitanja

U prethodnom koraku smo dodali novo ponašanje, ali to nije urađeno na dobar način. Metod `Pitanje::ProveriOdgovor` je postao prilično neuredan, zato što se u

[50] Iako šabloni funkcija često mogu da se upotrebljavaju bez eksplicitnog navođenja tipa, ovde moramo da navedemo tip zato što konstanta 5 i metod `size()` imaju različite celobrojne tipove. Alternativa je da konstantu eksplicitno zapišemo kao neoznačenu „`5u`“.

jednoj istoj funkciji poeni računaju na različite načine, a u zavisnosti od vrste pitanja. Takođe, i metod `Pitanje::PostaviPitanje` se ponaša različito za različita pitanja.

Uobičajeno sredstvo za razdvajanje ponašanja koje zavisi od vrste objekta jeste pravljenje hijerarhije klasa. U našem slučaju možemo da prepoznamo da postoje dve različite vrste pitanja i da napravimo odgovarajuće klase za njih. Štaviše, možemo i da se pripremimo za dodavanje novih vrsta pitanja pravljenjem dovoljno uopštene bazne klase hijerarhije.

Ipak, jedan od prvih problema koje moramo da rešimo je priprema našeg koda za prelazak na rad sa hijerarhijom klasa. Rad sa hijerarhijama klasa i dinamičko vezivanje metoda prema konkretnim klasama objekata u programskom jeziku C++ zahtevaju upotrebu objekata putem referenci ili pokazivača. Do sada smo testu dodavali automatske objekte pitanja, a sada ćemo morati da ih pravimo dinamički i da ih dodajemo putem pokazivača. Ta priprema mora da prethodi pravljenju hijerarhije klasa.

Najpre menjamo tip kolekcije pitanja:

```
vector<const Pitanje*>  Pitanja_;
```

Pa metod `Test::Dodaj`:

```
void Dodaj( const Pitanje* p )
{
    Pitanja_.push_back(p);
}
```

Neophodan nam je i destruktor:

```
~Test()
{
    for( unsigned i=0; i<Pitanja_.size(); i++ )
        delete Pitanja_[i];
}
```

Da ne bismo morali da se bavimo kopiranjem testova, naglasićemo da ne želimo da se implementiraju podrazumevani konstruktor kopije i operator dodeljivanja[51].

[51] Alternativno, može da se primeni klasičan pristup (koji je ujedno bio i jedini do verzije C++11) da se metodi deklarišu kao privatni i da se ne implementiraju. Na taj način se sprečava njihova eksplicitna ili implicitna upotreba. Eventualna upotreba ovih metoda van klase `Test` proizvela bi grešku pri prevođenju zbog pokušaja upotrebe privatnog metoda. Sa druge strane, u slučaju njihove eventualne upotrebe u okviru same klase

Punu ispravnu implementaciju ovih metoda ostavljamo za vežbu. Napominjemo da je potrebno da se posveti posebna pažnja ispravnom kopiranju testova i pojedinačnih pitanja.

```
private:
    Test( const Test& ) = delete;
    Test& operator=( const Test& ) = delete;
```

Iako smo samo zabranili podrazumevani konstruktor kopije, zapravo smo na taj način eksplicitno deklarisali jedan konstruktor, što je dovoljno da više ne postoji podrazumevani konstruktor klase `Test`. Zato moramo da eksplicitno napišemo konstruktor bez argumenata:

```
    Test() {}
```

Menjamo i ostale metode klase `Test` u kojima se koriste pitanja:

```
    void PostaviPitanja()
    {
        ...
            Odgovori_.push_back(Pitanja_[i]->PostaviPitanje());
        ...
    }

    void Izvestaj( ostream& ostr ) const
    {
        ...
            int brojPoena = Pitanja_[i]->ProveriOdgovor(
                                        Odgovori_[i], rezultat );
        ...
    }
```

Na kraju, menjamo i način pravljenja i dodavanja pitanja u glavnoj funkciji programa:

```
int main( int argc, char** argv )
{
    Test test;
```

`Test` će biti prijavljena greška pri povezivanju, zato što smo metode samo deklarisali a ne i definisali.

Od standarda *C++11* postoji posebna sintaksa za „izbacivanje“ podrazumevanih ili nasleđenih metoda, kao i sprečavanje nekih podrazumevanih konverzija – dovoljno je da se na kraju deklaracije metoda navede „`= delete`“.

```
    test.Dodaj( new Pitanje(
        ...
    ));
    test.Dodaj( new Pitanje(
        ...
    ));
    test.Dodaj( new Pitanje(
        ...
    ));

    test.Testiranje();
}
```

Korak 11 – Izdvajanje klase AbcdRedosled

U prethodnom koraku smo se pripremili za uvođenje hijerarhije klasa pitanja. Nova vrsta pitanja predstavlja specijalizaciju ranije postojećih pitanja sa više ponuđenih odgovora. Prvo pravimo novu klasu `AbcdRedosled`, koja će da nasledi postojeću klasu `Pitanje`:

```
class AbcdRedosled : public Pitanje
{
public:
    AbcdRedosled( const string& tekstPitanja,
                  const string& tacanOdgovor,
                  const string& odgovor1,
                  const string& odgovor2 = "",
                  const string& odgovor3 = "",
                  const string& odgovor4 = "",
                  const string& odgovor5 = ""
        )
        : Pitanje( tekstPitanja, tacanOdgovor,
                odgovor1, odgovor2, odgovor3, odgovor4, odgovor5 )
        {}
};
```

Menjamo i glavnu funkciju:

```
int main( int argc, char** argv )
{
    ...
    test.Dodaj( new AbcdRedosled(
        "Poredjati tipove od najmanjeg do najveceg "
            "po broju bajtova.",
        "badc",
        "a - short",
        "b - char",
        "c - long double",
        "d - long"
    ));
```

```
    ...
}
```

U ovom trenutku imamo ispravan kod, ali se naša nova klasa još uvek ni po čemu ne razlikuje od stare, a nismo ni popravili kod. Drugim rečima, sve ovo je bila priprema, a tek sada možemo da razdvojimo ponašanje ovih klasa. Cilj je da sve elemente ponašanja klase `Pitanje`, koji su specifični za klasu `AbcdRedosled`, premestimo u izvedenu klasu `AbcdRedosled`. To je tehnika refaktorisanja *Spuštanje ponašanja niz hijerarhiju*.

Najpre u klasi `Pitanje` deklarišemo kao virtualne one metode koji bi trebalo da se ponašaju različito za različite vrste pitanja. Dodajemo i virtualni destruktor:

```
virtual ~Pitanje()
{}

virtual string PostaviPitanje() const
{...}

virtual int ProveriOdgovor( const string& odgovor,
                            string& rezultat ) const
{...}
```

Zatim ih iskopiramo u klasu `AbcdRedosled`[52]:

```
string PostaviPitanje() const override
{...}
int ProveriOdgovor( const string& odgovor,
                    string& rezultat ) const override
{...}
```

Sada iz metoda `Pitanje::PostaviPitanje` i `Pitanje::ProveriOdgovor` izbacimo viškove i ostavimo samo ono što nam je neophodno u klasi `Pitanje` – tj. vraćamo ih u stanje koje je prethodilo dodavanju nove vrste pitanja:

```
virtual string PostaviPitanje() const
{
    cout << TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0; j<PonudjeniOdgovori().size(); j++ )
        cout << PonudjeniOdgovori()[j] << endl;
```

[52] U izvedenoj klasi ne mora da stoji deklaracija `virtual`, mada nije problem ni ako se navede. Umesto toga preporučuje se da se navede deklaracija `override`, kako bi se naglasilo da je to promena ponašanja u odnosu na baznu klasu.

```
    cout << "----------------------------------------"
         << endl
         << "Upisite slovo koje stoji "
            "ispred tacnog odgovora " << endl
         << "ili znak @ ako ne znate: ";
    string odgovor;
    cin >> odgovor;
    return odgovor;
}

virtual int ProveriOdgovor( const string& odgovor,
                            string& rezultat ) const
{
    if( odgovor == TacanOdgovor() ){
        rezultat = "Tacan odgovor.";
        return 5;
    }else if( odgovor == "@" ){
        rezultat = "Nije odgovoreno.";
        return -1;
    }else{
        rezultat = "Netacan odgovor.";
        return -3;
    }
}
```

Slično uradimo i u klasi `AbcdRedosled` – u njoj nema razloga da stoji ništa osim opisa ponašanja koje je specifično za novu vrstu pitanja:

```
string PostaviPitanje() const override
{
    cout << TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0; j<PonudjeniOdgovori().size(); j++ )
        cout << PonudjeniOdgovori()[j] << endl;
    cout << "----------------------------------------"
         << endl
         << "Navedite tacan redosled odgovora "
            "upisivanjem slova" << endl
         << "navedenih ispred odgovora (npr: acbd)"
         << endl
         << "ili znak @ ako ne znate: ";
    string odgovor;
    cin >> odgovor;
    return odgovor;
}

int ProveriOdgovor( const string& odgovor,
                            string& rezultat ) const override
{
```

```
        if( odgovor == TacanOdgovor() ){
            rezultat = "Tacan odgovor.";
            return max<unsigned>(5,TacanOdgovor().size());
        }else if( odgovor == "@" ){
            rezultat = "Nije odgovoreno.";
            return -1;
        }else{
            int poena = 0;
            for( unsigned i=0;
                 i<TacanOdgovor().size() && i<odgovor.size();
                 i++
                )
                if( TacanOdgovor()[i] == odgovor[i] )
                    poena++;
            if( poena>1 ){
                rezultat = "Delimicno tacno.";
                return poena;
            }else{
                rezultat = "Netacan odgovor.";
                return -3;
            }
        }
    }
```

Korak 12 – Izdvajanje uopštene bazne klase Pitanje

Klasa `Pitanje` je napravljena prema prvobitno jedinoj vrsti pitanja. Očigledno je da ona sada predstavlja vrlo specifičan vid pitanja, a ne uopšteno pitanje, kako bi naziv klase mogao da sugeriše. Na primer, ako bismo dodavali novu vrstu pitanja na koja se očekuje upisivanje tačnog odgovora (na primer, na pitanje „Izračunaj 2+3“ mora da se upiše tačan odgovor „5“), onda nam za to ne bi bili potrebni ponuđeni odgovori. Zbog toga ćemo sada da uvedemo novu baznu klasu `Pitanje`.

Promenu ćemo da napravimo u dva manja koraka: najpre ćemo da postojećoj klasi `Pitanje` promenimo naziv u `AbcdPitalica`, a zatim i da izdvojimo uopštenu osnovu ove klase u novu klasu `Pitanje`. Pri tome se klasa `Test` praktično ne menja, ali pri pravljenju testa u glavnoj funkciji sada pitanja moraju da se prave kao objekti klase `AbcdPitalica`.

Promena imena klase se izvodi jednostavno – menjamo ime u definiciji klase, u definiciji izvedene klase `AbcdRedosled` i u glavnoj funkciji programa:

```
class AbcdPitalica
{
public:
    AbcdPitalica( ... )
    ...
};
```

```
class AbcdRedosled : public AbcdPitalica
{
public:
    AbcdRedosled( ... )
        : AbcdPitalica( tekstPitanja, tacanOdgovor,
                odgovor1, odgovor2, odgovor3, odgovor4, odgovor5 )
        {}
    ...
};
...

int main( int argc, char** argv )
{
    ...
    test.Dodaj( new AbcdPitalica( ... ));
    test.Dodaj( new AbcdPitalica( ... ));
    test.Dodaj( new AbcdPitalica( ... ));
    ...
}
```

Zatim pravimo novu klasu `Pitanje`. Moramo odmah da joj dodamo deklaraciju interfejsa, da bi klasa `Test` mogla da je koristi. Sve ostalo ćemo dodavati kasnije, pa ova klasa na početku može da izgleda ovako:

```
class Pitanje
{
public:
    virtual ~Pitanje() {}
    virtual string PostaviPitanje() const = 0;
    virtual int ProveriOdgovor( const string& odgovor,
                                string& rezultat ) const = 0;
};
```

Potrebno je da izmenimo i klasu `AbcdPitalica` tako da nasleđuje klasu `Pitanje`:

```
class AbcdPitalica : public Pitanje
...
```

Korak 13 – Uređivanje odgovornosti klasa hijerarhije

Nakon što smo završili inicijalno pravljenje novih klasa, sada možemo da pristupimo postepenom uređivanju njihovih odgovornosti. Potrebno je da uočimo šta je odgovornost koje klase u hijerarhiji i da odgovarajuće ponašanje podignemo ili spustimo kroz hijerarhiju.

Svako pitanje bi trebalo da ima tekst pitanja i tačan odgovor, pa je odgovarajuće elemente potrebno da premestimo iz klase `AbcdPitalica` u klasu `Pitanje`. Preduzimamo refaktorisanje *Podizanje ponašanja uz hijerarhiju*. Dodajemo i

odgovarajući konstruktor i u skladu sa time menjamo konstruktor klase `AbcdPitalica`:

```
class Pitanje
{
public:
    Pitanje( const string& tekstPitanja, const string& tacanOdgovor)
        : TekstPitanja_(tekstPitanja),
          TacanOdgovor_(tacanOdgovor)
        {}

    virtual ~Pitanje() {}
    virtual string PostaviPitanje() const = 0;
    virtual int ProveriOdgovor( const string& odgovor,
                                string& rezultat ) const = 0;

    const string& TekstPitanja() const
        { return TekstPitanja_; }
    const string& TacanOdgovor() const
        { return TacanOdgovor_; }

private:
    string TekstPitanja_;
    string TacanOdgovor_;
};

class AbcdPitalica : public Pitanje
{
public:
    AbcdRedosled( const string& tekstPitanja,
                  const string& tacanOdgovor,
                  const string& odgovor1,
                  const string& odgovor2 = "",
                  const string& odgovor3 = "",
                  const string& odgovor4 = "",
                  const string& odgovor5 = ""
        )
        : Pitanje( tekstPitanja, tacanOdgovor )
    ...
};
```

Postavljanje pitanja je veoma slično u obe implementirane konkretne klase. Možemo da uočimo da bi postavljanje pitanja trebalo da podelimo na tri dela: ispisivanje pitanja, ispisivanje uputstva i čitanje odgovora. Za sada vidimo da može biti razlike u ispisivanju pitanja, već imamo razliku u uputstvu, ali ne vidimo da je potrebna razlika pri čitanju odgovora. U skladu sa time, u klasi `AbcdPitalica` izdvajamo metode `IspisiPitanje` i `Uputstvo`:

```
string PostaviPitanje() const
{
    IspisiPitanje();
    cout << Uputstvo();
    string odgovor;
    cin >> odgovor;
    return odgovor;
}

virtual void IspisiPitanje() const
{
    cout << TekstPitanja() << endl
         << "----------------------------------------"
         << endl;
    for( unsigned j=0; j<PonudjeniOdgovori().size(); j++ )
        cout << PonudjeniOdgovori()[j] << endl;
    cout << "----------------------------------------"
         << endl;
}

virtual const char* Uputstvo() const
{
    return  "Upisite slovo koje stoji "
            "ispred tacnog odgovora\n "
            "ili znak @ ako ne znate: ";
}
```

Ako u klasi `AbcdRedosled` implementiramo odgovarajuću verziju metoda `Uputstvo`, onda iz nje možemo da obrišemo posebnu verziju metoda `PostaviPitanje`:

```
const char* Uputstvo() const override
{
    return  "Navedite tacan redosled odgovora "
            "upisivanjem slova\n"
            "koja stoje ispred odgovora (npr: acbd)\n"
            "ili znak @ ako ne znate: ";
}
```

Napisana implementacija metoda `PostaviPitanje` je sada dovoljno uopštena da može da bude u baznoj klasi hijerarhije, pa ovaj metod podižemo uz hijerarhiju. Štaviše, nakon premeštanja on više ne mora da bude virtualan. Umesto toga, u klasi `Pitanje` deklarišemo nove virtualne metode `IspisiPitanje` i `Uputstvo`:

```
class Pitanje
{
public:
    ...
```

```
    string PostaviPitanje() const
    {
        IspisiPitanje();
        cout << Uputstvo();
        string odgovor;
        cin >> odgovor;
        return odgovor;
    }

    virtual void IspisiPitanje() const = 0;
    virtual const char* Uputstvo() const = 0;
    ...
};
```

Sledeći korak uopštavanja može da bude premeštanje najopštijeg dela ponašanja iz metoda `AbcdPitalica::IspisiPitanje` u `Pitanje::IspisiPitanje`:

```
class Pitanje
{
public:
    ...
    virtual void IspisiPitanje() const
    {
        cout << TekstPitanja() << endl
             << "-----------------------------------------"
             << endl;
    }
    ...
};

class AbcdPitalica : public Pitanje
{
public:
    ...
    void IspisiPitanje() const override
    {
        Pitanje::IspisiPitanje();
        for( unsigned j=0; j<PonudjeniOdgovori().size(); j++ )
            cout << PonudjeniOdgovori()[j] << endl;
        cout << "-----------------------------------------"
             << endl;
    }
    ...
};
```

Rezultat

Sledeći korak bi možda moglo da bude brisanje nekih suvišnih metoda: tekstu pitanja sada pristupamo samo iz klase `Pitanje`, pa možemo da obrišemo odgovarajući pristupni metod i njegovu upotrebu zamenimo eksplicitnim pristupanjem.

Slično je i sa nizom ponuđenih odgovora, koji se sada koristi samo u klasi u kojoj je i definisan.

Zatim bi moglo da se pristupi daljem strukturiranju metoda `ProveriOdgovor`. Na primer, metod bi mogao da se uopšti uvođenjem novih virtualnih metoda:

```
int ProveriOdgovor( const string& odgovor,
                    string& rezultat ) const
{
    if( OdgovorJeTacan(odgovor) )
        return OceniTacanOdgovor(rezultat);
    else if( odgovor == "@" ){
        rezultat = "Nije odgovoreno.";
        return -1;
    else
        return OceniNetacanOdgovor( odgovor, rezultat );
}
```

U nekim daljim koracima bismo mogli da razdvojimo apstrakciju pitanja od apstrakcije postavljanja pitanja, tako da se kroz testiranje ne menja test nego neki drugi objekat koji opisuje rezultate testiranja.

Ipak, ovde ćemo da zastanemo. Uvek možemo da idemo dalje i da napravimo bolji kod, ali nekad mora i da se stane. Predstavljeno je dovoljno refaktorisanja da čitalac može da stekne utisak o suštini procesa, a struktura programskog koda je pritom dovoljno unapređena da može da se razmišlja o daljem dodavanju ponašanja. Preduzimanje predloženih refaktorisanja, kao i eventualno dalje unapređivanje programa za testiranje, ostavljamo za vežbu.

Nakon svih napravljenih izmena program izgleda ovako:

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

//-------------------------------------------------------------
class Pitanje
{
public:
    Pitanje( const string& tekstPitanja, const string& tacanOdgovor)
        : TekstPitanja_( tekstPitanja ),
          TacanOdgovor_( tacanOdgovor )
        {}

    virtual ~Pitanje() {}
```

```
    string PostaviPitanje() const
    {
        IspisiPitanje();
        cout << Uputstvo();
        string odgovor;
        cin >> odgovor;
        return odgovor;
    }

    virtual void IspisiPitanje() const
    {
        cout << TekstPitanja() << endl
             << "----------------------------------------"
             << endl;
    }

    virtual const char* Uputstvo() const = 0;
    virtual int ProveriOdgovor( const string& odgovor,
                                string& rezultat ) const = 0;

    const string& TekstPitanja() const
        { return TekstPitanja_; }
    const string& TacanOdgovor() const
        { return TacanOdgovor_; }

private:
    string TekstPitanja_;
    string TacanOdgovor_;
};

//---------------------------------------------------------------
class AbcdPitalica : public Pitanje
{
public:
    AbcdPitalica( const string& tekstPitanja,
                  const string& tacanOdgovor,
                  const string& odgovor1,
                  const string& odgovor2 = "",
                  const string& odgovor3 = "",
                  const string& odgovor4 = "",
                  const string& odgovor5 = ""
        )
        : Pitanje( tekstPitanja, tacanOdgovor )
    {
        PonudjeniOdgovori_.push_back( odgovor1 );
        if( odgovor2 != "" )
            PonudjeniOdgovori_.push_back( odgovor2 );
        if( odgovor3 != "" )
            PonudjeniOdgovori_.push_back( odgovor3 );
        if( odgovor4 != "" )
            PonudjeniOdgovori_.push_back( odgovor4 );
```

```
        if( odgovor5 != "" )
            PonudjeniOdgovori_.push_back( odgovor5 );
    }

    const vector<string>& PonudjeniOdgovori() const
        { return PonudjeniOdgovori_; }

    void IspisiPitanje() const override
    {
        Pitanje::IspisiPitanje();
        for( unsigned j=0; j<PonudjeniOdgovori().size(); j++ )
            cout << PonudjeniOdgovori()[j] << endl;
        cout << "----------------------------------------"
             << endl;
    }

    const char* Uputstvo() const override
    {
        return "Upisite slovo koje stoji "
               "ispred tacnog odgovora \n"
               "ili znak @ ako ne znate: ";
    }

    virtual int ProveriOdgovor( const string& odgovor,
                        string& rezultat ) const
    {
        if( odgovor == TacanOdgovor() ) {
            rezultat = "Tacan odgovor.";
            return 5;
        } else if( odgovor == "@" ) {
            rezultat = "Nije odgovoreno.";
            return 1;
        } else {
            rezultat = "Netacan odgovor.";
            return -3;
        }
    }

private:
    vector<string> PonudjeniOdgovori_;
};

//---------------------------------------------------------------
class AbcdRedosled : public AbcdPitalica
{
public:
    AbcdRedosled( const string& tekstPitanja,
                  const string& tacanOdgovor,
                  const string& odgovor1,
                  const string& odgovor2 = "",
                  const string& odgovor3 = "",
```

```
                    const string& odgovor4 = "",
                    const string& odgovor5 = ""
        )
        : AbcdPitalica( tekstPitanja, tacanOdgovor,
                odgovor1, odgovor2, odgovor3, odgovor4, odgovor5 )
        {}

    const char* Uputstvo() const override
    {
        return  "Navedite tacan redosled odgovora "
                "upisivanjem slova\n"
                "koja stoje ispred odgovora (npr: acbd)\n"
                "ili znak @ ako ne znate: ";
    }

    int ProveriOdgovor( const string& odgovor,
                        string& rezultat ) const override
    {
        if( odgovor == TacanOdgovor() ) {
            rezultat = "Tacan odgovor.";
            return max<unsigned>( 5, TacanOdgovor().size() );
        } else if( odgovor == "@" ) {
            rezultat = "Nije odgovoreno.";
            return 1;
        } else {
            int poena = 0;
            for( unsigned i=0;
                 i<TacanOdgovor().size() && i<odgovor.size();
                 i++
                )
                if( TacanOdgovor()[i] == odgovor[i] )
                    poena++;
            if( poena>1 ){
                rezultat = "Delimicno tacno.";
                return poena;
            } else {
                rezultat = "Netacan odgovor.";
                return -3;
            }
        }
    }
};

//-----------------------------------------------------------------
class Test
{
public:
    Test()
    {}
```

```
    ~Test()
    {
        for( unsigned i=0; i<Pitanja_.size(); i++ )
            delete Pitanja_[i];
    }

    void Dodaj( const Pitanje* p )
        { Pitanja_.push_back(p); }

    void Testiranje( ostream& ostr = cout )
    {
        PostaviPitanja();
        Izvestaj( ostr );
    }

    void PostaviPitanja()
    {
        Odgovori_.clear();
        for( unsigned i=0; i<Pitanja_.size(); i++ ) {
            cout << endl
                 << "*** Pitanje br. " << (i+1) << " ***"
                 << endl << endl;
            Odgovori_.push_back( Pitanja_[i]->PostaviPitanje() );
            cout << endl;
        }
    }

    void Izvestaj( ostream& ostr ) const
    {
        ostr << endl << "Rezultat testiranja:" << endl
             << "-----------------------------------------"
             << endl;
        int zbirPoena = 0;
        for( unsigned i=0; i<Pitanja_.size(); i++ ) {
            string rezultat;
            int brojPoena = Pitanja_[i]->ProveriOdgovor(
                                        Odgovori_[i], rezultat );
            zbirPoena += brojPoena;
            ostr << (i+1) << ".  ("
                 << setw(2) << brojPoena
                 << " ) " << rezultat << endl;
        }
        ostr << "-----------------------------------------"
             << endl
             << "Rezultat: " << zbirPoena << " poena" << endl;
    }

private:
    Test( const Test& ) = delete;
    Test& operator=( const Test& ) = delete;
```

```
    vector<const Pitanje*> Pitanja_;
    vector<string> Odgovori_;
};

//---------------------------------------------------------------
int main( int argc, char** argv )
{
    Test test;
    test.Dodaj( new AbcdPitalica(
        "Sta je to 'std::map'?",
        "b",
        "a - Putokaz do gusarskog blaga.",
        "b - Asocijativni niz u standardnoj biblioteci p.j. C++.",
        "c - Mocvarni teren opasan po zivot."
    ));
    test.Dodaj( new AbcdPitalica(
        "Koliko tacnih odgovora ima ovo pitanje?",
        "c",
        "a - Tri.",
        "b - Dva.",
        "c - Jedan."
    ));
    test.Dodaj( new AbcdRedosled(
        "Poredjati tipove od najmanjeg do najveceg "
            "po broju bajtova.",
        "badc",
        "a - short",
        "b - char",
        "c - long double",
        "d - long"
    ));

    test.Testiranje();
}
```

Već na prvi pogled možemo da vidimo da je programski kod sada značajno veći nego što je bio, ali i da je bolje oblikovan. Često ćemo u praksi imati slučajeve da refaktorisanje povećava obim koda, ali to nikako ne bi smelo da predstavlja prepreku refaktorisanju. Naprotiv, ako je kod duži, ali je čitljiviji, jasniji i bolje strukturiran, onda će se on mnogo lakše održavati, pa dodatna veličina ne predstavlja nikakvu smetnju.

10.5 Problemi pri refaktorisanju

Refaktorisanje počiva na nizu malih transformacija. Zbog toga možemo da kažemo da je jedno složeno refaktorisanje teško koliko i najkomplikovanija pojedinačna transformacija. Zaista, ako su sve transformacije umereno složene i obimne, onda postupak refaktorisanja nije posebno težak, iako ponekad može da zahteva malo više truda. Sa druge strane, ako je neka od elementarnih transformacija

veoma složena ili se odnosi na relativno veliki deo programskog koda, onda to može da predstavlja problem.

U nekim slučajevima refaktorisanje ne može da se svede na jednostavne elementarne transformacije programskog koda. Tada je potrebno da se dobro razmisli da li je uopšte *neophodno* refaktorisati, kao i da se proveri da li postoji *neko drugo rešenje*.

Kada ne treba refaktorisati

Dobar primer teškog refaktorisanja su izmene koje obuhvataju promene strukture baze podataka. U zavisnosti od arhitekture softvera, sasvim je moguće da promena strukture baze podataka ima veoma široke posledice po programski kod[53].

Lokalne promene, koje se odnose na implementaciju neke klase (ili skupa klasa), često nemaju nikakav uticaj na druge programske celine (klase i komponente). Sasvim je drugačije u slučajevima kada je radi popravljanja strukture koda neophodno da se promeni interfejs neke klase. Ako se taj interfejs upotrebljava samo u uskom delu koda, onda ni to nije veliki problem – samo je potrebno da se pažljivo izmene sva mesta na kojima se on koristi. Međutim, ako se radi o interfejsu koji se intenzivno upotrebljava, onda njegovo menjanje može da bude veoma neugodno. Pre početka transformisanja koda potrebno je da se najpre sagleda obim potrebnih izmena, pa tek onda sme da se započne menjanje. U takvim slučajevima može da se primeni refaktorisanje *Zamenjivanje interfejsa*. Ako se stari interfejs koristi na previše mesta da bi zamenjivanje imalo smisla, onda može da bude opravdano da se (makar samo privremeno) zadrže oba interfejsa. Zbog složenosti zamenjivanja interfejsa, preporučuje se da se javni interfejs neke klase (ili komponente) ne stavlja u upotrebu sve dok ne postane dovoljno stabilan.

Iako to može da izgleda čudno, nekada je programski kod koji želimo da refaktorišemo toliko složen da pojedina refaktorisanja mogu samo da nanesu štetu, tj. da pokvare strukturu programa umesto da ga poprave. Na primer, ako refaktorisanje dodaje novu apstrakciju kodu, a u datom složenom kodu već postoji veći broj apstrakcija, tada nova apstrakcija može da doprinese preteranoj složenosti

[53] U slučaju baza podataka poseban problem predstavlja migracija sadržaja baze iz prethodne verzije u novu verziju baze sa novom strukturom podataka. Taj postupak može da bude veoma složen, a uvek je veoma osetljiv. Eventualne greške u procesu migracije mogu da budu veoma skupe, kako zbog potencijalno gubljenja podataka, tako i zbog veoma teškog kasnijeg ispravljanja grešaka. Posebno je neugodno ako se greške uoče sa zakašnjenjem, kada neispravno migrirani podaci već uđu u redovnu upotrebu. To je jedan od razloga što mnoge agilne metodologije odlažu pravljenje baze podataka sve do trenutka kada je ona neophodna, jer je migracija skladišta podataka zasnovanog na datotekama često daleko jednostavnija nego što je to slučaj sa bazama podataka.

klase, tako da dobijeni kod ima više slabosti nego prethodni. U takvom slučaju složenog dizajna koda, refaktorisanje obično može da se primeni, ali je potrebno da se razmotre različite tehnike da bi se dobio što bolji rezultat.

U nekim slučajevima nije dobro refaktorisati. Na primer, ako imamo veliki broj različitih slabosti u kodu, koje su međusobno isprepletane. Tada refaktorisanja koja popravljaju jedne slabosti mogu da naprave još veće probleme pogoršavajući neke druge slabosti. U takvim slučajevima može da se dogodi da neka refaktorisanja uopšte ne smeju da se primene, zato što mogu da promene postojeće ponašanje. Kao što je već istaknuto na samom početku ovog poglavlja, refaktorisanje nije dopušteno ni ako postojeći kod ne radi. Nije dobro započinjati veća refaktorisanja ni ako su suviše blizu krajnji rokovi za završetak razvojnog ciklusa. U tom periodu obično ima prečih poslova, a refaktorisanje nije dobro raditi u žurbi.

Refaktorisanje nije lek za sve slabosti u programskom kodu. U nekim slučajevima kod može da bude toliko loše strukturiran da je mnogo jednostavnije i bolje da ga napišemo od početka nego da ga popravljamo refaktorisanjem. Da ne bi došlo do toga, refaktorisanje mora da se primenjuje redovno.

Refaktorisanje i performanse izvršavanja

Pri pisanju i održavanju programa često razmišljamo o različitim vidovima optimizacija, najčešće u odnosu na trajanje izvršavanja ili zauzeće nekih resursa. Tu se neizbežno postavlja pitanje uticaja refaktorisanja na performanse. Dobro strukturiran programski kod je pogodan za održavanje, ali podizanje nivoa strukture koda često ima za posledicu spuštanje nivoa performansi, posebno ako se poredi sa programskim kodom koji je namenski optimizovan za efikasno izvršavanje.

U praksi se pokazuje da obično sasvim mali delovi koda imaju presudan uticaj na performanse. Na primer, ako bismo pisali program za obradu slika, svi oni elementi u programu koji se odnose na interakciju sa korisnikom, prikupljanje parametara i izdavanje rezultata mogu biti mnogo obimniji nego deo koda koji zaista obrađuje sliku, a da pri tome nemaju praktično nikakav uticaj na performanse. Sa druge strane, i ti delovi koda podležu redovnom održavanju i popravljanju, možda čak u većem obimu nego delovi koji obrađuju sliku.

Često se ne uzima dovoljno u obzir da je za najveći deo koda mnogo važnije koliko će biti lak za održavanje, nego koliko će biti efikasan pri izvršavanju. Za najveći deo programskog koda troškovi razvoja su važniji od troškova eksploatacije. Pri tome je negativan uticaj refaktorisanja na performanse obično daleko niži od ostvarenog pozitivnog uticaja na dizajn. Zbog toga refaktorisanje ne bi trebalo da se izbegava „radi očuvanja performansi“.

Štaviše, kao što je već naglašeno pri predstavljanju motivacije, refaktorisanje predstavlja vid optimizacije koda u odnosu na performanse održavanja. Preporučljivo je da se što manji deo programskog koda optimizuje u odnosu na

performanse izvršavanja, a da se sav ostali kod optimizuje u odnosu na performanse održavanja. Dobrim lokalizovanjem (tj. značajnim sužavanjem okvira) programskog koda koji ima presudan uticaj na efikasnost, možemo da obezbedimo da za najveći deo koda prioritet budu dobra strukturiranost i lakoća održavanja, a da samo za manji deo koda kao prioritet odredimo efikasnost izvršavanja. Takav pristup omogućava uspostavljanje uravnoteženog odnosa između lakoće održavanja i dobrih performansi izvršavanja.

10.6 Umesto zaključka

Refaktorisanje je jedna od najvažnijih tehnika agilnih metodologija razvoja softvera. Agilni razvoj podrazumeva uzdržano planiranje i veoma česte izmene u zahtevima, pa zatim i u projektu i implementaciji programa, a učestale izmene programskog koda neminovno imaju za posledicu postepeno urušavanje kvaliteta dizajna. Refaktorisanje nam služi da ne dopustimo da se dizajn naruši preko neke razumne mere. Umesto toga, redovnim refaktorisanjem održavamo dobru strukturu našeg programa i omogućavamo i lakši i sigurniji razvoj i održavanje programa.

Ovo poglavlju predstavlja samo uvod u refaktorisanje. Za detaljnije izučavanje tehnika refaktorisanja preporučuje se knjiga *Refaktorisanje: Poboljšanje dizajna postojećeg koda* [*Fowler 1999*]. Knjiga je propraćena i veb-lokacijom autora *Refactoring.Com*, na kojoj se mogu pronaći katalog refaktorisanja i dodatni izvori informacija. Pored toga, refaktorisanje je zastupljeno na još mnogo drugih lokacija na vebu, kao što su *Source Making* ili *Refactoring Guru*.

11 - Polimorfizam

Ja sam izmislio termin "objektno-orijentisano",
i mogu da kažem da nisam imao na umu C++.
Alan Kej

11.1 Pojam i vrste polimorfizma

Za neki *izraz* ili *vrednost* u programu kažemo da su *polimorfni* ako mogu da imaju za domen više različitih tipova. U skladu sa time, za neki *programski kod* kažemo da je *polimorfan* ako sadrži polimorfne vrednosti ili izraze.

Polimorfizam ima veliki značaj za programiranje. Ako programiranje posmatramo kao postupak formalnog modeliranja nekih procesa, izraza ili odnosa u posmatranom domenu, onda polimorfizam predstavlja jedno od osnovnih tehničkih sredstava za apstrahovanje predmeta modeliranja. Primenom polimorfizma se dobija programski kod koji može da se upotrebi u različitim kontekstima, za različite tipove podataka, pa čak i za veoma različite postupke. Zbog takve uloge, polimorfizam u svakodnevnoj programerskoj praksi predstavlja veoma korisno sredstvo za ostvarivanje višestruke upotrebljivosti koda.

Polimorfizam, njegove vrste, elementi i načini ostvarivanja predstavljaju u osnovi temu iz oblasti paradigmi programiranja. Ipak, imajući u vidu veliki značaj i uticaj koji primena polimorfizma ima na savremeni razvoj softvera, procenili smo da je važno da se u ovoj knjizi polimorfizmu posveti zaslužena pažnja.

Razmatranje karakteristika polimorfizma ćemo da započnemo posmatranjem vrlo jednostavnog segmenta koda, koji za dati objekat `lik` izračunava odnos između površine i obima:

```
odnos = lik.Povrsina() / lik.Obim();
```

Osnovne karakteristike različitih vidova polimorfizma ćemo da razmotrimo kroz:

- odnos između univerzalnog i promenljivog dela;
- način i trenutak proveravanja saglasnosti tipova;
- statičko i dinamičko vezivanje i
- vrste polimorfizma, tj. tehnički aspekt ostvarivanja polimorfizma u programskim jezicima.

Univerzalan i promenljiv deo

Svaki polimorfan deo programa (isečak koda, funkcija, metod, klasa i drugo) ima jedan *univerzalan deo*, koji predstavlja osnovu višestruke upotrebljivosti. To je deo koji se uvek odvija na isti način, bez obzira na specifične okolnosti u kojima se upotrebljava. U prethodnom primeru, možemo da primetimo da se odnos površine i obima uvek izračunava na isti način, bez obzira na tip objekta `lik`, kao i da se rezultat uvek dodeljuje promenljivoj `odnos`. Znači, univerzalan deo u ovom primeru koda čine način izračunavanja i čuvanja odnosa.

Ako hoćemo da budemo do kraja precizni, onda moramo da primetimo da tu ima još nekoliko faktora koji mogu da utiču na način izvršavanja ovog segmenta koda. Na primer, u zavisnosti od tipa rezultata metoda `Povrsina` i `Obim`, ovde može da bude reč o celobrojnom deljenju ili o deljenju sa pokretnom zapetom. Takođe, od tipa promenljive `odnos` zavisi da li će rezultat deljenja biti konvertovan pre dodeljivanja, kao i način na koji će se odvijari operacija dodeljivanja. Radi jednostavnosti, za sada pretpostavimo da se u svim ovim slučajevima radi o tipu `float`, tj. da se navedene operacije uvek prevode i odvijaju na isti način.

Deo koji nije u potpunosti „pod kontrolom“ u ovom primeru jeste način izračunavanja površine i obima. To je *potencijalno promenljiv deo*, koji predstavlja osnovu razlikovanja različitih slučajeva upotrebe našeg višestruko upotrebljivog isečka koda. Kažemo da je „potencijalno“ promenljiv zato što neki elementi tog dela koda mogu načelno da budu isti za veliki broj raziičitih tipova, ali je suština u tome da oni *mogu da se razlikuju* za različite tipove. Na primer, obim kvadrata i pravougaonika možemo da računamo na isti način, sabirajući dužine svih stranica formulom „`2a+2b`“, ali za kvadrat možemo da ga računamo i na drugačiji način korišćenjem specifične formule „`4a`“. Sa druge strane, obim kruga ćemo svakako računati drugačije nego obim pravougaonika.

Potencijalno promenljiv deo je neophodan da bismo neki programski kod mogli da smatramo polimorfnim. Kada ne bi postojao potencijalno promenljiv deo, onda se ne bi radilo o polimorfnom delu programa, već o programskom kodu čiji su tip i ponašanje uvek do kraja precizno određeni.

Univerzalan deo je takođe neophodan, ali u nekim slučajevima on može da ostane na nivou apstrakcije i algoritma. Na primer, ako bi u prethodnom primeru

tipovi metoda `Povrsina` i `Obim` i tip promenljive `odnos` mogli da budu različiti u različitim okolnostima, onda bi i operacije deljenja, konvertovanja i dodeljivanja bile potencijalno različite, tako da nijedna operacija u posmatranom izrazu ne bi bila doslovno univerzalna. Tada bi jedino algoritam zadržao univerzalnost, ali bi to i dalje bio polimorfan isečak koda.

Način i trenutak proveravanja saglasnosti tipova

Proveravanje saglasnosti tipova je značajan korak kojim se sprečavaju greške u programima. Kada se ne bi proveravala saglasnost tipova, onda bi moglo bi da se dogodi da na nekom objektu pokušamo da izvršimo nepostojeći metod. Način i trenutak izvođenja provere tipova zavise od konkretnog programskog jezika, pa čak i od implementacije.

Kod *slabo tipiziranih programskih jezika*, provera tipova, ili bar njen veći deo, se ostavlja za fazu izvršavanja programa. Programi se pokrenu i onda se u toku rada proverava da li svaka pojedinačna vrednost sa kojom se radi ima onaj tip koji je neophodan da bi izvršavanje moglo da se nastavi. Obično se radi na isti način, bez obzira na to da li se radi o polimorfnom ili nepolimorfnom delu programa.

Sa *strogo tipiziranim programskim jezicima* stvari stoje drugačije. Kod njih sve provere tipova moraju da se izvrše unapred, u fazi prevođenja (ako se radi o klasičnim prevodiocima) ili u fazi pripreme za izvršavanje (ako se radi o interpretatorima). U trenutku izvršavanja programa i svakog njegovog dela već mora da bude pouzdano utvrđeno da su svi tipovi operanada saglasni sa tipovima odgovarajućih operacija.

Da ne bude nesporazuma, to ne znači da će pri započinjanju izvršavanj programa svaka konkretna vrednost i svaki konkretan izraz imati tačno ustanovljen jedan konkretan tip. Provera saglasnosti tipova nije uvek nadležna da utvrdi *tačan tip* koji će na nekom mestu da se upotrebljava, već da pouzdano utvrdi da će sve operacije biti izvršavane isključivo nad objektima čiji su tipovi *saglasni* sa tipom operacije. U zavisnosti od konteksta u kome se posmatrani deo koda koristi, kao i u zavisnosti od upotrebljenog programskog jezika, isti polimorfni izrazi u različitim slučajevima mogu da imaju različite konkretne tipove.

Da bismo ilustrovali složenost problema proveravanja saglasnosti tipova, razmotrićemo kao primer sledeći isečak programskog koda:

```
a = b + c;
```

Kog tipa su promenljive `a`, `b` i `c`? Kog tipa je izraz `b+c`? Kog tipa su operacije sabiranja i dodeljivanja? Ako posmatramo samo navedeni isečak koda, onda ne možemo da procenimo navedene tipove sasvim precizno. Mogli bismo da zaključimo da `a`, `b` i `c` moraju da budu istog tipa, kao i da za taj tip postoje definisani

operatori sabiranja i dodeljivanja, ali takvo zaključivanje ne bi bilo dovoljno uopšteno.

Opštije zaključivanje bi trebalo da počiva na izvođenju i analiziranju skupa uslova koji moraju da budu ispunjeni. U našem primeru taj skup bi trebalo da obuhvati naredne uslove:

- postoji neki binarni operator „+“, koji za prvi argument tipa *T1* i drugi argument tipa *T2* izračunava rezultat tipa *T3*;
- ime `b` je tipa *Tb*, za koji važi da je podtip od *T1*[54] ili da postoji implicitna konverzija iz tipa *Tb* u *T1*;
- ime `c` je tipa *Tc*, za koji važi da je podtip od *T2* ili postoji implicitna konverzija iz tipa *Tc* u *T2*;
- postoji operator dodeljivanja koji promenljivoj tipa *TV* dodeljuje vrednost tipa *TE*;
- *T3* je podtip od *TE* ili postoji implicitna konverzija iz tipa *T3* u *TE*;
- ime `a` je tipa *Ta*, koji mora da bude isti[55] kao *TV*.

Iz navedenog skupa uslova (čak i bez zalaženja u njihovu analizu i izvođenje nekih složenijih zaključaka o tipovima) možemo da vidimo da čak i za sasvim jednostavne izraze proveravanje saglasnosti tipova predstavlja relativno složen postupak, u kome figuriše mnogo uslova koji moraju da se usklade. Skup uslova bi bio još mnogo složeniji kada bismo formalno definisali pojam *podtipa* i uveli u razmatranje i sva odgovarajuća pravila koja bi sledila iz takve definicije. Saznanje o složenosti ovog postupka može da nam pomogne da imamo malo više razumevanja prema određenoj sporosti koju ispoljavaju neki prevodioci kada se suoče sa polimorfnim izrazima.

---

[54] Primetimo da je specijalan slučaj podtipa upravo jednakost tipova. Ako je *A* podtip tipa *B* i *B* podtip tipa *A*, onda su *A* i *B* jednaki tipovi.

[55] Pri razmatranju tipa *levog argumenta* operatora dodeljivanja moramo da uzimamo u obzir ne samo tipove već i semantiku konkretnog programskog jezika. U slučaju programskog jezika C++, operator dodeljivanja se podrazumevano ne nasleđuje. Čak i kada se eksplicitno nasledi iz bazne klase (pomoću klauzule `using`), to se tumači kao dodatna definicija. U suprotnom, kada bi se nasleđivao, tj. kada bi umesto navedenog pravila važilo da je „*Ta* podtip tipa *TV*“, onda bi moglo da dođe do narušavanja integriteta (na primer, ako bismo promenljivoj tipa `Kvadrat` mogli da dodelimo vrednost tipa `Pravougaonik`, onda to više ne bi bio kvadrat).

Statičko i dinamičko vezivanje

Za svaku *zapisanu* operaciju u programu (operator, funkcija, metod), u nekom trenutku pre njenog izvršavanja mora da se odredi koja će *konkretna* operacija biti izvršena. Na primer, da li će operacija sabiranja biti izvršena kao celobrojno ili realno sabiranje, ili iz koje klase će biti odabran i izvršen metod `Povrsina`. Postupak odabiranja konkretne operacije se naziva vezivanje zapisane operacije za konkretnu implementaciju (obično koristimo kraće termine *vezivanje operacija* ili samo *vezivanje*).

Vezivanje operacija može da se odvija statički i dinamički. *Statičko vezivanje* se odvija unapred, pri prevođenju ili pripremanju programa za izvršavanje. *Dinamičko vezivanje* se odvija neposredno pre izvršavanja konkretne operacije.

Videli smo da se programski jezici na različit načine odnose prema proveri tipova i polimorfizmu. Način vezivanja nije eksplicitno povezan sa načinom proveravanja tipova, već zavisi od semantike i implementacije programskog jezika, ali možemo da uočimo određene veze. Kod slabo tipiziranih jezika se malo toga odvija u fazi prevođenja (ili pripremanja), već se praktično sve provere tipova odvijaju u fazi izvršavanja programa. Posledica toga je da onda i vezivanje operacija mora da se odvija u fazi izvršavanja, tj. na sceni je isključivo dinamičko vezivanje. Sa druge strane, kod strogo tipiziranih jezika se provere tipova odvijaju unapred, pa onda može (ali ne mora) da se koristi i statičko vezivanje.

Odnos univerzalnog i promenljivog dela polimorfnog koda prema načinu vezivanja nije isti. Doslovno univerzalni deo koda (tj. onaj čija je implementacija ista) uvek može da se vezuje statički. Sa druge strane, mogućnost statičkog vezivanja načelno univerzalnog dela (onaj čija je apstrakcija univerzalna) i promenljivog dela zavisi od više faktora.

Statičko vezivanje operacija možemo da dalje podelimo na dva podtipa – *površinsko* statičko vezivanje i *dubinsko* statičko vezivanje. Površinsko statičko vezivanje podrazumeva da se u trenutku vezivanja bira jedna od raspoloživih implementacija operacije sa kojom se konkretan zapis vezuje. Kada se kaže samo „statičko vezivanje“ obično se misli samo na površinsko statičko vezivanje. Neka operacija može da se vezuje površinski statički ako nije polimorfna ili ako svi njeni promenljivi delovi moraju da se vezuju dinamički.

Dubinsko statičko vezivanje podrazumeva da se po potrebi pravi i prevodi *nova implementacija* operacije, za konkretne upotrebljene tipove. Dubinsko statičko vezivanje se obično naziva *instanciranje*, zato što se prevodi posebna instanca polimorfne operacije za svaki tip za koji se ona upotrebljava. Operacija može da bude predmet instanciranja ako ima bar jedan promenljivi deo koji ne mora da se vezuje dinamički.

Kombinovanje dinamičkog i statičkog vezivanja podrazumeva da prethodno može da se izvrši provera saglasnosti tipova, čak i kada nije unapred poznat konkretan tip operanda. To znači da saglasnost tipova mora da se proveri na

uopštenom interfejsu, tj. da posmatrana operacija mora da ima uopšten interfejs, koji ne zavisi od konkretnih tipova operanada za koje se koristi. Drugim rečima, u kontekstu dela programa koji se prevodi statički, preduslov za dinamičko vezivanje nekih operacija je da sve varijante posmatrane operacije imaju isti interfejs.

Vratimo se na početni primer i pretpostavimo da je sličan isečak koda upotrebljen za definisanje polimorfne funkcije `odnos`, a da je zatim taj deo koda upotrebljen u konretnom slučaju:

```
...
float odnos( ... lik ) {
    return povrsina( lik ) / obim( lik );
}

...
Kvadrat k {...};
float k_odnos = odnos( k );
...
```

Kakvo vezivanje se primenjuje pri pozivanju `odnos(k)`?

Funkcija `odnos` je potencijalno polimorfna – polimorfna je ako ima promenjive delove, tj. ako je polimorfna neka od funkcija `povrsina` i `obim`. Funkcija u konkretnom slučaju može da se vezuje statički. Pri tome način vezivanja zavisi od prirode polimorfnih operacija `povrsina` i `obim`.

- Ako operacije `povrsina` i `obim` imaju uvek isti tip i vezuju se dinamički, onda je jedna verzija operacije `odnos` odgovarajuća za sve tipove. Upotreba operacije `odnos` može da se vezuje površinski statički. Upotrebe operacije `povrsina` i `obim` u njenoj definiciji se vezuju dinamički.
- Ako operacije `povrsina` i `obim` (ili bar jedna od njih) imaju potencijalno različite tipove rezultata za različite tipove argumenata (npr. za neke tipove vraćaju `int` a za neke druge `float`), onda one ne mogu da se vezuju `dinamički` u statički prevedenom kodu. Dalje, to znači da operacija `odnos` ne može da se prevede u samo jednu instancu i da se zatim vezuje površinski statički (zato što nam je u nekim slučajevima potrebno celobrojno a u nekim drugim realno deljenje), ali može da se vezuje dubinski statički, tj. da se instancira.
- Ako se radi o saglasnim tipovima, onda dinamičko vezivanje može uvek da se primenjuje, ukoliko ga podržava programski jezik. Ako se ograničimo na metode (i ne bavimo se funkcijama) i pri tome sve instance metoda (tj. sve njegove implementacije u nekoj hijerarhiji) imaju isti interfejs, onda nema prepreke za dinamičko vezivanje.

Način vezivanja je obično povezan sa suštinskim konceptima programskog jezika. Sposobnost potencijalno promenljivog dela da se vezuje statički zavisi od njegovih tehničkih karakteristika i od načina ostvarivanja polimorfizma u konkretnom programskom jeziku. Lako se uočava da je dinamičko vezivanje manje efikasno od statičkog, zato što u fazi izvršavanja mora da se uradi više posla. Instanciranje zbog toga ima posebno značajnu ulogu u optimizaciji programa, zato što omogućava da se implementacija posebno optimizuje za svaki konkretan slučaj.

Neki programski jezici imaju samo dinamičko vezivanje. Takav koncept je uobičajen za interpretirane jezike i skript-jezike (na primer *JavaScript*).

Programski jezici koji imaju samo statičko vezivanje ne mogu da u punoj meri podrže hijerarhijski polimorfizam (više o tome u narednom odeljku), ili uopšte nemaju koncept klase. Primer takvog jezika je *C*.

U većini savremenih programskih jezika se primenjuje kombinovanje statičkog i dinamičkog vezivanja, prvenstveno zbog toga što statičko vezivanje nudi veću efikasnost a dinamičko veću fleksibilnost. Na primer, moglo bi se reći da su *Java* i *C++* dva ekstremno udaljena primera kombinovanog vezivanja. U programskom jeziku *Java* se svi metodi klasa smatraju za potencijalno promenljive (tj. mogu da se razlikuju u različitim klasama hijerarhije) pa se vezuju dinamički, osim ako su deklarisani kao `final` u baznoj klasi. Za razliku od *Jave*, u programskom jeziku *C++* se za potencijalno promenljive metode smatraju samo oni koji su deklarisani kao `virtual`, dok se sve ostale operacije vezuju statički. Praktično, *Java* ima podrazumevano dinamičko vezivanje, a statičko samo kao vid optimizacije, a *C++* ima podrazumevano statičko vezivanje, a dinamičko samo kao eksplicitan vid implementiranja promenljivih delova klasa.

Vrste polimorfizma

Sveprisutnost objektno-orijentisanog programiranja i objektno-orijentisanih metodologija ima za posledicu da se polimorfizam često poistovećuje sa konceptom nasleđivanja. Iako je hijerarhijski polimorfizam konceptom nasleđivanja klasa čvrsto ugrađen u samu suštinu objektno-orijentisanog programiranja, to je ipak samo jedan od vidova polimorfizma. U savremenom razvoju softvera se prepoznaju četiri osnovne vrste polimorfizma:

- hijerarhijski polimorfizam;
- parametarski polimorfizam;
- implicitni polimorfizam i
- ad-hok polimorfizam.

Parametarski i implicitni polimorfizam imaju sve veći značaj, zato što ih sve više programskih jezika implementira. Za razliku od hijerarhijskog polimorfizma, koji je

veoma prisutan u literaturi, parametarskom i implicitnom polimorfizmu se obično posvećuje manja pažnja. Zato ćemo se u nastavku ovog poglavlja, nakon kratkog predstavljanja različitih vrsta polimorfizma, posvetiti pre svega parametarskom polimorfizmu.

Hijerarhijski polimorfizam

Objektno-orijentisano programiranje počiva na konceptima klase i nasleđivanja klasa. Nasleđivanje klasa u osnovi nije ništa drugo do deklarisanje da jedna klasa predstavlja specijalan slučaj (ili *potklasu*) druge klase, tj. da druga klasa predstavlja uopštenje (tj. *natklasu*) prve klase. Relacija *potklasa* je specijalan slučaj relacije *podtip*, a relacija *natklasa* specijalan slučaj relacije *nadtip*. Ako tipove posmatramo kao skupove, onda je skup svih vrednosti nekog *podtipa* podskup skupa vrednosti njegovog *nadtipa*. Zato se nasleđivanje često opisuje kao odnos *jeste* (engl. *is a*), zato što svaki objekat *potklase* jeste istovremeno i objekat *natklase*. O tome je već bilo reči u odeljku *Specijalizacija i generalizacija*, na strani 34.

Hijerarhijski polimorfizam koristi formalno uspostavljene hijerarhijske odnose *nadtip* i *podtip* među klasama kao sredstvo ostvarivanja polimorfizma. Formalizam uspostavljanja tih odnosa predstavlja tekst programskog koda, a u skladu sa sintaksom programskog jezika. Hijerarhijski polimorfizam je u osnovi osobine objektno-orijentisanih programskih jezika da programski kod, koji je napisan da radi sa objektima jedne klase, može da radi i sa objektima svih njenih potklasa.

Na primer, neka imamo hijerarhiju klasa geometrijskih likova, gde je bazna klasa `Lik`, a jedna od klasa hijerarhije je i klasa `Kvadrat`. Ako u baznoj klasi `Lik` postoji metod `Polozaj()` koji izračunava položaj lika, onda će on moći da se upotrebi i na objektima bilo koje konkretne klase, koja je neposredno ili posredno izvedena iz klase `Lik`, kao na primer na objektima klase `Kvadrat`:

```
void f( Kvadrat& k ){
    ... k.Polozaj() ...
}
```

Nosilac hijerarhijskog polimorfizma je hijerarhija klasa. Ulogu univerzalnog dela polimorfnog koda imaju interfejsi, odnosno metodi koji se nasleđuju i operacije koje koriste objekte hijerarhije klase. Promenljive delove predstavljaju različite implementacije interfejsa.

Da bi hijerarhijski polimorfizam mogao da se dosledno implementira, neophodno je da programski jezik podržava dinamičko vezivanje. Bez toga nije moguće ostvariti polimorfizam u pogledu različitih implementacija istog interfejsa u klasama hijerarhije. Na primer, ako u nekom nizu možemo da imamo objekte različitih klasa hijerarhije, onda ne možemo da u fazi prevođenja predvidimo koje tačne konkretne operacije ćemo koristiti za različite pojedinačne elemente niza, pa je zato neophodno da koristimo dinamičko vezivanje.

Kao što smo već istakli i predstavili primerom (u odeljku *3.4 – Slabosti objektno-orijentisanih koncepata*, na strani 37), odnosi koji upravljaju nasleđivanjem nisu do kraja formalno definisani, što ima neke veoma neugodne posledice. Činjenica da u izvedenim klasama neki metodi mogu da se napišu tako da ne rade isto kao u baznoj klasi predstavlja osnovu polimorfnog ponašanja klasa hijerarhije, a istovremeno potencijalno narušava osnovne pretpostavke hijerarhijskog polimorfizma, odnosno važenje relacije *jeste*.

Parametarski polimorfizam

Parametarski polimorfizam nema praktično nikakve veze sa hijerarhijskim polimorfizmom. Za razliku od hijerarhijskog polimorfizma, on ne počiva na deklarisanim odnosima među tipovima ili klasama. Parametarski polimorfizam počiva na upotrebi tzv. *tipskih promenljivih*. Tipske promenljive se koriste za simboličko označavanje tipova vrednosti, promenljivih i izraza. Svaki put kada se upotrebi programski kod napisan uz upotrebu parametarskog polimorfizma, simbolički tipovi (tipske promenljive) se zamenjuju konkretnim tipovima i program se prevodi (instancira) u nepolimorfnom obliku.

Da bi prevođenje bilo uspešno, neophodno je da u svim delovima tog polimorfnog koda zamenjivanje simboličkih tipova konkretnim tipovima proizvodi ispravan programski kod. Ako to nije slučaj, tj. ako je konkretan tip takav da čini da programski kod nije ispravan, onda će prevodilac prijaviti grešku. U zavisnosti od programskog jezika (ili čak verzije programskog jezika), ispravnost se ustanovljava ili pokušavanjem prevođenja i ustanovljavanjem da li je ono uspešno izvedeno ili ne, ili proveravanjem nekih eksplicitno deklarisanih uslova, ili oboje.

Određivanje konkretnih tipova, koji se koriste u nekom konkretnom slučaju, može da se izvodi eksplicitno, navođenjem tipa, ili implicitno, na osnovu pravila prevođenja konkretnog programskog jezika.

Kod parametarskog polimorfizma univerzalan deo predstavljaju algoritmi i strukture koje opisujemo uz parametrizovanje promenljivih delova. Parametarski polimorfizam može da se implementira primenom statičkog vezivanja i posebno dubinskog statičkog vezivanja – instanciranja. Punu snagu ostvaruje samo uz primenu stroge provere tipova i statičkog prevođenja.

Parametarski polimorfizam se često naziva i *generičkim programiranjem*. Kada se govori o tehnici programiranja, onda je češće u upotrebi termin generičko programiranje, a kada se govori o tehnikama ostvarivanja polimorfizma i višestruke upotrebljivosti programskog koda, onda se obično koristi termin parametarski polimorfizam. U svakom slučaju, radi se o suštinski istim stvarima.

Parametarski polimorfizam je zastupljen u sve većem broju programskih jezika. Smatra se da se po prvi put pojavio 1976. godine u okviru funkcionalnog programskog jezika *ML*. Predstavljao je važnu tehniku programiranja i u

programskom jeziku *Ada*. Tokom 1980-ih je uveden u veliki broj funkcionalnih jezika, pa možemo reći da danas većina značajnih funkcionalnih programskih jezika koristi neki oblik parametarskog polimorfizma, sa izuzetkom onih jezika koji umesto parametarskog koriste implicitni polimorfizam.

Programski jezik *C++* je od samog početka podržavao parametarski polimorfizam, ali njegov značaj se vremenom povećavao. Ključan doprinos je ostvarila grupa istraživača koji su radili na oblikovanju i razvoju biblioteka. Među njima se izdvajaju Aleksandar Stepanov i Meng Li, koji su početkom 1990-ih definisali *Standardnu biblioteku šablona* (engl. *Standard Template Library*), koja se (u nekom od svojih oblika) distribuirala uz većinu tadašnjih prevodilaca [*Stepanov 1994*]. Biblioteka je potom došla i do tela za standardizaciju programskog jezika *C++* i od nje je nastala Standardna biblioteka programskog jezika *C++*, što je formalizovano ISO standardom *C++98*. Standardna biblioteka *C++*-a skoro u potpunosti počiva na intenzivnoj upotrebi parametarskog polimorfizma. Po uzoru na *C++*, parametarski polimorfizam su, na sličan način ali uz značajna ograničenja, podržali i programski jezici *Java* i *C#*.

Podrška za parametarski polimorfizam u programskom jeziku *C++* je implementirana kroz koncept *šablona funkcija* i *klasa*. *Šablon funkcije* predstavlja funkciju u kojoj se upotrebljava neki parametarski (simbolički) tip. Slično tome, *šablon klase* je definicija klase koja počiva na upotrebi jednog ili više simboličkih tipova. Ove dve vrste šablona predstavljaju veoma važnu tehniku programiranja, pa ćemo ih u ovom poglavlju malo temeljnije obraditi.

Implicitni polimorfizam

Implicitni polimorfizam podrazumeva da se pri pisanju koda uopšte ne navode tipovi vrednosti i izraza. Pretpostavlja se da će prevodilac analizirati svaki konkretan segment programskog koda i sam zaključiti za koje tipove može da se prevede. Ako neki deo koda može da se uspešno prevede za više različitih tipova, onda se prevodi i koristi kao polimorfan deo koda. Implicitni polimorfizam predstavlja uopštenje parametarskog polimorfizma, tj. može se reći i da je parametarski polimorfizam dobijen uvođenjem određenih ograničenja i dodatnih specifikacija u implicitni polimorfizam, radi olakšavanja posla prevodiocu. Zbog toga, kao i u slučaju parametarskog polimorfizma, ni za implicitni polimorfizam nema posebnih preduslova u odnosu na način proveravanja tipova i vezivanje.

Implicitni polimorfizam se koristi u velikom broju dinamičkih programskih jezika, kod kojih se proveravanje tipova izraza (a time i proveravanje ispravnosti koda) odlaže do faze izvršavanja programa (na primer *Lisp*), ali i kod značajnog broja programskih jezika kod kojih se statičkom proverom tipova u fazi prevođenja tačno ustanovljava (potencijalno polimorfan) tip svakog izraza u kodu.

Na primer, naredna definicija funkcije `max` u programskom jeziku *Wafl* [*Malkov 2010*] je polimorfna i radi za sve tipove za koje postoji definisan binarni operator „>“:

```
max(x,y) = if x>y then x else y;
```

Još opštija bi bila definicija funkcije koja pronalazi najmanji element liste elemenata, pri čemu se za poređenje koristi data binarna funkcija `less`, a u slučaju prazne liste se vraća `zero`:

```
min( lst, less, zero ) =
    if lst.empty() then zero
    else foldl( lst.tl(),
                \x,y [less]: if less(x,y) then x else y,
                tl.hd() );
```

gde se u primeru definiše i koristi polimorfna binarna lambda (neimenovana) funkcija koja za dati operator `less` vraća manji od dva argumenta `x` i `y`.

Zanimljivo je da istaknemo da specifičan vid implicitnog polimorfizma predstavljaju i makroi programskog jezika *C*. Na primer, ako definišemo makro `max` na uobičajen način:

```
#define max(x,y) ((x)>(y) ? (x) : (y))
```

onda napisani makro možemo da upotrebimo za izračunavanje većeg od dva argumenta za bilo koje tipove podataka za koje je definisan operator poređenja „>“. Makroi u *C*-u predstavljaju vrlo ograničen vid implicitnog polimorfizma.

Ad-hok polimorfizam

Ad-hok polimorfizam počiva na osobini nekih programskih jezika da dopuštaju višeznačnost imena funkcija i operatora (engl. *overloading*). Ako se napiše više funkcija, tako da imaju isto ime i isti broj argumenata, ali su im argumenti različitih tipova, pri čemu sve funkcije rade suštinski istu stvar, prilagođenu različitim tipovima argumenata, onda one predstavljaju primer ad-hok polimorfizma.

U većini programskih jezika višeznačnost imena funkcija i operatora je ograničena samo na ugrađene osnovne elemente jezika (npr. operator sabiranja), ali postoje programski jezici koji programerima dopuštaju da sami pišu višeznačne operatore i funkcije. U programskom jeziku *C++* dopušteno je da se napiše više funkcija sa istim imenom, sve dok se sve takve funkcije mogu nedvosmisleno razlikovati po broju ili tipovima argumenata. Na primer, možemo da napišemo dve različite implementacije funkcije `potencijalnoZanimljivBroj`:

```
int potencijalnoZanimljivBroj( int n ){
    return n%3==0 && n%5!=0;
}
double potencijalnoZanimljivBroj( double x ){
    return potencijalnoZanimljivBroj( round(x) );
}
```

Ad-hok polimorfizam deli neke osobine parametarskog polimorfizma, pa ga neki autori smatraju njegovim posebnim slučajem i ne prepoznaju kao posebnu vrstu polimorfizma. Funkcije koje dele zajedničko ime, moraju biti *konceptualno usklađene* da bi se moglo govoriti o vidu polimorfizma. Ako neke funkcije imaju isto ime, ali za različite tipove argumenata izračunavaju suštinski različite stvari ili izvode suštinski različite operacije, onda nije u pitanju polimorfizam, nego samo višeznačnost imena funkcije. Činjenica da programer mora da se stara o toj konceptualnoj usklađenosti predstavlja značajan faktor razlikovanja između ad-hok i parametarskog polimorfizma.

Ad-hok polimorfizam ima i neke dodirne tačke sa hijerarhijskim polimorfizmom. On omogućava da za različite tipove definišemo različito ponašanje, što je slično kao u slučaju virtualnih metoda. Međutim, u programskom jeziku C++ ad-hok polimorfizam se uvek statički prevodi i vezuje, za razliku od virtualnih metoda.

Primena ad-hok polimorfizma može da se lepo nadopunjuje sa parametarskim polimorfizmom. Na primer, ako se napiše generička funkcija `g` (primena parametarskog polimorfizma) koja koristi neku funkciju `f` za parametarski tip `T`, onda se eksplicitnim definisanjem funkcije `f` za neke tipove `T1` i `T2` (primena ad-hok polimorfizma) omogućava da se i funkcija `g` primenjuje na `T1` i `T2`. Drugi vid je tzv. specijalizacija šablona, o kojoj će biti više reči u daljem tekstu.

Iako se naziva polimorfizmom, ad-hok polimorfizam je zapravo pre *dopuna* parametarskog i implicitnog polimorfizma, tj. tehnika koja dodatno olakšava i unapređuje njihovu primenu. Kao *pravi* polimorfizam se odnosi vrlo usko samo na funkcije i operatore koji imaju različito definisane tipove za različite vrste argumenata.

11.2 Šabloni funkcija

Šabloni funkcija u programskom jeziku C++ predstavljaju sredstvo za pisanje polimorfnih funkcija. Polimorfizam se ostvaruje apstrahovanjem nekih tipova i/ili konstanti koji se pojavljuju u deklaraciji i implementaciji funkcije. Najčešće se apstrahuju tipovi argumenata i rezultata, ili konstante koje se odnose na veličinu ili strukturu argumenata ili rezultata, ali se mogu apstrahovati i tipovi i konstante koji se koriste u samoj implementaciji.

Deklaracijama i definicijama šablona funkcija mora da prethodi *deklaracija parametara šablona*. Deklaracija parametara šablona se sastoji od ključne reči `template`, iza koje se u uglastim zagradama navode parametri šablona. Svaki parametar se opisuje *vrstom* i *imenom*. Parametar šablona može da predstavlja tip (tzv. *tipski parametar*) ili konstantu (tzv. *konstantni parametar*). Ako se radi o tipu, onda

se kao oznaka vrste upotrebljava ključna reč `typename`[56], dok se u slučaju konstante kao oznaka vrste upotrebljava tip konstante. Na primer, u sledećoj deklaraciji šablona se navodi da postoje dva parametra – tip `T` i celobrojna konstanta `K`:

```
template< typename T, int K >
```

Nakon deklaracije šablona navodi se deklaracija ili definicija funkcije. Ni deklaracija ni definicija se ne razlikuju značajno od uobičajenog načina deklarisanja i definisanja funkcija. Jedina razlika je u tome što i u deklaraciji i u definiciji mogu da se upotrebljavaju parametrizovani tipovi i konstante.

Zamenjivanje parametara šablona konkretnim tipovima i konstantama naziva se *instanciranje šablona*. Rezultat instanciranja šablona je jedna konkretna funkcija, koja može da se prevede i upotrebi[57]. Šabloni funkcije mogu da se instanciraju *eksplicitnim vezivanjem parametara* ili *implicitnim vezivanjem* (*automatskim zaključivanjem*) *parametara*. Eksplicitno vezivanje parametara podrazumeva da programer na mestu upotrebe šablona funkcije eksplicitno navede vrednosti parametara. Vrednosti parametara se navode neposredno iza imena funkcije, u uglastim zagradama, u istom poretku u kome su deklarisani. Na primer, u narednom izrazu se šablon funkcije `f` instancira eksplicitnim navođenjem vrednosti parametara `int` i `10`:

```
f<int,10>(...)
```

Ako pri pozivanju funkcije, na osnovu navedenih konkretnih argumenata, može da se jednoznačno ustanovi koje su to vrednosti parametara šablona koje daju odgovarajuću instancu šablona funkcije, onda može da se primeni implicitno instanciranje. U tom slučaju šablon funkcije možemo da koristimo kao da je u pitanju obična funkcija, bez eksplicitnog navođenja vrednosti parametara šablona, ili uz

---

[56] Umesto ključne reči `typename` može da se upotrebi i ključna reč `class`. U prvim verzijama programskog jezika *C++* je u ovom kontekstu upotrebljavana samo ključna reč `class`, prvenstveno da se ne bi uvodile nove ključne reči. Ipak, kada je međunarodni komitet radio na standardizaciji, procenili su da je ovakva upotreba ključne reči `class` problematična, najpre zbog toga što parametrizovani tipovi ne moraju biti samo klase već mogu da budu i ugrađeni tipovi, funkcije i drugo, ali i zato što ova primena ključne reči nema mnogo veze sa njenim osnovnim značenjem. Zbog toga je uvedena nova ključna reč `typename` i preporučena je njena upotreba u ovom kontekstu, iako je zbog kompatibilnosti sa starim izvornim kodim omogućeno da se i dalje upotrebljava i ključna reč `class`.

[57] Naravno, pod uslovom da definicija funkcije može da se uspešno prevede i izvrši za konkretne vrednosti parametara. Načinu prevođenja šablona i neophodnim uslovima za uspešno prevođenje šablona ćemo se više posvetiti u jednom od narednih odeljaka.

eksplicitno označavanje da se radi o šablonu, ali bez navođenja vrednosti parametara:

```
...f(...)...
...f<>(...)...
```

Ako programer upotrebi šablon na implicitan način (bez eksplicitnog instanciranja, očekujući da će prevodilac uspeti da jednoznačno prepozna odgovarajuće vrednosti parametara), a prevodilac ustanovi da ne postoji jednoznačno rešenje (tj. da ne postoji nijedna odgovarajuća konfiguracija parametara, ili da, nasuprot tome, postoji više podjednako prihvatljivih konfiguracija parametara), onda će prevodilac prijaviti odgovarajuću grešku. Takođe, ako eksplicitno navedeni parametri daju definiciju funkcije koja ne može da se prevede (npr. za navedene tipove ne postoje neke od operacija koje se upotrebljavaju u definiciji šablona funkcije), biće izdata odgovarajuća greška pri prevođenju. Pri tome bi trebalo da imamo u vidu da obe vrste grešaka pri instanciranju šablona mogu da budu veoma prikrivene ili tek posredno iskazane, posebno u složenim slučajevima, kada se u definiciji jednog šablona upotrebljavaju neki drugi šabloni.

Ako se navede upotreba bez navođenja parametara šablona i čak bez navođenja uglastih zagrada (tj. u obliku u kome može da se pozove obična funkcija), a pri tome postoji funkcija sa istim imenom i brojem argumenata i saglasnim tipovima argumenata, onda će prevodilac uvek pre birati tu funkciju nego odgovarajuću instancu šablona. Nasuprot tome, ako navedemo prazne uglaste zagrade, onda će to značiti da na tom mestu mora da se instancira šablon, tj. da ne sme da se upotrebi funkcija[58].

Ne postoje nikakva posebna ograničenja u pogledu broja različitih instanciranja šablona, sve dok su sva pojedinačna instanciranja ispravna. Jedan isti šablon funkcije može da se instancira na različite načine u jednom istom izrazu, bilo eksplicitno bilo implicitno.

Proces prevođenja šablona funkcija ima nekoliko specifičnosti. U „prvom prolazu“, pri „prevođenju“ same definicije šablona funkcije, prevodilac proverava samo da li definicija zadovoljava osnovne normative koje propisuje sintaksa programskog jezika C++. Zbog toga što u tom trenutku još uvek nisu poznate vrednosti parametara, nije moguće ni tačno prevođenje ni dosledno proveravanje ispravnosti sintakse. Tek pri instanciranju šablona funkcije prevodilac „ponavlja“ prevođenje šablona za konkretne navedene vrednosti parametara.

[58] Pisanje funkcije i šablona funkcije koji imaju isto ime je veoma loša ideja, bez obzira na to da li imaju isto ponašanje ili ne. Ako imaju isto ponašanje, onda je nešto od toga suvišno, a ako nemaju, onda bi trebalo da se nazovu različitim imenima.

Posledica takvog načina prevođenja je da se mnoge greške načinjene pri pisanju definicije šablona funkcije ispoljavaju tek pri njenom instanciranju, što može da otežа prepoznavanje grešaka. Zbog toga je pri testiranju šablona funkcije i posebno pri pisanju testova jedinica koda kojima se proverava ispravnost šablona funkcije, neophodno da se testiranje sprovede na različitim instancama šablona. Štaviše, često je potrebno da se napravi veći broj suštinski različitih instanci.

Šablon funkcije max2

Za ilustraciju ćemo se poslužiti jednostavnom funkcijom `max2`[59], koja računa veći od dva data cela broja:

```
int max2( int x, int y ){
    return x > y ? x : y;
}
```

Funkciju `max2` možemo da zamenimo polimorfnom implementacijom, šablonom funkcije `max2`, na sledeći način:

```
template< typename T >
T max2( T x, T y ){
    return x > y ? x : y;
}
```

Primetimo da su sve načinjene izmene sasvim jednostavne. Najpre je ispred same definicije funkcije dopisana deklaracija parametara šablona: `template<typename T>`, a zatim su u samoj definiciji funkcije sva pojavljivanja tipa `int` zamenjena tipskim parametrom `T`. Tako smo dobili polimorfnu implementaciju `max2`.

Šablon funkcije `max2` možemo da instanciramo i upotrebimo uz eksplicitno ili implicitno vezivanje odgovarajućeg tipa kao vrednosti parametra `T`. Ako se tipski parametar `T` zameni tipom `int`, dobija se celobrojna funkcija, koja je po svemu (a pre svega po ponašanju i efikasnosti) identična prethodno razmatranoj celobrojnoj funkciji. Ako bismo, umesto toga, tip `T` zamenili nekim drugim tipom, na primer tipom `double`, onda bismo dobili implementaciju koja je odgovarajuća za taj drugi tip. Na primer, eksplicitno instanciranje tipom `int` može da se zapiše:

```
... max2<int>(a,b) ...
```

[59] U primeru je upotrebljeno ime `max2` da pri eventualnom prevođenju primera ne bi dolazilo do mešanja sa šablonom funkcije `max` standardne biblioteke.

U narednom primeru se u istom izrazu šablon funkcije `max2` dva puta implicitno instancira na različite načine – najpre kao celobrojna funkcija, sa parametrom `T=int`, a zatim kao realna funkcija, sa parametrom `T=double`:

```
... max2(2,5) + max2(1.3,2.4) ...
```

Naglasili smo da će u slučaju neispravnog implicitnog instanciranja prevodilac prijaviti da je prepoznao grešku. Na primer, ako pokušamo da primenimo implicitno instanciranje šablona funkcije `max2` sa argumentima različitih tipova, poput `max2(1, 2.5)`, prevodilac će izdati ovakvu ili sličnu poruku:

```
primer.cpp: In function 'int main()':
primer.cpp:14:21: error: no matching function for call to 'max2(int,
double)'
  cout << max2(1, 2.5) << endl;
                     ^
primer.cpp:14:21: note: candidate is:
primer.cpp:5:17: note: template<class T> const T& max2(const T&,
const T&)
 inline const T& max2( const T& x, const T& y )
                 ^
primer.cpp:5:17: note:   template argument deduction/substitution
failed:
primer.cpp:14:21: note:   deduced conflicting types for parameter
'const T' ('int' and 'double')
  cout << max2(1, 2.5) << endl;
                     ^
```

Ako eksplicitno navedemo tip parametra, bilo `int` ili `double`, prevođenje će uspeti i u prvom primeru će instanca biti celobrojna funkcija, koja izračunava vrednost `2`, a druga instanca će biti realna funkcija, koja izračunava vrednost `2.5`:

```
...max2<int>(1,2.5)...
...max2<double>(1,2.5)...
```

Radi kompletnosti, naglasićemo da se pri pisanju ovakvih šablona obično radi sa što opštijim tipovima, da bi se prevodiocu ostavio prostor za širu primenu i automatske optimizacije. Zbog toga bi šablon `max2` trebalo pisati na sledeći način, kako bi se izbeglo eventualno kopiranje u slučaju primene na složenije tipove:

```
template< typename T >
const T& max2( const T& x, const T& y ){
    return x > y ? x : y;
}
```

Jednostavni šabloni funkcija se zbog efikasnosti često pišu kao *inline* funkcije, koje se ne prevode kao posebni delovi koda nego se fizički ugrađuju u kod na svakom mestu upotrebe (poput makroa programskog jezika C)[60]:

```
template< typename T >
inline const T& max2( const T& x, const T& y ){
    return x > y ? x : y;
}
```

11.3 Šabloni klasa

Kao što funkcije mogu da se uopšte šablonima funkcija, tako i klase mogu da se uopšte pomoću šablona klasa. Šabloni klasa u programskom jeziku *C++* predstavljaju sredstvo za definisanje polimorfnih tipova podataka. Polimorfizam se ostvaruje apstrahovanjem nekih tipova i/ili konstanti u deklaraciji i definiciji klase. Najčešće se apstrahuju oni tipovi podataka i konstante koji se odnose na strukturu elemenata klase ili na argumente najvažnijih metoda klase.

Sintaksa je veoma slična sintaksi šablona funkcije: pri opisivanju šablona klase se pre deklaracije i definicije klase navodi deklaracija parametara šablona. Takođe, ako se neki metod klase implementira van definicije klase, ispred implementacije metoda mora da se navede ista deklaracija parametara šablona.

Za razliku od šablona funkcija, u slučaju šablona klasa do skoro je bilo dopušteno samo eksplicitno instanciranje. Implicitno instanciranje u opštem slučaju nije moguće, zato što tačne vrednosti parametara obično ne mogu da se implicitno ustanove pri deklarisanju ili pravljenju objekata nekog tipa, zato što parametri šablona mogu da utiču na mnoga različita mesta na kojima se objekti i metodi klase upotrebljavaju. Ipak, od C++17 je podržano implicitno instanciranje u slučajevima kada se na osnovu upotrebe mogu jednoznačno zaključiti i vrednosti parametara.

Kao i u slučaju šablona funkcija, ne postoje ograničenja u pogledu broja različitih instanciranja šablona klasa, sve dok su sva pojedinačna instanciranja ispravna.

Proces prevođenja šablona klasa je sličan prevođenju šablona funkcija, ali uz neke dodatne specifičnosti. U „prvom prolazu“, pri „prevođenju“ same definicije šablona klase, prevodilac proverava samo da li definicije klase i svih metoda zadovoljavaju osnovne normative koje propisuje sintaksa programskog jezika. Pri instanciranju šablona klase prevodilac „ponavlja“ prevođenje strukture klase i uočava eventualne

[60] Novije verzije prevodilaca uglavnom ignorišu ključnu reč `inline` i ponašaju se kao da je ona uvek navedena u definiciji šablonske funkcije, pa pokušavaju da samostalno procene da li je bolje da se konkretna instanca šablona funkcije prevede kao posebna funkcija ili da se ugradi na mestu upotrebe.

probleme u samoj strukturi. Međutim, prevođenje instanci metoda se i dalje odlaže, sve dok se ne naiđe na njihovu konkretnu upotrebu. To u praksi ima dve veoma značajne posledice. Prva je da se prevođenje instance šablona klase praktično distribuira kroz različite module programa, tako da se svaki metod prevodi tek na mestu upotrebe, pa se i greške ispoljavaju pri prevođenju različitih jedinica koda. Druga je da metodi koji se u programu ne koriste (ili se bar ne koriste za konkretnu instancu šablona), neće ni biti prevođeni. Zbog takvog načina prevođenja može da se dogodi da neke greške u definiciji šablona klase ostanu prilično dugo prikrivene, pa čak i da šablon uđe u široku upotrebu, a da neki problem nije blagovremeno uočen.

Ako smo ranije nagovestili da opširno izveštavanje o greškama pri instanciranju šablona funkcija može da bude neugodno, onda na ovom mestu to moramo još više da istaknemo. Zaista, poruke prevodilaca u vezi sa neispravnim instanciranjem šablona klasa mogu da budu mnogo opširnije, a pronalaženje stvarnih uzroka grešaka mnogo teže, nego što je to slučaj sa šablonima funkcija. Uzrok je, naravno, u potencijalno mnogo većoj složenosti šablona klase u odnosu na šablon funkcije, kao i u brojnim međuzavisnostima različitih elemenata šablona klase. Svaki metod šablona klase je u osnovi jedan šablon funkcije, pri čemu većina metoda koristi neke druge metode istog šablona klase, tako da se povećava i dubina na kojoj se problemi mogu ispoljiti.

Prevođenje šablona i ustanovljavanje neispravnog instanciranja (tj. pokušaja instanciranja sa vrednostima parametara koje zapravo nisu dopuštene) se značajno olakšava uvođenjem *koncepata* u standardu C++20.

Šablon klase Tacka

Šablone klasa ćemo da ilustrujemo jednostavnim primerom klase `Tacka`, koja predstavlja model tačke u trodimenzionalnom prostoru. Pretpostavimo da imamo klasu čije su koordinate celobrojne i koja ima samo konstruktor i odgovarajući operator za ispisivanje:

```
class Tacka
{
public:
    int x,y,z;
    Tacka(int x0, int y0, int z0)
        : x(x0), y(y0), z(z0)
        {}
};

ostream& operator<<( ostream& ostr, const Tacka& t )
{
    ostr << "(" << t.x << "," << t.y << "," << t.z << ")";
    return ostr;
}
```

Polazeći od ove klase možemo da napravimo šablon klase `Tacka` tako što ćemo da dodamo deklaraciju parametara šablona i da zamenimo odgovarajuće tipove:

```
template <class T>
class Tacka
{
public:
    T x,y,z;
    Tacka(T x0, T y0, T z0)
        : x(x0), y(y0), z(z0)
        {}
};

template <class T>
ostream& operator<<( ostream& ostr, const Tacka<T>& t )
{
    ostr << "(" << t.x << "," << t.y << "," << t.z << ")";
    return ostr;
}
```

Kao što je već istaknuto, do standarda C++17 šablon klase je morao da se instancira eksplicitno, kao u narednom primeru koda, gde su svaki put pri upotrebi tipa šablona eksplicitno navođene vrednosti parametara:

```
Tacka<int> t(1,2,3);
cout << t << endl;
cout << Tacka<int>(1.2, 1.3, 1.4) << endl;
cout << Tacka<float>(1.2, 1.3, 1.4) << endl;
```

Od standarda C++17 vrednosti parametara ne moraju da se navode u slučajevima kada mogu da se automatski raspoznaju. Tako, na primer, pri konstrukciji objekta `t` ne mora da se navede da se radi o celobrojnom tipu, kao što ni pri poslednjoj konstrukciji tačke sa realnim koordinatama ne mora da se navede da se radi o realnom tipu.

Ipak, sa implicitnim instanciranjem šablona klasa moramo da budemo mnogo pažljiviji nego što je to slučaj sa funkcijama. Iako će konstrukcija objekta `Tacka(1.2,1.3,1.4)` biti automatski prepoznata i prevedena, to neće biti isto što i `Tacka<float>(1.2,1.3,1.4)`, zato što će u slučaju automatskog raspoznavanja biti napravljen objekat tipa `Tacka<double>`. Pogrešne pretpostavke o automatski prepoznatim tipovima mogu da dovedu do razlika u strukturi i veličini objekata, što ponekad može da ima odložen ili čak prikriven uticaj na ispravnost programa.

11.4 Šablonske promenljive

Od standarda C++14 programeri mogu da koriste i *šablonske promenljive*. Definišu se slično kao i obične promenljive, s tom razlikom da će za svaki konkretan

instancirani tip da se napravi posebna instanca šablonske promenljive. Unutar klasa mogu da se koriste samo za definisanje statičkih promenljivih klase, a ne i za definisanje podataka koji ulaze u sastav objekata[61].

Često se koriste za definisanje konstanti odgovarajuće veličine ili odgovarajućeg tipa. Na primer, možemo da definišemo više instanci konstante `pi`:

```
template<class T>
constexpr T pi = T(3.1415926535897932385L);
```

i da ih zatim koristimo bez mnogo brige o konverzijama tipova:

```
...pi<float>...
...pi<double>...
...pi<long double>...
...pi<int>...
```

Šablonske promenljive se često koriste za pojednostavljivanje rada sa šablonskim klasama. Na primer, šablon klase `std::is_integral<T>` sadrži definiciju logičke konstante `value`, koja ima tačnu vrednost ako i samo ako je tip `T` celobrojan (C++11). To možemo da iskoristimo u svojim definicijama šablona da bismo napravili različite implementacije u zavisnosti od vrste parametarskog tipa:

```
template<typename T>
int f( T x ) {
    ...
    if( std::is_integral<T>::value )
        { ... }
    else
        { ... }
    ...
}
```

Standardom C++14 se definišu mnoge šablonske promenljive, koje služe za pojednostavljivanje pristupa takvim konstantama. Na primer, šablonska promenljiva `is_integral_v<T>` se definiše kao:

```
template<typename T>
inline constexpr bool is_integral_v = is_integral<T>::value;
```

[61] Zbog toga što ne može unapred da se zna za koje će se sve različite tipove takve promenljive instancirati, ako bi se one mogle definisati na nivou objekata, onda ne bismo mogli da unapred znamo veličinu objekata. To bi onemogućilo prevođenje programa.

i može da se koristi na sledeći način:

```
if( std::is_integral_v<T> ) ...
```

11.5 Eksplicitna specijalizacija

Šabloni funkcija i klasa, kao što smo već naglasili, mogu da se instanciraju za sve vrednosti parametara za koje definicija može da se prevede. Međutim, nisu retki slučajevi da neki kod može da se prevede ali da nije odgovarajući rezultat njegovog izvršavanja, odnosno smisao tog koda u kontekstu konkretnih vrednosti parametara.

Vratimo se na ranije definisan šablon funkcije `max2` i razmotrimo šta će biti rezultat njegove primene na pokazivače:

```
int a(2),b(3);
cout << *max2(&a,&b) << endl;
```

U većini slučajeva rezultat će biti broj 2. Razlog je u tome što će rezultat primene šablona `max2` na pokazivače biti pokazivač na podatak koji se nalazi *na većoj adresi* u memoriji. U ovom slučaju tipski parametar `T` je dobio vrednost „`int*`“ i operacija poređenja je primenjena na pokazivače, a ne na objekte na koje oni pokazuju. Većina prevodilaca smešta lokalne podatke na stek tako da prva definisana promenljiva ide na više adrese, pa će tako i promenljiva `a` biti smeštena na višoj memorijskoj adresi u odnosu na `b` i predstavljat rezultat funkcije `max2`. Vrednosti promenljivih pri tome ne igraju nikakvu ulogu.

Ako bismo želeli da uporedimo objekte (u ovom slučaju brojeve) i vratimo adresu onog objekta čija je vrednost veća, onda naš šablon funkcije ne bi dobro radio. Jedno moguće rešenje je da napišemo novi šablon, na primer `max_p`, koji bi radio samo sa pokazivačima. Drugo rešenje je da ostavimo prethodnu i dodamo još jednu verziju postojećeg šablona, koja je prilagođena za slučaj sa pokazivačima[62]:

```
template< typename T >
T max2( T x, T y ){
    return x > y ? x : y;
}
```

[62] Ovaj primer je samo ilustracija koncepta. Ne preporučuje se pisanje ovakvih šablona. Umesto toga je bolje da se uvek očekuje da se radi sa objektima (ili referencama), a da se u slučaju kada se funkcija primeni na pokazivače onda zaista i želi upoređivanje njihovih primitivnih vrednosti. Izuzetak je, kao što ćemo uskoro videti, ako se radi o pokazivačima na niske.

```
template< typename T >
T* max2( T* x, T* y ){
    return *x > *y ? x : y;
}
```

Pravila prevođenja propisuju da prevodilac, ako ima na raspolaganju više verzija nekog šablona koje mogu da se primene, uvek mora da izabere onu čija deklaracija (funkcije ili klase) je manje opšta. Alternativno tumačenje je da se bira ona za koju će vrednosti tipskih parametara biti *jednostavniji* tipovi. Ako prevodilac to ne može jednoznačno da prepozna, prijaviće grešku.

U konkretnom slučaju (prethodni primer) prevodilac ima na raspolaganju dve definicije šablona. Prva je opštija, zato što deklaracija funkcije prihvata sve tipove, dok druga prihvata samo pokazivače. Zbog toga će prevodilac, ako može da bira između ove dve implementacije, uvek birati drugu. Po alternativnom tumačenju, ako bi prevodilac primenio prvu definiciju, tipski parametar `T` bi imao vrednost `int*`, a ako bi primenio drugu, tipski parametar `T` bi imao vrednost `int`. Zato što je tip `int` *jednostavniji* od tipa `int*`, prevodilac bi birao drugu verziju. Nećemo sada ulaziti u formalno definisanje poređenja tipova i složenosti tipova, već ćemo neformalno smatrati da je tip A jednostavniji od tipa B ako se oni razlikuju i tip B se gradi od A.

Pođimo sada još jedan korak dalje i razmotrimo šta će biti rezultat izraza:

```
cout << max2("niska2","niska1") << endl;
```

U ovom slučaju se radi o pokazivačima, pa će biti primenjena druga verzija šablona. Za slučaj pokazivača `const char*` naš programski kod *ne zna* da se radi o nizovima, pa poredi samo prve znakove niski i, pošto prvi znak prve niske nije veći od prvog znaka druge, rezultat će biti druga niska[63].

Ovde možemo da primenimo još jednu specijalizaciju šablona. Međutim, ako navedemo kompletan tip „`char*`“, da li nam je uopšte potreban parametar `T`? Zaista, ne samo da nam tip `T` nije potreban, već bi njegovo navođenje proizvelo suprotan efekat – zbog toga što bi kod bio ispravan za *svaki* tip `T` (zato što se on nigde ne koristi pa može da bude bilo šta), implicitno instanciranje šablona funkcije bi *uvek* bilo obavljano na osnovu drugih definicija, a ne na osnovu te nove, zato što je ona *opštija*. Zbog toga pri eksplicitnoj specijalizaciji, u slučaju kada se neki parametar više ne pojavljuje u definiciji funkcije, taj parametar može (a vidimo da praktično i mora) da se izostavi:

[63] Da nismo napisali drugu verziju šablona, i ovde bi se, kao u slučaju pokazivača na cele brojeve, poredile adrese niski, što svejedno ne bi valjalo. Znači, drugom definicijom šablona nismo napravili ovaj problem nego samo izmenili njegov oblik.

```
template<>
char* max2( char* x, char* y ){
    return strcmp(x,y)>0 ? x : y;
}
```

Iako može da izgleda da smo dovršili posao, prevođenje dovoljno dobrim prevodiocem (tj. dovoljno saglasnim sa standardom) će proizvoditi program koji i dalje ne radi ispravno!? Problem je u tipovima niski.

Po standardu jezika, svaka eksplicitno navedena niska ima tip `const char*`, a ne `char*`. Kako konstantan tip ne može da se konvertuje u nekonstantan, ispada da naša specijalizacija ne može da se primeni u konkretnom slučaju. Rešenje je da napravimo izmenjenu verziju za konstantne tipove[64]:

```
template<>
const char* max2( const char* x, const char* y ){
    return strcmp(x,y)>0 ? x : y;
}
```

Čim se pogleda navedena specijalizacija, prirodno se postavlja pitanje – šta bi se dogodilo da smo takvu specijalizaciju napisali bez deklaracije šablona bez parametara, tj. kao običnu funkciju? Da li bi to bilo isto ili ne? Da li bi ponašanje prevedenog programa bilo isto?

Ponašanje programa bi izvesno bilo isto, ali moramo da uočimo da specijalizacija bez parametara i obična funkcija *nisu* isto. Štaviše, možemo da napišemo i jedno i drugo i prevodilac neće prijaviti grešku:

```
template<>
const char* max2( const char* x, const char* y ){
    return "specijalizacija";
}

const char* max2( const char* x, const char* y ){
    return "funkcija";
}
```

Ako postoje i šablon bez parametara i funkcija sa istim imenom i tipom, prevodilac će uvek birati funkciju, osim ako je upotrebljena sintaksa eksplicitnog instanciranja šablona bez parametara. Na primer, sledeće naredbe će ispisati, redom, „funkcija“ i „specijalizacija“:

[64] Da budemo do kraja precizni, i prethodnu opštu definiciju šablona za pokazivače bi trebalo napisati za konstantne pokazivače, zato što ne menja objekte sa kojima radi. Međutim, ona bi svejedno trebalo da radi u prethodnim primerima.

```
cout << max2( "niska2", "niska1" ) << endl;
cout << max2<>( "niska2", "niska1" ) << endl;
```

11.6 Podrazumevane vrednosti parametara

Pri definisanju šablona funkcija i klasa mogu da se upotrebljavaju i podrazumevane vrednosti parametara šablona.

Primetimo da su podrazumevane vrednosti parametara šablona funkcija dopuštene tek od standarda *C++11*. Ranije verzije jezika su dopuštale samo implicitno instanciranje šablona funkcija, pa podrazumevane vrednosti parametara nisu imale mnogo smisla.

Kao i u slučaju podrazumevanih vrednosti argumenata funkcija, neki parametar šablona može da ima podrazumevanu vrednost samo ako i svi parametri koji se navode iza njega takođe imaju podrazumevane vrednosti. Važno je da se pri opisivanju podrazumevanih vrednosti parametara mogu upotrebljavati čak i prethodno navedeni parametri. Ako svi parametri šablona imaju podrazumevane vrednosti, onda se eksplicitno instanciranje može izvesti i navođenjem prazne liste parametara: „<>“. Ako se ne navede lista parametara biće primenjeno implicitno instanciranje.

Na primer, ako definišemo šablon funkcije `max2` na sledeći način:

```
template <typename T1, typename T2=T1, typename T3=T1>
inline T3 max2( const T1& x, const T2& y )
{
    return x > y ? x : y;
}
```

onda će u izrazu `max2(1,2.5)` šablon funkcije biti instanciran sa vrednostima parametara, redom, `int`, `double` i `int`, a vrednost izraza će biti `2`, dok će u izrazu `max2<double>(1,2.5)` vrednost svih parametara biti `double`, a vrednost izraza `2.5`.

Primetimo da u ovom primeru ne smemo da upotrebimo vraćanje rezultata po referenci, zato što u slučaju da se bar jedan od tipova `T1` i `T2` razlikuje od tipa `T3`, prevodilac mora da implicitno konvertuje odgovarajući argument u tip `T3`, pa dobijeni privremeni objekat neće smeti da se vrati po referenci. Posebno, ako su `T1` i `T2` različiti tipovi, onda će bar jedan od njih će biti različit od tipa `T3`, pa ćemo imati isti problem. Odgovarajuća greška bi bila ispoljena tek pri instanciranju šablona sa opisanim razlikama između tipova, tj. mogla bi da dugo ostane neprimećena. Prevodioci u nekim potencijalno problematičnim slučajevima mogu da prijave upozorenje.

Mogućnost određivanja podrazumevanih vrednosti šablona klasa je prisutna praktično od početnih verzija programskog jezika C++. Između ostalog,

podrazumevane vrednosti parametara se intenzivno upotrebljavaju u standardnoj biblioteci. Šabloni klasa u standardnoj biblioteci često imaju parametre sa podrazumevanim vrednostima, koji se u uobičajenom radu veoma retko eksplicitno navode, a omogućavaju upravljanje specifičnim aspektima rada klase. Na primer, sve standardne kolekcije (`vector`, `list`,...) imaju tipski parametar `Alloc` koji omogućava da se eksplicitno navede način upravljanja memorijom, ali se u osnovnim kursevima programskog jezika on obično i ne pominje, zato što je njegova podrazumevana vrednost u najvećem broju slučajeva dovoljno dobra:

```
template< typename T, typename Alloc = std::allocator<T> >
class vector;
```

11.7 Algoritmi i funkcionali

Praktičnu primenu šablona predstavićemo na primeru *algoritma*, koji radi sa *funkcionalima*. U terminologiji C++-a, *algoritmima* se nazivaju uopštene implementacije šablona funkcija koje se upotrebljavaju za različite tipove podataka i kolekcija, koje su često parametrizovane čak i operacijama koje se upotrebljavaju u tim algoritmima.

Funkcionalima se, strogo posmatrano, nazivaju klase čije instance mogu da se primene kao funkcije (t.j. imaju implementiran operator „`()`“). Šire od toga, sve ono što može da se upotrebi sa sintaksom funkcije (pre svega funkcije i instance funkcionala) se naziva *funkcijskim objektima*. Često se kaže da funkcijski objekti imaju *opšti funkcijski tip*. Ipak, ovi termini se često mešaju, što u nekim složenijim slučajevima može da dovede do nesporazuma, ali najčešće ne pravi probleme. U skladu sa navedenim, ako neki parametar algoritma može da bude funkcijski objekat (bez obzira na to da li je u pitanju funkcija ili funkcional), onda taj parametar ima funkcijski tip.

U programskom jeziku C++ funkcijski objekti se veoma često koriste pri implementaciji uopštenih algoritama, da bi se omogućila njihova primena na što većem skupu različitih tipova. U konkretnim slučajevima se zatim često koriste funkcionali.

Početni primer

Funkcionale i funkcijske objekte ćemo predstaviti na primeru programskog koda koji prebrojava koliko elemenata neke kolekcije zadovoljava neki uslov. Početni primer ćemo napisati bez upotrebe šablona, kao uobičajeni način brojanja neparnih elemenata u nizu celih brojeva. Štaviše, u prvih nekoliko koraka nećemo koristiti funkcionale, nego obične funkcije:

```
unsigned prebrojNeparne( const vector<int>& niz )
{
    unsigned n=0;
```

```
    for( unsigned i=0; i<niz.size(); i++ )
        if( niz[i] % 2 )
            n++;
    return n;
}

int main(){...
    cout << prebrojNeparne(niz) << endl;
...}
```

Ovakvo rešenje je ispravno, ali može da se primeni samo na cele brojeve sadržane u datom nizu, a i uslov koji se proverava je fiksiran i ne može da se menja. Kroz nekoliko koraka ćemo pokazati kako pomoću šablona možemo da uopštimo praktično sve aspekte problema – od tipa elemenata do tipa kolekcije i uslova koji se proverava.

Uopštavanje tipova elemenata i kolekcija

U prvom koraku ćemo samo da izdvojimo proveru uslova u posebnu funkciju, čime se u određenoj meri smanjuje spregnutost funkcije koja izvodi brojanje sa kontekstom brojanja:

```
bool neparan( int n ){
    return n%2;
}

unsigned prebrojNeparne( const vector<int>& niz )
{
    unsigned n=0;
    for( unsigned i=0; i<niz.size(); i++ )
        if( neparan( niz[i] ))
            n++;
    return n;
}
```

U drugom koraku ćemo da apstrahujemo uslov na način koji je uobičajen i za programski jezik *C*. Umesto fiksiranja uslova u samoj implementaciji brojanja, prenećemo uslov kao argument funkcije:

```
unsigned prebroj( const vector<int>& niz, bool(*uslov)(int) )
{
    unsigned n=0;
    for( unsigned i=0; i<niz.size(); i++ )
        if( uslov( niz[i] ))
            n++;
    return n;
}
```

```
int main(){...
    cout << prebroj(niz,neparan) << endl;
...}
```

Prva stvar koju možemo da apstrahujemo šablonom je tip elemenata kolekcije

```
template<typename T>
unsigned prebroj( const vector<T>& niz, bool(*uslov)(T) )
{
    unsigned n=0;
    for( unsigned i=0; i<niz.size(); i++ )
        if( uslov( niz[i] ))
            n++;
    return n;
}
```

Upotreba novog šablona je potpuno ista kao i upotreba prethodno napisane funkcije `prebroj`, zato što može da se primenjuje implicitno instanciranje šablona funkcije:

```
    cout << prebroj(niz,neparan) << endl;
```

Ako promenimo način prolaska kroz kolekciju, tako da se umesto opsega indeksa upotrebljavaju iteratori, onda ćemo nakon toga moći da apstrahujemo i kolekciju. Znači, najpre iteratori:

```
template<typename T>
unsigned prebroj( const vector<T>& niz, bool(*uslov)(T) )
{
    unsigned n=0;
    typename vector<T>::const_iterator
        i = niz.begin(),
        e = niz.end();
    for( ; i!=e; i++ )
        if( uslov( *i ))
            n++;
    return n;
}
```

a zatim i apstrahovanje čitave kolekcije uopštenom kolekcijom tipa `TK`:

```
template<typename T, typename TK>
unsigned prebroj( const TK& kolekcija, bool(*uslov)(T) )
{
    unsigned n=0;
    typename TK::const_iterator
        i = kolekcija.begin(),
        e = kolekcija.end();
```

```
    for( ; i!=e; i++ )
        if( uslov( *i ))
            n++;
    return n;
}
```

Ovako napisan metod može da izvrši brojanje odgovarajućih elemenata bilo koje kolekcije koja ima iteratore.

Primetimo da je u prethodnim primerima pri deklarisanju iteratora upotrebljena ključna reč `typename`. Pri upotrebi šablona prevodilac može da bude u nedoumici u vezi sa time šta programer očekuje da neko ime predstavlja u parametrizovanom tipu (u našem slučaju prevodiocu nije jasno šta u tipu `TK` predstavlja ime `const_iterator`, tj. da li je to ime tipa ili ime metoda). Imajući u vidu da imena najčešće predstavljaju podatke ili metode, prevodilac sva upotrebljena imena tako i tumači, ali to onda može da proizvede probleme ako neko ime ne predstavlja ni podatak ni metod nego ime tipa. Zbog toga, ako se u izrazu upotrebljava neki tip za koji se očekuje da bude definisan u parametrizovanoj klasi, onda takav izraz mora da bude označen ključnom reči `typename`, da bi prevodilac znao da se radi o imenu tipa.

Od standarda *C++11* postoji nova sintaksa naredbe `for`, koja omogućava obilazak cele date kolekcije, a koja se interno implementira kao obilazak pomoću iteratora, praktično na isti način kao što smo to prethodno napisali. Navodimo odgovarajući kod radi ilustracije, ali ćemo u nastavku ipak upotrebljavati verziju sa iteratorima, zato što je opštija i pruža nam neke dodatne mogućnosti:

```
template<typename T, typename TK>
unsigned prebroj( const TK& kolekcija, bool(*uslov)(T) )
{
    unsigned n=0;
    for( const auto& x : kolekcija )
        if( uslov( x ))
            n++;
    return n;
}
```

Iteratori omogućavaju sekvencijalno obilaženje elemenata kolekcija, čak i kada je njihova interna struktura daleko složenija (na primer, ako predstavlja drvo, graf ili neku drugu nelinearnu strukturu). Možemo reći i da iteratori *prevode* kolekcije u odgovarajuću sekvencijalnu reprezentaciju. Kao posledica toga se pojavljuje veoma često korišćena mogućnost da se par iteratora upotrebljava za označavanje opsega elemenata koje je potrebno da obradimo – prvi iterator određuje početak opsega (tj. njegova vrednost je iterator na prvi element opsega koji želimo da obradimo), a drugi određuje kraj opsega (tj. njegova vrednost je iterator *iza* poslednjeg elementa koji želimo da obradimo).

Ako u implementaciji šablona funkcije `prebroj` već koristimo iteratore, onda možemo da odemo i korak dalje – umesto da se ograničavamo samo na brojanje u celoj kolekciji, možemo da se bavimo i samo izabranim delovima kolekcije. Radi zadržavanja kompatibilnosti sa prethodnim verzijama, sada pišemo novu funkciju sa istim imenom, ali zadržavamo i prethodnu, s tim da je implementiramo preko nove. Parametar nove funkcije više nije tip kolekcije `TK` nego tip iteratora `Iterator`. Dodatnim primerom upotrebe ilustrovaćemo kako mogu da se prebroje svi neparni brojevi u delu niza sa indeksima od 20 do 34:

```
template<typename T, typename Iterator>
unsigned prebroj( Iterator beg, Iterator end, bool(*uslov)(T) )
{
    unsigned n=0;
    for(Iterator i=beg; i!=end; i++ )
        if( uslov( *i ))
            n++;
    return n;
}

template<typename T, typename TK>
unsigned prebroj( const TK& kolekcija , bool(*uslov)(T) )
{
    return prebroj( kolekcija.begin(), kolekcija.end(), uslov );
}

int main(){...
    cout << prebroj(niz,neparan) << endl;
    cout << prebroj(niz.begin()+20, niz.begin()+35,
        neparan) << endl;
...}
```

Uopštavanje provere uslova

Oblik u kome smo do sada navodili funkciju zahteva da bude ispunjeno nekoliko relativno strogih uslova. Prvi je da rezultat provere uslova mora biti tipa `bool`, što nije uvek slučaj u *C*-u i *C*++-u, gde se kao rezultati logičkih funkcija često izračunavaju celi brojevi. U ovom koraku ćemo da apstrahujemo sve moguće funkcije (štaviše sve moguće funkcijske objekte) koje smeju da se upotrebe u datom kontekstu, tj. za koje program može da se prevede i da ispravno radi. Umesto konkretnog tipa funkcije uvešćemo novi tipski parametar `Predikat`. Zbog uvođenja novog tipskog parametra, raniji tipski parametar `T` se više ne koristi u telu šablona, pa nam više nije potreban:

```
template<typename Iterator, typename Predikat>
unsigned prebroj( Iterator beg, Iterator end, Predikat uslov )
{...}
```

```
template<typename TK, typename Predikat>
unsigned prebroj( const TK& kolekcija, Predikat uslov )
{...}
```

Termin *predikat* se u programiranju veoma često upotrebljava da označi tip funkcije ili konkretnu funkciju koja izračunava logičku vrednost, tj. proverava ispunjenost nekog uslova za date podatke. U našem slučaju se radi o *unarnom* predikatu, ali to nećemo eksplicitno naglašavati u imenu tipa.

Ovako napisan šablon funkcije `prebroj` može da broji elemente bilo koje kolekcije, ili dela kolekcije, koja podržava iteratore. Da bi mogao da se instancira šablon potrebno je da kolekcija podržava iteratore i da predikat predstavlja unarnu funkciju, čiji je argument nekog tipa u koji tip elemenata kolekcije može implicitno da se konvertuje. Posledica povećane fleksibilnosti je da sada za proveravanje uslova možemo da upotrebimo i funkciju koja očekuje argument tipa `double`, na primer funciju `veciOd5`:

```
bool veciOd5( double n ){
    return n > 5;
}

int main(){...
    cout << prebroj(niz,veciOd5) << endl;
    cout << prebroj(niz.begin(), niz.begin()+20, veciOd5) << endl;
...}
```

Operator ()

Na ovom mestu ćemo da razmotrimo jednu veoma zanimljivu osobinu programskog jezika C++ – programibilnost operatora „`()`“. Programski jezik C++ dopušta programerima da u klasi napišu operator „`()`“ sa proizvoljnim brojem argumenata i sa proizvoljnim tipom rezultata. Štaviše, dopušteno je da se u jednoj klasi napiše proizvoljno mnogo različitih implementacija ovog operatora, sve dok se one razlikuju po broju ili tipovima argumenata. Klase koje imaju definisan bar jedan operator „`()`“ predstavljaju *funkcionale*, tj. objekti koji predstavljaju njihove instance mogu da se upotrebljabvaju poput funkcija. Kao prvi primer funkcionala napisaćemo sasvim jednostavnu implementaciju klase `Neparan`:

```
class Neparan
{
public:
    bool operator()( int n ) const
        { return n%2; }
};
```

Pre nego što nastavimo dalje, uočimo specifičnu sintaksu definicije operatora „`()`“ – prvi par zagrada predstavlja deo imena operatora, dok drugi par zagrada služi za navođenje argumenata.

Objekti ove klase mogu da se upotrebljavaju kao funkcije koje imaju jedan celobrojni argument i izračunavaju logičku vrednost, na primer:

```
Neparan a;
cout << a(3) << endl;
cout << Neparan()(5) << endl;
```

gde u prvom primeru koristimo automatski objekat `a` kao funkciju, a u drugom primeru izrazom `Neparan()` pravimo privremeni neimenovani objekat i koristimo ga kao funkciju.

Možda nije sasvim očigledno koji je smisao pravljenja ovakvog funkcionala, kada je rezultat isti kao da smo napisali funkciju, s tim da je čak neophodno i malo više pisanja. Zaista, klasa `Neparan` u našem primeru ne donosi suštinski ništa novo u odnosu na do sada upotrebljavanu funkciju `neparan`. Da bismo razumeli suštinu moramo da se podsetimo da za razliku od funkcije, koja je jedna i uvek ista, klasa može da ima više instanci (objekata), koje se mogu razlikovati. U slučaju klase `Nepran` sve instance će biti praktično iste i zato se ne vidi značajan doprinos ovakvog pristupa, ali u slučaju klase koja ima neko unutrašnje stanje (tj. čiji objekti sadrže neke podatke), svaki objekat može da predstavlja posebnu funkciju, a klasa praktično predstavlja oblik *dinamičkog* šablona funkcije sa konstantnim parametrima.

Vratimo se funkciji `veciOd5`, sa kojom smo se susreli nešto ranije, u kojoj postoji unapred određena i fiksirana konstanta 5 u odnosu na koju se porede argumenti funkcije. Ako bismo želeli da parametrizujemo tu konstantu, jedan način je da napišemo šablon funkcije:

```
template<int osnovaPoredjenja>
bool veciOd( int n ){
    return n > osnovaPoredjenja;
}
```

Takav šablon možemo da upotrebimo za pravljenje različitih instanci funkcija, na primer:

```
int main(){...
    cout << prebroj(niz,veciOd<5>) << endl;
    cout << prebroj(niz,veciOd<15>) << endl;
...}
```

Šablon funkcije može da bude dobro rešenje za neke slučajeve, ali ipak ima značajno ograničenje: parametri šablona se navode, a instance šablona prave,

isključivo *tokom prevođenja programa*. Drugim rečima, šabloni omogućavaju isključivo *statičko* instanciranje, koje je definisano u tekstu programa i predvidivo u trenutku prevođenja. Nije moguće odložiti instanciranje do izvršavanja programa i uraditi, na primer, nešto kao:

```
for( int i=0; i<100; i+=5 )
    cout << i << " : " << prebroj(niz,veciOd<i>) << endl;
```

Tu stupaju na scenu funkcionali, zato što se instance funkcionala prave *dinamički*, u fazi izvršavanja programa. Odgovarajući funkcional možemo da napišemo kao klasu sa jednim podatkom:

```
class VeciOd
{
public:
    VeciOd( int n )
        : N_(n)
    {}

    bool operator()( int n ) const
        { return n > N_; }

private:
    int N_;
};
```

Zbog toga što ima unarni operator „`()`“, koji je implementiran na odgovarajući način, klasa `VeciOd` zadovoljava sve kriterijume potrebne da se tipski parametar `Predikat` instancira klasom `VeciOd`, a konkretni argument `uslov` objektom ove klase:

```
cout << prebroj(niz,VeciOd(5)) << endl;
```

Štaviše, sada možemo da napišemo i ono što prethodno nismo mogli pomoću šablona funkcije `veciOd`:

```
int main(){...
    for( int i=0; i<100; i+=5 )
        cout << i << " : " << prebroj(niz,VeciOd(i)) << endl;
...}
```

Na ovom primeru vidimo da funkcionali donose veoma značajan nov koncept u programiranju, a koji bez primene šablona ne bi mogao da dođe u potpunosti do izražaja. Upotrebom funkcionala i šablona omogućeno je parametrizovanje ne samo podataka nego i operacija koje se izvršavaju u nekom kontekstu – i to u fazi izvršavanja programa. Na taj način čitavi algoritmi mogu da se apstrahuju i

implementiraju nezavisno od konkretnih tipova podataka, ali i od konkretnih elementarnih operacija koje se u algoritmima koriste.

Funkcionali se upotrebljavaju u mnogim segmentima standardne biblioteke programskog jezika *C++*, uključujući algoritme za pretraživanje, uređivanje, particionisanje i transformisanje kolekcija podataka i mnoge druge operacije. Radi ilustracije i daljeg upoznavanja ove oblasti preporučuje se da se analiziraju implementacije algoritama iz biblioteke `<algorithm>`. Na primer, jedan od algoritama te biblioteke je i šablon funkcije `count_if`, koji radi praktično isto što i naš šablon funkcije `prebroj`.

Vezivanje argumenta funkcije

Videli smo kako možemo da pišemo funkcionale i da ih koristimo u algoritmima. Nešto što predstavlja potencijalan problem je potreba da se pišu odgovarajuće klase, čak i kada već postoje implementirane funkcije za poređenje. Na primer, ako imamo neku funkciju koja proverava ispunjenost nekog odnosa između dva argumenta (pretpostavimo da postoje neki tipovi `T1` i `T2`), moraćemo da napišemo odgovarajuću klasu da bismo mogli da dinamički instanciramo proveravanje uslova u odnosu na različite izabrane vrednosti drugog argumenta:

```
bool proveraOdnosa( T1 a, T2 b ){...}

class ProveraOdnosa {
public:
    ProveraOdnosa( T2 b )
        : B_(b)
    {}

    bool operator()( T1 a ) const
        { return proveraOdnosa(a,B_); }

private:
    T2 B_;
};
```

Ako malo bolje pogledamo takvu klasu, možemo da uočimo da tu postoji obrazac koji se ponavlja. Zaista, ako izdvojimo samo ono što je zajedničko za sve takve klase, onda možemo da napišemo odgovarajući šablon klase, na primer ovako:

```
template<typename Fn, typename T>
class VeziDrugiArg
{
public:
    VeziDrugiArg( Fn fn, T arg2 )
        : Fn_(fn), Arg2_(arg2)
    {}
```

```
    bool operator()( T n ) const
        { return Fn_(n,Arg2_); }

private:
    Fn Fn_;
    T  Arg2_;
};
```

Objekti ovog šablona, tj. njegovih instanci, sadržaće u sebi vezanu informaciju o funkciji koja se upotrebljava za proveravanje odnosa i o vrednosti koju je potrebno navoditi kao drugi argument pri proveravanju. Instance ovog šablona predstavljaju pojedinačne klase poput predstavljene klase `ProveraOdnosa`. Na primer, naredna dva izraza bi izračunavala identične funkcionale:

```
ProveraOdnosa(v2)
VeziDrugiArg<bool (*)(T1,T2), T1>(proveraOdnosa,v2)
```

Ako imamo šablon klase `VeziDrugiArg`, onda više nije potrebno da pišemo pojedinačne odgovarajuće klase, već možemo da ih instanciramo iz univerzalnog šablona. Preostaje još jedan problem, a to je neugodna sintaksa[65] – svaki put moramo da eksplicitno navodimo tipove funkcije i argumenta, što otežava pisanje koda i daje nečitak rezultat. Srećom, i to može da se prevaziđe. Kao što smo već videli, instanciranje klase uvek zahteva da se tipovi eksplicitno navode i to ne možemo da izbegnemo, ali možemo da zaobiđemo. Iskoristićemo činjenicu da je, za razliku od šablona klasa, u slučaju šablona funkcija dopušteno implicitno instanciranje. Napravićemo šablon funkcije koji ne radi ništa drugo osim što pravi objekat odgovarajuće instance šablona klase:

```
template<typename Fn, typename T>
VeziDrugiArg<Fn,T> veziDrugiArg( Fn fn, T arg2 )
{
    return VeziDrugiArg<Fn,T>( fn, arg2 );
}
```

Sada ovaj šablon funkcije možemo da upotrebimo bez navođenja tipova i da napravimo isti funkcional kao u prethodnim slučajevima:

```
ProveraOdnosa(v2)
VeziDrugiArg<bool (*)(T1,T2), T1>(proveraOdnosa,v2)
```

[65] Podsetimo se, od C++14 prevodioci mogu da u velikom broju slučajeva izađu na kraj sa automatskim raspoznavanjem tipova i implicitnim instanciranjem šablona klasa. Ipak, valja imati u vidu i predstavljeno rešenje sa funkcijom, zato što neće u svim slučajevima biti moguće automatsko raspoznavanje tipova.

```
veziDrugiArg( proveraOdnosa, v2 )
```

Primetimo da naši šabloni `VeziDrugiArg` i `veziDrugiArg` nisu potpuno uopšteni. Oni podrazumevaju da funkcija za poređenje ima dva argumenta i da je rezultat logičkog tipa ili može da se konvertuje u logički tip. Dalje uopštavanje bi zahtevao mnogo više prostora nego što nam je na raspolaganju, pa ga ovde nećemo opisivati.

U standardnoj biblioteci `<functional>` implementirano je više različitih operacija nad funkcionalima, među kojima je i vezivanja argumenata. Ako bismo koristili delove te biblioteke, onda bismo odgovarajući funkcional (bez ograničenja koje ima naše rešenje) mogli da napravimo ovako:

```
bind2nd( ptr_fun(proveraOdnosa), v2 )
```

a u *C++*-u 11 čak i ovako:

```
bind( proveraOdnosa, _2, v2 )
```

11.8 Upravljanje ponašanjem klase

Šabloni mogu da se upotrebljavaju za efikasno upravljanje ponašanjem objekata ciljne klase. Kada se za upravljanje ponašanjem koriste neki parametri koji se dinamički proveravaju (na primer podaci sadržani u objektu, a zadati prilikom pravljenja objekta), onda se svaki put pri njihovom proveravanju troši procesorsko vreme i gubi se na performansama. Ako se način ponašanja objekata ne menja tokom njihovog života, onda je potencijalno efikasnije da se ponašanje odredi statički, u programskom kodu, kao i da se proveravanje uskladi i optimizuje sa statički podešenim parametrima. Upravo takvo ponašanje pružaju nam šabloni klasa.

Najpre ćemo da predstavimo primenu šablona za upravljanje ponašanjem klase na primeru implementacije nizova koji omogućavaju proveravanje ispravnosti opsega indeksa, a kasnije ćemo pokušati da uočimo opštija pravila. Za početak ćemo napraviti nadgradnju bibliotečkog šablona klase `vector` tako da obuhvati proveravanje ispravnosti opsega indeksa. Javnim nasleđivanjem nasledićemo i sve javne metode šablona klase. Da bi naša klasa bila jednako funkcionalna trebalo bi dodati još nekoliko konstruktora:

```
template<typename T>
class Niz : public vector<T>
{
public:
    Niz()
        : vector<T>()
    {}
```

```
    Niz( unsigned int sz )
        : vector<T>( sz )
    {}

    T& operator[]( unsigned i )
    {
        if( i >= vector<T>::size() )
            throw out_of_range("Indeks van opsega");
        return vector<T>::operator[](i);
    }

    const T& operator[]( unsigned i ) const
    {
        if( i >= vector<T>::size() )
            throw out_of_range("Indeks van opsega");
        return vector<T>::operator[](i);
    }
};
```

U prethodnom primeru smo napisali dve implementacije operatora „`[]`“. Razlog je u tome što želimo da ovaj operator koristimo na različite načine za konstantne i nekonstantne nizove. Ako niz nije konstantan, onda operator mora da vrati referencu na element niza, da bismo izrazom „`niz[i]=x`“ mogli da promenimo vrednost elementa niza. Sa druge strane, ako je niz konstantan, onda operator ne sme da vrati takvu referencu na element niza, zato što bismo tako posredno menjali sadržaj konstantnog niza. Rešenje je u pisanju dve implementacije[66] operatora „`[]`“.

Ponašanje ovako napisanog niza možemo da proverimo na sledeći način:

```
int main()
{
    try {
        Niz<int> niz(20);

        for( unsigned i=0; i<niz.size(); i++ )
            niz[i] = i*i;
        for( unsigned i=0; i<=niz.size(); i++ )
            cout << i << " : " << niz[i] << endl;
```

[66] Na prvi pogled ove dve implementacije imaju isti broj i tip argumenata, ali zapravo nije tako. Svaki metod klase, pored svih eksplicitno navedenih argumenata, ima i implicitan argument – pokazivač `this`, koji pokazuje na objekat na kome se metod izvršava. U navedenim verzijama operatora pokazivač `this` ima različite tipove: u prvom slučaju pokazivač `this` ima tip `Niz<T>*`, a u drugom, konstantnom slučaju, ima tip `const Niz<T>*`. Na osnovu tipa objekta na kome se operator izračunava, može da se odredi koja od dve verzije implementacije bi trebalo da se koristi.

```
    }catch( exception& e ){
        cerr << "*** GRESKA: " << e.what() << endl;
    }

    return 0;
}
```

Kolekcije u standardnoj biblioteci, uključujući i klasu `vector`, ne proveravaju ispravnost indeksa, zato što je primarni cilj efikasnost, a pretpostavlja se da će programeri *znati šta rade*. Sada imamo našu verziju, koja može da proverava efikasnost, ali je sigurno da će biti manje efikasna nego `vector`. Bilo bi idealno kada bi jedna ista klasa (ili šablon) mogla da se upotrebljava i sa proverom i bez provere indeksa, a zavisno od potreba programera – na primer tako da se tokom razvoja indeksi proveravaju, a da se u konačnoj verziji programa, koja mora da bude efikasnija, indeksi ne proveravaju, zato što se pretpostavlja da je program dovoljno dobro testiran.

Ako bi se prilagođavanje ponašanja odvijalo u fazi izvršavanja programa, onda bi to zahtevalo neki vid proveravanja, poput:

```
T& operator[]( unsigned i )
{
    if( ... potrebnoProveravatiOpsegIndeksa ... )
        if( i >= vector<T>::size() )
            throw out_of_range("Indeks van opsega");
    return vector<T>::operator[](i);
}
```

U slučaju takve implementacije bi najpre moralo da se svaki put proverava „da li je potrebno proveravati opseg“, pa tek onda eventualno i da se proverava da li je opseg ispravan. Zbog toga bi u oba slučaja rešenje bilo sporije nego bez te opcije. Jedini način da se ovo unapredi je da se o načinu izvršavanja operatora odlučuje pre njegovog izračunavanja, tj. ili u fazi pravljenja objekta ili u fazi prevođenja programa. Odlučivanje u fazi pravljenja objekta bi se svodilo na dinamičko vezivanje metoda, što je već umereno sporije od statički vezanog operatora, dok bi odlučivanje u fazi prevođenja programa moglo da ponudi najbolje moguće performanse. U idealnom slučaju se deo koda koji proverava indekse uopšte ne bi ugrađivao u izvršni kod ako provere nisu potrebne.

Proveru indeksa i izbacivanje izuzetka možemo da izdvojimo u posebnu funkciju, na primer ovako:

```
T& operator[]( unsigned i )
{
    proveraIndeksa( i, vector<T>::size() );
    return vector<T>::operator[](i);
}
```

Međutim, uobičajeno rešenje je da se umesto posebne funkcije (ili metoda) napravi posebna klasa `ProveraIndeksa`, koja ima odgovarajući statički metod `Provera`. Takav pristup zahteva malo više pisanja, ali nam pruža i veće mogućnosti:

```
T& operator[]( unsigned i )
{
    ProveraIndeksa::Provera( i, vector<T>::size() );
    return vector<T>::operator[](i);
}
```

U našem primeru je dovoljno da klasa `ProveraIndeksa` ima samo statički metod `Provera`, koji definiše specifično ponašanje našeg niza.

```
class ProveraIndeksa {
public:
    static void Provera( unsigned indeks, unsigned opseg ){
        if( indeks >= opseg )
            throw out_of_range("Indeks van opsega");
    }
};
```

Izvedena izmena predstavlja vid refaktorisanja – nismo promenili ponašanje našeg niza, već samo strukturu koda. Indeksi se još uvek bezuslovno proveravaju, ali sada se već nazire način apstrahovanja problema. Ako pored klase `ProveraIndeksa` napišemo i klasu `BezProvereIndeksa`, a koja ima isti interfejs kao prethodna klasa ali ne radi doslovno ništa, onda možemo da navođenjem jedne od ovih klasa kao dodatnog parametra šablona niza, odredimo ponašanje niza u odnosu na proveravanje indeksa. Štaviše, izbor ponašanja će biti obavljen statički, u fazi prevođenja programa.

U klasi `BezProvereIndeksa` metod `Provera` ne radi doslovno ništa:

```
class BezProvereIndeksa{
public:
    static void Provera( unsigned indeks, unsigned opseg ) {}
};
```

Šablonu `Niz` dodajemo novi tipski parametar `ProveravacIndeksa`. Proveravanje indeksa prepuštamo metodu `Provera` upotrebljenog konkretnog tipa koji se navede kao vrednost ovog novog tipskog parametra:

```
template<typename T, typename ProveravacIndeksa>
class Niz : public vector<T>
{
public:
```

```
    T& operator[]( unsigned i )
    {
        ProveravacIndeksa::Provera( i, vector<T>::size() );
        return vector<T>::operator[](i);
    }

    const T& operator[]( unsigned i ) const
    {
        ProveravacIndeksa::Provera( i, vector<T>::size() );
        return vector<T>::operator[](i);
    }
};
```

Ako se kao `ProveravacIndeksa` upotrebi klasa `ProveraIndeksa`, onda će se proveravati ispravnost opsega indeksa, a ako se upotrebi klasa `BezProvereIndeksa`, onda će se operator indeksnog pristupa izvršavati bez proveravanja ispravnosti indeksa. Štaviše, ako se ne vrši provera opsega indeksa, a zbog toga što se šabloni prevode za svaku instancu posebno, i naš operator i cela klasa `Niz` će se prevesti doslovno kao da smo upotrebljavali `vector` – bez ikakve cene po performanse[67].

Ako se neki oblik ponašanja koristi češće nego drugi, onda podrazumevano ponašanje može da se navede putem navođenja odgovarajuće podrazumevane vrednosti tipskog parametra. U našem slučaju, možemo da navedemo de je podrazumevano da ne želimo da proveravamo opseg indeksa:

```
template<typename T,
         typename ProveravacIndeksa=BezProvereIndeksa>
class Niz : public vector<T>
{...};
```

11.9 Rekurzivni šabloni

Šabloni dobijaju sasvim novu dimenziju upotrebljivosti kada se omogući njihovo rekurzivno definisanje. Upotrebu rekurzije u šablonima ćemo da predstavimo na primeru implementacije višedimenzionih „matrica“.

Pod višedimenzionom matricom podrazumevamo kolekciju podataka koja se indeksira po više dimenzija i koristi nalik na višedimenzione nizove programskog

[67] Ovde moramo da istaknemo da zbog različitog pristupa prevođenju i optimizovanju programa, može da se desi da u slučaju verzija koje su prilagođene za debagovanje u izvršnoj verziji programa ostanu neki delovi „suvišnog“ koda, kako bi lakše mogao da se prati tok izvršavanja programa. Sa druge strane, optimizovana produkciona izrvšna verzija bi trebalo da bude „u bajt“ identična kao u slučaju upotrebe klase `vector`.

jezika *C*, ali uz potencijalno veće mogućnosti. U ovom odeljku upotrebljavamo naziv *matrica*, a ne *niz* da bi u tekstu bilo jasnije kada govorimo o apstrahovanoj višedimenzionoj strukturi (matrici), a kada o jednodimenzionom nizu, koji koristimo pri implementaciji matrice. Za slične strukture se često koristi i naziv *tenzor*.

Implementacija bi trebalo da nam omogući da matricu i njene elemente koristimo kao u narednom primeru:

```
Matrica<int,3> m(2,3,4);   // 3 dimenzije, 2x3x4
...
m[i][j][k] = ...;
...
```

Implementacija višedimenzione matrice zavisi od tipa elemenata i broja dimenzija. Zbog toga ćemo da je implementiramo kao šablon klase sa tipskim parametrom `T` i neoznačenim celobrojnim parametrom `Dim`:

```
template<typename T, unsigned Dim>
class Matrica {...}
```

Matricu dimenzije `Dim` možemo da apstrahujemo kao niz matrica koje imaju dimenziju nižu za jedan. Matrica niže dimenzije će imati tip `Matrica<T,Dim-1>`. To će ujedno biti i tip elemenata niza kojim implementiramo matricu[68]:

```
template<typename T, unsigned Dim>
class Matrica
{
public:
    typedef Matrica<T,Dim-1> tipPodmatrice;
...
private:
    vector<tipPodmatrice> Podmatrice_;
};
```

Ovakvo rešenje počiva na primeni rekurzije po celobrojnom parametru šablona. Kao i svaka druga rekurzija, i ova mora da ima završni korak, koji je eksplicitno definisan. U slučaju šablona klasa, završni korak se definiše eksplicitnom specijalizacijom završnog slučaja. U ovom slučaju, rekurzivni korak ne može da se primeni u slučaju matrice koja ima dimenziju 1. Zato matrice dimenzije 1 implementiramo

[68] Znači, matrica dimenzije 3 će imati za elemente matrice dimenzije 2, a one će imati za elemente matrice dimezije 1, koje će na kraju sadržati niz skalara. Takva organizacija memorije je veoma neefikasna, ali je koristimo kao zanimljiv primer rekurzivnih šablona.

pomoću eksplicitne specijalizacije. Osnovna razlika je u tome što matricu dimenzije 1 implementiramo kao niz elemenata:

```
template<typename T>
class Matrica<T,1>
{
...
private:
    vector<T> Elementi_;
};
```

Sve potrebne metode moramo da razvijamo paralelno za opštiji slučaj, kada je dimenzija veća od 1, i za specijalan slučaj dimenzije 1. Da bismo mogli da pristupamo elementima matrice, potrebno je da implementiramo operator „[]". Napisaćemo za svaku verziju matrice po dve verzije operatora, konstantnu i nekonstantnu:

```
template<typename T, unsigned Dim>
class Matrica { ...

    tipPodmatrice& operator[]( unsigned i ) {
        return Podmatrice_[i];
    }

    const tipPodmatrice& operator[]( unsigned i ) const {
        return Podmatrice_[i];
    }

... };

template<typename T>
class Matrica<T,1> { ...

    T& operator[]( unsigned i ) {
        return Elementi_[i];
    }

    const T& operator[]( unsigned i ) const {
        return Elementi_[i];
    }

... };
```

U slučaju dimenzije veće od 1, operator indeksiranja vraća kao rezultat referencu na odgovarajuću podmatricu, pa na nju može ponovo da se primeni operator indeksiranja (ali ne isti, već za odgovarajući tip podmatrice). U slučaju jednodimenzione matrice, operator indeksiranja vraća referencu na konkretan element matrice.

Naše matrice su za sada prazne – podrazumevani konstruktor će napraviti matricu čiji niz podmatrica (ili elemenata) će biti prazan. Neophodno je da napišemo metod `PostaviVelicinu` za inicijalizovanje veličine i sadržaja matrice. Pretpostavićemo da metod ima onoliko argumenata koliko matrica ima dimenzija. Prvi argument određuje veličinu po prvoj dimenziji i na osnovu njega menjamo veličinu niza koji sadrži podmatrice niže dimenzije. Ostale argumente prosleđujemo rekurzivno svakoj od podmatrica. Otprilike ovako:

```
void PostaviVelicinu( unsigned d1, unsigned d2,..., unsigned dN ) {
    Podmatrice_.resize( d1 );
    for( auto& m: Podmatrice_ )
        m.PostaviVelicinu( d2, ..., dN );
}
```

Napisan kod ilustruje osnovnu ideju implementacije, ali sintaksa nije ispravna i on ne može da se prevede. Od verzije standarda *C++11* su podržani tzv. *varijadički argumenti*, tj. parametri šablona funkcija i metoda koji mogu da odgovaraju proizvoljnom broju argumenata, što konceptualno u potpunosti odgovara prethodnom primeru, ali uz nešto eksplicitniju sintaksu. Varijadički argumenti omogućavaju primenu rekurzije u odnosu na spisak argumenata. U našem slučaju svi argumenti metoda imaju isti tip, ali to u opštem slučaju ne mora da važi.

Ispravna implementacija metoda `PostaviVelicinu` izgleda ovako:

```
template<typename T, unsigned Dim>
class Matrica { ...

    template<typename... Args>
    void PostaviVelicinu( unsigned dim, Args... args ) {
        Podmatrice_.resize( dim );
        for( auto& m: Podmatrice_ )
            m.PostaviVelicinu( args... );
    }

... };
```

Kao što vidimo, osnovna ideja je ista, ali metod `PostaviVelicinu` pravimo kao šablon u zavisnosti od broja argumenata. To može da izgleda kao suvišno uopštavanje, zato što je nama za svaku verziju matrice potrebna po tačno jedna implementacija – ona koja ima broj argumenata koji je jednak broju dimenzija matrice. Ipak, zbog toga što matricu implementiramo kao šablon klase, moraćemo da i ovaj metod pišemo kao šablon, a u praksi će se prevoditi i koristiti tačno po jedna odgovarajuća implementacija ovog metoda za svaku instancu šablona klase.

Opštiji oblik šablona klase pokriva slučajeve matrica čija je dimenzija veća od 1. U skladu s tim i implementacija metoda `PostaviVelicinu` radi u slučajevima sa više

od jednog argumenta. Prvi argument je naveden eksplicitno kao `unsigned dim`. Ostali argumenti su apstrahovani varijadičkim argumentom `Args... args` koji odgovara parametru šablona `typename... Args`. Kada postavljamo dimenzije podmatrica, za svaku od njih pozivamo metod `PostaviVelicinu` sa svim argumentima osim prvog: `args...`.

Za specifičan slučaj jednodimenzione matrice moramo eksplicitno da obezbedimo drugačiju implementaciju:

```
template<typename T>
class Matrica<T,1> { ...

    void PostaviVelicinu( unsigned dim ) {
        Elementi_.resize(dim);
    }

... };
```

Sada možemo da dodamo i konstruktore matrice i metod `Velicina(d)` koji vraća veličinu tražene dimenzije matrice. Konstruktore implementiramo tako da koriste metod `PostaviVelicinu`. Moramo da implementiramo i konstruktor bez argumenata, da bismo mogli da pravimo nizove matrica:

```
template<typename T, unsigned Dim>
class Matrica { ...

    Matrica() {}

    template<typename... Args>
    Matrica( Args... args ) {
        PostaviVelicinu( args... );
    }

    unsigned Velicina( unsigned d ) const {
        if( d )
            return Podmatrice_[0].Velicina( d-1 );
        else
            return Podmatrice_.size();
    }

... };


template<typename T>
class Matrica<T,1> { ...
    Matrica() {}
    Matrica( unsigned d  ) {
        PostaviVelicinu( d );
    }
```

```
    unsigned Velicina( unsigned d ) const {
        if( d ) return 0;   // Greška, ovo ne bi smelo da se desi
        else return Elementi_.size();
    }

... };
```

Radi ilustracije upotrebe, napravićemo matricu koja ima tri dimenzije 2x3x4 i ukupno 24 elementa. Popunićemo je nekim vrednostima i zatim ćemo da ispišemo njen sadržaj:

```
int main()
{
    Matrica<int,3> m(2,3,4);
    cout << m.Velicina(0) << " x "
         << m.Velicina(1) << " x "
         << m.Velicina(2) << endl << endl;

    for( unsigned i=0; i<m.Velicina(0); i++)
        for( unsigned j=0; j<m.Velicina(1); j++ )
            for( unsigned k=0; k<m.Velicina(2); k++ )
                m[i][j][k] = (i+1)*100 + (j+1)*10 + (k+1);

    for( unsigned i=0; i<m.Velicina(0); i++){
        for( unsigned j=0; j<m.Velicina(1); j++ ){
            for( unsigned k=0; k<m.Velicina(2); k++ )
                cout << m[i][j][k] << ' ';
            cout << endl;
        }
        cout << endl;
    }
    return 0;
}
```

Program ispisuje sledeći rezultat:

```
2 x 3 x 4

111 112 113 114
121 122 123 124
131 132 133 134

211 212 213 214
221 222 223 224
231 232 233 234
```

Ovakav šablon matrice je upotrebljiv, ali ima još mnogo prostora za unapređenja. Na primer, u implementaciji bismo umesto kolekcije `vector` mogli da koristimo prethodno napisan šablon klase `Niz`, koji može da proverava ispravnost indeksa.

Drugo unapređenje se odnosi na organizaciju memorije. Opisana implementacija matrice se svodi na pravljenje velikog broja malih objekata. Na primer, ako napravimo matricu dimenzija 10x20x30, onda bi se „velika“ matrica sastojala od 10 manjih podmatrica dimenzija 20x30, a svaka od njih od po 20 nizova sa po 30 elemenata. Alokacija memorije za takvu matricu bi podrazumevala alokaciju 211 nizova, što predstavlja vrlo neefikasan oblik organizacije memorije. U slučaju većih matrica ili većeg broja dimenzija, broj alokacija može da bude i mnogo veći.

Bolje rešenje je da se svi podaci koji čine matricu alociraju u jednom koraku. Takvo rešenje bi bilo efikasnije i za alokaciju i za upotrebu[69]. Međutim, takvo rešenje zahteva nešto složeniji način implementiranja operatora indeksiranja, tj. pristupanja elementima matrice. U prethodnom primeru prva primena operatora indeksiranja je vraćala referencu na matricu niže dimenzije, koja fizički postoji u implementaciji matrice, ali u alternativnom slučaju takva matrica ne postoji fizički kao poseban objekat i zato ove operatore moramo da implementiramo drugačije. Jedan način implementacije je da se operatori indeksiranja naprave tako da izračunavaju privremene objekte slične iteratorima, koji bi se takođe implementirali kao šabloni. Drugi način je da se umesto njih implementiraju operatori „`()`“ sa varijadičkim argumentima. Treći način, ali tek od C++23, je da se koriste operatori „`[]`“ sa više argumenata.

11.10 Lambda izrazi i funkcionali

Lambda izrazi nisu neposredno vezani za parametarski polimorfizam, ali se veoma često upotrebljavaju zajedno sa šablonima, a i implementiraju se pomoću funkcionala i kao zamena za funkcionale, kojima smo već posvetili pažnju u ovom poglavlju. Zato ćemo im na ovom mestu posvetiti neophodnu i zasluženu pažnju.

U prethodnim odeljcima smo predstavili rad sa funkcionalima ali smo se zadržali na dobrim stranama te tehnike. Međutim, funkcionali imaju i neke slabosti. U praktičnom radu se najviše ispoljavaju dva problema. Prvo, ako se neka funkcija (ili funkcional) upotrebljava na samo jednom mestu u programu, onda se njenim definisanjem možda nepotrebno opterećuje prostor imena. Šta se dešava ako su za implementaciju našeg programa potrebne stotine takvih funkcija? Veliki broj imena može da ima za posledicu zatrpavanje prostora imena, ali i biranje nekih generičkih imena, koja nisu lako razumljiva.

Drugi problem je verovatno još važniji – takav način implementiranja primorava programera da fizički razdvoji opis ponašanja konkretne funkcije od mesta na kome

[69] Alokacija i dealokacije bi se izvodili u po jednom koraku, a takva organizacija podataka bi bila i mnogo bolje prilagođena keširanju.

se ona upotrebljava. Time se potencijalno otežavaju razumevanje i održavanje programa. Značaj takve razdvojenosti se dodatno povećava ako je to jedino mesto upotrebe takve funkcije (ili funkcionala), a posebno ako se radi o relativno jednostavnim funkcijama.

Ako razmotrimo način definisanja funkcionala, onda možemo da uočimo da svi imaju tipski oblik: definišemo koje vrednosti iz opsega primene želimo da vežemo, a zatim definišemo telo funkcije koja se izračunava. Stiče se utisak da bi to možda moglo da se uradi i tako da sve navedemo na istom mestu, nekako nalik na:

```
cout << prebroj(niz, ... n%2 ... ) << endl;
```

Programski jezik *C++*, od verzije 11, ima mogućnost definisanja i korišćenja umetnutih neimenovanih pomoćnih funkcija, koje se uobičajeno nazivaju *lambda funkcije* ili *lambda izrazi*. Osnovna namena lambda izraza je upravo prevazilaženje opisanih problema pri korišćenju funkcija i funkcionala. Lambda izrazi apstrahuju opisane sličnosti između svih funkcionala i omogućavaju da se na mestu upotrebe navede programski kod koji se kasnije ponaša praktično ekvivalentno kao da su upotrebljeni funkcionali. Lambda izrazi omogućavaju da se u programskom kodu, na mestu na kome se očekuje da se navede referenca na neku ranije definisanu funkciju ili na neki funkcijski objekat (funkcional), umesto toga navede upravo definicija funkcije. U pitanju je tehnika koja je uobičajena za funkcionalne programske jezike, a njen naziv potiče iz lambda računa[70].

Lambda izrazi u programskom jeziku *C++* imaju sledeću osnovnu sintaksu:

```
[ <vezani objekti> ] ( <argumenti> ) { <telo> }
```

Argumenti predstavljaju argumente funkcije, a telo predstavlja telo funkcije i navode se na potpuno isti način kao i u slučaju uobičajenog definisanja imenovanih funkcija. Ono što se razlikuje je što (1) ne navodimo naziv funkcije, (2) najčešće ne moramo da navodimo tip rezultata funkcije i (3) možemo da navedemo dodatno vezivanje objekata. Puna sintaksa (od *C++20*) omogućava i definisanje šablonskih lambda izraza, koji imaju određene specifične primene u šablonima, ali to ovde nećemo detaljnije razmatrati.

[70] Lambda račun predstavlja jedan od formalnih načina za definisanje pojma algoritma. Oblikovao ga je Alonso Čerč 1930-ih godina, u cilju formalizacije rukovanja matematičkim izrazima. Jedan od osnovnih elemenata lambda računa je apstrakcija unarne funkcije, tzv. lambda izraz. Na primer, ako bismo imali na raspolaganju odgovarajuće operatore, onda bi se sintaksom lambda izraza funkcija neparan definisala kao: `λx.x%2!=0` [*Church 1936*].

Na primer, ispravna definicija lambda izraza, koji proverava da li je dati neoznačen ceo broj neparan, može da se napiše ovako:

```
[]( unsigned n ){ return n%2; }
```

a njegova primena pri prebrojavanju neparnih elemenata niza bi izgledala ovako:

```
cout << prebroj(niz, [](unsigned n){return n%2;} ) << endl;
```

Naziv funkcije se ne navodi, zato što nije potreban – da jeste, pisali bismo klasičnu funkciju (ili funkcional) a ne lambda izraz. Alternativno, ime može da se uvede i kao ime promenljive funkcijskog tipa, na primer:

```
auto neparan = []( unsigned n ){ return n%2; };
cout << prebroj(niz, neparan ) << endl;
```

Tip rezultata funkcije obično može da se ustanovi automatski, pa zato ne mora da se navodi. Ako tip rezultata funkcije ipak ne može da se nedvosmisleno automatski ustanovi, ili on nije u skladu sa tipom koji se očekuje na mestu primene lambda izraza, onda možemo da eksplicitno odredimo tip rezultata na sledeći način:

```
[ <vezani objekti> ] ( <argumenti> ) -> <tip rezultata> { <telo> }
```

Lambda funkcija iz prethodnog primera ima rezultat tipa `unsigned`. Ako želimo da vratimo logičku vrednost, onda to možemo eksplicitno da naglasimo:

```
[]( unsigned n ) -> bool { return n%2; }
```

Vezivanje objekata

Vezivanje objekata omogućava da se definišu lambda izrazi u čijem telu se ne koriste samo argumenti nego i neka ili sva imena (tj. objekti) koja su vidljiva u opsegu u kome je definisan lambda izraz. Ako se ne navede nijedan objekat niti parametar vezivanja, onda u telu lambda izraza ne može da se koristi nijedan drugi objekat osim onih koji se navode kao argumenti.

U prethodnim primerima smo pravili funkcional `VeciOd`. Za razliku od tog primera, upotreba lambda izraza sa vezivanjem podataka nam daje mnogo kompaktniji programski kod. Umesto da pišemo celu klasu, sada možemo da napišemo samo lambda izraz u kome ćemo naglasiti da je potrebno da se veže objekat `i`, tako što ćemo navesti `i` u spisku vezanih objekata:

```
for( int i=0; i<100; i+=5 )
    cout << i << " : "
         << prebroj(niz,[i](int n){return n>i;}) << endl;
```

Kao što vidimo iz primera, da bi objekat mogao da se koristi u telu lambda izraza, potrebno je da se njegovo ime navede u spisku vezanih objekata. Podrazumevano je da se vezivanje objekata ostvaruje kopiranjem. Ako želimo da se menjanjem vezanog objekta u telu lambda izraza zapravo menja i spolja vezani objekat, onda je potrebno da se vezivanje odvija po referenci. To se postiže tako što se u spisku vezanih imena ispred imena objekta navede „`&`“.

Imajući u vidu veličinu objekata, obično ima smisla da se vezivanje kopiranjem zameni vezivanjem po referenci na konstantan objekat, ali moramo imati u vidu da to nije uvek moguće. Na primer, ako se napravljen lambda izraz vraća kao rezultat funkcije, a vezani su neki privremeni objekti, onda je jasno da tu mora da se izvodi kopiranje, zato što bismo inače koristili reference na objekte koji su u međuvremenu obrisani. Izbor načina vezivanja je važan i kada se radi o većim objektima, zato što neoprezna upotreba vezivanja objekata po vrednosti može da naruši performanse.

Kao što su lambda izrazi bez vezanih objekata konceptualno ekvivalentni funkcijama, iz navedenog primera može da se vidi da su lambda izrazi sa vezivanjem objekata konceptualno ekvivalentni funkcionalima. Zaista, sve što možemo da implementiramo pomoću funkcionala, možemo da implementiramo i pomoću lambda izraza sa vezivanjem objekata i obrnuto[71].

U spisku vezanih objekata može da se navede i „`this`“, čime se postiže da je u telu lambde na raspolaganju pokazivač `this`, koji pokazuje na objekat u čijem se metodu pravi lambda izraz, tj. vezuje se vrednost koju je imao pokazivača `this` u trenutku pravljenja lambda funkcije. Za `this` je podrazumevano je da se objekat prenosi po referenci i da može da se menja, što je i logično, jer se radi o prenošenju pokazivača, a ne objekta. Ako želimo da objekat ne može da se menja, ili da se iz nekog drugog razloga vezuje njegova kopija, onda u spisku vezanih imena umesto `this` mora da stoji `*this`, pri čemu je to moguće tek od *C++17*.

Ako želimo da se vezuju sva raspoloživa imena, onda navodimo „`[=]`“. Tada će sva imena (koja se vide na tom mestu, a koja se koriste u telu lambda izraza) biti vezana po vrednosti, kao kopije. Ako želimo da sva imena budu vezana po referenci, onda navodimo „`[&]`“. Ako se navede „`[=]`“, onda i dalje za neke objekte može da se naglasi da se vezuju po referenci, na primer „`[=,&x]`“.

[71] Naravno, to stoji samo ako pod funkcionalima podrazumevamo klase koje od ponašanja imaju samo konstruktor i operator (). Neki složeniji oblici takvih funkcionala, koji pri računanju menjaju i svoje unutrašnje stanje, nisu mogli da se opišu lambda izrazima u *C++11*. Međutim od uvođenja inicijalizacije vezanih objekata, tj. od verzije *C++14*, čak je i to moguće. Ako funkcionali rade i neke druge stvari i imaju i neke druge metode, onda ne mogu da se implementiraju lambda izrazima.

Od verzije *C++14* uz vezivanje može da se navede i inicijalizacija, što praktično znači da se pravi i inicijalizuje novi objekat sa datim imenom. To je isto kao da se neposredno pre definicije lambda izraza navede odgovarajuća definicija pomoćnog objekta i da se zatim on veže.

Prevođenje lambda izraza

Lambda izrazi, koji ne koriste vezivanje imena, mogu da se prevedu i obično se prevode kao funkcije. U postupku optimizacije takve funkcije mogu da se integrišu u kod funkcije iz koje se pozivaju, ali to ne menja suštinu – upotreba lambda izraza umesto običnih definisanih imenovanih funkcija ne donosi značajne razlike u odnosu na veličinu i efikasnost izvršnog programa. U osnovi ne bi trebalo da bude nikakvih razlika, ali praktično ipak može da bude manjih odstupanja, posebno ako se prevođenje radi u režimu za debagovanje.

Kada je reč o lambda izrazima koji koriste vezivanje objekata, već smo ranije ukazali na konceptualnu ekvivalentnost takvih izraza sa funkcijskim objektima. Zaista, takvi lambda izrazi se prevode kao funkcionali. Svi vezani objekti (ako se zaista i koriste u telu lambda izraza) se implementiraju kao članovi podaci funkcijskih objekata. Ni u ovom slučaju ne bi trebalo da bude razlika u veličini i efikasnosti izvršnog programa, bez obzira na to da li u programu eksplicitno definišemo funkcionale ili koristimo lambda izraze sa vezivanjem objekata.

std::function

Kada pišemo funkciju (ili metod, ili šablon funkcije ili metoda) koja za argument može da ima neku funkciju, lambda izraz ili funkcional, onda u nekim slučajevima u postupku prevođenja može da dođe do problema pri automatskom ustanovljavanju tipova. Problemi nastaju, pre svega, ako pokušavamo da sačuvamo dobijene funkcijske parametre ili ako želimo da kao rezultat funkcije vratimo neku lambda funkciju.

Radi rešavanja tog problema u standardnoj biblioteci je definisan šablon klase `std::function`, koji omogućava da se ujednači rad sa svim vrstama „funkcija“, tj. svih objekata na koje može da se primeni operator aplikacije „`()`“ sa odgovarajućim brojem i tipom argumenata. Parametar šablona je suštinski tip funkcije, a ne stvarni tip objekta sa kojim radimo. Suštinskim tipom funkcije smatramo osnovni tip funkcije poput koje se naš tip ponaša. Na primer, ako pravimo funkcional koji se ponaša kao funkcija koja preslikava dva cela broja u logičku vrednost, onda je suštinski tip funkcije za taj funkcional „`bool(int,int)`“.

U sledećem primeru pravimo funkcional koji poredi dati broj sa 5 i čuvamo ga pod imenom `veci0d5`. Suštinski tip tog funkcionala je „`bool(int)`“:

```
std::function<bool(int)> veci0d5 = [](int x){ return x>5; };
```

11.11 Koncepti

Rad sa šablonima može da bude neugodan zato što prevođenje zna da dugo traje, a poruke o greškama zahtevaju pažljivo čitanje i razumevanje. Uzrok većine grešaka sa šablonima je što se pokušava njihova primena na neke konkretne tipove sa kojima iz nekog razloga ne rade. Pri tome, iz deklaracije šablona, posebno ako su složeni, najčešće nije očigledno koje uslove moraju da zadovolje tipovi da bi mogli da se koriste pri instanciranju tipskih parametara. Da bi se taj problem rešio uvedeni su *koncepti*.

Koncepti u programskom jeziku C++ imaju sličnu ulogu kao, na primer, klase tipova u Haskelu. Oni služe da opišemo neke klase (ili grupe, skupove) tipova i da zatim pri definisanju šablona ne navodimo uobičajene opšte tipske parametre, nego da umesto toga označimo da parametar mora da pripada nekoj opisanoj klasi tipova, ili da, prema terminima programskog jezika C++, *zadovoljava dati koncept*.

Koncept se definiše u obliku:

```
template<...>
concept ime = izraz;
```

gde `ime` predstavlja ime koncepta a `izraz` je logički izraz koji navedeni tipski parametri moraju da zadovolje da bi bili smatrani delom tog koncepta. Na primer, koncept `IzvedenaIz<A,B>` će važiti ako je klasa `A` izvedena iz klase `B`:

```
template< class A, class B >
concept IzvedenaIz = std::is_base_of_v<B,A>;
```

Nakon što se napravi, koncept može da se koristi pri pravljenju šablona. Može da se navodi odmah iza klauzule deklarisanja tipskih parametara, ili na kraju dela deklaracije, a pre definicije. Koncepti se upotrebljavaju u obliku klauzule `requires` i izraza `requires`.

Klauzula `requires` ima oblik:

```
requires izraz
```

gde je `izraz` neki konstantan logički izraz (izračunljiv u fazi prevođenja) i predstavlja uslov koji mora da bude ispunjen.

Naredni primer naglašava da klasa `T1` mora da bude izvedena iz `T2`:

```
template<class T1, class T2> requires IzvedenaIz<T1,T2>
...
```

Izraz `requires` ima logičku vrednost i koristi se u jednom od dva oblika:

```
requires { izraz }
requires ( argumenti ) { izraz }
```

gde `izraz` predstavlja neki izraz (ili naredbu ili niz naredbi) programskog jezika, kome eventualno prethode deklaracije tipova nekih imena koja se pojavljuju u izrazu, i naglašava da taj izraz mora da bude prevodiv za konkretne tipove. Izraz `requires` se koristi i pri definisanju koncepata.

Naredni primer definiše koncept `ImaSabiranje` koji obuhvata sve tipove koji imaju operator sabiranja:

```
template<class T>
concept ImaSabiranje = requires ( T x ){ x + x; };
```

Uslovi koji se definišu, pa tako i odgovarajući koncepti, mogu da budu relativno složeni. Standardna biblioteka je zbog toga dopunjena velikim brojem unapred pripremljenih koncepata i još većim brojem šablonskih predikata (šablona klasa koji imaju statičku logičku vrednost `value` izračunljivu u toku prevođenja, kao na primer `std::is_base_of<B,A>`).

Upotreba koncepata omogućava prevodiocima da pri instanciranju nekog šablona, pre njegovog prevođenja najpre provere da li su ispunjeni uslovi odgovarajućih koncepata. Ako nisu, onda se prevođenje i ne pokušava, već se programeru saopštavaju vrlo čitke i jasne poruke o greškama.

11.12 Generički tipovi u Javi i *C#*-u

Pod uticajem programskog jezika *C++* i tehnika programiranja u nekim funkcionalnim programskim jezicima, nekim savremenim programskim jezicima je naknadno dodata podrška za mehanizme koji liče na šablone programskog jezika *C++*.

Programski jezik Java od verzije 5 ima podršku za generičke tipove (ili *generičke parametre*, engl. *generics*). Generički tipovi u Javi su razvijeni u velikoj meri po uzoru na šablone u programskom jeziku *C++*, ali se od njih suštinski razlikuju. Dok šabloni u programskom jeziku *C++* predstavljaju implementaciju suštinskog parametarskog polimorfizma, generički tipovi u Javi imaju pre svega sintaksnu primenu, kako bi se jedan broj grešaka u kodu uočio već u fazi prevođenja programa, umesto u fazi njegovog izvršavanja.

Da bismo razumeli kontekst, podsetimo se da su kolekcije i mnoge druge stvari u Javi izvorno radile samo sa uopštenom osnovnom klasom `Object`, pri čemu je često bilo neophodno da se vrši eksplicitna konverzija tipova. Na primer, naredni isečak

programa bi se uspešno preveo, ali bi proizveo grešku pri izvršavanju, zato što bi došlo do neuspešnog pokušaja konverzije niske u ceo broj:

```
List v = new ArrayList();
v.add("niska");
Integer i = (Integer)v.get(0);  // Greška u fazi izvršavanja
```

Osnovni cilj uvođenja generičkih tipova u Javu je bio da se spreče takvi problemi, tako što bi se omogućilo da se u kodu eksplicitno navode tipovi elemenata kolekcija, kao i da zatim ne moraju da se eksplicitno navode konverzije tih elemenata:

```
List<String> v = new ArrayList<String>();
v.add("niska");
Integer i = v.get(0);           // Greška u fazi prevođenja
```

Posledica takvog pristupa je da se generički tipovi ponašaju značajno drugačije od šablona i imaju mnogo manje mogućnosti. Navešćemo neke od najvažnijih razlika:

- Generički tipovi u Javi nisu uvedeni sa ciljem omogućavanja punog generičkog polimorfizma, već prvenstveno sa ciljem olakšavanja pisanja koda i prevencije određenih vrsta grešaka;
- Čak i tako neke greške ipak mogu da se dešavaju u fazi izvršavanja[72];
- U programskom jeziku *C++* šabloni se prevode tek pri instanciranju, a parametarski tip se ne koristi za ozbiljnije sintaksne i semantičke provere, dok se generički tipovi u Javi proveravaju i koriste pre svega u fazi prevođenja;
- Da bi uspelo prevođenje generičkog koda u Javi, ako se na generičkim tipovima primenjuju neki metodi, onda mora biti naznačeno koja su postojeća ograničenja (tj. implementirani interfejsi ili nasleđene klase) za te generičke tipove;
- Prevođenje generičkog koda u Javi se odvija uz tzv. *brisanje tipova*. Prevod generičkog koda će biti potpuno isti kao da je svuda upotrebljena uopštena klasa `Object` ili druga odgovarajuća bazna klasa koja je navedena kao ograničenje, uz odgovarajuće eksplicitne konverzije[73];

[72] Na primer, objekat tipa `List<String>` može da se dodeli objektu tipa `List`, koji se zatim koristi kao da sadrži brojeve.

[73] Korisna posledica ovakvog pristupa je da se prevodi generičkog koda mogu upotrebljavati u programima pisanim na starijim verzijama Jave, naravno, ako osim generičkih tipova nisu korišćene druge specifičnosti novih verzija.

- Svaka instanca šablona u *C++*-u se prevodi posebno i proizvodi poseban prevod, tako da se i statički podaci šablona klasa u *C++*-u razlikuju za svaku konkretnu instancu. Sa druge strane, generičkom tipu u Javi odgovara samo jedan deljeni prevod pa se i statički podaci dele za sve primene generičkog tipa. Štaviše, statički podaci u Javi zbog toga ne smeju da imaju generički tip;
- Prevod šablona u *C++*-u će biti drugačiji i posebno optimizovan za svaki konkretan tip, dok se generički kod u Javi ni na koji način niti posebno prevodi niti posebno optimizuje za konkretan tip;
- U Javi se instanciranje može obavljati samo klasama (naslednicima uopštene klase `Object`), dok se u *C++*-u mogu upotrebljavati proizvoljni tipovi;
- U Javi ne postoje vrednosni parametri, već samo tipski.

Kao što vidimo, iako ima sintaksnih sličnosti, suštinske razlike su zapravo mnogo veće, pa se sa pravom može reći da generički tipovi u Javi predstavljaju pre sintaksnu olakšicu nego punu implementaciju parametarskog polimorfizma.

Slično Javi, i programskom jeziku *C#* je u verziji 2 dodata podrška za generičke tipove. U osnovi se radi o sličnom konceptu kao u slučaju Jave. Sintaksa je veoma slična, a razlike se uglavnom odnose na način prevođenja. U slučaju programskog jezika *C#* podrška za generičke tipove je izvedena na nivou virtualne mašine, a ne samo na nivou prevodioca. Zbog toga prevođenjem može da se dobije različit kod za različite instance generičkih tipova, što omogućava sprovođenje određenih optimizacija. Ipak, i pored toga su generički tipovi u jeziku *C#* mnogo bliži istoimenom konceptu kod Jave nego šablonima u jeziku *C++*. Neke od najvažnijih razlika u odnosu na *C++* su:

- Kao i u slučaju Jave, generički tipovi u *C#*-u dopuštaju samo tipske parametre;
- Nije podržana eksplicitna ni parcijalna specijalizacija;
- Tipski parametar ne može da se koristi kao bazna klasa generičke klase;
- Ne postoje podrazumevane vrednosti parametara;
- U slučaju *C#*-a programski kod mora da može da se ispravno prevede za sve vrednosti tipskih parametara koje zadovoljavaju zadato ograničenje.

11.13 Haskel

Razmatranje polimorfizma bi ostalo nepotpuno ako bismo propustili da pomenemo programski jezik *Haskel* (engl. *Haskell*). Haskel je čisto funkcionalan

programski jezik sa lenjom semantikom, koji je po mnogo čemu veoma napredan. Mnogo elemenata Haskela je vremenom pronašlo put do drugih programskih jezika i široke primene u praksi, ali ono što nam je ovde posebno važno jeste Haskelov sistem tipova i rad sa tipovima.

Polimorfizam u Haskelu počiva na automatskom raspoznavanju tipova i statičkoj analizi tipova. Automatsko raspoznavanje tipova nam omogućava da programski kod pišemo uglavnom bez eksplictnog navođenja tipova. Navođenje tipova argumenata i rezultata pri definisanju funkcija predstavlja opcioni vid dokumentovanja i dodatne provere saglasnosti.

Haksel ima *klase*, ali njegove klase ne predstavljaju klase objekata, kao u OOP, nego se radi o *klasama tipova*. Opisivanjem klase tipova se određuje koje osobine neki tipovi moraju da imaju da bi mogli da se svrstaju u odgovarajuću klasu tipova. Klasa tipova predstavlja apstraktnu specifikaciju tipova, odnosno vid opisa interfejsa tipa, ali može da ima i elemente njegove strukture. Za razliku od klasičnih interfejsa u OO programskim jezicima, klase tipova se definišu u odnosu na izabrane parametre i omogućavaju daleko veću fleksibilnost pri definisanju.

U tom smislu, klase tipova u Haskelu su veoma slične konceptima u C++-u. Oba mehanizma služe za parametarsko definisanje polimorfnih interfejsa (i strukture) i omogućavaju veoma veliku slobodu i fleksibilnost programeru, a uz istovremeno pružanje solidne podrške u automatizovanju provera saglasnosti tipova i ispravnosti programskog koda. Uvođenje koncepata u C++ predstavlja značajan doprinos u isticanju značaja parametarskog polimorfizma i tehnika programiranja koje se usvajaju iz funkcionalnih programskih jezika.

Naravno, sama priroda programskih jezika se veoma značajno razlikuje. Filozofija jezika i način prevođenja su potpuno različiti. Na primer, Haskel kao jedan od primarnih ciljeva ima modularnost, dok je kod C++-a na vrhu prioriteta efikasnost, zbog čega instanciranje (tj. prevođenje) šablona praktično ne može da se obavlja na nivou modula, već tek na mestu upotrebe. Koncepti ne predstavljaju funkcionalan deo sistema tipova, već samo pomažu pri automatskom raspoznavanju i proveravanju tipova. Sa druge strane, klase tipova su sastavni deo sistema tipova u Haskelu.

Detaljno razmatranje i upoređivanje karakteristika programskih jezika C++ i Haskel je veoma zanimljiva tema, ali bi nam uzelo previše prostora. Toplo preporučujemo čitaocima da upoznaju Haskel [*Lipovača 2011*]. Više informacija o poređenju karakteristika C++-a i Haskela može da se pronađe u [*Bernardy 2008*].

11.14 Umesto zaključka

Polimorfizam ima veoma važno mesto u savremenom razvoju softvera. Značaj polimorfizma je u tome što nam omogućava značajno podizanje nivoa apstrakcije

programskog koda. Apstraktniji programski kod može da se upotrebi u više konkretnih slučajeva, pa primena polimorfizma neposredno povećava upotrebljivost napisanog koda. Polimorfizam nam omogućava uopštavanje koncepata poput funkcionala ili iteratora, kao i pravljenje daleko upotrebljivijih biblioteka kolekcija i biblioteka uopštenih algoritama.

Savremeni programski jezici se u velikoj meri oslanjaju na upotrebu polimorfizma. Svi programski jezici koji su bar delimično objektno-orijentisani, imaju i neki vid hijerarhijskog polimorfizma. Dinamički jezici su obično slabo tipizirani i ne proveravaju tipove unapred nego tek u fazi izvršavanja programa, što kod većine takvih jezika ima za posledicu mogućnost primene implicitnog polimorfizma. Veliki broj funkcionalnih programskih jezika, uključujući i one koji su strogo tipizirani i proveravaju ispravnost i saglasnost tipova u fazi prevođenja, takođe podržavaju implicitni polimorfizam. Videli smo i da *C++* ima punu podršku za parametarski polimorfizam, kao i da ga Java i *C#* podržavaju u izvesnoj meri.

Veliki podsticaj razvoju savremenih tehnika implementacije i primene parametarskog polimorfizma predstavljala su istraživanja i publikacije Andreja Aleksandreskua, a pre svega njegova knjiga [*Alexandrescu 2001*]. Njegovi rezultati su imali značajan uticaj i na kasnija unapređenja standarda programskog jezika *C++*. Andrej se 2007. godine priključio Volteru Brajtu na razvoju programskog jezika *D*, koji se u velikoj meri zasniva na daljem unapređenju koncepata polimorfizma [*D*].

Čitaocima koji su zainteresovani za teorijske osnove polimorfizma i automatizacije izvođenja tipova preporučujemo da izučavanje započnu od radova [*Cardelli 1985*] i [*Cook 2009*].

12 - Debagovanje

Debagovanje je dvostruko teže
nego pisanje programa koje mu prethodi.
Znači, ako pišete program najpametnije što umete,
onda, po definiciji, niste dovoljno pametni da ga debagujete.
Brajan Kernigen

12.1 Greške

Razvoj softvera je složen proizvodni proces. Kao i u svakom drugom proizvodnom procesu, i u toku razvoja softvera neminovno dolazi do pravljenja određenih propusta, koji se ispoljavaju na različite načine. U zavisnosti od faze razvijanja softvera u kojoj nastaju, propusti se mogu značajno razlikovati po složenosti, prikrivenosti i načinu ispoljavanja.

12.1.1 Greška, propust ili bag

Propusti u razvoju softvera se često nazivaju *greškama* ili *bagovima*. Primetimo da postoji određen problem sa upotrebom termina *greška*, zato što je u praksi prilično uobičajeno da se pojam procesa razvoja softvera izjednačava sa njegovim konačnim proizvodom – softverom, kao što se i konačan proizvod često povezuje sa jednom od završnih faza izrade – programiranjem. Kada se kaže da u nekom programu postoji *greška* ili *bag*, obično se to povezuje sa propustima u programiranju. Međutim, termin *greška* u razvoju softvera ima mnogo šire značenje i obuhvata sve one propuste koji mogu da nastanu u svim fazama razvoja softvera, kao što su, na primer, analiza zahteva, planiranje, projektovanje ili programiranje.

U prvim odeljcima ovog poglavlja ćemo se baviti svim vrstama grešaka, a ne samo greškama u fazi programiranja, pa ćemo ravnopravno koristiti termine *greška* i

propust. Kasnije, kako se budemo više posvećivali konkretnom problemu lociranja i otklanjanja grešaka u programskom kodu, *greške* će postepeno istisnuti *propuste* iz teksta.

Propusti mogu da imaju različite vrste posledica. *Propustom* smatramo sve ono što ima za posledicu *neispravno ponašanje* softvera. U zavisnosti od konteksta, *neispravno ponašanje* može da se ispolji kao izračunavanje neispravnog rezultata, nedopustivo veliko trajanje izračunavanje, neprihvatanje novih zahteva korisnika, neizdavanje rezultata i drugo.

Najuopštenije posmatrano, propust u razvoju softvera (greška, bag) je sve ono što stvara probleme u funkcionisanju softvera kao završnog proizvoda. Često se smatra da propusti predstavljaju sve ono što ima za posledicu da se softver ne ponaša u skladu sa specifikacijom[74].

12.1.2 Dinamička priroda grešaka

Softver ima izrazito dinamičnu prirodu. Sve vreme tokom izvršavanja programa odvijaju se složene, međusobno povezane i uslovljene operacije promene stanja programa i računarskog sistema. Ako svedemo posmatranje na pojedinačne atomične promene stanja, onda možemo da kažemo da je svaka od njih izrazito jednostavna, ali da ih je neverovatno mnogo – svake sekunde po nekoliko miliona ili čak milijardi. Sa druge strane, ako ih posmatramo po grupama, kao nešto veće složene promene stanja, onda se njihov broj smanjuje, ali nam njihova unutrašnja složenost otežava praćenje i analizu.

Posledica takve prirode softvera je da i greške u softveru imaju dinamičku prirodu. Andreas Zeler [*Zeller 2006*] je dinamičku prirodu grešaka pokušao da opiše prolaskom kroz četiri glavne faze od nastajanja do ispoljavanja greške. Taj put se obično naziva sekvencom *kvar-infekcija-neuspeh*:

1. Programer[75] piše neispravan programski kod. Neispravnost u programu predstavlja kvar (engl. *defect*), koji je zatim uzročnik *infekcije*.
2. Kvar prouzrokuje infekciju. Sa izvršavanjem programa izvršava se i neispravan deo programa, što kao posledicu ostavlja za sobom neispravno stanje programa.

[74] Ni ova neformalna definicija nije savršena, zato što ne može da se primeni na propuste koji nastaju tokom izrade specifikacije i prethodnih faza.

[75] Neispravnost u programski kod neposredno unosi programer. Ali to ne znači da je programer kriv, zato što, kao što smo već videli, to može biti posledica greške u projektovanju, ili čak u definisanju zahteva. U svakom slučaju, na kraju dobijamo neispravan programski kod, u kome imamo bag ili .

3. Infekcija se prenosi. Neispravno stanje programa utiče na izvršavanje drugih delova programa. Posledica je da se neispravnosti postepeno pojavljuju i u drugim elementima stanja programa.
4. Infekcija izaziva neuspeh. Neispravno stanje programa na kraju dovodi do spolja vidljive neispravnosti u ponašanju programa.

Svaka od ovih faza ima svoje specifičnosti. Najpre, u prvom koraku, kada se pri pisanju programa napravi greška, ona vrlo često ne ide sama, već se u njenoj neposrednoj blizini često nalazi još neka greška. Bagovi često nastaju zbog nepažnje ili nerazumevanja problema i okolnosti u koijma će pisani deo programa da radi, pa ako programer pri pisanju nekog dela programa ne pazi ili nešto ne razume, onda je vrlo verovatno da će da napravi više bagova, a ne samo jedan.

Put infekcije je vrlo retko linearan – pre bi se moglo reći da ima prirodu eksponencijalnog širenja, nalik na grananje drveta ili grafa. Ako se neispravan programski kod izvršava više puta, na različitim podacima, onda je sasvim moguće da jedna greška neposredno proizvede mnogo infekcija, ostavljajući veći broj pojedinačnih neispravnih elemenata stanja programa. Koliko god čudno izgledalo, takvi slučajevi su poželjniji, zato što se tada infekcija vrlo brzo širi i skoro neminovno dolazi do skorog neuspeha i uočavanja da postoji problem. Nasuprot tome, mnogo su nezgodniji slučajevi kada se infekcija prenosi povremeno i sporo, zato što takvi problemi mogu da dugo ostanu neprimećeni.

Slično je i u trećem koraku – kao što greške nemaju istu stopu izazivanja infekcije, tako se i infekcije različitom brzinom prenose i manifestuju. Jedan broj infekcija se zadržava u unutrašnjem stanju programa i veoma retko se prenosi na spoljašnje elemente stanja i izaziva neuspehe. To je slično kao slučaj kvara koji retko izaziva infekcije – ponovo imamo problem koji će dugo ostati neprimećen.

Ovakva priroda grešaka ima za posledicu da skoro svaki put kada se bavimo otkrivanjem grešaka, čiji su neuspesi uočeni, moramo da se bavimo ne samo traženjem i izučavanjem grešaka, nego najpre traženjem i izučavanjem *infekcija* i puteva infekcija. Iako ćemo u daljem tekstu uglavnom govoriti o greškama u kodu, zapravo se podrazumeva da svaki put kada proveravamo ispravnost ponašanja delova programa mi zapravo proveravamo ispravnost *stanja* programa, tj. proveravamo da li ima nekih znakova infekcije. Svaka neispravnost (osim možda slabih performansi) se u nekom trenutku manifestuje kao neispravan podatak. Neispravni podaci su osnovni vid kvarova i infekcija, a oni čine neispravno stanje programa.

Ispravnost stanja programa se praktično identifikuje sa odsustvom infekcije, ili da budemo još precizniji – pretpostavka ispravnosti stanja programa je ekvivalentna sa odsustvom uočenih infekcija. Kao što ne znamo da li ima nekih skrivenih infekcija,

tako najčešće ne znamo ni da li je stanje programa ispravno, već to možemo samo da pretpostavljamo[76].

12.1.3 Vrste propusta

U literaturi se sreću različite klasifikacije propusta. Jedna od uobičajenih deli propuste prema načinu ispoljavanja na:

- nekonzistentnosti u korisničkom interfejsu;
- neispunjena očekivanja;
- slabe performanse;
- padove softvera i oštećenja podataka i
- bezbednosne propuste.

Nekonzistentnosti u korisničkom interfejsu

Nekonzistentnosti u korisničkom interfejsu se odnose na različite vrste neusklađenosti između onoga što korisnik želi da uradi, načina na koji mora da prenosi svoje zahteve softveru, načina na koji softver izvršava zahtevanu operaciju i načina na koji softver izveštava korisnika o toku ili rezultatu operacije.

Na primer, većina programa za *MS Windows* koristi prečicu *Ctrl+F* za započinjanje operacije traženja. Međutim, neke verzije programa *MS Outlook* su tu prečicu koristile za prosleđivanje primljene elektronske poruke (engl. *forward*). To je dovodilo do neusklađenosti između očekivanja korisnika, koji želi da pronađe neki tekst u poruci, i ponašanja programa, koji započinje prosleđivanje poruke.

Propusti koji se manifestuju kao nekonzistentnosti u korisničkom interfejsu obično imaju uzroke u lošoj specifikaciji. Ona obično nastaje zbog loše urađene analize u fazi planiranja i projektovanja korisničkog interfejsa. Dobro sredstvo za prevenciju ove vrste problema jesu prototipovi. Oni pomažu da se u ranim fazama razvoja uoče eventualni konceptualni problemi sa korisničkim interfejsom.

Neispunjena očekivanja

Dobijanje neočekivanog (tj. neispravnog) rezultata predstavlja verovatno najneugodniju vrstu propusta. Neispravni rezultati mogu da imaju različite oblike:

[76] Ovde ne razmatramo formalnu verifikaciju programa i dokazivanja ispravnosti, ali čak i tada postoji mogućnost da su neispravni neki elementi specifikacije ili neke pretpostavke, pa da zbog toga i dalje postoje neispravnosti i u gotovom proizvodu.

- neispravan numerički ili drugi rezultat – izračunata vrednost nije ispravna;
- neispravno reagovanje na akciju korisnika ili neispravan prikaz u korisničkom interfejsu;
- pogrešan pozitivan rezultat – dobijen je potvrdan odgovor umesto ispravnog odričnog odgovora i
- pogrešan negativan rezultat – dobijen je odričan odgovor umesto ispravnog potvrdnog odgovora.

U opštem slučaju, sve probleme ovog tipa možemo da svedemo na prvi oblik – svaki drugi oblik je samo drugačije posmatran rezultat izračunavanja. Suština je u tome da je specifikacijom programa definisano da će on nešto da izračuna, a u praksi program to računa pogrešno ili računa nešto sasvim drugo.

Uzroci neispunjenih očekivanja mogu da se nalaze u različitim fazama razvoja softvera, pa njihovo pronalaženje i otklanjanje može da bude veoma složeno. Propusti ovog tipa mogu da nastanu već u fazi definisanja zahteva, kada zbog propusta u komunikaciji može da dođe do nerazumevanja između klijenta i razvojnog tima. Ako se taj deo posla uradi kako valja, onda ova vrsta problema može da nastane u skoro svakoj od faza analize i projektovanja softvera, pa da se kasnije ispolji u obliku propusta u projektovanju ili propusta u dokumentovanju određenih faza projekta. Na kraju, čak i kada je projektovanje urađeno u potpunosti ispravno, do problema može da dođe pri pisanju programa (tj. pri kodiranju) u vidu grešaka programera (neispravan odabir ili implementacija algoritma, neispravno čitanje ili tumačenje projektne dokumentacije,...).

Slabe performanse

Slabe performanse se obično ispoljavaju kao stalno ili povremeno sporo reagovanje programa na zahteve korisnika, ili kao preterano zauzeće resursa računarskog sistema tokom rada programa. Iako same po sebi ne moraju da utiču na ispravnost konačnog rezultata, slabe performanse mogu da imaju presudan uticaj na upotrebljivost softvera.

Uzroci slabih performansi se nalaze u svim fazama razvoja softvera, od početnog procenjivanja opterećenja i količine raspoloživih resursa, preko projektovanja i do samog kodiranja. Ako su uzroci slabih performansi u kasnijim fazama projektovanja ili u kodiranju, onda često mogu da se otklone primenom bolje prilagođenih algoritama ili tehnika (npr. upotreba više niti), ali ako su greške načinjene u fazama procenjivanja i planiranja opterećenja ili ranim fazama projektovanja, onda se mnogo teže prevazilaze.

Problemu slabih performansi i načinima unapređenja performansi ćemo se opsežnije posvetiti u poglavlju *13 - Optimizacija softvera*.

Padovi softvera i oštećenja podataka

U najopasnije vrste propusta spadaju oni koji se manifestuju različitim vidovima *padova softvera* ili oštećenja podataka. Pod padom softvera se podrazumevaju svi različiti slučajevi prekida rada programa, bilo da se radi o neplaniranom zatvaranju programa ili prestanku prijema komandi korisnika. Nisu retki slučajevi da pad jednog programa izaziva pad i nekih drugih programa ili čitavog operativnog sistema. Takvo indukovanje širokih padova najčešće je povezano sa velikim zauzećem sistemskih resursa od strane nestabilnog programa. Nisu svi operativni sistemi jednako otporni na ovu vrstu problema. Operativni sistemi predviđeni primarno za interaktivan rad korisnika (tzv. desktop operativni sistemi) mnogo slabije izoluju i podnose ovakve probleme nego sistemi koji su specijalizovani za pružanje usluga (tzv. serverski operativni sistemi).

Oštećenja podataka su posebno neugodna. Za razliku od padova sistema, oštećenja podataka mogu da ostanu prikrivena, pa čak i da se veoma dugo dešavaju neprimećeno, a sa potencijalno veoma ozbiljnim posledicama. Ako se takve neispravnosti uoče tek posle dužeg vremenskog perioda, onda njihovo nagomilavanje može da proizvede veoma ozbiljne posledice po ukupnu ispravnost podataka i ispravnost rada čitavog sistema, pa prevazilaženje takvih problema može da bude veoma složeno, dugotrajno i skupo.

Karakteristika ovih propusta je da, osim u fazama projektovanja i programiranja, oni mogu nastati i u fazama povezivanja različitih komponenti složenih sistema. Bilo da je reč o neispravnoj primeni protokola povezivanja, upotrebi neispravnih protokola ili nekim drugim propustima, za većinu propusta koji nastaju u ovoj fazi je zajedničko da predstavljaju posledicu neispravne, nepotpune ili neispravno upotrebljavane dokumentacije.

Naravno, ovakvi problemi mogu da budu i posledice grešaka u kodiranju. Ako greške u kodiranju dovode do neispravnog izračunavanja i zatim neispravnog menjanja podataka, onda se problem rešava kao i u drugim slučajevima neispunjenih očekivanja. Sa druge strane, ako greška u kodu neposredno dovodi do neplaniranog menjanja podataka ili čak do pada sistema, onda se obično radi o nekontrolisanom pristupu pogrešnim memorijskim lokacijama, pa se takva greška relativno lako ispravlja, kada se jednom pronađe, ali njeno pronalaženje može da bude veoma teško.

Bezbednosni propusti

Bezbednosnim propustima smatramo sve propuste koji neposredno ili posredno mogu da dovedu do ugrožavanja bezbednosti računarskog sistema na kome će da se izvršava softver koji razvijamo. Predmet ugrožavanja bezbednosti mogu da budu poverljivost, integritet i raspoloživost informacija u okviru računarskog sistema.

Uzroci ugrožavanja mogu da budu namerni napadi, ali i slučajne nezgode. U skladu sa tim i karakteristike bezbednosnih propusta mogu da budu veoma različite.

Bezbednosni propusti koji mogu da dovedu do neplaniranih oštećenja ili gubitaka podataka se uglavnom ne razlikuju mnogo od drugih propusta sa sličnim posledicama. Obično se radi o nedovoljnom staranju o integritetu podataka i softvera ili o problemima u kodiranju. Jedan od glavnih načina sprečavanja takvih propusta je pisanje robusnog programskog koda[77].

Nasuprot tome, bezbednosni propusti koji omogućavaju ili olakšavaju namerne napade ili slučajno i neplanirano narušavanje poverljivosti, obično imaju specifične karakteristike. Njihova manifestacija je uobičajeno manje očigledna nego u slučaju drugih vrsta propusta, zato što oni najčešće ne ometaju korisnika u radu (ili bar ne neposredno). Zbog toga se oni teže uočavaju. Štaviše, relativno se često dešava da nedobronamerni korisnici koji ih uoče to prećute, kako bi možda kasnije pokušali da ih zloupotrebe.

Zbog njihovog otežanog uočavanja, akcenat u radu sa bezbednosnim propustima se stavlja na prevenciju u toku razvoja i na detaljno praćenje upotrebe softvera i detektovanje eventualnih problema u fazi upotrebe softvera. Da bi se napravio bezbedan softver neophodno je sistematično posvećivanje bezbednosnim rizicima, praćeno strogom disciplinom pri razvoju.

Jednom kada se bezbednosni problem uoči, onda se njegovo rešavanje odvija na sličan način kao i otklanjanje drugih vrsta propusta.

Dobar uvod u pisanje bezbednog softvera predstavlja knjiga [*Kohnfelder 2021*]. Detaljnija analiza specifičnih bezbednosnih problema pri upotrebi programskih jezika *C* i *C++* i tehnike za njihovu prevenciju su izloženi u knjizi [*Seacord 2013*].

12.1.4 Upravljanje propustima

Različiti razvojni timovi imaju različita iskustva sa propustima. U nekom timu se propusti češće prave, u nekom drugom se lakše pronalaze, a u nekom trećem se lakše otklanjaju. Ozbiljnost propusta takođe nije ujednačena. Ako analiziramo različite timove, način rada u njima, kao i način na koji svaki pojedinačan član tima pristupa svom poslu, onda možemo da uočimo mnogo razlika u različitim posmatranim oblastima. Razlike postoje u kadrovskoj politici (sastav timova, način odabira članova tima,...), načinu upravljanja timom (organizaciona struktura, podela odgovornosti,...), primenjenoj razvojnoj metodologiji (koncepcija rada, fokus, skup metoda i tehnika,...), konkretnim primenjenim alatima (programski jezici, prevodioci, alati za izgradnju i testiranje,...) i drugim aspektima koji utiču na

[77] O testiranju robusnosti koda je bilo reči u odeljku *9.2 Testovi jedinica koda*.

svakodnevni rad u timu. Čak i različiti članovi istog tima često imaju sasvim različita iskustva u vezi sa greška i propustima. Neko greši manje a neko više, neko pravi ozbiljnije a neko bezbolnije propuste, neko ih pronalazi i ispravlja relativno lako, a nekome to predstavlja veliki napor.

Zbog toga postoji potreba da se rad sa propustima sistematizuje i unapređuje na nivou čitavog tima. Sve one aktivnosti koje za predmet bavljenja imaju propuste ili neke teme koje su povezane sa propustima, čine tzv. *upravljanje propustima*. Značaj upravljanja propustima je često presudan za kvalitet razvojnog procesa i konačnog proizvoda i predstavlja deo aktivnosti koje se šire nazivaju *obezbeđivanjem kvaliteta*.

Poslovi koji čine upravljanje propustima mogu prema vremenu odvijanja da se podele na dva velika skupa poslova – na prevenciju propusta i otklanjanje propusta. Prevencija propusta se zasniva na stalnom staranju o okolnostima koje imaju neposredan ili posredan uticaj na nastajanje propusta i njihovo kasnije otklanjanje. Odvija se tokom čitavog razvoja, od prvih koraka na definisanju i preciziranju ciljeva i zahteva, pa sve do puštanja sistema u rad. Prevencijom propusta ćemo se baviti u završnom delu ovog poglavlja. Otklanjanje propusta se odvija tokom čitavog perioda razvoja, ali i kasnije, tokom perioda održavanja softvera. Obuhvata sve aktivnosti koje imaju za cilj pronalaženje uzroka uočenih problema i otklanjanje pronađenih uzroka problema.

12.2 Otklanjanje grešaka

Otklanjanje grešaka iz programskog koda se među programerima uobičajno naziva *debagovanje* (engl. *debugging*). Doslovni prevod engleskog termina na srpski jezik bi mogao da bude *dezinsekcija* ili *uklanjanje buba*, ali su ipak uobičajeni termini *otklanjanje grešaka* i *debagovanje*.

Kao i svaki drugi složen posao, debagovanje je mnogo lakše ako mu se pristupi sistematično. Iako ponekad može da se stekne utisak da je neko „pravi talenat“, a neko „potpuni antitalenat“ za debagovanje, te da sam proces debagovanja u velikoj meri zavisi od ličnih sklonosti ili intuicije i osećaja, stvari ipak stoje malo drugačije. Tačno je da lične sposobnosti i sklonosti mogu da imaju nezanemarljiv uticaj na kvalitet i efikasnost debagovanja, ali je značaj sistematičnog pristupa i obučenosti razvijalaca ipak daleko veći.

Debagovanje od programera zahteva visok nivo stručnosti i dobro poznavanje problema, primenjenih algoritama, programskog jezika, svih povezanih delova sistema, a uz sve to još i sistematičnost, sposobnosti dobrog usmeravanja svoje pažnje na problem, sagledavanja problema iz različitih uglova i kritičkog razmišljanja i sklonost kako analitičkom tako i konstruktivnom razmišljanju. Na sve to valja dodati i poznavanje pravila i tehnika debagovanja, kojima ćemo u ovom delu teksta posvetiti posebnu pažnju.

Najvažnije aktivnosti u okviru debagovanja su:

- uočavanje da propust postoji;
- analiziranje i razumevanje propusta;
- lociranje propusta i
- ispravljanje propusta.

Uočavanje neispravnosti se najčešće dešava u okviru testiranja softvera, kao deo širih aktivnosti u oblasti prevencije propusta ili obezbeđivanja kvaliteta, ali se neispravnosti često uoče i tek pri upotrebi završenog i isporučenog softvera. Uočavanje neispravnosti u suštini nije predmet debagovanja, ali se tokom debagovanja elementi softvera veoma pažljivo posmatraju, pa se često dešava da se tokom rada na otklanjanju jednog problema uoče i evidentiraju i neki drugi problemi. Zato se uočavanje i evidentiranje propusta ne svrstava uvek u aktivnosti u oviru debagovanja.

Osobe kojima debagovanje ide od ruke
su obično one koje su, svesno ili nesvesno,
usvojile i primenjuju osnovna pravila debagovanja.
Dejvid Agans

U idealnom slučaju, pri uočavanju propusta se on detaljno evidentira. Navode se tačne okolnosti koje su dovele do uočene neispravne manifestacije, što obuhvata tačnu verziju softvera, ulazne podatke, niz koraka koji je doveo do ispoljavanja problema, mesto i vreme i sve drugo što bi moglo da pomogne. Međutim, ako imamo u vidu da uočavanje problema često dolazi od strane osoba koje ne poznaju internu strukturu softvera i tokove podataka, već se prvenstveno bave pitanjima upotrebe i funkcionalnosti, onda će nam biti jasno da prijavljeni opis manifestacije problema često nije dovoljno detaljan.

Naredni korak, koji se često smatra i za prvi korak *pravog* debagovanja, čine analiziranje i razumevanje problema. Osnovni cilj analize problema je da se prikupi dovoljno informacija da bi problem mogao da se reprodukuje. Sve dok nismo u stanju da ponovimo problem, teško možemo da kažemo da razumemo kako on nastaje.

Da bi propust mogao da se razume, potrebno je da se poznaju šire okolnosti manifestovanja propusta, kao i struktura i povezanost podsistema u kome se propust manifestuje. Razumevanje propusta obuhvata povezivanje njegove manifestacije sa stvarnim dešavanjem u sistemu. Obično se pri uočavanju propusta dokumentovanje zadržava na informacijama koje se dobijaju kroz korisnički interfejs, dok je za puno razumevanje problema neophodno da se posmatraju i interni elementi sistema, koji

učestvuju u širenju infekcije od mesta nastajanja do mesta ispoljavanja. Korisnik uočava neuspeh, ali najčešće ne razume ni gde je kvar ni koji su putevi infekcije od kvara do uočenog neuspeha. Rezultat analiziranja i razumevanja propusta bi trebalo da bude skup pretpostavki o onome što se interno dešava u sistemu i dovodi do odgovarajuće manifestacije.

Lociranje propusta je postupak pronalaženja kvara, tj. tačnog segmenta programskog koda (ili više takvih segmenata) u kome postoje greške koje za posledicu imaju opisan problem. Lociranje propusta obuhvata razumevanje puteva infekcije i pronalaženje konkretnog kvara. Zato je za uspešno lociranje propusta neophodno dobro razumevanje njegove manifestacije, ali i temeljno razumevanje šireg posmatranog programskog koda. Ako se nedovoljno dobro razume propust, ili se nedovoljno dobro poznaje i razume odgovarajući programski kod, onda može da se utroši mnogo vremena na neuspešno traženje greške u delovima programskog koda koji su samo prividno povezani sa problemom.

Razumevanje propusta i lociranje konkretne greške u programskom kodu čine nerazdvojivu celinu, koja predstavlja najsloženiji i najteži deo procesa debagovanja. Analiza i razumevanje problema se bave delom puta infekcije koji je bliži tački ispoljavanja, dok se lociranje greške bavi delom puta infekcije koji je bliži uzroku. U nekim slučajevima, ako je put infekcije kratak, a mesto ispoljavanja vrlo blizu uzroku problema, može da se dogodi da se tokom analize i pokušaja razumevanja problema relativno lako otkrije i tačan uzrok, ali to je pre izuzetak nego pravilo.

Razumevanje i lociranje se veoma često odvijaju naizmenično (nakon pokušaja razumevanja sledi pokušaj lociranja, a u slučaju neuspeha se ponavlja pokušaj razumevanja, pa tako sve dok se greška ne locira) ili čak do te mere objedinjeno da imaju karakteristike jednog složenog koraka.

Kada se greška pronađe, njeno ispravljanje je najčešće daleko jednostavniji korak od razumevanja i lociranja. Težina ispravljanja propusta najviše zavisi od nivoa na kome je on načinjen. Locirane greške u implementaciji programskog koda se ispravljaju relativno lako, ali prevazilaženje propusta pronađenih u poslovnim pravilima ili formalnim zahtevima nekada može da bude veoma zahtevno i skupo – čak i do te mere da se više isplati da se delovi programskog koda ponovo razvijaju.

Predstavljeni koraci debagovanja mogu da se izvode uz primenu odgovarajućih strategija ili različitih skupova principa i pravila. Prema tome kako se odvijaju, možemo da razlikujemo i različite pristupe debagovanju.

12.3 Neformalno debagovanje

Praktično svaki programer se, makar u početnom periodu svog programerskog stvaralaštva, oprobao u tzv. *neformalnom debagovanju*. Neformalno debagovanje predstavlja najjednostavniji, ali istovremeno i najpovršniji metod debagovanja.

Počiva na pretpostavci da je propust jednostavan i da može brzo i lako da se pronađe i ispravi.

Osnovno sredstvo za rad su znanje o rešavanom problemu i napisanom programskom kodu i intuicija. Opis postupka čine samo dva koraka:

1. Pokušati sa sasvim jednostavnom[78] popravkom.
2. Sve dok program ne proradi, ponavljati korak 1.

Važno je da znamo da neformalnim putem može da se dođe do rešenja samo u sasvim ograničenom prostoru slučajeva. Jednostavne instant popravke mogu da se pronalaze „zatvorenih očiju" samo u slučajevima:

- jednostavnih malih programa;
- programa ili delova programa u kojima je prostor problema veoma uzak i lako sagledliv;
- problema čiji prostor širenja infekcije je sasvim ograničen;
- programa ili delova programa koje piše jedan programer za najviše nekoliko dana;
- kada je domen problema izuzetno dobro poznat programeru.

U ovu grupu slučajeva u prvom redu spada rešavanje problema koji su uočeni već tokom pisanja koda. Tada programer vrlo dobro sagledava zadatak i kod koji piše, pa može neposredno i lako da ispravlja jednostavne previde u kodiranju. Manje sintaksne greške, zaboravljanje nekih promena podataka, pogrešno postavljanje granica blokova, neispravni indeksi i slične greške su primeri propusta koji često mogu da se lako i jednostavno reše. Ali nisu svi previdi koji se prave pri kodiranju dovoljno jednostavni za takvo rešavanje. Neke greške nastaju zbog neispravnog koncepta rešenja ili pogrešnog zaključivanja. One uglavnom ne mogu da se reše primenom neformalnog metoda.

Kod naknadno ustanovljenih grešaka stvari obično stoje sasvim drugačije. U savremenom razvoju softvera se koriste testovi jedinica koda, kao i testovi ponašanja i prihvatljivosti, pa ako nijedan od tih testova ne prepozna problem, nego se on ispolji tek kasnije, pri manuelnom testiranju ili eksploataciji sistema, onda takav

[78] Ovde možemo dodati još neke atribute uz popravku, bilo da smo uvereni u sopstvene sposobnosti ili u jednostavnost problema, kao na primer: očigledna, originalna, genijalna i drugo.

problem najverovatnije nema jednostavan uzrok i zaslužuje ozbiljniji tretman nego što može da pruži neformalni metod.

Iako neformalan pristup u nekim slučajevima može da radi, on vlo često može da bude kontraproduktivan. Nipošto ne smemo da se zavaravamo da takvo debagovanje može da doprinese rešavanju svih problema. Naprotiv, ako neuspešna primena većeg broja pokušaja popravljanja nije propraćena i redovnim i potpunim brisanjem neuspelih „popravki“, onda će programski kod vrlo verovatno da nam postane bogatiji za čitavo brdo novih problema.

Deset malih bagova u kodu se skrilo.
Pre poslednje popravke, devet ih je bilo.

SM

Ako rešavanje nekog problema neformalnim putem ne dovede do rešenja iz manjeg broja pokušaja, to je obično jasan znak da je vreme za ozbiljniji pristup problemu.

12.4 Empirijski naučni metod debagovanja

Razumevanje i lociranje propusta su analitičke aktivnosti. Počivaju na posmatranju velike količine informacija i izvođenju zaključaka na osnovu tog posmatranja. To se u osnovi ne razlikuje mnogo od istraživačkog pristupa u prirodnim naukama.

Za većinu prirodnih nauka je uobičajen *empirijski naučni metod* istraživanja, koji se odlikuje izvođenjem opštih zaključaka na osnovu mnogobrojnih pojedinačnih posmatranja. Empirijski naučni metod počiva na sledećem algoritmu:

1. Posmatrati prostor problema (ili nečije zapise o ranije obavljenom posmatranju);
2. Oblikovati (izmisliti) hipotezu (neprovereno tvrđenje) koja je u skladu sa rezultatima posmatranja;
3. Na osnovu hipoteze napraviti predviđanje ponašanja u nekim novim slučajevima;
4. Eksperimentalno (ili daljim posmatranjima) proveriti da li je takvo predviđanje bilo ispravno;
5. Ponavljati korake 2, 3 i 4, uz popravljanje postojeće ili oblikovanje nove hipoteze, sve dok postoje neslaganja između predviđanja i odgovarajućih eksperimenata i posmatranja i dok ima još prostora za dalje preciziranje hipoteze i odgovarajućih predviđanja.

Opisan empirijski metod može da se primeni i na problem razumevanja i lociranja propusta u procesu debagovanja. Takav pristup se naziva *naučno debagovanje* ili *sistematsko debagovanje*:

1. Posmatrati uočen problem;
2. Postaviti hipotezu o uzroku problema;
3. Na osnovu hipoteze napraviti predviđanje ponašanja sistema;
4. Eksperimentalno proveriti ispravnost predviđanja;
5. Ponavljati korake 2, 3 i 4, uz popravljanje ili zamenjivanje hipoteze, sve dok se ne potvrdi ispravnost hipoteze ili ne ponestanu mogućnosti za njeno dalje unapređivanje.

Rezultat uspešne primene opisanog postupka bi trebalo da bude vrlo jasno i precizno definisana hipoteza. Jedan deo hipoteze mora da se odnosi na razumevanje uzroka problema, a drugi na tačnu lokaciju greške (ili grešaka) u kodu.

Primetimo da naučni metod zahteva da se pri debagovanju eksplicitno prave i zapisuju hipoteze. To je jedan veliki kvalitet, zato što implicitno rezonovanje o problemu ima tendenciju da stvara više teškoća nego koristi. U debagovanju je u igri ogromna količina raznovrsnih informacija i implicitno rukovanje njima vodi veoma krivudavim putem, često u nove probleme. Jedna od posledica implicitnog pristupa je da je programeru teško da odgovara na pitanja svojih kolega o problemu. Kada se zaključci eksplicitno prepoznaju i evidentiraju kao uočene činjenice i hipoteze, umesto da postoje samo u neformalnom obliku kao zamisli i ideje, onda je rezonovanje na osnovu njih mnogo lakše i pouzdanije, a i lakše se o njima razmenjuju mišljenja sa saradnicima.

Pri proveravanju važenja hipoteze, postoji određena sloboda u pogledu mogućnosti menjanja sistema radi olakšavanja posmatranja i proveravanja. Međutim, pri tome mora dobro da se pazi da se neodgovarajućim ili preteranim izmenama ne dovede do *efekta posmatrača*. Kao i u prirodnim naukama, tako i pri debagovanju, svakim korakom koji načinimo prema detaljnijem i obuhvatnijem posmatranju, svakim zahvatom u programskom kodu ili primenom nekog spoljašnjeg alata, mi neminovno menjamo stanje sistema i okolnosti događaja koji posmatramo. Efektom posmatrača se nazivaju sve one posledice posmatranja koje dovode do netrivijalnih promena u ponašanju posmatranog sistema. Kao posledica načinjenih izmena može da nastupi promena u načinu manifestovanja problema, pa čak i potpuno maskiranje problema koji pokušavamo da posmatramo.

Da ne bi došlo do efekta posmatrača, moramo da imamo u vidu dinamičku prirodu problema, koju smo upoznali na početku ovog poglavlja. Sva naša posmatranja moramo da sprovodimo tako da ne remetimo prirodan tok ispoljavanja greške – od kvara, preko nastajanja i širenja infekcije, pa sve do manifestacije

neuspeha. U suprotnom može da se dogodi da nekim neopreznim zahvatima privremeno prekinemo ili promenimo ovaj tok, a time i izmenimo ili sprečimo nastupanje neuspeha. To nije ništa drugo do jedan vid ostvarivanje efekta posmatrača.

Najveći problem u primeni naučnog metoda je u tome što nam on daje neke opšte naznake kako i šta da radimo da bismo se približili rešavanju problema, ali nam ne pomaže mnogo u konkretnim slučajevima. Najteži deo posla u empirijskom debagovanju je *oblikovanje hipoteza*. Metod nas uči da je to neophodan i važan korak, ali nam ne daje čak ni približne naznake kako bi to trebalo da ga izvodimo, a da to bude i dobro i efikasno.

12.5 Heurističko debagovanje

Heuristički metod debagovanja počiva na primenjivanju određenih pravila i principa, koji se oblikuju na osnovu ranijeg iskustva. Različiti autori predlažu različite skupove pravila, koji su uglavnom oblikovani prema njihovom konkretnom načinu razmišljanja. U nekim knjigama se navodi i po više različitih skupova pravila[79].

Kao jedan od dobrih skupova pravila može da se izdvoji onaj koji je predstavio Dejvid Agans [*Agans 2006*]. Dalji tekst odeljka o heurističkom metodu debagovanja se u velikoj meri oslanja na pravila izložena u toj knjizi. Agansova pravila su dobro oblikovana i sistematično i dobro izložena. Pojedinačna pravila se najviše odnose na razumevanje i lociranje grešaka, ali celovit skup pravila pokriva i ostale aspekte debagovanja. Pravila su u osnovi jednostavna, što olakšava njihovo razumevanje, ali ne i primenu, koja u slučaju kompleksnih sistema može biti prilično teška.

> *Kada nam je trebalo mnogo vremena da pronađemo neku grešku,*
> *to je bilo zato što smo zanemarili neko od osnovnih pravila debagovanja*
> *– jednom kada smo ga primenili, brzo smo pronašli problem.*
>
> *Dejvid Agans*

Njihova primena zahteva da se čitavom procesu pristupi veoma sistematično, uz redovno preispitivanje urađenog i istovremeno posvećivanje i detaljima i celini

[79] Na primer, u [*Metzger 2004*] se neka pravila navode u vidu heuristika, dok se neka druga pravila navode u vidu *taktika*. Iako su taktike uglavnom navedene u formi manjih postupaka koji se obavljaju u okviru šire oblikovanih heuristika, i tako definisana pravila imaju preklapanja sa skupovima pravila koje predstavljaju drugi autori. Pri kraju odeljka o heuristikama ćemo ukratko razmotriti i neke od predloženih taktika.

posmatranog problema. Jedna od najvažnijih karakteristika procesa programiranja je posmatranje i formalno opisivanje problema na različitim nivoima apstrakcije, pa zato i proces debagovanja mora da se odlikuje sličnim svojstvima, kako bi uspeo da obuhvati i prevaziđe sve potencijalne propuste koji su načinjeni pri kodiranju.

Dejvid Agans ističe devet osnovnih pravila za debagovanje. Pravila su formulisana jednostavno i razumljivo:

1. Razumeti sistem
2. Navesti sistem na grešku
3. Najpre posmatrati pa tek onda razmišljati
4. Podeli pa vladaj
5. Praviti samo jednu po jednu izmenu
6. Praviti i čuvati tragove izvršavanja
7. Proveravati naizgled trivijalne stvari
8. Zatražiti tuđe mišljenje
9. Ako je nismo ispravili, onda greška nije ispravljena

U narednim odeljcima ćemo prodiskutovati najvažnije aspekte ovih pravila i pokušati da objasnimo zašto su toliko važna za uspešno otklanjanje grešaka.

Pravilo 1 – Razumeti sistem

Da bi se neki problem prevazišao, potrebno je razumeti ga, pronaći gde je načinjen propust i otkloniti propust. Ali da bi problem mogao da se razume, najpre mora da se razume okruženje čiji je problem deo, a to je, najšire posmatrano, računarski sistem na kome se ispoljava posmatrani problem. Pojam računarskog sistema se u ovom kontekstu upotrebljava u svojoj najširoj definiciji, koja obuhvata razvijani softver, operativni sistem, sve ostale softverske komponente sistema (kako one sa kojima razvijani softver ostvaruje komunikaciju, tako i one koje naizgled sa njim nemaju ama baš nikakve veze), hardver računarskog sistema (uključujući sve hardverske uređaje, pa i one koji se naizgled ne upotrebljavaju), mrežne i distribuirane komponente sistema i korisnike.

Posmatranje sistema se pri debagovanju najčešće ograničava na najuži mogući deo – na razvijani softver ili neku njegovu pojedinačnu komponentu, ali ovde moramo da istaknemo da je nesistematična primena takvog pristupa potencijalno veoma loša i kontraproduktivna. Iako u praksi, srećom, za rešavanje većine problema ne moramo da sagledavamo doslovno sve elemente posmatranog računarskog sistema, ograničavanje posmatranja ipak mora da se radi sistematično. Detaljnije objašnjenje ostavljamo za odeljak o pravilu *4 - Podeli pa vladaj*.

Razumevanje sistema počiva na poznavanju njegovih opštih i specifičnih karakteristika. Opšte karakteristike se odnose na uobičajene hardverske i softverske komponente sistema, zvaničnu dokumentaciju i formalno definisana pravila korišćenja sistema i razvijanog softvera. Specifične karakteristike obuhvataju sve one elemente sistema koji mogu da se manje ili više razlikuju između različitih sistema na kojima se razvijani softver koristi. Na primer, ako je softver razvijan za jedan konkretan operativni sistem, onda taj operativni sistem predstavlja opštu karakteristiku sistema, a jezik korisnika i druga regionalna podešavanja spadaju u specifične karakteristike sistema.

Uputstva i tehnička dokumentacija se često ne čitaju u punom obimu, ili se čitaju relativno površno. Kada se upoznajemo sa nekim sistemom, takvo površno čitanje je najčešće sasvim dovoljno (ili bar težimo da tako sebi predstavimo), zato što je puna dokumentacija za ozbiljne sisteme najčešće preobimna da bi neko mogao da je prouči od reči do reči pre nego što počne da ih koristi. Razvojna dokumentacija savremenih operativnih sistema, sistema za upravljanje bazama podataka i drugih složenih softverskih sistema može da obuhvata po nekoliko hiljada stranica prepunih detaljnih specifikacija načina upotrebe ili opisa ponašanja u različitim situacijama. Iluzorno je očekivati da neko može da drži u glavi sve informacije iz tolike dokumentacije. U krajnjem slučaju, dok bi početnik proučio svu tu dokumentaciju, vrlo verovatno bi bila objavljena neka novija verzija softvera sa novom i još obimnijom pripadajućom dokumentacijom.

Sa druge strane, kada dođe do nekog problema, čitanje dokumentacije je ipak nezamenljivo. Najveći deo dokumentacije upravo tome i služi. Najčešće je dovoljno da se pri čitanju usmerimo na određene delove dokumentacije, posvećene nekim konkretnim pitanjima, koja su bliska problemu. Koliko god da je to teško i neprijatno, za neke probleme se može prepoznati uzrok ili pronaći rešenje tek posle čitanje više različitih knjiga dokumentacije o različitim komponentama sistema (operativni sistem + sistem za upravljanje bazama podataka + razvojni alat + neke biblioteke + ...). Posebno je važno da se odgovarajući delovi dokumentacije prouče detaljno, tako da se ne propuste neke „sitnice“ koje mogu da značajno utiču na ponašanje sistema i razvijanog softvera.

Poseban problem sa dokumentacijom je da ona često ima tendenciju da ne bude ažurna. Uobičajena je praksa da se dokumentacija pravi paralelno sa izradom softvera ili da blago kaska za izradom softvera. Zato je veoma čest slučaj da se aktuelna javno dostupna verzija dokumentacije zapravo odnosi na prethodnu verziju softvera. U takvim slučajevima obično postoji tzv. „beta“ verzija dokumentacije, koja se odnosi na aktuelnu verziju softvera, ali još nije do kraja „ispeglana“, pa se u njoj mogu naći neke „omaške“. Čak i kada je dokumentacija deklarisana kao ažurna, ona često ima različite vrste slabosti: objašnjenja nekih novih elemenata softvera ili novih oblika ponašanja softvera nisu dovoljno detaljna ili sasvim nedostaju, postojeća objašnjenja nisu usklađena sa poslednjim izmenama u

softveru ili čak postoje neki viškovi, koji opisuju elemente softvera koji su izbačeni u odnosu na prethodnu verziju ili iz nekog razloga ipak nisu ušli u aktuelnu verziju iako su bili planirani.

Sastavni deo razumevanja sistema je i poznavanje očekivanog ponašanja razvijanog softvera. Da bi se mogao uočiti, analizirati i razumeti neki problem, neophodno je znati kako bi sistem trebalo da se ponaša u idealnim uslovima. Nije retkost da se prijavi postojanje nekog problema i da se na njegovu analizu utroši mnogo vremena, a da se na kraju ispostavi da je u pitanju ponašanje koje je u potpunosti u skladu sa planovima i specifikacijom. Neke stvari mogu neupućenom korisniku da izgledaju neintuitivno, čudno, prekomplikovano ili čak pogrešno, a da u širem kontekstu predstavljaju dobro zamišljeno i implementirano rešenje. U takvim slučajevima mnogo vremena u fazama testiranja i analize prijavljenih nedostataka može da se uštedi dobrim poznavanjem očekivanog ponašanja sistema.

Savremeni razvoj softvera počiva na upotrebi različitih softverskih i hardverskih komponenti i njihovoj međusobnoj komunikaciji i saradnji. U takvim okolnostima poznavanje ponašanja razvijanog softvera obuhvata i poznavanje tokova podataka između različitih podsistema, a često i poznavanje formata tih podataka. Potrebno je znati kako je organizovana raspodela odgovornosti između komponenti sistema – koja komponenta je zadužena za koje aktivnosti, šta su ulazne informacije na osnovu kojih ona obavlja svoj posao, koje izlazne informacije proizvodi i kojim putem ih izdaje. U razumevanju ovog segmenta sistema mogu da budu od velike pomoći dobro napravljeni dijagrami. Tu je danas nezamenljiva uloga *UML*-a, kao široko prihvaćenog jezika za modeliranje softvera.

Sve što je rečeno u vezi sa čitanjem dokumentacije u vezi sa komponentama sistema, odnosi se i na razvojne alate. Razvojni alati su svi oni programi koji članovim razvojnog tima pomažu u proizvodnji softvera. Uobičajeno je da se u razvojne alate svrstavaju prevodioci, editor teksta, debageri i/ili integrisani razvojni alati (engl. *Integrated Development Environment – IDE*), ali tu spadaju i mnogi drugi alati, kao sistemi za praćenje verzija, sistemi za praćenje poslova i problema, alati za rad sa bazama podataka i drugo. Štaviše, u savremenom razvoju softvera je uobičajeno da se za pisanje različitih komponenti sistema koriste i različiti programski jezici, pa i različiti sistemi za upravljanje bazama podataka, a nije retko ni da se različite komponente prilagođavaju različitim operativnim sistemima.

Dokumentacija služi da se čita i to je potrebno raditi što je ranije moguće. Ako smo iz bilo kog razloga propustili da pročitamo dokumentaciju već pri započinjanju pisanja softvera, onda moramo da joj se posvetimo već na početku debagovanja.

Mnoge greške u kodiranju su posledica površinskih sličnosti između razvojnih alata. Iako nekada može da nam se učini da su neki elementi različitih programskih jezika međusobno ekvivalentni, njihova slična sintaksa može da prikrije sasvim različitu semantiku, što je idealna podloga za greške. Slično važi i za druge vrste

razvojnih alata, pa je dobro poznavanje svih alata i jezika koji se koriste u razvoju veoma važno kao preventiva pravljenja grešaka. A kada je greška već načinjena, za njeno razumevanje i lociranje je neophodno da se poznaje jezik programskog koda, kao i specifičnosti alata koje eventualno mogu da utiču na ponašanje.

Savremeni softverski sistemi su veoma složeni, što značajno otežava njihovo razumevanje. No, niko nije rekao da je programerski posao lak – a ako želimo da ga radimo dobro, tj. da pišemo programe sa malo grešaka i da te greške dobro i efikasno otklanjamo, onda smo neizbežno prinuđeni da poznajemo različite aspekte softverskih sistema koje pravimo i računarskih sistema u okviru kojih će oni da se koriste. Pri tome nije dovoljno da se zadržimo na površnom upoznavanju, već moramo da se bavimo i detaljima.

Neophodna pretpostavka za dobro razumevanje računarskog sistema je dovoljno široko i temeljno poznavanje teorijskih i praktičnih osnova računarstva. Štaviše, iskustvo pokazuje da u ovom kontekstu „dovoljno“ nikada nije dovoljno.

Pravilo 2 – Navesti sistem na grešku

Da bi neki problem mogao da se reši, najpre je potrebno da bude pažljivo posmatran i analiziran. U slučaju propusta u razvoju softvera, teškoća je što izvršavanje programa nije jedna stabilna statička slika koja može da se proizvoljno dugo posmatra, već je u pitanju dinamički proces čiji tok ne može uvek lako da se predvidi. Dinamičnost izvršavanja programa i promenljivost stanja programa značajno otežavaju uočavanje i posmatranje problema. Zbog toga je veoma važno da se pronađe tačan redosled koraka posle koga se ispoljava uočen problem. Ako problem može da se ponovi, onda on može lakše da se posmatra iz više uglova i u različitim okolnostima, što je neophodno za njegovo razumevanje.

Omogućavanje posmatranja problema je jedan od najvažnijih razloga za ponavljanje greške, ali ne i jedini. Drugi važan razlog za ponavljanje greške jeste omogućavanje posmatranja uzroka njenog ispoljavanja. Uzrok problema je obično povezan sa nekim od koraka koji su doveli do manifestovanja greške. Ako ne znamo koji su to koraci, onda smo veoma daleko od razumevanja problema i ustanovljavanja uzroka. Tek kada poznajemo mehanizam ponavljanja greške, onda možemo da razmatramo i različite potencijalne uzroke i da pristupimo postepenoj lokalizaciji problema. Posmatranje koraka koji dovode do greške nam olakšava praćenje širenja infekcije i lokalizovanje kvara.

Postoji i treći važan razlog za određivanje niza koraka koji dovode do greške – provera da li je greška otklonjena ili ne. Ako znamo kako da ponovimo grešku, onda imamo relativno jednostavno sredstvo za proveravanje da li je ispravljanje greške bilo uspešno – nakon sprovođenja pokušaja njenog otklanjanja, možemo da pokušamo da ponovimo grešku i da posmatramo ishod. Ako greška više ne može da se ponovi, onda to može da bude vid potvrde da je otklonjena.

Uobičajen metod za ponavljanje manifestovanja greške je isprobavanje uz dokumentovanje različitih nizova koraka i rezultata. I ovde vidimo značaj sistematičnosti pri debagovanju. U najboljem slučaju, kada neko po prvi put uoči problem, znaće kako je do toga došlo i dokumentovaće sasvim precizno odgovarajući niz koraka. Naravno, to nije uvek tako. Vrlo često je specifikacija uočenog problema nedovoljno dobra da bi se on mogao ponoviti, pa je potrebno da se eksperimentiše sa različitim aktivnostima, koje se ne udaljavaju previše od navedenog niza koraka. Kada god mora da se eksperimentiše, neophodno je da dokumentujemo sve pokušaje i ishode, zato što u suprotnom vrlo lako može da se dogodi da se isti pokušaji ponavljaju po više puta i dragoceno radno vreme troši uzalud.

Jednom kada ponavljanje problema uspe, ne bi trebalo da nas iznenadi ako na nekom drugom sistemu, ili na istom sistemu ali u drugim okolnostima, ista sekvenca koraka ne dovodi do ispoljavanja problema. Za uspešno ponavljanje greške često nije dovoljno dokumentovati samo niz koraka koji do nje dovode nego i detaljan opis uslova, na primer u obliku niza koraka koji pripremaju čitavo okruženje za rad. Na ponavljanje greške mogu da utiču neke očekivane stvari, kao npr. različiti podaci koji se obrađuju, postojeći sadržaj baze podataka i sl. ali i neke naizgled sasvim nebitne stvari kao npr. kodni raspored tastature, vreme i datum na računarskom sistemu, vremenska zona, da li je korisnik levoruk ili desnoruk, da li istovremeno na istom računaru radi još neki program i drugo.

Neke probleme je potrebno da ponovimo mnogo puta da bi se izvršilo dovoljno različitih analiza i ustanovio uzrok. Ako je postupak ponavljanja greške složen i/ili dugotrajan, a mora da se ponovi više puta, onda može biti korisno da se napravi skript koji simulira aktivnosti korisnika i automatizuje ponavljanje greške. Simuliranje omogućava i da se programski simulira ponašanje sistema za različite vrednosti parametara. Grafički korisnički interfejsi savremenih operativnih sistema uglavnom počivaju na arhitekturi upravljanoj događajima, tako da se simulacija aktivnosti korisnika svodi da simuliranje događaja koje korisnik proizvodi, tj. simulaciju upotrebe miša i tastature. Elementarni alati za simuliranje aktivnosti korisnika su sadržani i u nekim distribucijama operativnih sistema. Obično su koncipirani tako da omogućavaju snimanje aktivnosti korisnika i njihovo kasnije ponavljanje. Naprednije verzije ovih alata omogućavaju i manuelno održavanje sačuvanih zapisa aktivnosti, pa i upotrebu kontrolnih struktura i pravljenje pravih programa za podražavanje rada korisnika.

Ako se pri debagovanju naprave neki pomoćni alati, čija je namena da omoguće ponavljanje greške, takve alate nakon otklanjanja konkretne greške nije dobro odbacivati. Preporučljivo je da se sačuvaju, zato što se lako može dogoditi da nam ponovo zatrebaju. U prirodi grešaka je da su reko usamljene. Greške nastaju u trenucima umora ili smanjene pažnje programera, pa ako u nekom delu koda imamo grešku, to znači da je taj deo koda pisan (ili menjan) kada je programer bio u

„nepažljivoj fazi“, a to opet znači da je vrlo verovatno da su u istom delu koda načinjene i još neke greške. Čuvanje pomoćnih alata za ponavljanje grešaka može da nam uštedi mnogo vremena u slučajevima naknadnog ispoljavanja „srodnih“ problema.

Ako smo u nekom trenutku uspeli da rekonstruišemo niz koraka koji dovode do greške, tu ne treba da se zaustavimo. Poželjno je da razmotrimo mogućnost da do istog problema može da se dođe i na neki drugi način. Posmatranje je potrebno da se usmeri na različite srodne postupke, koji služe za izvođenje iste ili sličnih operacija i da se poveri da li je istu ili sličnu manifestaciju moguće proizvesti i u delimično izmenjenim okolnostima. Primarna motivacija za ovakvu analizu je u tome što vezivanjem za sasvim specifičan niz koraka možemo da se usmerimo na otklanjanje samo jednog dela propusta, pa da neispravljeni ostatak nastavi da se manifestuje u izmenjenim okolnostima. Suviše precizno posmatranje ponekad može da nas navede da otklonimo samo simptom, a ne i uzrok problema. Alternativni postupci za ponavljanje greške mogu da nam prošire pogled na grešku i njene uzroke, ali i da omoguće pouzdaniju proveru nakon otklanjanja.

Pri traženju alternativnih postupaka za ponavljanje greške ne treba ići predaleko. Ako se neka dva niza koraka, koja vode do iste greške, veoma značajno međusobno razlikuju, onda možda nije reč o dva puta do ispoljavanja istog problema, već o dve potpuno različite greške sa sličnim manifestacijama. Naravno, nije loše da pri popravljanju jednog problema proverimo i što više srodnog koda, ali, na žalost, to nije uvek prihvatljivo iz poslovnog ugla i to iz bar dva razloga: najpre zato što može da bude veoma važno da se konkretan problem otkloni u što kraćem periodu, a zatim i stoga što redovnim proveravanjem široke okoline grešaka možemo doći do toga da neke delove koda nepotrebno poveravamo po više puta.

Traženje više alternativnih nizova koraka koji dovode do greške može dalje da se uopšti ponavljanjem (ili simuliranjem) ne same greške nego uslova u kojima se ona ispoljava.

Neke greške je teško ponoviti. Za to može biti više razloga, od složenih okolnosti u kojima se pojavljuje, do zahtevane visoke preciznosti specifičnog niza koraka. Složene okolnosti podrazumevaju veliki broj različitih elemenata koji moraju da se usklade da bi se greška manifestovala. Primer visoke preciznosti uslova nastajanja problema može da bude kada se problem ispoljava samo u slučajevima kada se pokazivač miša pomeri sa jedne tačno određene pozicije na neku drugu tačno određenu poziciju, ili za tačno određen broj piksela udesno.

Veoma je važno da se neuspešno ponavljanje greške ne tumači kao njeno odsustvo. Moramo imati određeno poverenje u osobu koja je prijavila postojanje problema – zašto bi neko gubio vreme da prijavljuje problem koji ne postoji? Sasvim je moguće da prijavljene okolnosti nisu dovoljno tačne, da možda i samo ispoljavanje nije dovoljno dobro opisano, ili da je u pitanju ispravno ponašanje koje je korisniku

izgledalo kao greška, ali činjenicu da problem postoji ne smemo olako da dovodimo u pitanje. Najlakše je da na prijavljenu grešku odgovorimo komentarom da je „opisan problem nemoguće ponoviti“, ali moramo da imamo na umu da greška koju nismo otklonili nije otklonjena (pravilo 9).

Ako smo prepoznali niz koraka koji samo u nekim slučajevima dovodi do greške, onda to znači da nam okolnosti ispoljavanja greške nisu do kraja poznate. Najbolji put je da se daljim analiziranjem okolnosti i eksperimentisanjem popravi niz koraka tako da se učestalost ispoljavanja greške poveća. Ipak, neke greške će se uporno pojavljivati samo povremeno, npr. jedanput u 10, 100 ili čak i više hiljada ponavljanja niza koraka. Da bi se ponovile takve greške, potrebno je da se razmotre svi oni uslovi koji do tada nisu kontrolisani. To može da obuhvata praćenje i podešavanje:

- neinicijalizovanih podataka;
- slučajno generisanih podataka;
- ulaznih podataka;
- trenutka ili trajanja izvršavanja;
- međusobne sinhronizacije niti i procesa i
- spoljašnjih uređaja.

U najvećem broju slučajeva će određivanje fiksiranog ponašanja za neki od ranije nekontrolisanih parametara uticati na učestalost ponavljanja greške. Ako se učestalost poveća, to znači da smo se približili tačnim uslovima. Ako se smanji, to znači da smo se udaljili, ali da uslov koji je kontrolisan ipak utiče na pojavljivanje greške. Tada je potrebno da nastavimo kontrolisanje istog uslova, ali sa nekim drugim vrednostima.

Nekada svi pokušaji uvođenja dodatne kontrole uslova imaju za rezultat smanjivanje učestalosti ponavljanja problema. To su posebno neugodni slučajevi, zato što ukazuju na to da se greška ispoljava samo u vrlo strogo definisanim uslovima, te da je takve uslove veoma teško ponoviti. Primarni cilj u takvim slučajevima jeste da se prepozna koji su to parametri koji utiču na ponavljanje greške, a tek sekundarni cilj je da se ustanove i odgovarajuće vrednosti tih parametara. Pri prepoznavanju značajnih parametara može da bude od pomoći upravo suprotan metod od do sada pominjanog – povećavanje stepena slučajnosti vrednosti parametra. Time što neki parametar konfigurišemo da pri svakom ponavljanju uzima neku drugu slučajnu (tj. pseudo-slučajnu) vrednost i posmatramo da li se i u kojim uslovima greška ponavlja, možemo da postepeno dođemo do skupa parametara čije menjanje utiče na ispoljavanje problema. Takav postupak nije ni jednostavan ni brz, ali ako uobičajen analitički pristup ne dovede do rezultata, onda nam je to jedna od najprihvatljivijih alternativa. Problem sa ovakvim pristupom je što postavljanje slučajnih vrednosti parametara može da proizvede neke druge

greške, što ima za posledicu brojne potencijalne stranputice tokom rešavanja jednog konkretnog problema. Od velike je važnosti da se slučajne vrednosti parametara primenjuju uz primenu još jednog pravila debagovanja – *6. Praviti i čuvati tragove izvršavanja*. Bez toga je praktično nemoguće analizirati ponašanje sistema u brojnim ponovljenim izvršavanjima sa različitim vrednostima parametara.

Ako uradimo sve opisano, a problem se i dalje ponavlja samo povremeno, onda je potrebno promeniti redosled aktivnosti. Ne smemo da zapadnemo u apatiju zato što je problem „svojeglav" – problem nikada nije svojeglav. Uvek se ispoljava u nekim sasvim preciznim ulovima, a to što ne umemo da ih ponovimo je samo otežavajući faktor. U takvim slučajevima možemo da odustanemo od striktnog ponavljanja problema i da se neposredno posvetimo upravo ciljevima ponavljanja. Ciljevi ponavljanja su, podsetimo se, omogućavanje posmatranja problema i njegovih uzroka, a kasnije i provera da li je problem otklonjen. Za proveru nam može poslužiti čak i procedura koja samo povremeno proizvodi problem – ako se problem pojavljuje jedanput u 1000 izvršavanja, onda možemo da smatramo da smo ga otklonili (sa određenom verovatnoćom), ako se ne pojavi nijedanput u npr. 50.000 izvršavanja. Takav metod proveravanja nije savršen, ali je često prihvatljiv.

Što je neki problem teže ponoviti, to je važnije da ga pažljivije posmatramo onda kada se ponovi. Potrebno je da prikupljamo sve relevantne informacije i da njihovim analiziranjem pokušamo da ustanovimo koji su to uzroci problema. Da bi to bilo moguće, često je potrebna intenzivna primena pravila *6 – Praviti i čuvati tragove izvršavanja*.

Pravilo 3 – Najpre posmatrati pa tek onda razmišljati

Videli smo da je potrebno da se problem ponovi da bismo mogli da ga posmatramo. Posmatranje je neophodno radi prikupljanja što veće količine informacija o ispoljavanju problema i uslovima njegovog ispoljavanja. Informacije su nam neophodne zato što samo uz dovoljno informacija možemo razmišljanjem da dođemo do dobrog zaključka. Razmišljanje kome bismo pristupili bez relevantnih informacija nam ne bi bilo od velike pomoći, već bi moglo da nas zatrpa neispravnim ili beskorisnim zaključcima. Pravilo „najpre posmatrati pa tek onda razmišljati" ima za cilj da nam uštedi vreme, koje bi bilo uloženo u razmatranje i implementiranje pogrešnih zaključaka.

Iskusni programeri mogu da na osnovu stečenog iskustva donose zaključke i sa mnogo manje raspoloživih informacija nego početnici. To je dobro – uostalom, upravo tome i služe iskustvo i eksperti. Međutim, za dobre zaključke „mnogo manje" ne sme biti i „nedovoljno". Nipošto ne smemo da se zalećemo i da nudimo rešenje pre nego što sagledamo okolnosti. Ne smemo da donosimo zaključke pre nego što uspemo da ponovimo grešku. Izuzetno, ako imamo veliko poverenje u onoga ko je prijavio problem i u njegov izveštaj, onda informacije sadržane u tom izveštaju mogu da budu dovoljne, ali takve prečice smeju da se prave samo u

iskusnim timovima u kojima se članovi tima odlično sporazumevaju. Čak i tada je preporučljivo da se problem ponovi pre njegovog rešavanja.

Pri posmatranju moraju da se uzimaju u obzir i detalji i celina. Svaka pojedinačna informacija može da nam ukaže na potencijalan uzrok ili neki element mehanizma koji dovodi do problema. Sa druge strane, međusoban odnos uočenih pojedinosti je često mnogo rečitiji nego svaka od njih za sebe.

Da bi posmatranje moglo da se obavlja dovoljno dobro, u mnogim razvojnim projektima se već u fazi projektovanja softvera isplanira obezbeđivanje alata za pomoć pri posmatranju. O tzv. unutrašnjim alatima će biti više reči u odeljku 12.6 Tehnike i alati za debagovanje. Na srodna pitanja se odnosi i već više puta pominjano pravilo 6.

Pri posmatranju grešaka često se primenjuje privremeno isključivanje delova koda. Cilj isključivanja je da se istovremeno suzi prostor oko greške (pravilo 4), skrati vreme potrebno za ponavljanje i posmatranje greške i smanji količina informacija koju je potrebno analizirati.

Jedna od opasnosti pri upotrebi unutrašnjih i spoljašnjih alata je softverska verzija efekta posmatrača: „Svako posmatranje menja sistem, zato što su i posmatranja sastavni deo sistema“. Čak i sasvim sitne promene programskog koda, koje se uvode radi olakšavanja posmatranja, mogu da imaju uticaja na ponašanje sistema, a time i na uslove i načine ispoljavanja posmatranih grešaka. Ovo ne znači da kod ne treba aktivno posmatrati, već da je potrebno da se to čini veoma pažljivo. Preporučljivo je da se programski kod menja radi posmatranja samo toliko koliko je neophodno da se uoče posmatrani elementi, ali ne i više od toga. Pravljenjem suvišnih izmena se nepotrebno povećava verovatnoća menjanja ponašanja sistema, a bez značajnih pozitivnih posledica.

Put do pouzdanih zaključaka o uzroku greške vodi preko pretpostavki (hipoteza). Pretpostavke služe samo radi daljeg lokalizovanja uzroka ispoljenih problema. Neproverene pretpostavke ne smeju da se koriste kao osnova za otklanjanje grešaka – neproverene hipoteze su ništa drugo do nagađanje. Svi zaključci moraju (1) da počivaju na dovoljnim i ispravnim informacijama i (2) da budu nedvosmisleno potvrđeni.

Pravilo 4 – Podeli pa vladaj

Osnovna ideja pravila „podeli pa vladaj“ je da se postepenim aproksimacijama sužava oblast traženja greške. Najpre se postavljaju hipoteze o lokaciji greške (naravno, u skladu sa pravilom 3). Takve hipoteze ne bi trebalo da pokušavaju da sasvim precizno lociraju grešku, već da podele posmatranu oblast na dve podoblasti približnih veličina. Zatim se testovima ustanovljava tačnost ili netačnost hipoteze i u skladu sa rezultatima testiranja se sužava oblast posmatranja. Ponavljanjem ovih

koraka se oblast postepeno sve više sužava, da bi se, u idealnom slučaju, došlo do sasvim precizno lociranog uzroka greške.

Motivacija za ovakav pristup potiče iz realnih ograničenja posmatrača i velikog obima i složenosti kompletnog sistema. Iako bi bilo idealno da se uvek posmatra ceo sistem, to u praksi uglavnom nije moguće. Ako bi neko čak mogao da isprati sva dešavanja u složenom sistemu, to bi onda sigurno bilo isuviše sporo. Znači, već pri posmatranju greške postoji potreba da se posmatranje usmeri na one delove sistema, koji povezuju uzrok problema i uočene posledice.

Jedna od najčešćih grešaka u primeni ovog pravila se pravi već na samom početku, izborom nedovoljno širokih okvira polazne posmatrane oblasti. Od presudne važnosti za uspešnu primenu pravila „podeli pa vladaj" jeste da svi uzroci problema budu u okviru polazne posmatrane oblasti. Ako to nije slučaj, onda nikakvim hipotezama i sužavanjima oblasti posmatranja ne mogu da se pronađu svi uzroci problema, pa neispravna primena ovog pravila često ima za rezultat samo protraćeno vreme.

Nije uvek jednostavno da se nedvosmisleno utvrdi da je posmatrana oblast dovoljno široka, tj. da obuhvata sve uzroke. To je posebno teško u slučajevima kada se posmatra mrežno ili distribuirano okruženje. Neophodan preduslov za pravljenje dobrih početnih procena je dobro poznavanje sistema (pravilo 1).

Pri izboru hipoteza potrebno je da se vodi računa o veličini potprostora na koje se hipotezom deli posmatrani prostor, ali i o tome da se hipotezom nedvosmisleno ukazuje na jedan od tih potprostora kao na lokaciju greške. Ako jedan od ova dva uslova nije zadovoljen, tada je i hipoteza verovatno beskorisna.

Jedan način za postavljanje hipoteza je da se pođe od neuspeha prema potencijalnim uzrocima. Mesto ispoljavanja greške je obično relativno poznato. Pretpostavljanjem da je uzrok ne dalje od nekih hipotezom određenih okvira omogućava se postepeno sužavanje posmatrane oblasti bez posvećivanja pažnje svakom pojedinačnom potencijalnom uzroku. Mesta ispoljavanja problema obično ima mnogo manje nego potencijalnih uzroka, pa je zbog toga kretanje od uočenog neuspeha prema kvarovima koji su ga proizveli najčešće efikasnije od kretanja u suprotnom smeru i postepenog odbacivanja jednog po jednog potencijalnog uzroka. Od presudne važnosti je da se počne od *svih* oblika ispoljavanja problema. Ako se zanemari neki od aspekata ispoljavanja problema, onda posledica može da bude „rešavanje" nekih simptoma, a ne njihovog uzroka.

Pri traženju uzroka mogu da se pojave šumovi. Šumove stvaraju različiti dinamički delovi sistema, koji svojim promenljivim ponašanjem mogu da prikriju stvarne uzroke problema, ili da navedu posmatrača da traži uzroke u pogrešnom delu sistema. Poseban problem je što jedna greška može biti uzrok druge greške. Zbog toga bi i posle pronalaženja i lociranja greške trebalo da se nastavi dalje

sužavanje posmatrane oblasti, kako bi se proverilo da ne postoji i još neki dodatni uzrok problema.

Šumovi ponekad mogu da se otklone privremenim isključivanjem delova koda za koje se pretpostavlja da ne sadrže uzrok greške. U duhu primene ovog pravila, isključivanje dela koda predstavlja upravo vid testa odgovarajuće hipoteza o lokaciji greške.

Neke vrste grešaka se teško uočavaju na „živim" podacima. Zbog toga se često prave veštački uzorci ulaznih podataka (ili širih uslova izvršavanja programa), za koje je poznat ispravan rezultat. Veštački uzorci mogu da olakšaju proveru nekih hipoteza, kao i da dopuste formulisanje nešto slobodnijih hipoteza, kakve možda ne bi imale mnogo smisla sa realnim podacima.

Pored šumova i teškoća sa živim podacima, najozbiljniji problem sa kojim se suočavamo pri sužavanju oblasti posmatranja su slučajevi sa više uzroka koji su rasuti po različitim segmentima sistema. Tada se sužavanje na uobičajen način obično završava na relativno širokoj oblasti koja obuhvata sve uzroke i možda još poneki dodatni element softvera. Kao dodatni elementi su obično obuhvaćene komponente koje povezuju delove softvera koji sadrže uzroke ili puteve prenošenja infekcija, ili su u relativno složenim odnosima sa takvim delovima.

Problem širokih oblasti, koje obuhvataju više uzroka, može da se prevaziđe pomoću ciljano oblikovanih veštačkih uzoraka ulaznih podataka. Biranjem ulaznih podataka sa različitih granica opsega obično može da se neutrališe uticaj nekih uzroka, pa i da se omogući dalje sužavanje posmatrane oblasti. Biranjem različitih veštačkih uzoraka obično mogu da se lakše lokalizuju neki od uzroka problema. Po otklanjanju uočenih uzroka, izborom drugačijih ulaznih podataka mogu da se potraže i drugi uzroci i da se tako nastavi rad do potpunog rešavanja problema.

Pravilo 5 – Praviti samo jednu po jednu izmenu

Svaki put kada menjamo postojeći kod, bilo zbog implementiranja novog ili izmenjenog ponašanja, refaktorisanja ili debagovanja, moramo da imamo u vidu da naknadne izmene predstavljaju plodnije okruženje za nastajanje grešaka nego što je to slučaj sa pisanjem sasvim novog koda. Razlika je u tome što kada pišemo novi kod, onda obično imamo široku sliku o tome šta se zbog čega piše i zašto se nešto piše na konkretan izabran način, dok se pri menjanju postojećeg koda uže fokusiramo na lokalni kod, pa postoji povećana opasnost da neki deo konteksta, u kome se nalazi ili funkcioniše programski kod koji se menja, nije u potpunosti shvaćen ili čak uopšte nije ni uzet u razmatranje.

Naravno, menjanje postojećeg koda je neophodno. Štaviše, debagovanje podrazumeva učestalo menjanje postojećeg programskog koda. U fazi lokalizacije greške kod se menja da bi se privremeno isključilo funkcionisanje nekog dela koda, ili da bi se proverila ispravnost hipoteze. U fazi otklanjanja greške kod se menja da bi

se otklonili lokalizovani uzroci greške. U uslovima kada moramo da pravimo izmene, a one predstavljaju opasnost u vidu potencijalnog pravljenja novih grešaka, potrebno je da sagledamo načine za što bezbednije pravljenje izmena. U tome je osnovni smisao pravila „Praviti samo jednu po jednu izmenu“. Izmene nikada ne bi trebalo da se prave u paketu, već samo pojedinačno, da bi se na taj način olakšalo uočavanje eventualno napravljenih propusta. Kada govorimo o „jednoj izmeni“ ili o „paketu izmena“, to se, pre svega, odnosi na određen broj izmena koje se prave u okviru jednog „atomičnog“ koraka rešavanja nekog problema, u toku koga se ne proveravaju posledice načinjenih izmena.

Među potencijalne probleme pri pravljenju paketa izmena spada mogućnost da jedna izmena reši problem, a da je druga suvišna. U tom slučaju, druga izmena uopšte nije poželjna, zato što može da napravi nepotrebne nove probleme. Pri menjanju koda je potrebno da se uvek zadržimo na najmanjem mogućem broju izmena, te da pravimo samo one izmene koje su neophodne za rešavanje problema. Tim pre se te izmene mogu i moraju praviti svaka za sebe, jedna po jedna, uz učestalo proveravanje njihovih posledica.

Može da se dogodi i da nijedna od načinjenih izmena nije dobra. Naravno, to možemo da ustanovimo proveravanjem, ali će biti mnogo teže da se pokaže da izmene načinjene u paketu nisu dobre nego kada bi se to radilo sa jednom po jednom izmenom. Nije retkost da se načinjenim izmenama istovremeno rešava jedan i pravi neki drugi novi problem. I u tom slučaju je uočavanje neispravnosti i njihovo povezivanje sa konkretnim izmenama jednostavnije i pouzdanije ako se izmene prave postepeno, jedna po jedna.

Pre nego što načinimo prvu izmenu, najpre je potrebno da prepoznamo ključni element koda koji se menja, pa zatim da najpre menjamo baš taj element koda, a tek posle, kao vid finog podešavanja, da popravljamo i druge delove koda. Svaka izmena bi trebalo da bude dobro izolovana i lokalizovana. Ako je zaista uočeno u čemu je problem, onda je jedna izmena uglavnom dovoljna. Ako postoji utisak da je potrebno da se napravi više izmena da bi se rešio problem, onda je obično potrebno da nastavimo posmatranje koda, da bismo bolje lokalizovali problem.

Česta greška pri debagovanju je tzv. „zaletanje“ u izmene, tj. menjanje koda na osnovu neutemeljenih ili neproverenih pretpostavki. Takav pristup obično ima za rezultat veći broj nepotrebno načinjenih izmena, što stvara uslove za pravljenje novih problema. Zbog toga je pri debagovanju potrebno da budemo *uzdržani*. Pre svake izmene je neophodno da dobro razmislimo:

- Da li je ta izmena neophodna?
- Da li ona sigurno rešava problem?
- Da li postoji alternativa?

Ukoliko se ispostavi da načinjena izmena, suprotno očekivanjima, ne rešava problem, to znači da je lokalizacija problema loše ili nepotpuno sprovedena. Tada je potrebno da se, pre nastavka debagovanja, najpre izmenjen deo koda vrati u početno stanje, da ne bi uticao na ponašanje sistema i nastavak procesa lokalizacije, kao i da ne bi ostao kao potencijalna nova greška.

Pravljenje privremenih izmena koda ima značajnu ulogu u procesu lokalizacije problema, pre svega u okviru testiranja hipoteza. U jednom broju slučajeva pri lokalizaciji mogu da se uoče potencijalne izmene koje otklanjaju posmatrani problem, ali koje ne smeju da se prave zbog toga što bi predstavljale suviše velike izmene ponašanja. Iako mogu da predstavljaju uzrok posmatranog problema, takva mesta u kodu su često samo posledica nekih manjih grešaka, koje bi trebalo pronaći. Ako sprovedena lokalizacija ipak ne može da dovede do drugih, manje invazivnih izmena, onda i ovakve uočene izmene mogu da se primene. Ideja je da se ovakve „velike" izmene načine iako je jasno da nisu odgovarajuće, pa da se onda posmatra ponašanje sistema nakon menjanja i uporedi sa ponašanjem pre menjanja koda. Pri tome je potrebno da se porede programski kod, ulazni i izlazni podaci, tragovi izvršavanja, dnevnici izvršavanja i sve ostale raspoložive informacije. Analizom prikupljenih informacija se nekad mogu izvesti novi zaključci o lokaciji greške u kodu, koji omogućavaju njeno preciznije lokalizovanje. Takav pristup preduzimanja velikih pripremnih promena je zapravo oblik refaktorisanja, ali u uslovima postojanja bagova, što je veoma rizično i zbog toga mora da se radi veoma pažljivo.

Od velikog značaja pri lokalizovanju grešaka, a u vezi sa pravljenjem izmena, jeste čuvanje istorije verzija koda. Kada se uoči neki problem, onda može da bude veoma korisno da se proveri da li je problem postojao u prethodnim verzijama koda. Pronalaženjem poslednje verzije koda u kojoj problem nije postojao i prve verzije koda u kojoj se on pojavljuje omogućava nam da poređenjem tih verzija dođemo do značajnih informacija o uzrocima problema.

Posledice načinjenih izmena se proveravaju i logičkom analizom i testiranjem. Svaki od metoda ima svoje prednosti i mane. Logička analiza je pouzdanija od ograničenog broja testova, ali ona najčešće nije efektivno primenjiva pri debagovanju složenijih programa, zbog velikog broja potencijalno značajnih uslova. Sa druge strane, testovi zahtevaju veliku opreznost, zato što svaki pojedinačan test proverava ponašanje sistema samo u jednom vrlo specifičnom kontekstu i nikako ne može da predstavlja pouzdanu garanciju da će se softver uvek ponašati na ispravan način.

Neki programeri se relativno olako odlučuju da u okviru debagovanja preduzimaju refaktorisanje, kako bi popravili dizajn nekih delova koda, a radi olakšavanja pronalaženja greške. Koliko god da je refaktorisanje poželjno, ono ne bi trebalo da se odvija paralelno sa debagovanjem, ili bar ne dok nismo sasvim sigurni da smo locirali grešku. Ako bismo pokušavali da refaktorišemo tokom razumevanja i lociranja greške, onda bismo morali da se zapitamo kako da budemo sigurni da

refaktorisanjem nećemo promeniti ponašanje, kada mi uopšte i ne znamo kako se softver ponaša?

Koraci refaktorisanja i debagovanja bi trebalo da budu jasno razdvojeni, zato što je njihova priroda potpuno drugačija – refaktorisanjem popravljamo *dizajn* koda (tj. njegovu strukturu) i nipošto ne bi trebalo da menjamo ponašanje, a debagovanjem težimo da popravljamo *neispravno ponašanje*. Različita priroda ovih procesa ima za posledicu da je istovremeno ili naizmenično preduzimanje debagovanja i refaktorisanja veoma rizično i teško proverivo. Refaktorisanje ne sme da menja rezultate postojećih (i eventualno novih) testova, a debagovanju je upravo cilj da promeni rezultate testova koji potvrđuju postojanje problema (ne menjajući rezultate drugih testova). Zbog toga što su koncepti na kojima ovi procesi počivaju potpuno različiti i čak prilično suprotstavljeni, razlikuju se i mehanizmi njihovog sprovođenja. Njihovim eventualnim spajanjem bi se dobila nekonzistentna masa pravila i postupaka, koja u praksi donosi više problema (čitaj „novih grešaka“) nego što ih rešava.

U nekim slučajevima ispravljanje greške može da zahteva pravljenje „širokih“ izmena u različitim delovima koda. Takve greške su najčešće posledica propusta na nivou projekta, ili čak na nivou analize ili definisanja zahteva. U takvim slučajevima moramo da podelimo izmene na dve potpuno razdvojene celine: refaktorisanje i debagovanje. Pre nego što se pristupi refaktorisanju, potrebno je da se postave jasni ciljevi i da se nedvosmisleno potvrdi tačna lokacija greške. Ciljevi se oblikuju na osnovu uočenih ograničenja postojeće strukture koda, tako da po njihovom ispunjavanju bude moguće rešavanje problema menjanjem relativno malog izolovanog i lokalizovanog dela koda. Prema tako postavljenim ciljevima se odabiru koraci refaktorisanja kojima će se ciljevi ispuniti. Svaki od koraka refaktorisanja se pravi u skladu sa pravilima refaktorisanja, sa testiranjem jedinica koda i ponašanja koda kao celine. Tek kada se refaktorisanje privede kraju, tj. kada su postavljeni ciljevi ispunjeni, a da ponašanje pri tome *nije izmenjeno*, onda možemo da pristupimo menjanju ponašanja, tj. otklanjanju greške.

Pravilo 6 – Praviti i čuvati tragove izvršavanja

Tragovi izvršavanja (engl. *trace*) su jedan od najvažnijih unutrašnjih alata za rešavanje problema. Oni su nezamenljivo sredstvo za obezbeđivanje informacija o toku izvršavanja u prostornoj i vremenskoj okolini greške. Tragovi izvršavanja su posebno korisni u slučajevima kada je neophodno da se izvršavanje ponovi veliki broj puta da bi se greška ispoljila, ali i u slučajevima kada je praćenje sistema otežano iz bilo kojih razloga. Na primer, interaktivno praćenje izvršavanja distribuiranih sistema je veoma teško, pa se u distribuiranim okuženjima problemi po pravilu rešavaju uz pravljenje detaljnih tragova izvršavanja.

Kada se govori o tragovima izvršavanja, obično se pod time podrazumevaju delovi programskog koda koji automatski zapisuju informacije o stanju programa i

toku njegovog izvršavanja u pomoćnim izlaznim datotekama (tzv. *tragovi*, engl. *trace file*). Delovi koda koji služe za pravljenje tragova se obično pišu i upotrebljavaju samo u fazi debagovanja[80].

Drugi značajan oblik tragova predstavljaju tzv. manuelni tragovi izvršavanja, tj. dnevnici aktivnosti korisnika koji testira softver. Kao što im ime kaže, manuelni tragovi se ne prave automatski, iako mogu da se koriste alati koji olakšavaju njihovo pravljenje. Obično predstavljaju niz zapisa, koje programer zapisuje u svesci ili u nekoj pomoćnoj datoteci. Manuelni tragovi mogu da budu od presudnog značaja za razumevanje problema. Ako se redosled koraka zapisuje tek nakon ispoljavanja propusta, onda postoji mogućnost da korisnik slučajno zaboravi neku od prethodno načinjenih aktivnosti ili previdi neki od aspekata stanja sistema koje je prethodilo izvođenju zapisanog niza koraka. Može da se zameri da zapisivanje svih aktivnosti tokom testiranja predstavlja usporavajući faktor pri testiranju, ali je važnije što se na taj način štedi mnogo više vremena pri kasnijoj analizi i otklanjanju problema. Manuelni tragovi izvršavanja su posebno korisni u slučajevima kada neki naizgled beznačajan korak ili spoljašnji faktor utiče na ispoljavanje problema.

Pri pravljenju manuelnih tragova potrebno je da se zapisuju:

- detaljan opis stanja koje prethodi nizu aktivnosti;
- tačan redosled preduzetih aktivnosti;
- način izvođenja svih aktivnosti, relativno detaljno;
- sve uočene posedice preduzetih aktivnosti.

Detaljan opis stanja koje prethodi nizu aktivnosti može da se izostavi samo ako se testiranje nadovezuje na neki prethodno zapisan niz koraka, zato što tada stanje može da se rekonstruiše na osnovu tog prethodnog zapisa. Ipak, ako se tokom radnog dana obavlja veći broj različitih testiranja, dobro je da se bar povremeno ponovi opisivanje trenutnog stanja sistema.

Jedan od najvažnijih ciljeva upotrebe tragova izvršavanja je uspostavljanje relacija između simptoma (tj. ishoda izvršavanja) i potencijalnih uzroka (tj. uslova koji su doveli do ispoljavanja problema). Povezivanje simptoma i uzroka na osnovu tragova izvršavanja je moguće samo ako su i simptomi i uzroci navedeni u tragu

[80] U nekim izuzetnim slučajevima (na primer ako znamo da neki deo programa nije dovoljno dobro testiran, a rokovi ili drugi vidovi pritiska nam nalažu da softver mora da ide u produkciju), može da bude dobro da se neki elementi pravljenja tragova izvršavanja zadrže i u finalnoj verziji programa, kako bi se eventualno pojavljivanje problema bolje dokumentovalo. U takvim slučajevima često dolazi do slučajnog ili namernog preplitanja odgovornosti dnevnika (engl. *log*) i tragova izvršavanja.

izvršavanja. Ako bismo unapred znali simptome ili uzroke, onda bismo mogli da ograničimo sadržaj tragova samo na potrebne informacije. Međutim, onda nam tragovi izvršavanja verovatno ne bi ni bili potrebni. Teškoće pri debagovanju i nastaju zato što ne možemo da unapred pouzdano znamo ni da li smo uočili sve simptome ni koji su sve njihovi stvarni uzroci. Zbog toga se obično kaže da pisani trag izvršavanja (bilo automatski ili manuelni) nikada nije previše detaljan.

Ako je trag izvršavanja prevelik za manuelno iščitavanje i analiziranje, uvek se može pronaći (ili napraviti) neka alatka za automatsko pretraživanje, filtriranje i analiziranje teksta, a koja može da se koristi za obradu tragova izvršavanja. Već smo više puta naglasili da su svi detalji važni i da je razlog pravljenja tragova upravo u tome da sprečimo da nam neki važan detalj promakne. I pored toga, ipak postoje relativno retki slučajevi kada trag izvršavanja može da bude preobiman. Tipičan primer su automatski tragovi izvršavanja kojima se zapisuje tok nekog složenog iterativnog računanja. Na primer, izračunavanja visoke složenosti i veliki broj iteracija mogu da proizvedu ogromne tragove, čije zapisivanje može da značajno uspori sam tok izračunavanja, čak do te mere da to usporavanje prikrije posmatrani problem ili obesmisli konkretno izvršavanje.

Pravilo 7 – Proveravati naizgled trivijalne stvari

Uzroci i rešenja problema su često sasvim jednostavni, ali je to uglavnom veoma teško da se ustanovi. Da bi se razumelo o kom nivou jednostavnosti se ovde radi, možda je najbolje da navedemo nekoliko trivijalnih primera:

- Ako se pita nije ispekla, da li je rerna uopšte bila uključena?
- Ako automobil ne može da se pokrene, pre nego što ga rastavimo, možda bi bilo dobro da proverimo da li ima goriva?

Ovi primeri mogu da izgledaju kao da je u pitanju šala – ali nije. Čitaocima koji nemaju značajnija iskustva u debagovanju, možda je teško da poveruju da takve greške u programiranju uopšte postoje, ali onima koji takva iskustva imaju, verovatno ih je izlišno predočavati.

Pri analiziranju i rešavanju nekog problema, veoma se često zanemaruju potencijalno jednostavni uzroci i rešenja. Suština problema je u spontanom podrazumevanju nekih očiglednih stvari. Kada se suočimo sa nekim problemom, obično smo unapred spremni na težak i ozbiljan rad na njegovom rešavanju. Samim tim, pokušavamo da razmišljamo istovremeno sveobuhvatno i apstraktno i da sagledavamo čitav sistem i njegove elemente. Pri tome smo veoma često skloni da zanemarimo mogućnost da je problem u tome što nije ispunjena neka od osnovnih pretpostavki za uspešno funkcionisanje.

Da ne bismo dolazili u priliku da jednostavne i „očigledne“ uzroke i rešenja uočavamo tek posle više sati napornog rada, potrebno je da poštujemo ovo pravilo i

da na samom početku posmatranja problema najpre dovedemo u pitanje sve „podrazumevane“ pretpostavke. Ako smo iz bilo kog razloga pretpostavili da je neka komponenta ispravna, to ne znači automatski da ona zaista *jeste* ispravna. Zbog toga je potrebno da analizu *uvek* započnemo proveravanjem da li su ispunjeni svi preduslovi za izvršavanje problematičnog dela posla. Na primer, neki od uobičajenih „trivijalnih“ preduslova su:

- Da li je korisnik pokrenuo ispravnu verziju softvera?
- Da li je softver ispravno instaliran?
- Da li je odgovarajući deo koda uopšte ugrađen u softver?
- Da li su ulazno izlazni uređaji ispravni?
- ... i razni drugi preduslovi.

Naravno, svaki razvojni projekat ima neke specifičnosti, pa i specifične preduslove koje ima smisla proveravati na samom početku traženja grešaka. Na primer, ako se sistem sastoji od više distribuiranih komponenti softvera, onda provere ispunjenosti preduslova za obavljanje posla mogu da obuhvate i sledeća pitanja:

- Da li je računar *R1* uključen?
- Da li je na računaru *R1* pokrenuta komponenta softvera *K1*?
- Da li funkcioniše mreža?
- Da li su računar *R1* i komponenta *K1* dostupni na mreži?
- ... i druge slične specifične provere.

Osim provere ispunjenosti početnih uslova za obavljanje posla, u trivijalne probleme spadaju i različiti aspekti pogrešne upotrebe razvojnih alata. U ovom kontekstu kao razvojne alate valja računati i programske jezike, prevodioce i različite biblioteke koje se koriste u projektu. Nije retkost da programeri misle da alat nešto radi na jedan način, a da se to zapravo odvija na neki sasvim drugi način. Na primer, početnici često greše pri upotrebi makroa u programskom jeziku *C*, zato što očekuju da se oni ponašaju kao funkcije, a njihovo stvarno ponašanje je sasvim drugačije.

Alata ima mnogo i relativno su složeni, pa je zato mnogo i potencijalnih pretpostavki o njihovom ponašanju. Zbog toga je praktično nemoguće dovoditi u pitanje doslovno svaku pretpostavku o njihovom ponašanju, posebno ne kao prvu stvar u debagovanju. Kada se radi o alatima, suština primene ovog pravila je u tome da se naglasi da (1) sve pretpostavke o alatima ne predstavljaju mrtvo slovo na papiru i *mogu* da se dovode u pitanje, kao i da je (2) neke pojedinačne pretpostavke

potrebno dovoditi u pitanje, ali tek onda kada je problem dovoljno lokalizovan da takvih pretpostavki ima razumno mnogo.

Drugi aspekt ovog pravila se odnosi na proveravanje čak i onih uslova koji nam izgledaju nemoguće ili neverovatno. Sve dok nam nešto izgleda nemoguće, to znači da se ne radi o potvrđenoj pretpostavci već samo o hipotezi. A već smo videli (pravilo 3) da svaka hipoteza mora da se potvrdi, pre nego što počnemo da je koristimo kao činjenicu. Koliko god da nam je „očigledno" da nešto ne može da bude uzrok problema, tu mogućnost ipak ne smemo da odbacimo dok je ne proverimo. Tek kada različitim hipotezama i njihovim proverama pouzdano isključimo mogućnost da nešto pripada prostoru potencijalnih uzroka (pravilo 4), onda možemo to da isključimo iz daljeg razmatranja.

Pravilo 8 – Zatražiti tuđe mišljenje

U razvoju softvera, kao ni drugde uostalom, ne bi trebalo da bude egoizma, sujete ili ponosa. Ali ih, kao i drugde, ima i mogu da sprečavaju programera da zatraži savet od saradnika, prijatelja ili nekog nepoznatog eksperta. Argumentacija obično liči na pravdanja poput: „Ovo je moj deo softvera. Ja sam ga pisao i samo ja znam šta se tu dešava. Niko osim mene ne može da reši ovaj problem." Ali, trebalo bi da se vratimo korak nazad: „Da, ovo jeste moj deo softvera i niko drugi ga nije pisao osim mene. Ako problem postoji, onda sam ga *ja napravio*." A greške izvesno postoje, zato što se inače tim delom softvera ne bismo ni bavili u procesu debagovanja. Zašto bismo verovali da onaj ko je napravio grešku jeste pozvaniji i sposobniji da je otkloni nego neko drugi?

Naravno, deo argumentacije je sasvim ispravan – niko drugi ne poznaje neki deo koda bolje od njegovog autora. Ali u nekim slučajevima to nije prednost autora već njegov hendikep. U onim brojnijim slučajevima, kada je to prednost, autor je uglavnom u stanju da samostalno pronađe i otkloni grešku. Ali u onim preostalim slučajevima, u kojima ga traženje uzroka greške pošteno namuči, sva je prilika da bi neko sa strane mogao da bude od velike pomoći. Zašto?

Problem poznavanja nekog dela koda, ili čak i dugotrajnog bavljenja tim delom koda, na primer kroz pokušaje debagovanja, je u tome da poznavanje dovodi do pretpostavki. Svako dodatno iskustvo stvara nove pretpostavke. Hendikep je u tome što se takve pretpostavke prećutno, često i nesvesno, stvaraju i zatim više ne dovode u pitanje. Time se vraćamo na pravilo *7 - Proveravati i naizgled trivijalne stvari*. Ali, za razliku od uobičajene primene tog pravila, ovde postoji značajna otežavajuća okolnost – programer u praksi veoma teško uočava trivijalne pretpostavke koje su nastale kao rezultat njegovog sopstvenog iskustva u radu sa posmatranim delom koda. Jednostavno, obim podrazumevanja može da postane prevelik.

Spoljašnji posmatrač, koji ne poznaje dobro posmatrani deo koda, mogao bi da svojim pogledom „sa strane" bolje sagleda potencijalne slabosti koje promiču unutrašnjem posmatraču. Neko ko nije opterećen suvišnim pretpostavkama može

mnogo bolje da rasuđuje o posmatranom problemu. To je ujedno i jedna od najznačajnijih motivacija za traženje pomoći. Međutim, čak i kada zatražimo pomoć i dalje možemo da napravimo grešku. Iste lične slabosti koje mogu da nas sprečavaju u traženju pomoći, mogu da se ispolje i kada se pomoć zatraži, u vidu pravdanja i objašnjavanja šta smo to sve učinili bez uspeha. Takva suvišna objašnjenja mogu da imaju negativne posledice, zato što se kroz njih na saradnika prenose iste one pogrešne pretpostavke koje su nas i do tada ometale u radu. Ako svoje neispravne pretpostavke prenesemo na pomagača, onda možemo da potpuno obesmislimo njegovu pomoć.

Druga grupa motiva za traženje pomoći se odnosi na ograničene mogućnosti pojedinca u pogledu poznavanja brojnih tehnika i tehnologija koje se prepliću u savremenom razvoju softvera. Za svaku oblast postoje eksperti. Istovremeno, niko nije dovoljno dobar ekspert za sve oblasti koje su zastupljene u razvoju softvera. Koliko god da mislimo da dobro poznajemo neku konkretnu tehnologiju, uvek će postojati eksperti koji se bave *samo njom* i koji je poznaju mnogo bolje od nas. Umesto da gubimo sate i dane pokušavajući da razumemo neki problem, može da bude mnogo jednostavnije, efikasnije i jevtinije da se zatraži pomoć od eksperta. Sasvim je moguće da je neko već rešavao problem koji nas muči, pa nema razloga da ga ponovo rešavamo. U svetu mašinske industrije se pomalo šaljivo kaže da eksperti znaju gde da udare mašinu čekićem, pa da ona proradi. Naravno, većina nas bi čekićem samo nanela nova oštećenja. U softveru nema čekića, ali neki zahvati umeju da budu vrlo „grubi" i da se odnose na neke specifične konfiguracione parametre, za čije je ispravno određivanje potrebno veoma obimno i temeljno poznavanje i same biblioteke i nekih drugih vezanih tehnologija.

Od presudnog značaja je da se pri opisivanju problema saradniku navode samo simptomi. Pretpostavke se ne smeju saopštavati! Umesto da pozvanog saradnika zatrpamo svim bitnim i nebitnim, proverenim i neproverenim informacijama kojima raspolažemo, mnogo je produktivnije da sve te informacije sačuvamo za sebe sve dok nam ne bude postavljeno odgovarajuće pitanje, pa čak i tada moramo da budemo „štedljivi". Čak i kada mislimo da smo neke pretpostavke dokazali i da izvesno važe, ne smemo da ih olako prenosimo pomagaču. Potrebno je da pustimo pomagača da uči o sistemu neopterećen našim pogledom na problem. Naravno, pomagač će nam postavljati pitanja, ali bi odgovori koje mu dajemo trebalo da budu prvenstveno informativne prirode, tako da nude informacije a ne zaključke. Na primer, na pitanje „Da li je proveren taj i taj uslov?", odgovor ne bi nikako smeo da bude u obliku „Da, to je sigurno u redu!" već u opisnom obliku poput „Provereno je to i to na taj i taj način i rezultat je bio takav i takav." Ako smo od nekoga tražili pomoć, važno je da ga pustimo da proba da pronađe odgovore, umesto da mu ih sami nudimo – da su naši odgovori tačni, pomoć nam ne bi ni bila potrebna.

Danas je pristup informacijama mnogo lakši nego u prethodnim decenijama. Uloga Interneta u razvoju softvera je nezamenljiva. Dok je ranije „pasivna pomoć"

bila ograničena na različita uputstva, tutorijale, vodiče kroz probleme (engl. *troubleshooting guides*) i slične kolekcije iskustava, danas mnogo informacija može da se pronađe na ličnim veb lokacijama i blogovima, gde programeri zapisuju svoja zanimljiva iskustva. Diskusione grupe pružaju istovremeno sasvim specifičan vid aktivne i pasivne pomoći. Aktivnu pomoć možemo da zatražimo postavljanjem pitanja i očekivanjem da nam neko ponudi odgovor, a pasivna nam je na raspolaganju u obliku uvida u već vođene diskusije.

Pasivna pomoć je posebno korisna u slučajevima kada pretpostavljamo da smo naišli na neispravno ili nedokumentovano ponašanje neke komponente ili biblioteke koju koristimo. Princip je jednostavan – ako problem zaista postoji, relativno je velika verovatnoća da ga je neko drugi već uočio pre nas, kao i da ga je negde zapisao. Čak i ako taj problem nije u međuvremenu rešen i opisan, dodatni opis sličnog ili istog problema može da nam bude u pomoći pri njegovom rešavanju.

Pravilo 9 – Ako je nismo ispravili, onda greška nije ispravljena

Moglo bi se reći da je ovo pravilo toliko jasno da je nepotrebno, ali ipak nije tako. Izvesno ga nije teško razumeti. Osnovno značenje je sasvim jasno. Ipak, ovo pravilo ima nešto šire implikacije, koje u složenim okolnostima razvoja softvera ne smeju ni da se zanemare ni da se podrazumevaju.

Nije retkost da neki problem ne može da se ponovi. Pri tome se ne misli na problem čije ponavljanje ne umemo da izvedemo zbog nepoznavanja odgovarajućeg postupka (pravilo 2), nego na slučajeve kada procedura koja je proizvodila problem naprasno prestane da ga proizvodi. Razlozi za to su brojni: možda je neka druga izmena programskog koda uticala na (ne)pojavljivanje posmatranog problema, ili je neka komponenta zamenjena novom verzijom pa se ona sada ponaša drugačije, ili su neke okolnosti u okviru računarskog sistema izmenjene pa se zato problem ne ispoljava. Ipak, nama je važnije da je sasvim moguće da se problem i dalje ispoljava, ali na neki drugi način, pa mi to ne uočavamo. Možda je kvar i dalje prisutan, ali su izmenjeni putevi prenošenja infekcije, tako da više ne dolazi do manifestovanja očekivanog neuspeha.

Jedan od razloga da ne zanemarimo ovo pravilo je što praksa potvrđuje (a i čuveni Marfijev zakon nas uverava) da će problem koji nije rešen, a koji je naprasno prestao da se ispoljava, sasvim verovatno isto tako naprasno ponovo da se ispolji baš onda kada nam to bude najmanje odgovaralo. Ako je problem „nestao“ sam od sebe, to bi trebalo da nam sugeriše da nismo dovoljno dobro upoznali okolnosti u kojima se problem pojavljuje. Promena tih nepoznatih okolnosti je uticala na njegovo ispoljavanje. Ako se okolnosti ponovo promene, problem može ponovo da počne da se ispoljava, na isti ili drugi način i na istom ili drugom mestu.

Ako je promena nekih delova sistema imala za posledicu da se posmatrani problem više ne ispoljava, a pouzdano znamo da se niko nije bavio njegovim rešavanjem, to znači da je neka od načinjenih promena proizvela relativno široke

efekte, a to opet može da znači i da je proizvela i neke potencijalno loše posledice. Sasvim je moguće da su rešeni samo neki aspekti problema i da već neka minimalna promena niza koraka može ponovo da dovede do ispoljavanja problema.

Prestanak ispoljavanja problema često može da znači da je samo prestalo ispoljavanje uočenih simptoma. Ovaj slučaj možemo da uporedimo sa zamenjivanjem pregorelog električnog osigurača – uzroci pregorevanja i nakon zamene nastavljaju da postoje i mogu da se ispolje pregorevanjem novog osigurača, ali i na neki drugi, mnogo neugodniji način. Zbog toga se svim „nestalim" problemima mora posvetiti pažnja na isti način i u istoj meri kao problemima koji „nisu nestali".

Uobičajeno je da prvi korak u ovakvim slučajevima predstavlja ozbiljna provera da li je problem zaista popravljen. Jedan od načina je da se vratimo do poslednje verzije koda u kojoj se problem ispoljavao i da pokušamo da ga lokalizujemo. Zatim lokalizovan deo koda uporedimo sa novijim verzijama koda, u kojima se problem ne ispoljava. Ako uspemo da prepoznamo da je neka konkretna promena izmenila ponašanje na odgovarajući način, onda je moguće i da je problem sada zaista rešen.

Ovde je neophodno da budemo posebno oprezni – slučajno ili usputno rešavanje drugih problema je često nedovoljno temeljno. Zato proveru da li je greška popravljena moramo da izvodimo veoma pažljivo. Nekada je korisno da posle lokalizacije pristupimo i rešavanju problema na staroj verziji koda, pa da tek posle toga upoređujemo naše izmene sa onima koje je neko drugi napravio. Na taj način možemo da smanjimo uticaj već načinjenih izmena na naše rasuđivanje o problemu, što je važno zato što bismo inače pod takvim uticajem mogli da pomisimo da je slučajno rešenje potpunije i kvalitetnije nego što zaista jeste.

Jedan značajan aspekt ovog pravila se odnosi na pitanja kvaliteta razvojnog procesa. Ako imamo neke greške, to znači da razvojni proces ima nekih slabosti. Pri rešavanju problema smo često u prilici da uočimo uzroke njihovog nastajanja i da na taj način postepeno popravljamo i unapređujemo razvojni proces. Ako bismo poverovali da je neki problem nestao sam od sebe, ili da je „slučajno" rešen kroz neke druge popravke, time bismo propustili da sagledamo propuste u razvojnom procesu koji su doveli do njegovog nastajanja. Samim tim, mogli bismo da očekujemo da će se pojaviti neki drugi problem, kao posledica istih slabosti u procesu razvoja, a što bi moglo da se izbegne dodatnim posvećivanjem pažnje „nestalom" problemu.

Dodatna pravila

Većina skupova pravila debagovanja su uglavnom ekvivalentni sa ovde predstavljenim Agansovim skupom pravila ili nekim njegovim podskupom, ali

postoje i neka pravila koja nisu doslovno obuhvaćena Agansovim skupom, na primer[81]:

- Nacrtati dijagram;
- Opisati problem nekom standardnom metodologijom;
- Redukovati ulazne podatke koji su neophodni za ponavljanje problema;
- Razumeti zahteve;
- Pojednostaviti test-slučaj koji dovodi do ponavljanja problema;
- Pročitati tačnu poruku o grešci;
- Koristiti odgovarajuće alate;
- Ispratiti ispravljenu grešku novim testom
- i druga

Ako razmotrimo ova pravila, možemo da vidimo da, iako nisu neposredno obuhvaćena Agansovim pravilima, većina njih može da se izvede kao posledica ili specifičan vid primene njegovih pravila. Na primer, crtanje dijagrama i opisivanje problema nekom standardnom metodologijom su samo vrlo konkretne primene pravila 6 „Praviti i čuvati tragove izvršavanja", razumevanje zahteva je samo podskup šireg razumevanja sistema (pravilo 1) i slično.

Dodavanje novih dobrih pravila u našu svakodnevnu praksu ne može da škodi. Naprotiv, može da unapredi svakodnevnu praksu debagovanja. Sa druge strane, detaljnije posvećivanje većem broju pravila pri učenju heurističkog metoda debagovanja može da nepoželjno proširi priču i oteža razumevanje suštine metoda. Agansov skup predstavlja dobru polaznu osnovu, a svako od nas je slobodan da proširuje i prilagođava taj skup pravila prema sopstvenim merilima i potrebama.

12.6 Tehnike i alati za debagovanje

Tehnike i alate za prevenciju i otklanjanje grešaka u programskom kodu delimo na *spoljašnje* i *unutrašnje*.

Spoljašnje tehnike i alati su različiti programski alati koji su napravljeni za opštu primenu, a ne za neke specifične slučajeve. Takve tehnike i alati se koriste u svakodnevnom radu pri debagovanju različitih projekata.

Unutrašnje tehnike i alati su elementi koji se ugrađuju u programski kod, a nemaju nikakvu svrhu u kontekstu izvršavanja posla kome je program namenjen,

[81] Navedena pravila predstavljaju podskup od 13 pravila iz [*Grotker 2008*] i 9 pravila iz [*Metzger 2004*], a koja nisu neposredno ekvivalentna sa Agensovim pravilima.

već služe isključivo radi pomoći u prevenciji i otklanjanju grešaka. Često predstavljaju jednokratna sredstva, koja se razvijaju za rešavanje jednog specifičnog problema. Neki unutrašnji alati mogu da posluže i za rešavanje više različitih problema, ali vrlo retko van relativno ograničenog dela posmatranog programskog koda.

12.6.1 Spoljašnje tehnike i alati

Prevodilac

Kada se traže greške ili se radi na njihovom otklanjanju, veoma je važno da se program izvršava baš onako kako ga je programer napisao. Iako može da izgleda da je to normalno stanje stvari, u praksi stvari stoje malo drugačije. Prevodioci često optimizuju programski kod tako da prevod ne odgovara baš doslovno izvornom kodu. Na primer, prevodilac može da promeni redosled nekih delova koda ili nekih mašinskih instrukcija tako da se dobije ekvivalentno ali efikasnije izvršavanje, ili može da izbaci iz petlje neke delove koda i slično. Iako takve transformacije programa mogu ra doprinesu efikasnosti, one mogu da otežaju praćenje izvršavanja programa pa i pronalaženje i razumevanje grešaka. Zbog toga prevodioci imaju različite opcije prevođenja, koje mogu da nam olakšaju debagovanje.

Uobičajeno je da prevodioci i sistemi za izgradnju programa omogućavaju bar tri osnovna režima prevođenja:

- debagovanje (engl. *debug*);
- objavljivanje (engl. *release*) i
- objavljivanje sa debagovanjem (engl. *release with debug*).

Pri prevođenju za debagovanje je uobičajeno da se isključe sve ili skoro sve optimizacije. Na primer, ne dopušta se promena redosleda instrukcija, ne vrši se umetanje koda nego se svaka tzv. umetnuta funkcija prevodi kao prava funkcija i drugo. Pored toga, u prevedenu izvršnu datoteku (ili u pomoćnu prateću datoteku) se zapisuju pomoćne informacije za debagovanje, koje kasnije omogućavaju da se svaka mašinska instrukcija u prevedenom programu poveže sa konkretnom naredbom izvornog koda čiji deo prevoda ona predstavlja. Na osnovu toga, alati za debagovanje mogu da paralelno analiziraju mašinski kod i izvorni kod programa. Ovako prevedeni programi se najlakše debaguju, ali se najčešće izvršavaju daleko sporije nego programi prevedeni za objavljivanje.

Pri prevođenju za objavljivanje se omogućavaju sve optimizacije koje su programeri izabrali. Ne čuvaju se nikakve dodatne informacije o prevodu. Na taj način se dobija efikasan program, ali je njegovo eventualno debagovanje moguće vršiti samo na nivou mašinskog koda, zato što nema dovoljno informacija da se on

poveže sa izvornim kodom od koga potiče. Takva izgradnja programa je uobičajena na kraju razvoja, pri puštanju programa u upotrebu.

Treći režim prevođenja predstavlja kombinaciju prethodna dva režima. Sve optimizacije se omogućavaju, a u izgrađenu izvršnu datoteku (ili dodatnu pomoćnu datoteku) se zapisuju pomoćne informacije za debagovanje. Usled primenjenih optimizacija, u ovom režimu može da dođe do umetanja funkcija i menjanja redosleda instrukcija, pa je teže lokalizovati neke vrste problema, ali postoji dovoljno informacija da se svaka mašinska instrukcija u prevedenom programu poveže sa delom izvornog koda čiji prevod predstavlja. Ovako prevedeni programi su obično ili jednako efikasni ili umereno sporiji nego programi prevedeni za objavljivanje.

Većina alata za debagovanje podrazumeva da su raspoloživi pomoćni podaci za debagovanje. U tom smislu, oni često rade na isti način bilo da se radi o programu koji je preveden u režimu za debagovanje ili u režimu za objavljivanje sa dopunskim informacijama za debagovanje. Ipak, kao što je već naglašeno, program preveden za objavljivanje sa informacijama za debagovanje je nešto teže debagovati od programa prevedenog u režimu za debagovanje.

Pored prevođenja u različitim režimima, prevodioci mogu da nam pomažu u debagovanju i putem obezbeđivanja podrške za statičku ili dinamičku analizu koda, za čistače ili alate za proveru pokrivenosti koda. O tome će biti više reči u narednim odeljcima.

Debager

Najvažniji spoljašnji alat za debagovanje je debager. Debager (engl. *debugger*) je alat koji omogućava detaljno posmatranje stanja računarskog sistema i izvornog koda programa tokom izvršavanja programa. Primenom različitih tehnika se korisniku (tj. programeru koji otklanja neki problem) pruža pomoć u praćenju izvršavanja i posmatranju stanja programa, a sa osnovnim ciljem pružanja što informativnijeg uvida u rad programa i pomaganja pri lociranju uzroka problema.

Debageri mogu da budu sastavni deo većih razvojnih alata, tzv. integrisanih razvojnih okruženja (engl. *Integrated Development Environment - IDE*) ili samostalni proizvodi. Prednost integrisanih debagera je u tome što su obično vrlo dobro uklopljeni u razvojni alat i povezuju delove okruženja za razvoj i debagovanje. Sa druge strane, prednost samostalnih debagera je u mogućnosti njihovog povezivanja sa različitim razvojnim okruženjima i programskim jezicima i njihovim implementacijama. Primeri razvojnih okruženja sa integrisan debagerom su *MS Visual Studio* i *Eclipse*, dok je najpoznatiji primer samostalnog debagera *GNU* debager *GDB*, koji se koristi iz komandne linije ili pomoću neke od brojnih dodatnih implementacija vizualnog interfejsa.

Sve ono što može da se uradi pomoću debagera moglo bi (uz malo ili malo više truda) da se uradi i pomoću odgovarajućih unutrašnjih alata. Prednost debagera je u

tome što je daleko jednostavnije koristiti gotove alate nego ih praviti svaki put iznova kada nam zatrebaju. Navešćemo neke od osnovnih alata kojima raspolažu savremeni debageri:

Izvršavanje programa „korak-po-korak“ (engl. *step-by-step*). Programer ima priliku da prati izvršavanje programa od samog početka njegovog izvršavanja, tako što eksplicitno zahteva da se izvrši jedna po jedna naredna naredba. U slučaju svakog poziva potprograma može da se bira da li će se ući u telo potprograma i pratiti njegovo izvršavanje korak-po-korak (obično se takva komanda naziva „korak-unutra“, engl. *step into*) ili će se izvršiti pozivanje potprograma kao jedan veći korak (komanda „korak-iznad“, engl. *step over*). Većina debagera ima i opciju „izlazak iz potprograma“ kojom se automatski izvršavaju svi koraci tekućeg potprograma, zaključno sa povratkom na mesto odakle je on pozvan (komanda „korak-van“, engl. *step out* ili *step return*). Neki debageri imaju i opciju „izvršavanje do kursora“ (engl. *run to*), kojom se nalaže da se nastavi izvršavanje dok se ne dođe do naredbe koju je programer izabrao (npr. pozicioniranjem kursora editora).

Postavljanje tačaka prekida (engl. *breakpoint*). Izvršavanje velikih programa metodom korak-po-korak može da bude dugotrajno. Umesto toga, mnogo je efikasnije da pustimo da se program izvršava dok ne dođe do neke konkretne naredbe. Tačke prekida su oznake koje se postavljaju na naredbe u programu i koje sugerišu debageru da mora da stane kada dođe do neke od njih. Po dostizanju neke od tačaka prekida izvršavanje programa se privremeno zaustavlja i prepušta se kontrola programeru. Izvršavanje može da se nastavi korak-po-korak ili nastavljanjem izvršavanja do naredne tačke prekida (komanda „nastavak“, engl. *Resume* ili *Continue*).

Uslovne tačke prekida (engl. *conditional breakpoint*). Često je potrebno da se tačka prekida postavi na naredbi koja se izvršava veliki broj puta, ali tako da je poželjno da se program ne zaustavlja svaki put kada se dođe do tačke prekida, već samo u nekim specifičnim slučajevima. Uslovne tačke prekida omogućavaju da se uz tačku prekida definiše uslov koji će automatski da se proverava kada se ta tačka dostigne i na osnovu koga će debager da odlučuje da li je potrebno da se program zaustavi ili da se nastavi izvršavanje. Alternativa je da se broje prolasci kroz tačku prekida i da se program zaustavi tek nakon zadatog broja prolazaka.

Tačke prekida na podacima (engl. *data breakpoint*). Osim na naredbama, tačke prekida mogu da se postavljaju i na podacima. Tačka prekida na podatku se vezuje za neku oblast u memoriji i nalaže debageru da svaki put kada program pristupi toj zoni memorije, onda debager mora da prekine njegovo izvršavanje. Kao i u slučaju tačaka prekida u programu, i tačke prekida na podacima mogu da budu uslovne. Uslov može da obuhvati i vrstu pristupa, tako da se, na primer, program zaustavi samo ako se pokuša pisanje.

Tačke prekida u slučaju izbacivanja izuzetaka. Većina debagera omogućava da se automatski prekine izvršavanje programa neposredno pre izbacivanja izuzetka. Na taj način se programeru omogućava da posmatra stanje programa koje je dovelo do izbacivanja izuzetka.

Lokalne promenljive. Razlog za zaustavljanje programa u nekom trenutku ili za izvršavanje programa korak-po-korak je to što programer želi da posmatra stanje programa. Jedan od osnovnih elemenata stanja programa predstavljaju lokalne promenljive. Dok je program privremeno zaustavljen, debager omogućava detaljan uvid u lokalne promenljive.

Globalne promenljive. Lokalne promenljive predstavljaju samo uzak lokalni deo stanja programa. Za dobijanje šire slike moraju da se prate i posmatraju i vrednosti globalnih promenljivih. Globalnih promenljivih je često mnogo, pa nije praktično da se sve prikazuju. Zato debageri pružaju korisnicima mogućnost da sami navedu koje su to promenljive čije stanje žele da prate.

Pregledanje podataka. Složene strukture podataka ne mogu lako da se sagledaju samo posmatranjem vrednosti promenljivih, već je potrebno da se pažljivo pregledaju neki elementi podataka ili povezani podaci (atributi objekata, objekti na koje pokazuju pokazivači, atributi objekata na koje pokazuju pokazivači,...). Na primer, neki debageri umeju da prepoznaju i na odgovarajući način prikažu strukture podataka poput niza ili liste.

Praćenje stanja steka. Programski stek služi za prenošenje argumenata i rezultata potprograma i za čuvanje adrese povratka iz potprograma. Analizom stanja steka može da se utvrdi kako je tok programa došao od glavnog programa (funkcije `main` u jeziku *C*/*C++*) do potprograma koji se trenutno izvršava. Prikazivanjem sadržaja steka u preglednom obliku, debager omogućava programeru da vidi odakle je pozvan tekući potprogram, odakle je pozvan prethodni potprogram i tako dalje sve do nivoa glavnog programa. Štaviše, debager može da pruži programeru uvid u stanje lokalnih promenljivih i vrednosti argumenata u svakom od tih potprograma. Analiza sadržaja steka je jedan od najvažnijih aspekata upotrebe debagera. Praćenjem steka i stanja lokalnih promenljivih na različitim nivoima pozivanja može da se ustanovi odakle potiču neke eventualno neispravne vrednosti argumenata potprograma.

Interaktivno izvršavanje naredbi. Neki debageri omogućavaju da se, u periodima kada je izvršavanje programa privremeno zaustavljeno, izvršavaju izabrane ili eksplicitno napisane naredbe programskog jezika. Na taj način je moguće ostvariti složenije operacije čitanja ili analiziranja stanja programa, čime se omogućava detaljnije ispitivanje okolnosti koje dovode do problema ili predstavljaju posledicu problema. Štaviše, moguće je ostvariti i menjanje stanja programa, što nam omogućava da popravimo uočene neispravnosti ili da namerno proizvodimo specifične okolnosti koje će nas brže dovesti do ponavljanja problema.

Praćenje stanja niti. Debagovanje programa koji se izvršavaju u više niti je veoma složeno i naporno. Problem je u otežanom praćenju izvršavanja, analiziranju i praćenju sinhronizacije i komunikacije među nitima, što ima za posledicu otežanu lokalizaciju problema. Debager mora da omogući praćenje stanja svake od niti. Obično debageri omogućavaju da se jedna ili više izabranih niti privremeno suspenduju, kako bi se smanjio obim istovremenih promena stanja i olakšalo fokusiranje posmatranja na preostale niti. Iako se tako olakšava posmatranje stanja i toka izvršavanja programa, smanjivanje nivoa konkurentnosti može da ima za posledicu privremeno maskiranje uzroka i onemogućavanje nastajanja ili ispoljavanja problema, pa stoga može da bude kontraproduktivno i da nam otežа pronalaženje problema.

Praćenje toka programa na nivou mašinskih instrukcija. Obično je mnogo lakše, bolje i brže analizirati izvršavanje programa uz posmatranje izvornog koda programa, u obliku u kome je originalno napisan na konkretnom programskom jeziku, ali u nekim specifičnim slučajevima može da bude potrebno da se program posmatra i izvršava na nivou mašinskih instrukcija procesora. Većina debagera nam pruža tu mogućnost. Ona je korisna u slučajevima kada je potrebno proveriti da li je optimizacija izvedena na odgovarajući način ili da li je neki deo koda preveden tako da može da se izvršava neatomično. Posebno, mnogi savremeni prevodioci omogućavaju da se u okviru teksta programa na višem programskom jeziku, između konstrukcija tog jezika, eksplicitno navodi asemblerski zapis mašinskih instrukcija. To se naziva „umetnuti asemblerski kod“ (engl. *inline assembler*) i omogućava da se i programi koji zahtevaju maksimalne performanse razvijaju najvećim delom na višem programskom jeziku, a da se samo oni delovi koji su posebno osetljivi dodatno optimizuju primenom asemblera. Debagovanje delova programa koji su pisani sa umetnutim asemblerskim kodom je obično nemoguće bez praćenja izvršavanja programa na nivou mašinskih instrukcija. Sve što je navedeno za izvršavanje programa korak-po-korak i postavljanje tačaka prekida, važi i kada se program izvršava na nivou mašinskih instrukcija, s tim da se kao jedinice izvršavanja ne posmatraju naredbe i konstrukcije višeg programskog jezika nego mašinske instrukcije procesora.

Praćenje stanja procesora. Ako se tok izvršavanja programa prati na nivou mašinskih instrukcija, onda se stanje programa, makar na lokalnom nivou, mora pratiti posmatranjem stanja procesora. Stanje procesora čine vrednosti registara, uključujući i stanje registara zastavica, koje opisuje režime rada procesora i privremene podatke veličine jednog bita. Debageri, koji omogućavaju praćenje izvršavanja programa na nivou mašinskih instrukcija, po pravilu omogućavaju i praćenje stanja procesora.

Udaljeno debagovanje. U distribuiranim okruženjima, ili kada se razvija softver za neki specifičan uređaj, često nije moguće interaktivno upotrebljavati debager na sistemu na kome se izvršava program koji se debaguje. Mnogi savremeni debageri

podržavaju tzv. „udaljeno debagovanje“ (engl. *remote debugging*), kada se jedna komponenta debagera izvršava na istom sistemu gde i debagovani program, a druga na radnoj stanici koju interaktivno upotrebljava korisnik (tj. programer koji debaguje). Prva komponenta obuhvata sve sistemske i funkcionalne elemente debagera, a druga samo predstavlja korisnički interfejs. Naravno, između ove dve komponente mora da postoji komunikacija. Najčešće se upotrebljava uobičajena mrežna infrastruktura, ali se mogu upotrebljavati i neke druge posvećene veze, npr. putem serijskog ili *USB* interfejsa. U slučaju udaljenog debagovanja, arhitekture sistema koji se debaguje i radne stanice mogu biti potpuno nezavisne. Jedino ograničenje je da konkretna verzija debagera mora da podržava obe vrste računarskih sistema. Ako se ima u vidu da se kao radne stanice obično koriste standardizovane platforme (npr. *PC + Linux* ili *PC + Windows* ili *Apple*), koje su dobro podržane u pogledu alata za debagovanje, onda je potencijalan problem samo obezbeđivanje udaljene komponente debagera za konkretan računarski sistem ili uređaj.

Simulirano debagovanje. Predstavlja podvrstu udaljenog debagovanja. U slučaju simuliranog debagovanja se debagovani program i funkcionalna komponenta debagera izvršavaju u okviru odgovarajućeg simulatora sistema. Arhitektura simulatora je vrlo slična virtualnim mašinama, s tim da se pretpostavlja veća nezavisnost u arhitekturi matičnog i gostujućeg sistema. Iz ugla debagovanja, stvari na konceptualnom nivou funkcionišu skoro potpuno isto kao u slučaju udaljenog debagovanja.

Profajler

Profajleri (engl. *profiler*) su programi koji prate i analiziraju upotrebu resursa tokom rada programa. Osnovna namena profajlera je da pomognu u procesu optimizacije softvera, tako što olakšavaju uočavanje delova programa koji predstavljaju usko grlo. Pošto slabe performanse mogu da se smatraju za bag, profajleri, kao i optimizacija, mogu da imaju svoje mesto i u procesu debagovanja.

O profajlerima će biti više reči u okviru poglavlja o optimizaciji, u odeljku *13.6 Profajleri*, na strani 460.

Alati za dinamičku analizu

Alati za dinamičku analizu programa prave posebno okruženje za pokretanje programa, koje omogućava da se prate različiti parametri programa tokom njegovog izvršavanja. Obično simuliraju standardne biblioteke operativnog sistema ili čak stvaraju okruženje koje podseća na virtualnu mašinu ili tzv. kutiju sa peskom (engl. *sandbox*), tako da izvršavani program „ima osećaj“ da se izvršava u realnim uslovima, a da je okruženje zapravo potpuno kontrolisano od strane alata za dinamičku analizu.

Jedan od najpoznatijih alata za dinamičku analizu je *Valgrind*. *Valgrind* je kolekcija alata za dinamičku analizu koja obuhvata programe za praćenje operacija sa memorijom (alokacija, dealokacija, upotreba) i detekciju neispravnih pristupanja, curenja memorije i drugih problema; za praćenje upotrebe keša i uočavanje delova programa koji potencijalno rade sporije zato što ne koriste keš na pravi način; za praćenje pozivanja funkcija, slično profajlerima ali na egzaktnom nivou; za praćenje upotrebe globalne memorije (hip); detekciju grešaka u komunikaciji i sinhronizaciji niti i drugo.

Alati za dinamičku analizu programa spadaju u alate za ranu detekciju bagova i potencijalnih slabosti u kodu. Imaju mesto u razvoju negde između testiranja jedinica koda i testova integracije. Često se prave posebni probni poslovi koji se podvrgavaju dinamičkoj analizi, kako bi se videlo da li ima nekih nepredviđenih dešavanja tokom rada programa.

Alati za statičku analizu

Alati za statičku analizu programa analiziraju programski kod bez njegovog pokretanja i ukazuju na uočene neispravnosti ili na delove koji sadrže potencijalne slabosti. Među otvorenim alatima za statičku analizu progama pisanih na programskom jeziku *C++* se izdvajaju *Cppcheck* i *Clang Static Analyzer*. Oni pronalaze različite vrste problema i svaki ima svojih specifičnosti. Mogu da se povežu sa alatima za upravljanje složenim projektima, kao što su *CMake*, *Visual Studio* i drugi.

Alati za statičku analizu programa spadaju u alate za prevenciju bagova, zato što nam pomažu da se neke potencijalne slabosti uoče na vreme. Redovna primena alata za statičku analizu bi trebalo da pri pisanju programa ima slično mesto kao i otklanjanje svih potencijalnih problema koji proizvode upozorenja prevodilaca. Oni nam pomažu da blagovremeno uočimo i popravimo neke nedoterane i potencijalno osetljive tačke u kodu.

Alati za proveru pokrivenosti koda

Alati za proveravanje pokrivenosti koda (engl. *coverage testing*) služe, pre svega, da pronađu delove programa koji pri izvršavanju nijedanput nisu upotrebljeni. Provera može da se izvodi kako na nivou potprograma, tako i na nivou pojedinačnih naredbi ili linija programskog koda. Provera pokrivenosti programskog koda može da se koristi:

- pri debagovanju, da se proveri da li se prolazi kroz neki deo programskog koda, čiji rezultati rada se možda ne vide;
- pri optimizaciji, da se vidi koji delovi programa se češće izvršavaju i
- pri testiranju jedinica koda, provera da li su svi delovi koda testirani.

Primer alata koji proverava pokrivenost koda je `gcov`.

Čistači

Čistači (engl. *sanitizer*, moglo bi se prevesti i kao *alati za dezinfekciju*) su alati koji se isporučuju u okviru prevodilaca i rade sličnu stvar kao alati za dinamičku analizu, ali tako što se fizički ugrađuju u prevedenu izvršnu verziju programa. Namena čistača je da prilikom izvršavanja programa evidentiraju sve aktivnosti koje su od značaja (u skladu sa upotrebljenim opcijama prevodioca) i izveštavaju programera o uočenim neispravnostima.

Prevodioci `Clang` i `g++` imaju veliki broj opcija oblika `-fsanitize=...`, kojima se uključuju odgovarajući čistači. Neke od važnijih opcija su:

- `address` – brza detekcija grešaka u radu sa memorijom;
- `pointer-compare` – detektuje se poređenje pokazivača iz različitih alociranih prostora;
- `pointer-subtract` – detektuje se oduzimanje pokazivača iz različitih alokaciranih prostora;
- `shadow-call-stack` – detekcija grešaka u radu sa stekom;
- `thread` – detekcija nadmetanja za resurse (engl. *race condition*);
- `leak` – detekcija curenja memorije i
- `undefined` – detekcija nedefinisanog ponašanja.

Čistači imaju neke prednosti, ali i neke mane u odnosu na programe za dinamičku analizu. Za razliku od alata za dinamičku analizu, čistači se ugrađuju u prevod programa i već pri ugradnji imaju neposredan pristup izvornom kodu programa, što im omogućava da neke pripremne radnje obave praktično u fazi prevođenja. Zbog toga su obično daleko brži. Sa druge strane, kada je program preveden, u njega su već ugrađeni neki čistači i ne mogu da se naknadno dodaju novi, što ovu tehniku čini manje fleksibilnom. Naravno, iako su čistači brži od alata za dinamičku analizu, njihovo prisustvo i rad svakako usporavaju izvršavanje programa.

Među važnije razlike između čistača i programa za dinamičku analizu možemo da ubrojimo:

- čistači su obično daleko brži;
- čistači mogu da se precizno podešavaju opcijama prevodioca;
- čistači su novija tehnologija i često su slabije dokumentovani;
- čistači ne mogu da pomognu ako je problem u nekoj upotrebljenoj biblioteci, a ne u našem kodu;
- neke opcije čistača ne mogu da se koriste zajedno, pa za neke složene provere može da bude potrebno da se program izgrađuje više puta sa različitim opcijama.

12.6.2 Unutrašnje tehnike i alati

Unutrašnje tehnike i alati su različiti elementi koji se ugrađuju u programski kod ali nemaju svrhu u kontekstu izvršavanja posla kome je program namenjen, već služe isključivo radi pomoći u prevenciji i otklanjanju grešaka. U najvažnije unutrašnje tehnike spadaju:

- proveravanje pretpostavki:
  - o stanju ulaznih argumenata na početku potprograma;
  - o stanju promenljivih u toku algoritma;
  - o stanju promenljivih na kraju potprograma;
- pravljenje tragova izvršavanja:
  - o prolasku kroz neku tačku programa;
  - o stanju promenljivih u nekim tačkama programa;
- umetanje specifičnih delova koda
  - o radi sprovođenja složenijih analiza stanja;
  - o radi izvođenja istog posla na drugi način;
  - o radi pravljenja čitljivog zapisa stanja ili dela stanja programa;
- dodavanje testova jedinica koda:
  - o u skladu sa proverenim pretpostavkama;
  - o u skladu sa neproverenim pretpostavkama;
- pisanje jasnog i razumljivog koda:
  - o dobro imenovanje promenljivih i potprograma;
  - o komentarisanje programskog koda u skladu sa proverenim pretpostavkama
  - o i drugi elementi.

Testovima jedinica koda je posvećeno posebno poglavlje (*9 - Razvoj vođen testovima*, na stranici 211), a pisanjem jasnog i razumljivog koda se bavimo kroz celu knjigu. Ovde ćemo da posvetimo pažnju preostalim navedenim tehnikama.

Proveravanje pretpostavki

Proveravanje pretpostavki (engl. *assertion*) u programskom kodu je tehnika koja omogućava da se na odgovarajućim mestima ugrade automatske provere nekih uslova, za koje se očekuje da na tim mestima moraju da važe. U slučaju evenualne neispunjenosti proveravanog uslova, uobičajeno je da se prekine rad programa i na određeni način saopšti odgovarajuća poruka. U retkim slučajevima se reaguje ispisivanjem poruke i nastavljanjem rada programa.

Pretpostavke se proveravaju nekim oblikom funkcije (ili makroa) `assert`.

U narednom primeru je predstavljeno kako u kodu pisanom na programskom jeziku *C++*[82] može da se proverava pretpostavka da je argument funkcije `fact` u odgovarajućem opsegu:

```
#include <cassert>
...
int fact( int n )
{
    assert( n>=0 );
    assert( n<100 );
    int r = 1;
    while( n > 1 )
        r *= (n--);
    return r;
}
```

Uobičajeno je da se proveravaju sasvim jednostavne pretpostavke. Ako je potrebno da se proveravaju neki složeni uslovi, preporučljivo je da se oni podele na manje delove pa da se svaki od njih posebno proverava. Razlog za to je jednostavan – ako neki složen uslov nije ispunjen, onda se pri analizi problema neće znati koji konkretan deo tog složenog uslova nije ispunjen. Sa druge strane, ako nije ispunjen neki jednostavan uslov, onda je jasnije o čemu se radi. Zato su u prethodnom primeru navedene dve odvojene pretpostavke za proveravanje uslova `n>=0` i uslova `n<100`.

U najčešće proveravane uslove spadaju provere ispravnosti opsega argumenata ili međusobne usklađenosti vrednosti različitih argumenata potprograma. Takve provere same za sebe ne govore mnogo, ali ako se pokaže da neki potprogram pozivamo sa neispravnim argumentima, onda znamo da je potrebno da problem tražimo na mestima gde se taj potprogram upotrebljava.

U složenijim potprogramima može da bude potrebno da se proverava stanje promenljivih u toku izvođenja nekog algoritma. Provere međurezultata se najčešće implementiraju kao provere ispravnosti grananja, provere ispravnosti iteracija i provere ispravnosti spajanja. Provere ispravnosti grananja se ugrađuju na početku svake od grana pri uslovnom grananju i predstavljaju dodatnu proveru važenja uslova koji u svakoj od grana moraju biti ispunjeni. Provera ispravnosti iteracije služi da se proveri da li su na početku iteracije ispunjeni uslovi koji moraju da važe svaki put pre početka narednog ponavljanja. Provera ispravnosti spajanja se koristi na mestima spajanja različitih grana (npr. iza bloka `if-then-else`) ili iza naredbi

[82] Skoro isto je i u programskom jeziku *C*, s tim da se uključuje zaglavlje `<assert.h>` umesto `<cassert>`.

ponavljanja, da bi se proverilo da li je odgovarajuća celina koda ispravno uradila posao za koji je zadužena.

Često može da bude korisno da se proveravaju i rezultati potprograma. Ako izlazne vrednosti potprograma ne zadovoljavaju određene uslove u odnosu na ulazne vrednosti, onda znamo da u konkretnom potprogramu imamo grešku.

Proveravanje ispravnosti ulaznih podataka predstavlja deo pisanja robusnih programa i vid prevencije grešaka. Za razliku od toga, proveravanje ispravnosti izlaznih podataka i stanja promenljivih u toku potprograma obično se preduzima tek u fazi debagovanja. Međutim, kada se provere pretpostavki jedanput napišu, dobro je da se one ostave u programu, tj. da se ne brišu nakon pronalaženja i ispravljanja tražene greške.

U većini programskih jezika i prevodilaca, kao i u nekim od alata ili biblioteka, postoji način da se prevodiocu naglasi da li je proveravanje pretpostavki potrebno da se uključi u prevedeni izvršni program ili ne. Na taj način se omogućava da se proveravanje pretpostavki ugradi u razvojne verzije, koje se upotrebljavaju u početnim fazama testiranja i pri debagovanju, a da se iz konačne izvršne verzije softvera one isključe da ne bi umanjivale performanse sistema. U slučaju programskih jezika *C* i *C++*, za proveravanje pretpostavki se obično koristi makro `assert`, koji proverava stanje datog uslova i prekida rad programa ako uslov nije zadovoljen (kao u prethodnom primeru). Pretpostavke se proveravaju samo u fazi debagovanja programa, dok je u slučaju produkcione verzije ovaj makro definisan kao prazan kod i ne proizvodi nikakvo dejstvo. Produkciona verzija se prepoznaje po predefinisanom makrou `NDEBUG`, koji nije definisan u verziji za debagovanje i testiranje.

U mnogim projektima se prave dve vrste pretpostavki, koje se razlikuju samo po tome što se jedna od njih zadržava i u izvršnim verzijama prevedenog softvera, a druga ne. Provere pretpostavki koje ostaju u izvršnim programima mogu da se naprave i tako da ne prekidaju rad programa, već da samo izdaju odgovarajuće obaveštenje, zapisuju informacije u dnevnik događaja i zatim omogućavaju nastavak rada programa. Takve provere omogućavaju prikupljanje informacija koje mogu da olakšaju otkrivanje i prepoznavanje problema uočenih u fazi eksploatacije softvera. Nije dobra praksa da se sve provere pretpostavki ostavljaju u izvršnom kodu, zbog potencijalnog ugrožavanja performansi. Čak i jednostavne provere, ako se nalaze u petljama i potprogramima koji imaju jednostavna tela a izvršavaju se veoma često, mogu da utiču veoma negativno na efikasnost izvršavanja.

U programskom jeziku *C++* (od verzije 11) postoji i mogućnost *statičkog* proveravanja pretpostavki. Radi se o pretpostavkama čije važenje se proverava u fazi prevođenja programa i eventualna greška se prijavljuje već u toj fazi, tako da ne uzima vreme u fazi izvršavanja. Takve pretpostavke mogu da se navode na različitim mestima, na primer:

```
template<typename T, unsigned N>
class Point {
    static_assert( N < 100, "Dimension too high" );
    ...
};

static_assert( sizeof(Point<double,70>) <= sizeof(buffer) );
```

U zavisnosti od korišćenih alata, ali i konkretnih razvojnih projekata, mogu da se definišu i koriste različiti alati za pisanje pretpostavki. Na primer, ako se ispostavi da detaljno proveravanje pretpostavki u toku nekog potprograma može da unese neprihvatljivo usporenje u program (iako samo u fazi debagovanja), onda opcijama prevođenja i specifičnim definicijama u kodu može da se upravlja uključivanjem i isključivanjem proveravanja pretpostavki u različitim delovima programa.

Pravljenje tragova izvršavanja

Pravljenje tragova izvršavanja programa se obično implementira kroz upotrebu različitih makroa ili potprograma, koji zapisuju odgovarajuće informacije u nekoj datoteci ili u izlaznom toku za greške. Tragovi izvršavanja programa mogu da budu veoma obimni. Zbog toga, da bi se lakše pratili i analizirali, uobičajeno je da svaki zapis sadrži tačno vreme i mesto izdavanja. Mesto izdavanja služi za lakše lociranje mesta u kodu na kome je poruka izdata i obično obuhvata naziv potprograma i/ili naziv izvornog fajla programa i broj reda u njemu.

Slično kao i proveravanje pretpostavki i pravljenje tragova može da opterećuje performanse softvera. Štaviše, pravljenje tragova može da bude mnogo intenzivnije i obimnije, pa i posledice po performanse mogu da budu mnogo veće. Zbog toga je uobičajeno da se tragovi izvršavanja ne uključuju u izvršne verzije programa, a ako je ipak neophodan neki vid praćenja rada radi kontrole kvaliteta, onda se to radi u sasvim ograničenom obimu.

U velikim projektima se često koriste ili čak implementiraju posebne biblioteke za pravljenje tragova, koje sadrže različite potprograme, kao i različite opcije detaljnosti ili formata ispisivanja tragova. Tako se postiže da potrebne informacije mogu da se izdaju u odgovarajućim prilagođenim formatima, ali i da tragovi izvršavanja mogu da se analiziraju, pretražuju ili upoređuju primenom različitih alata.

Na primer, na programskom jeziku *C++* bi makro `DEBUG_TRACE_CERR`, za pravljenja traga izvršavanja u standardnom toku za greške, mogao da se implementira i koristi na sledeći način:

```
#ifdef NDEBUG
   #define DEBUG_TRACE_CERR(msgs)
#else
   #define DEBUG_TRACE_CERR(msgs)                    \
      std::cerr                                      \
      << " * TRACE: " << __FUNCTION__ << std::endl   \
```

```
        << " *         " << __FILE__                    \
        << " [" << __LINE__ << "]" << std::endl         \
        << " *         " << msgs
#endif

...{...
   DEBUG_TRACE_CERR( "[x=" << x << "]" );
...}...
```

Umetanje specifičnih delova koda

Greške obično nastaju u složenim delovima programa ili su bar povezane sa složenim stanjem koje se prenosi (implicitno ili eksplicitno) kroz različite delove programa. Zbog toga ispisivanje jednostavnih tragova izvršavanja, koji sadrže samo lokacije ili pojedinačne vrednosti promenljivih često može da bude nedovoljno informativno.

U takvim slučajevima je obično neophodno da se dodaju posebni delovi programa koji služe isključivo za olakšavanje postupka debagovanja i ne bi trebalo da imaju bilo kakvog uticaja na produkcionu verziju softvera. Takvi delovi programa mogu da imaju različitu formu i namenu.

Po formi razlikujemo potprograme i isečke koda. Ako pišemo posebne potprograme, koji služe samo za debagovanje, onda nas oni ne brinu previše, zato što njihovo postojanje samo po sebi ne utiče na produkcionu verziju – ako se neki potprogram ne koristi, onda on ne troši vreme i resurse osim mesto u prevedenoj verziji programa, ali zapravo ne troši čak ni to, zato što se u fazi povezivanja programa obično odbacuju delovi koji se nigde ne koriste. Sa druge strane, ako u postojeće potprograme ubacujemo specifične isečke koji nam služe pri debagovanju, onda bi bilo dobro da imamo sredstvo za njihovo isključivanje iz produkcione verzije programa.

Najčešća namena umetnutih isečaka koda jeste pravljenje čitljivog zapisa stanja programa, obično prevođenjem neke složene strukture u tekst. Takav zapis zatim može da se koristi u okviru tragova izvršavanja ali može i da se posmatra u okviru primene nekog spoljašnjeg alata. Na primer, ako privremeno zaustavimo izvršavanje programa, onda možemo da vidimo stanje svih lokalnih promenljivih, pa tako i stanje tako pripremljenog čitljivog zapisa stanja.

U takvim slučajevima možemo da koristimo delove koda poput:

```
std::string opis1 = ...;
std::string opis2 = ...;
```

Da bismo takav isečak koda imali samo u verziji za debagovanje, možemo da napravimo i da koristimo makroe nalik na naredni primer:

```
#ifdef NDEBUG
   #define DEBUG_CODE(x)
#else
   #define DEBUG_CODE(x) x
#endif

...{...
   DEBUG_CODE(
      std::string opis1 = ...;
      std::string opis2 = ...;
   )
...}...
```

Pored ovakve namene, umetnuti segmenti programa mogu da imaju i mnogo složenije primene. Na primer, mogu da obavljaju dodatna izračunavanja ili analize da bi se automatski proverila ispravnost stanja programa na konkretnom mestu i u konkretnom trenutku. Zbog lakšeg snalaženja u programu, preporučljivo je da složenije analize i obrade budu implementirane u dodatnim pomoćnim potprogramima, a da se u potprograme koje debagujemo ubacuju samo njihovi pozivi.

Umetanje pomoćnih delova koda ima posebno važnu ulogu u kontekstu mogućnosti interaktivnog izvršavanje naredbi u okviru debagera.

12.6.3 Strategije i taktike debagovanja

Opisivanjem neformalnog, naučnog i heurističkog debagovanja nismo iscrpli sve pristupe debagovanju. Na primer, Čarls Mecger u svojoj knjizi [*Metzger 2004*] predlaže da se postupak debagovanja veže za matematički (naučni) metod i u okviru toga sugeriše da se koriste strategije, heuristike i taktike debagovanja. Heuristikama smo već posvetili pažnju, pa ćemo ovde ukratko prikazati strategije i taktike. Primetimo da postoji preklapanje nekih strategija i taktika sa predstavljenim heuristikama.

Strategije predstavljaju viši nivo organizovanja i planiranja postupka debagovanja. Uglavnom su oblikovane prema različitim strategijama pretraživanja složenih struktura podataka. Izražene su u vidu algoritama, gde ulaz predstavlja skup svih segmenata programskog koda u kome se pretpostavlja da bi trebalo da je greška, a izlaz je redukovan skup segmenata (poželjno jednočlan). Pomenimo neke od njih:

- strategija binarnog pretraživanja – delimo skup na dva dela i proveravamo u kom skupu je greška;
- strategija pohlepnog pretraživanja – bira se jedan po jedan segment iz ulaznog skupa i proverava se da li je u njemu greška, pri čemu metod izbora daje prioritet onim segmentima koji na osnovu nekog kriterijuma izgledaju kao verovatnija lokacija greške;

- strategija pretragom u širinu – polazeći od najšireg poziva u kome se detektuje greška, provera se svaki poziv potprograma, pa ako se u njemu detektuje greška, onda se svi njegovi pozivi potprograma dodaju kao kandidati na kraj liste za proveravanje;
- strategija pretragom u dubinu – slično kao prethodni slučaj, ali se pozivi potprograma dodaju na početak liste;
- strategija programskih isečaka – pravljenje i testiranje isečaka programa, tako da oni obuhvataju delove izračunavanja ili promene stanja povezanih sa manifestacijom greške;
- strategija deduktivne analize – u osnovi počiva na uopštenom postavljanju hipoteza deduktivnim zaključivanjem i
- strategija induktivne analize – u osnovi počiva na uopštenom postavljanju hipoteza induktivnim zaključivanjem.

Taktike su manji postupci (često elementarni), koji se preduzimaju u okviru primene širih heuristika i strategija, za efektivno proveravanje da li je greška u nekom posmatranom delu programskog koda. U knjizi je u vidu kataloga predstavljena 21 taktika. Veći broj taktika predstavlja primenu već obrađenih unutrašnjih i spoljašnjih tehnika, ili može da se svede na savete za upotrebu alata za dinamičku analizu i čistača. Ovde ćemo navesti samo neke od preostalih taktika:

- čitati izvorni kod i tražiti probleme;
- praviti izveštaje o kompletnom stanju memorije;
- zameniti lokalne promenljive globalnim;
- prevoditi program do nivoa asemblerskog koda;
- proveriti da li greška postoji i kada se koristi prevodilac drugog proizvođača;
- isprobati prevođenje i izvršavanje programa na drugom operativnom sistemu.

12.7 Prevencija propusta

U uvodnom delu ovog poglavlja smo videli da upravljanje propustima obuhvata aktivnosti na prevenciji propusta i aktivnosti na otklanjanju propusta. Do sada smo se uglavnom bavili otklanjanjem propusta. Videli smo da je debagovanje složen i zahtevan proces, koji uz to i nije posebno prijatan. Sada je vreme da se posvetimo prevenciji propusta i da vidimo da li postoji neki način da doprinesemo smanjenju broja grešaka koje se prave tokom rada na razvoju softvera, a time i smanjenju količine radnog vremena koje ćemo da provedemo „uživajući“ u debagovanju.

U radnom okruženju mogu da postoje neke okolnosti koje posebno doprinose nastanku propusta. Takođe, postoje i okolnosti koje doprinose smanjivanju učestalosti i ozbiljnosti propusta. Veliki značaj imaju i okolnosti koje ne utiču presudno na nastajanje propusta, ali kasnije mogu da doprinesu njihovom lakšem uočavanju, lociranju i otklanjanju.

Osnovni cilj prevencije propusta je stvaranje uslova za nastajanje što manjeg broja propusta i njihovo blagovremeno uočavanje. Redovnim sagledavanjem i proveravanjem uslova rada i odnosa u timu obično može da se blagovremeno uoči prisustvo ili odsustvo poželjnih ili nepoželjnih okolnosti. Zatim uslovi i način rada mogu da se unapređuju ili prilagođavaju, kako bi se stanje popravilo.

Okolnosti koje pogoduju nastajanju propusta

Među najvažnijim okolnostima, koje doprinose nastajanju propusta, su nedovoljna stručnost tima i nepotrebno povećan nivo stresa u timu.

Nedovoljna stručnost se različito ispoljava na različitim nivoima organizacije tima. Primer nestručnog ponašanja na poziciji programera je primena pristupa „kodiraj pa razmišljaj“. Brzopleto pisanje koda stvara uslove za nastajanje mnogih propusta, od pogrešnog tumačenja specifikacije, pa sve do pravljenja grešaka u algoritmima i njihovoj implementaciji. Na višem nivou, nestručnost i žurba često dovode do lošeg razumevanja zahteva, nedovoljno dobre analize problema i na kraju izrade neispravnog projekta, koji nije u skladu sa postavljenim zahtevima. Konačno, ako stručnost izostane na nivou rukovodstva tima, onda se kao posledica dobija odsustvo posvećenosti kvalitetu, a time se stvara i prostor za brojne propuste u svim fazama razvoja i svim delovima softvera koje tim razvija.

Nepotrebno povećanje nivoa stresa u razvojnom timu je najčešće posledica lošeg planiranja ili loših međuljudski odnosa u timu. Loše planiranje obuhvata sve vidove lošeg raspoređivanja resursa, uključujući i nedovoljan broj kadrova i njihovu slabu obučenost. Svi oblici nestručnosti u timu na neki način, pre ili kasnije, negativno utiču i na međuljudske odnose i na rokove za dovršavanje razvoja, pa time posredno i na povećavanje nivoa stresa u timu. Na rukovodstvu razvojnog tima leži najveća odgovornost za stvaranje dovoljno velikog i dovoljno stručnog tima. Ovaj aspekt problema je obično dobro prepoznat, ali to ipak često nije prepreka da se zapostavlja.

Loši međuljudski odnosi neminovno vode lošem razumevanju između članova tima, što stvara posebno pogodne uslove za različita tumačenja zahteva, specifikacija i drugih vidova dokumentacije. Pored toga, članovi tima koji koji nemaju dobru komunikaciju sa svojim saradnicima najpre postaju nezadovoljni, a zatim često i dodatno opterećeni činjenicom da moraju da sarađuju sa osobama sa kojima se ne razumeju. To podiže nivo stresa kod pojedinaca, što se neminovno preslikava i na tim kao celinu.

Osnovni problem sa povećanim nivoom stresa je u dugotrajnom izlaganju tima otežanim uslovima rada. Dok je većina ljudi u stanju da na kratkotrajan stres odreaguje pozitivno i da u takvim uslovima pruži možda i više nego što bi inače pružila, dotle su posledice izlaganja dugotrajnom stresu upravo suprotne – čak i veoma sposobni i stručni članovi tima će u slučaju produženog stresa početi da prave neuobičajeno mnogo grešaka. Različite su vrste stresa koje mogu da opterećuju članove tima i ceo tim. Neke od njih mogu biti prisutne u praktično svakom razvojnom okruženju, kao, na primer: prekovremeni rad, različite vrste pritisaka (obećavanje nagrada, pretnje kaznama, učestalo ponavljanje i insistiranje na neostvarivim planovima i drugo), loši međuljudski odnosi, loši ambijentalni uslovi (neprovetrenost, manjak svetla, loša oprema i slično), neprilagođene radne procedure (komplikovani i spori formalni postupci, mnogo dokumentacije,...) i drugo. Sa druge strane, svako konkretno radno okruženje ima neke specifične karakteristike, što sa sobom nosi i mogućnosti za nastanak nekih specifičnih vrsta stresa.

Na rukovodstvu razvojnog tima leži najveća odgovornost za stvaranje dobrih uslova za rad i prepoznavanje i suzbijanje nepotrebnog stresa i loših međuljudskih odnosa, ali rukovodstvo to ne može da postigne bez dobre saradnje svih članova tima. Važno je imati na umu da je dobar tim, koji je sastavljen od dobrih pojedinaca, često daleko uspešniji nego loš tim sastavljen od izuzetnih pojedinaca. Usklađenost članova tima i kvalitet njihovih međusobnih odnosa često su važniji od sposobnosti pojedinačnih članova tima.

Okolnosti za izbegavanje propusta

Kao što postoje okolnosti koje pogoduju nastajanju propusta, tako postoje i one koje smanjuju mogućnost nastajanja propusta. Neke od njih neposredno ili posredno sprečavaju ispoljavanje ili smanjuju značaj već pominjanih okolnosti koje pogoduju nastajanju propusta. Druge donose u razvojno okruženje neke nove kvalitete, koji doprinose uspešnom radu.

Neke od najznačajnijih okolnosti za izbegavanje (ili bar smanjivanje verovatnoće nastajanja) propusta su:

- dobri odnosi i komunikacija u timu;
- stalno usavršavanje članova tima;
- dobra komunikacija sa klijentom;
- relativno opušteni uslovi rada;
- sistematičnost i obezbeđivanje kvaliteta;
- i druge.

U prethodnom odeljku smo videli da je nedovoljna stručnost članova tima (uključujući i rukovodioce) jedan od glavnih faktora koji utiču na nastajanje grešaka.

u skladu sa tim, neke od navedenih okolnosti za izbegavanje propusta se, posredno ili neposredno, odnose na problem stručnosti. Naravno, osnovni put prema dobroj osposobljenosti članova tima vodi preko pažljivog odabira kadrova i njihovog stalnog usavršavanja. Ako je tim već formiran, onda nema mnogo prostora da se utiče na odabir kadrova, ali ako se ustanovi da među članovima tima postoji značajan nedostatak osposobljenosti u nekoj od oblasti, onda možda može da se sugeriše nadređenima da je potrebno obezbediti pojačanje timu, ili bar dodatnu obuku postojećih članova.

Uvek ima i mora da bude prostora za stalno usavršavanje članova tima. To je najpouzdaniji način za redovno i sistematično podizanje nivoa stručnosti i pojedinaca i celog tima. Ako govorimo o bagovima, onda se „stručnost" najpre odnosi na poznavanje alata koji se upotrebljavaju u svakodnevnom radu, a pre svega na dobro poznavanje principa programiranja, programskih jezika, prevodilaca i drugih alata. Međutim, ako se problem posmatra malo šire, onda se „stručnost" odnosi i na principe projektovanja, na šire razumevanje sistema koji se razvija, kao i okruženja u kome bi on trebalo da funkcioniše, kao i na različite aspekte problema čijem je rešavanju tim posvećen.

Često se prenebregava da je za usavršavanje članova tima potrebno vreme, a da bi se to vreme obezbedilo, neophodno je da se smanji angažovanje na nekoj drugoj strani. Ne može da se očekuje da će programeri, koji rade prekovremeno na nekim veoma složenim problemima, pa možda i u drugim stresnim uslovima, imati dovoljno vremena, motiva i energije da se dodatno usavršavaju. Zato je dobro da se planom rada predvidi neko vreme za čitanje i učenje, makar to bilo i svega par sati nedeljno. U radu tima povremeno može da nastupi period smanjene aktivnosti, kada iz nekog razloga postoji smanjeni pritisak rokova i obima posla (na primer, ako se čeka na neku značajnu odluku klijenta). Takve periode je dobro iskoristiti za usavršavanje članova tima, kako kroz pojedinačni rad, tako i kroz zajednički rad na nekim temama za koje je ustanovljeno da zahtevaju bolje poznavanje.

Kao drugi značajan faktor uticaja na nastajanje bagova smo naveli stresne uslove rada. Nezadovoljstvo i stres mogu da nastaju kao posledica mnogih dešavanja na radnom mestu, pa i suočavanje sa tim problemom mora da ima odgovarajuću širinu. Uticaj međuljudskih odnosa na nivo stresa je među najvažnijim faktorima, čak do te mere da predstavlja jedan od glavnih faktora zbog kojih se zaposleni odlučuju da ostanu u timu i preduzeću ili da počnu da traže novu radnu sredinu. Čak i neki od jednostavnih elemenata radnog okruženja mogu da neposredno utiču na opuštenu radnu atmosferu, na primer udobna radna stolica, odgovarajuća visina radnog stola, kvalitetan interfejs računara (monitor, tastatura, miš), prijatan ambijent i drugo, ali međuljudski odnosi u timu su svakako daleko važniji.

Sistematičnost i rad na obezbeđivanju kvaliteta imaju višestruk uticaj na nastajanje grešaka. Sa jedne strane, tehnike koje se tu koriste (testovi jedinica koda,

testovi prihvatljivosti, pisanje robusnog softvera, upravljanje rizicima i druge) imaju za osnovni cilj olakšavanje blagovremenog uočavanja problema, o čemu je već bilo reči. Ali sa druge strane, sistematičnost i konkretne pojedinačne tehnike značajno doprinose i ugodnosti na radnom mestu. Praktično sve tehnike koje se uvode sa ciljem uređivanja procesa razvoja istovremeno imaju i pozitivan uticaj na smanjenje stresa u timu. Sistematičan razvojni proces smanjuje prisustvo neizvesnosti i omogućava ujednačeniji radni ritam, što svima mnogo više odgovara od svakodnevne promene ritma.

Dobra komunikacija sa klijentom može da se posmatra i u okviru staranja o kvalitetu, ali i u okviru razmatranja kvaliteta međuljudskih odnosa. Ona najpre u velikoj meri utiče na kvalitet specifikacija i jasnoću ciljeva i planova, ali može da ima značajan ticaj i na motivaciju članova tima.

Okolnosti za lociranje propusta

Samo u trivijalnim projektima može da se govori o tome da propusti *mogu* da nastanu. U svakom iole složenijem projektu, koliko god da se pažnje posveti sprečavanju nastajanja propusta, oni *svakako* nastaju, samo je pitanje kada, gde i kako. Zbog toga se u oblasti razvoja softvera veliki značaj pridaje ne samo stvaranju radnog okruženja u kome će do propusta ređe dolaziti, nego i stvaranju preduslova da se nastali propusti lakše uočavaju, lociraju i otklanjaju.

Jedan od najlakših načina da se locira greška u programskom kodu jeste poređenje programskog koda verzije u kojoj greška postoji sa programskim kodom neke prethodne verzije, u kojoj greška nije postojala. Zbog toga je jedna od najvažnijih i nezamenljivih tehnika u razvoju programa čuvanje svih prethodnih verzija programskog koda. To se radi pomoću alata za praćenje verzija, kao što su *GIT*, *SVN* i drugi. U slučaju većih projekata može da bude relativno teško preuzimati i prevoditi različite sačuvane verzije programa, zbog čega je dobro da se čuvaju i različite verzije izgrađenih izvršnih verzija programa, kako bi moglo da se lakše proverava da li se i u kojoj od njih ispoljava neki problem.

Osim programskog koda, drugi značajan izvor informacija predstavlja prepiska između članova razvojnog tima, kao i prepiska sa klijentom. Za ovaj vid komunikacije može da se upotrebljava elektronska pošta, ali je pretraživanje obično daleko jednostavnije i uspešnije ako se upotrebljava neki od alata za praćenje poslova i problema. Savremeni alati za praćenje poslova i problema omogućavaju da se sve relevantne informacije o različitim elementima razvoja zapisuju sistematično i centralizovano. U praksi je upotreba ovakvih sistema daleko pouzdanija i efikasnija nego upotreba elektronske pošte, posebno ako se ima u vidu potencijalno ogroman obim cirkulacije poruka u iole većem razvojnom timu.

Da bi se greške ispoljile (što je neophodno da bi se uočile i zatim locirale), nekada nije dovoljno prevoditi samo izmenjene delove koda. Zbog toga se preporučuje često i plansko redovno *građenje koda*. Terminom *građenje koda* se označava postupak

prevođenja čitavog projekta, bez korišćenja ranije prevedenih međurezultata. U slučaju velikih projekata ovo može biti relativno dugotrajan proces, ali je njegov značaj dovoljan da ga ima smisla automatizovati tako da se odvija planski. Da problem bude još veći, često nije dovoljno testirati prevedeni program na računarima na kojima se razvija, već je potrebno izvršiti potpunu instalaciju softvera, kako bi okolnosti upotrebe u potpunosti odgovarale planiranoj ciljnoj platformi. U obuhvatnijim testiranjima (koja se ne odnose samo na uske delove sistema i pojedinačne operacije) je uobičajeno da se koristi samo softver koji je preveden punim građenjem koda i instaliran na ciljnoj platformi, zato što u drugim slučajevima neke greške mogu lakše da ostanu neprimećene.

Dodatna korist od redovnog građenja koda se ostvaruje ako se čuvaju sve (ili bar izabrane) izgrađene izvršne verzije programa. Na taj način se olakšava proveravanje kako su se ponašale prethodne verzije i da li se problem ispoljavao i u nekoj od njih.

Potencijalan problem u vezi sa prevođenjem programa može da bude relativno površan odnos programera prema *upozorenjima* koja se dobijaju od prevodilaca. Dok programeri moraju da se posvete porukama o *greškama* u prevođenju, zato što se program inače ne bi uspešno preveo, sa druge strane *upozorenja* uglavnom bezbolno mogu da se ignorišu. To je veoma loša praksa. Upozorenja prevodilaca se obično odnose na delove programa koji su potencijalno dvosmisleni i sadrže potencijalne greške, ili ostavljaju prostor za kasnije nastajanje grešaka. Na primer, upozorenja se dobijaju u slučaju implicitnih konverzija tipa, naredbi koje se nikada ne izršavaju, zapisivanja podatka veće dužine u promenljivoj manje dužine i slično. Da se ne bi ostavio prostor za greške, poželjno je pročistiti programski kod tako da se ne dobija nijedno upozorenje. Ako se to radi redovno, u pitanju je sasvim jednostavan posao, koji donosi veliku korist u odnosu na uložen napor.

Veoma čest problem pri lociranju grešaka može da predstavlja nečitak programski kod. Čitljivost programskog koda može da se ostvari na više načina, ali uobičajeno je da se u timovima uvode i poštuju pravila o načinu pisanja i formatiranja i razumnom komentarisanju programskog koda. Pravila o načinu pisanja koda obuhvataju načine imenovanja podataka i potprograma, kao i vizualno oblikovanje programskog koda. Dosledno poštovanje ovih pravila značajno olakšava programerima uočavanje i razumevanje delova programskog koda, što je preduslov za uspešno pronalaženje grešaka i održavanje koda. Savremeni programerski editori omogućavaju da se programski kod automatski vizualno uređuje i to konfigurabilno, u skladu sa pravilima koja su određena u timu.

Komentarisanje programa

Razumno komentarisanje programa podrazumeva uvođenje i poštovanje pravila o komentarisanju celina u programskom kodu (moduli, klase, potprogrami) i pojedinačnih elemenata koda. U savremenom razvoju je uobičajena upotreba alata za automatsko pravljenje dokumentacije na osnovu programskog koda i komentara koji

su napisani u skladu sa pravilima konkretnog alata. Ako dokumentacija već mora da se pravi (a obično mora), onda je to verovatno najbolji način da se ostvari ažurnost programske dokumentacije. Alat za generisanje dokumentacije *Doxygen* je u toj oblasti postao nezvanični standard [*Doxygen*]. Razvijen je 1997. godine i vrlo brzo je stekao zasluženu popularnost. Danas se koristi u velikom broju projekata, a postoji i veliki broj pomoćnih alata koji dodatno nadgrađuju ono što *Doxygen* napravi.

Obično se uvode pravila da bi pojedinačni elementi koda (naredbe, izrazi, blokovi) trebalo da se komentarišu samo onda kada nije sasvim očigledno šta se na nekom mestu radi ili izračunava, ili zašto je nešto urađeno na možda neuobičajen način. Obično se sugeriše da se komentarima ne objašnjavaju elementi programskog koda, nego motivacija za njihovo pisanje, razlozi zašto su neophodni i slično. Programeri bi trebalo da dovoljno dobro poznaju programski jezik i alate da mogu da razumeju šta neki element programskog koda radi (ili opisuje) i bez dodatnih komentara, a ako ipak nije očigledno čemu taj element služi u konkretnom kontekstu, onda bi komentarima trebalo opisati upravo njegovu ulogu i smisao, a ne njegovo tehničko značenje u kontekstu programskog jezika.

Komentari u programskom kodu predstavljaju veoma značajan vid dokumentacije. Postoji mnogo korisnih informacija koje mogu da se navedu u programskom kodu, ali da bi komentari imali smisla oni ne smeju da budu preterani. Trivijalne komentare bi svakako trebalo izbegavati, zato što nepotrebno opterećuju programera tokom čitanja koda.

Često je bolje da se nejasni delovi koda izdvoje u posebne potprograme, zato što onda na osnovu imena potprograma i argumenata i možda jednostavnog komentara, uloga tog dela koda može da bude znatno jasnija nego kada je detaljno prokomentarisan u sklopu neke veće celine.

Neke od uobičajenih vrsta komentara obuhvataju:

- **Zaglavlje datoteke.** Na početku svake datoteke sa izvornim kodom programa je poželjno da se vrlo kratko (sa svega par reči) navede kom pojektu ona pripada i šta je njen sadržaj.
- **Opis namene definicije.** Pre svake definicije (klase, funkcije, metoda, podatka) je poželjno da se navede kratak opis namene elementa koji se definiše. U slučaju klase se opisuje njena funkcija, možda ukratko glavni deo interfejsa ili uobičajen model upotrebe. Možda neka važnija napomena o mestu u hijerarhiji ili ulozi u nekom obrascu ili delu projekta. U slučaju funkcije (metoda) ili podatka (atributa) navode se uloga i celina kojoj pripada. U slučaju funkcije (metoda) navode se i aspekti menjanja stanja, ako nisu očigledni.

- **Opis ulaznih i izlaznih podataka.** U slučaju funkcija i metoda mogu da se opisuju ulazni ili izlazni argumenti, kao i rezultat funkcije. Ne opisuju se ako su očigledni.
- **Opis promene stanja.** Ako funkcija (metod) proizvodi bočni efekat, odnosno menja stanje nekog podatka (objekta), a da to nije očigledno iz naziva funkcije i argumenata, onda je to potrebno da se opiše.
- **Vlasništvo.** Ako funkcija preuzima vlasništvo nad nekim argumentom ili ne predaje vlasništvo nad rezultatom, to je poželjno da se objasni. Na primer, u programskom jeziku *C++* je uobičajeno je da se vlasništvo ne menja kada se radi sa referencama, ali nije uvek jasno šta se dešava kada se prenose pokazivači.
- **Algoritam.** Algoritam se objašnjava samo ako nije očigledan. Može da se objašnjava u celosti, u okviru jednog uvodnog komentara, ili po segmentima u okviru tela funkcije, na mestima gde se implementiraju pojedinačni koraci algoritma. Umesto opisa algoritma, češće je potrebno da se objasni zašto je izabran baš taj algoritam a ne neki drugi. Takvo objašnjenje može da spreči da neko u budućnosti izgubi vreme na razmatranju alternativnog algoritma, ako to u datom kontekstu možda nema smisla.
- **Objašnjenja nelogičnih elemenata koda.** Ponekad u programski kod moraju da se ugrade neki naizgled nepotrebni ili besmisleni elementi. To je, na primer, slučaj kada neka specifična konstrukcija, za koju bi se očekivalo da nema značajno dejstvo, ima posledice po način prevođenja i optimizovanja delova koda od strane neke verzije prevodioca. Takođe, i kada se radi manuelna optimizacija, vrlo često može da izgleda da je napisan kod besmislen, a da on ipak ima važnu ulogu u kontekstu optimizacije. Takva mesta u programu je neophodno pažljivo dokumentovati, da se ne bi dogodilo da ih neko slučajno „popravi“.

Kao što postoje stvari koje je dobro da se objašnjavaju komentarima, tako postoje i loši i neprimereni komentari. Kao što je već ranije navedeno, komentari ne smeju da budu trivijalni, tj. da obuhvataju očigledne informacije. Komentari koji opisuju nešto što je i bez njih sasvim jasno, predstavljaju opterećenje po korisnike izvornog koda. Svaki suvišan red komentara troši vreme čitaoca i istovremeno nepotrebno skreće pažnju sa bitnih detalja. Ako napišemo 20 rečenica komentara, od kojih su samo dve važne, a ostale su očigledne, onda je sasvim moguće da neki čitalac ne uoči te dve važne rečenice u masi beznačajnih. Tipični primeri trivijalnih komentara su oni poput „povećavamo brojač za 1“ ili „argument `brojElemenataNiza` određuje veličinu niza mereno brojem elemenata“. Iako je sasvim očigledno da od takvih komentara niko ne može da ima koristi, komentari tog tipa se iznenađujuće često

koriste, pa se čak u razvojnim timovima često potpuno nepotrebno insistira na navođenju bar po jedne rečenice objašnjenja za svaki argument potprograma ili svaki definisan podatak.

Neki od primera loših komentara su:

- **Trivijalni komentari.** To su obično komentari koji nepotrebno prevode program sa programskog jezika na govorni jezik ili objašnjavaju očigledne stvari.
- **Dobri komentari za loš kod.** Ako se komentarom opisuje jednostavan a nerazumljiv kod, onda to obično znači da bi taj kod trebalo napisati tako da bude razumljiviji. Nekada je dovoljno upotrebiti razumljivija imena promenljivih.
- **Dugački i nejasni komentari.** Ako se neki deo koda opisuje dugačkim komentarom, to obično znači da tu postoji neki problem sa implementacijom.
- **Neprecizne i nejasne reference.** Kada se u komentaru referiše na neki dokument ili veb lokaciju, onda je to potrebno da bude kratko ali nedvosmisleno. Neprecizne i nejasne reference prave više štete nego koristi.
- **Objašnjenja koja nisu primerena kontekstu.** Neprimereno je da se u komentaru izlaže nešto što se ne odnosi na konkretan segment programskog koda. Na primer, nije dobro da se arhitektura sistema opisuje u okviru datoteke izvornog koda koja implementira samo jedan njen manji deo. Umesto toga je bolje da se navede samo kratka rečenica sa referencom na deo dokumentacije gde može da se pronađe odgovarajući opis. Ako neko već gleda izvorni kod, onda verovatno već zna ponešto o arhitekturi. Problem je što se takvi komentari retko ažuriraju pri promeni šireg konteksta na koji se odnose, pa mogu da postanu dezinformacija.

Komentari relativno često mogu da budu naznaka da nešto u programu nije u redu. Martin Fauler to lepo opisuje, kada kaže da se komentari „često koriste kao dezodorans“ i da je preporučljivo da se pre pisanja komentara uvek najpre pokuša sa refaktorisanjem [*Fowler 1999*]. Zaista, suviše se često dešava da bi komentar mogao potpuno da se odstrani ako bi se odgovarajući segment koda preuredio. Kako se to često kaže, programski kod bi trebalo da govori za sebe, a ne da ga komentari zastupaju.

Iako se u oblasti agilnog razvoja može naići i na ekstremne stavove, koji predlažu skoro potpuno eliminisanje komentara, ipak bismo zaključili da je najbolje pronaći dobar balans i pisati komentare, ali samo tamo gde su zaista potrebni i u obimu u kome su korisni.

12.8 Umesto zaključka

Kada započinjemo novi projekat, često podstaknuti svojim i tuđim velikim pa i revolucionarnim planovima, poslednje o čemu želimo da razmišljamo su greške koje ćemo tokom rada na projektu da napravimo. Prirodno je da želimo da se fokusiramo na sve ono lepo i dobro što taj projekat nosi, a ne na ono što bi moglo da nam pokvari dan ili čak čitav projekat. Međutim, kako stvari stoje, greške su neizbežan deo svakog razvojnog procesa i ispada da je jedini način da sprečimo da nam one pokvare previše radnih dana (a možda čak i ceo projekat) da im se od samog početka vrlo temeljno i strpljivo posvetimo – najpre kroz prevenciju, a kasnije kroz sistematično pronalaženje i otklanjanje.

Što više naučimo o greškama i debagovanju, to će nam lakše biti ne samo da pronalazimo i rešavamo probleme, nego i da prepoznajemo okolnosti koje nas vode prema problemima. Postoji mnogo knjiga o debagovanju i nije lako izabrati i preporučiti neke od njih. U ovom poglavlju su već pominjani neki izvori, a pre svih su istaknute knjige Andreasa Zelera [*Zeller 2006*] i Dejvida Agansa [*Agans 2006*]. U prvoj se temeljno i argumentovano izlažu različiti aspekti procesa debagovanja, pa se na kraju čak navodi i jedna formalizacija postupka debagovanja. U drugoj se na veoma praktičan način izlažu neka od najvažnijih pravila debagovanja. Zanimljiv pristup debagovanju primenom *matematičkog metoda* se izlaže u knjizi Čarlsa Mecgera [*Metzger 2004*]. Knjiga „Vodič kroz debagovanje za razvijaoce“ [*Grotker 2008*] je više praktično orijentisana i nudi opise i uputstva za primenu alata za debagovanje i detektovanje problema sa memorijom, kao i elemente upotrebe profajlera, a sve to na primeru programskog jezika *C++*.

13 - Optimizacija softvera

Kompetentan programer je potpuno svestan
strogo ograničene veličine sopstvene glave;
zbog toga pristupa programerskom poslu sa velikom skromnošću
i pored ostalog izbegava pametne trikove kao kugu
Edsher Daikstra

13.1 Performanse softvera

Softverski projekti se odlikuju velikim brojem funkcionalnih i nefunkcionalnih zahteva, koji opisuju ciljne karakteristike proizvoda. U nefunkcionalne zahteve se svrstavaju, između ostalog, i pretpostavke o efikasnosti rada softvera. Efikasnost rada softvera obično se izražava kroz *performanse softvera*. Performanse softvera predstavljaju ocenu odnosa obima obavljenog posla i zauzeća ili utroška različitih resursa. Obično o performansama govorimo u množini, zato što posmatramo različite mere obima obavljenog posla, kao i različite vrste resursa. Razumevanje performansi softvera nam omogućava da procenimo za koje specifične poslove možemo da koristimo softver, do kog obima podataka je on upotrebljiv, kakav računar nam je potreban da bi softver radio dobro u predviđenom domenu i sa predviđenim obimom podataka i slično.

Obim obavljenog posla se obično meri brojem obrađenih slučajeva upotrebe ili obimom obrađenih podataka, ali može da se meri i dimenzijama pojedinačnih problema i na druge načine. Na primer, možemo da merimo količinu obrađenog teksta u pojedinačnim znacima ili u bajtovima, ili broj obrađenih transakcija, ili dužinu obrađenog video ili zvučnog zapisa u sekundama i drugo.

Resursi čije zauzeće posmatramo mogu da budu svi resursi koji su prisutni u savremenim računarskim sistemima:

- opterećenje procesora:
  - utrošeno procesorsko vreme;
  - broj paralelnih tokova izvršavanja;
  - broj izvršenih instrukcija;
  - broj izvršenih naredbi grananja;
  - broj pogrešno predviđenih uslovnih grananja;
- opterećenje grafičke jedinice (*GPU*);
- opterećenje radne memorije:
  - obim podataka koji se čuvaju u radnoj memoriji;
  - broj čitanja ili pisanja iz radne memorije;
  - obim podataka koji se pišu ili čitaju iz radne memorije;
- opterećenje keš memorije...;
- opterećenje virtualne memorije...;
- opterećenje diska...;
- opterećenje magistrale:
  - broj operacija čitanja ili pisanja;
  - obim podataka koji se čitaju ili pišu;
- opterećenje mreže...;
- utrošak električne energije;
- zagrevanje računarskog sistema;
- i drugo.

Odnos obima obavljenog posla i zauzeća nekog resursa često izražavamo i kao složenost programa[83] (ili algoritma), a u odnosu na posmatrani resurs. Pri analiziranju algoritama najčešće se posmatra utrošak procesorskog vremena, ali se vrlo često posmatra i opterećenje radne memorije. Međutim, kada imamo pred sobom neki konkretan softver, koji radi na nekom konkretnom računarskom sistemu, onda nam često nije dovoljno da se ograničimo na posmatranje složenosti pojedinačnih algoritama, već nas zanima kako softver radi *kao celina* i kakvo je ukupno zauzeće resursa.

[83] Pretpostavljamo da su se čitaoci već susretali sa pojmom složenosti algoritma kao uobičajenim vidom približnog opisivanja zavisnosti utroška procesorskog vremena od veličine problema. Na primer, kažemo da algoritam ima linearnu složenost ako je za rešavanje 10 puta većeg problema potrebno približno 10 puta više procesorskog vremena, a kvadratnu složenost ako je potrebno približno 10^2, tj. 100 puta više procesorskog vremena.

Različiti načini merenja obima obavljenog posla i raznovrsnost resursa koji nam mogu biti od interesa, primoravaju nas da pratimo veliki broj različitih parametara koji opisuju performanse softvera. Radi olakšavanja staranja o performansama, obično se fokusiramo na resurs koji trpi najveće opterećenje. Takav resurs nazivamo *usko grlo*. Često možemo da izdvojimo tačno jedno usko grlo, ali se dešava i da ih bude više.

Najčešći slučaj je da je glavno usko grlo procesorsko vreme, ali relativno često usko grlo može da bude i radna memorija. Na primer, povećavanje brzine rada programa često može da se ostvari tako što se algoritam oblikuje da pamti prethodne rezultate ili međurezultate, čime se opterećenje prebacuje sa procesora na radnu memoriju. U slučaju manjih samostalnih programa, čiji je zadatak da nešto izračunaju ili urade, ostali računarski resursi su najčešće raspoloživi u dovoljnoj meri.

Kod složenih sistema, kao što su distribuirani sistemi, ili sistemi koji su zaduženi za složenu i obimnu transakcionu obradu, ili sistemi sa jedinicama za masivnu paralelnu obradu (GPU), povećava se i opterećenje drugih resursa i dešava se da usko grlo predstavlja procesorska ili sistemska magistrala, mreža, diskovi ili nešto drugo.

13.2 Pojam optimizacije softvera

Performanse softvera su veoma važna karakteristika softvera. Iako se uobičajeno ne svrstavaju u funkcionalne karakteristike, one mogu da budu od presudnog značaja za procenjivanje dovršenosti ili prihvatljivosti softvera. Ako softver pravimo da radi u nekom domenu, a u tom domenu postoje slučajevi za koje su ostvarene performanse neprihvatljivo niske, onda takav softver verovatno nije prihvatljiv i klijent će smatrati da nije dovršen. Ako dva softvera rade isti posao, onda njihove performanse imaju veliki uticaj na procenjivanje njihovog kvaliteta.

Niske ili nedovoljne performanse imaju mnogo zajedničkih karakteristika sa bagovima – one mogu da se uoče rano ili kasno, a kada se uoče onda moraju da se popravljaju; tokom razvoja možemo da se staramo da do njih ne dođe, postoje uslovi koji pogoduju njihovom nastajanju ili predstavljaju dobru prevenciju i slično. U skladu sa tim i rad na unapređenju performansi često možemo da posmatramo kao vid debagovanja – ako debagovanje posmatramo kao skup aktivnosti koje preduzimamo da bismo popravili neispravno ponašanja programa, onda ono svakako obuhvata i aktivnosti usmerene na podizanje performansi softvera, zato što je i to vid popravljanja ponašanja.

Međutim, staranje o performansama ima i neke specifične elemente zbog kojih ga razlikujemo od debagovanja i posmatramo kao posebnu aktivnost u okviru razvoja softvera. Ključna razlika u odnosu na debagovanje je da pri popravljanju performansi mi zapravo ne želimo da promenimo ponašanje u smislu ispravnosti rezultata, već samo da spustimo cenu izračunavanja. Podizanje performansi ima sličnosti i sa

refaktorisanjem – u oba slučaja menjamo implementaciju, ali tako da pri tome ne želimo da promenimo rezultat izračunavanja. Razlika je u tome što je cilj refaktorisanja pravljenje *lepšeg* programa, a cilj podizanja performansi je pravljenje *efikasnijeg* programa. Kao što ćemo videti u nastavku ovog poglavlja, insistiranje na visokoj efikasnosti se često ne slaže najbolje sa dobrim dizajnom i lepotom programskog koda.

Staranje o performansama se uobičajeno naziva *optimizacijom softvera*. Obuhvata strategije optimizacije softvera, razmatranje ciljeva, planiranje vremena, mesta i količine optimizovanja, kao i određivanje mesta i uloge optimizovanja u okviru celovitog procesa razoja softvera. Kažemo da je to šire značenje pojma optimizacije softvera i da obuhvata sve aktivnosti koje preduzimamo da bi softver imao odgovarajuće performanse.

Sa druge strane, kada govorimo o konkretnim tehnikama optimizacije, onda se tu obično ne radi o optimizaciji u širem smislu, već o jednom mnogo užem i konkretnijem značenju ovog pojma. Optimizacija softvera, u užem smislu, je postupak menjanja strukture i implementacije programa radi postizanja boljih performansi. Bolje performanse najčešće postižemo rasterećivanjem resursa koji predstavlja usko grlo. Pri tome, rasterećivanje jednog resursa često (ali ne uvek) ima za posledicu povećavanje opterećenja nekog drugog resursa. Zbog toga možemo da kažemo da optimizacija softvera predstavlja preraspoređivanje opterećenja različitih računarskih resursa, u skladu sa potrebama i mogućnostima.

Optimizacija softvera se često pogrešno povezuje sa pojmom optimalnog rešenja. Kažemo da je neko rešenje *optimalno* ako predstavlja *najbolje moguće* rešenje[84]. U nekom idealnom slučaju, možda bismo mogli da kažemo da je cilj optimizacije softvera dostizanje tog optimalnog rešenja, ali to najčešće ne stoji. Umesto toga, optimizacija softvera skoro uvek ima za cilj dostizanje *dovoljno efikasnog rešenja*, tj. rešenja koje je dovoljno blizu nekog idealizovanog optimuma. Osnovni razlozi za to su cena razvoja i ograničenost raspoloživog vremena za završetak projekta. Da bismo znali da je neko rešenje optimalno, to moramo da dokažemo, što zahteva i odgovarajući složen matematički aparat i dovoljno dobru teorijsku podršku za rešavanje odgovarajućeg problema. Algoritmika i arhitektura računara su složene oblasti i veoma je teško dokazati da ne postoji baš nikakav način da se performanse nekog iole složenog posla još malo poprave. Sa druge strane, pronalaženje načina da se neki posao uradi efikasnije obično zahteva mnogo rada (analiziranje, eksperimentisanje, implementiranje, testiranje,...), pa razvojni tim nema uvek dovoljno prostora da mu se posveti. U realnom razvoju softvera često smo u prilici da znamo

[84] U zavisnosti od toga šta se i kako meri, u nekim slučajevima može da postoji više optimalnih rešenja.

da bi nešto moglo da se unapredi, ali da je u konkretnoj situaciji važnije da se projekat privede kraju i da radi ispravno, nego da se performanse dodatno podignu. U takvim slučajevima se pretpostavljene mogućnosti za unapređivanje evidentiraju u obliku budućih ili potencijalnih zadataka i ostavljaju za neko drugo vreme.

13.3 Nivoi optimizacije softvera

Optimizacija softvera može da se izvodi na više različitih nivoa, posmatrano u odnosu na nivo apstrakcije dela softvera koji se optimizuje. *Optimizacija visokog nivoa* obuhvata sve oblike optimizacije koji su *iznad* nivoa programskog koda, a *optimizacija niskog nivoa* se odnosi na one vidove optimizacije koji se tiču konkretnog programskog koda ili postupka izgradnje softvera od napisanog koda.

13.3.1 Optimizacija visokog nivoa

Optimizacija visokog nivoa se obično izvodi na nivou projekta ili na nivou algoritma. Optimizacija na nivou projekta se prvenstveno tiče arhitekture softvera. Cilj je da se arhitektura oblikuje uz puno razumevanje poznatih ili potencijalnih uskih grla. Optimizacija na nivou projekta proširuje skup alata za oblikovanja arhitekture dodatnom tehnikom – *dekomponovanjem prema resursima*. Ideja je da se najpre prepoznaju funkcionalnosti i njihov odnos prema uskim grlima i da se onda funkcionalnosti koje dele isto usko grlo razdvoje u različite komponente. U zavisnosti od procene, takve komponente mogu odmah da se projektuju tako da rade na različitim uređajima ili može da se ostavi prostor za njihovo eventualno kasnije distribuiranje. Na taj način se primenom modularizacije ostvaruje dodatna fleksibilnost u odnosu na uočen potencijalan problem preopterećenja konkretnog uskog grla.

Obično nije lako da se unapred proceni efikasnost neke arhitekture. U trenutku njenog oblikovanja još uvek ne postoji odgovarajući programski kod, ili postoje tek neki pojednostavljeni obrisi tog koda, pa nismo u prilici da izmerimo performanse. Ipak, pre nego što donesemo odluku da je neka arhitektura odgovarajuća, trebalo bi da nekako procenimo i da li je dovoljno efikasna. Tu može da nam bude od pomoći da napravimo različite eksperimente ili prototipove. Kao i sa drugim vrstama prototipova, njihovom pravljenju mora da se pristupa pažljivo i bez preterivanja.

Optimizacija na nivou algoritma se odnosi na unapređivanje ili čak zamenu konkretnog algoritma radi smanjivanja opterećenja uskog grla. Često za rešavanje nekog problema imamo na raspolaganju više algoritama, koji imaju različite osobine. Najčešće će nam najviše odgovarati onaj algoritam koji ima najmanju složenost u odnosu na procesorsko vreme, ali to nije uvek tako. Na primer, algoritmi koji imaju manju složenost, nekada mogu da imaju mnogo skuplji pojedinačan korak, tako da njihova efikasnost dolazi do izražaja tek na veoma velikim skupovima podataka. Slično tome, neki algoritmi imaju različitu složenost u odnosu na broj problema i

veličinu pojedinačnih problema, pa u zavisnosti od toga da li je potrebno da naš softver radi sa velikim brojem jednostavnih problema ili sa malim brojem složenih problema, neće uvek isti algoritam da predstavlja najbolji izbor.

Optimizacija visokog nivoa može da bude veoma skupa. Bilo da se radi na nivou arhitekture ili na nivou algoritma, optimizacija visokog nivoa može da dovede u pitanje upotrebljivost postojeće implementacije. Često moramo da ponovo implementiramo komponente ili algoritme koji smo već jednom implementirali.

Optimizacije visokog nivoa su skupe čak i kada se rade unapred, pre nego što smo bilo šta implementirali. Problem je u tome što se njima obično podiže nivo apstrakcije posmatranog dela softvera, uvode se složeniji odnosi među komponentama ili složeniji algoritmi, pa se potencijalno otežava implementacija. A pri tome je sasvim moguće da to sve nije ni potrebno, zato što bi možda i jednostavnije rešenje radilo sasvim dobro. Zato se pri razmatranju optimizacije visokog nivoa, koja se preduzima unapred, obično teži da se najpre pažljivo sagledaju svi rizici i njihova cena, što uključuje i što približnije procenjivanje efikasnosti mogućih rešenja, pa da se tek na osnovu tako prikupljenih informacija odabire rešenje koje je povoljnije.

13.3.2 Optimizacija niskog nivoa

Optimizacije niskog nivoa se odnose na već postojeći programski kod i tehnike izgradnje programa. Pojedinačne tehnike optimizovanja već postojećeg programskog koda se najčešće odvijaju na niskom nivou. Zbog toga se pod užim značenjem pojma optimizacije softvera uobičajeno podrazumevanju samo optimizacije niskog nivoa ili čak samo pojedinačne tehnike optimizovanja.

Optimizacije niskog nivoa mogu da se preduzimaju na nivou:

- izvornog koda;
- prevođenja;
- izgradnje koda;
- mašinskog koda ili
- izvršavanja.

Optimizacija na nivou izvornog koda predstavlja menjanje implementiranog programskog koda radi ostvarivanja veće efikasnosti. Obično počiva na promenama strukture koda, kojima se ostvaruje efikasniji rad, ali po cenu dobijanja manje opšteg ili slabije strukturiranog rešenja. Drugi vid optimizacije na nivou izvornog koda je prilagođavanje implementacije algortima specifičnostima konkretnog programskog jezika – nekada ima prostora da se pređe sa upotrebe nekih opštih programskih koncepata na upotrebu nekih koncepata specifičnih za konkretan jezik i da se tako dobije na performansama.

Optimizacija na nivou prevođenja se odnosi na upravljanje postupkom prevođenja, tako da se neke specifičnosti prevodilaca iskoriste za dobijanje efikasnijeg izvršnog koda. Obuhvata odabiranje verzije prevodioca i podešavanje opcija prevođenja. Praktično svi prevodioci imaju neke opcije kojima se uključuju različiti oblici internog optimizovanja prevedenog programa, ali neke od tih opcija ne donose uvek pozitivne efekte. Zato je često potrebno da isprobamo različite kombinacije finih podešavanja prevodilaca da bismo ostvarili najviše performanse. Nezgodna strana ovog nivoa implementacije je što mora posebno da se radi za različite platforme i prevodioce.

Optimizacija na nivou izgradnje programa se ostvaruje izborom odgovarajućih varijanti biblioteka i načina povezivanja. Na primer, neki delovi programa mogu biti toliko osetljivi u odnosu na performanse, da izbor statičkog ili dinamičkog povezivanja modula[85] može da proizvede značajne razlike. Ponekad izbor verzije neke biblioteke može da ima značajne posledice po efikasnost. Nije retkost da neka jednostavnija biblioteka pruža veće performanse, ali po cenu manjih mogućnosti. Ako su nam te umanjene mogućnosti dovoljne, onda takva biblioteka može da bude bolji izbor.

Optimizacija na nivou mašinskog koda se izvodi pisanjem delova programa na mašinskom jeziku, tj. na asembleru. To je jedini nivo pisanja programa na kojem do kraja mogu da se iskoriste sve mogućnosti računarskog sistema. Osnovni problem je u tome što se pisanjem mašinskog koda naša implementacija striktno vezuje za jednu klasu ili generaciju procesora, a često čak i za konkretnu generaciju procesora ili konkretan operativni sistem. To nas primorava da pišemo i održavamo više verzija optimizovanog koda za različite procesore i operativne sisteme.

Prostor za primenu ovog nivoa optimizacije se sve više sužava, kako zbog problema koje on donosi, tako i zbog činjenice da savremeni prevodioci imaju ugrađene vrlo napredne algoritme za optimizaciju mašinskog koda. Današnji prevodioci često mogu da prevedu naš program na mašinski jezik na skoro idealan način, pa čak i da promene redosled mašinskih naredbi tako da one rade efikasnije a sa istim rezultatom. Programerima je sve teže da manuelno napišu mašinski kod koji je efikasniji od onog koji bi napravio prevodilac. Zato se danas relativno retko celi programi, funkcije ili metodi pišu na mašinskom jeziku, već je uobičajeno da se na taj način manuelno optimizuju samo pojedini osetljivi delovi koda, u slučajevima kada

[85] Da ne bude nesporazuma, ovde je reč o povezivanju prevedenih modula, a ne o vezivanju metoda. Statičko povezivanje je povezivanje objektnog koda sa statičkim bibliotekama u fazi povezivanja programa, a dinamičko povezivanje je povezivanje izvršnog programa sa dinamičkim bibliotekama u fazi učitavanja programa u radnu memoriju, neposredno pre izvršavanja, ili čak u toku izvršavanja.

želimo da iskoristimo neku specifičnost konkretnog procesora ili operativnog sistema.

Optimizacija na nivou izvršavanja obuvata prilagođavanje programa specifičnostima arhitekture procesora i računara i načinu izvršavanja programa ili određenih operacija. Uobičajeno je da se na ovom nivou optimizacije vrši preraspoređivanje mašinskih instrukcija radi njihovog izvršavanja preko reda, upotreba operacija odloženog grananja ili paralelnog izvršavanja nekih instrukcija i slično. Takve optimizacije su na nižem nivou nego optimizacije na nivou mašinskog koda i u velikoj meri se preklapaju sa njima.

U optimizacije na nivou izvršavanja se ubrajaju i tehnike poput automatskog prevođenja ili optimizovanja neposredno pred izvršavanje, kao što je na primer prevođenje *na vreme* (ili *u pravo vreme*, engl. *JIT – just in time*) u slučaju *JVM* ili *CLR*.

13.4 Strategije

Najvažnija pitanja u vezi sa optimizacijom su *kada*, *šta* i *koliko* je potrebno da optimizujemo. Odgovori na ova pitanja određuju, redom, trenutak, predmet i dubinu optimizacije.

Odgovor na pitanje *kada* i odabir tačnog trenutka izvođenja optimizacije nazivaju se i *vremenskom lokalizacijom* optimizacije. Prema tome kada se određena optimizacija preduzima razlikujemo *optimizaciju unapred* i *optimizaciju unazad*.

13.4.1 Optimizacija unapred

Optimizacija unapred podrazumeva da se staranje o performansama odvija sve vreme tokom razvoja – od samog početka planiranja, kroz projektovanje arhitekture, oblikovanje ili odabiranje algoritma i kroz implementiranje i testiranje softvera koji se razvija. Cilj je da svi učesnici u razvoju tokom svih razvojnih aktivnosti rade na tome da naprave što efikasnija rešenja za probleme koje rešavaju. Optimizacija unapred se naziva i *kontinualnim staranjem o performansama*.

Optimizacija unapred omogućava da se dođe do veoma kvalitetnih rešenja, u pogledu performansi. Posvećenost razvojnog tima omogućava da se blagovremeno uoče i primene efikasna rešenja, što dodatno omogućava i da u svakom trenutku razvoja može da se sagleda da li su performanse u skladu sa planovima.

Međutim, optimizacija unapred nosi i nekoliko veoma važnih negativnih posledica. Najpre, zahteva mnogo više radnog vremena, što podiže inicijalnu cenu razvoja. Naravno, bar deo te cene će se isplatiti zato što neće biti potrebno da se preduzima naknadna optimizacija, ali ne možemo da se ne zapitamo da li je ta cena opravdana – pokazuje se da je u realnom razvoju veoma lako utrošiti na optimizaciju mnogo više vremena nego što je potrebno, a radi ostvarivanja zanemarljivo većeg ili nepotrebno visokog nivoa performansi.

Drugi problem je što je optimizovan softver najčešće značajno teži za održavanje. Ili su uvedene složenije apstrakcije na nivou arhitekture, ili se koriste složeniji algoritmi, ili je programski kod pisan uz fokusiranje programera na efikasnost, pa zato nije dovoljno lep i teži je za razumevanje. Razvijaoci će trošiti više vremena na razumevanje, menjanje i testiranje takvog programskog koda, pa se tako dodatno podiže cena razvoja.

Pri optimizovanju unapred se oslanjamo na procene o tome koji deo softvera je potrebno da se optimizuje i do kog nivoa efikasnosti. Pravljenje takvih procena obično nije nimalo jednostavno, a eventualno pogrešna procena može da ima za rezultat nepotrebno visoku cenu razvoja.

Ako se optimizuje šire i dalje nego što je potrebno, onda može da se proizvede softver koji je efikasniji nego što je potrebno, što samo po sebi nije loše, ali se zbog toga u optimizaciju ulaže mogo više rada nego što je neophodno, a verovatno je umesto toga taj rad mogao da se uloži u razvoj neke dodatne funkcionalnosti, koja bi za korisnike imala veću vrednost nego što je vrednost ostvarene dodatne efikasnosti. Dodatni problem je što nepotrebne optimizacije prouzrokuju i očigledno nepotrebne dodatne troškove pri kasnijem menjanju i održavanju softvera.

Sa druge strane, ako se optimizuje uže ili nedovoljno daleko, onda se za rezultat često dobija nedovoljno efikasan softver, pa je na kraju neophodno da se primeni i optimizacija unazad. U takvim slučajevima se sa pravom postavlja pitanje da li je od čitave optimizacije unapred uopšte bilo nekakve koristi ili taj uloženi rad predstavlja beskorisno utrošen resurs?

Optimizacija unapred može da ima smisla kada se razvijaju kritični delovi programa, čije performanse presudno utiču na njegovu upotrebljivost. Može da se primenjuje i na određene elemente optimizacije visokog nivoa. Sa druge strane, ona nije uobičajena za optimizovanje celih složenih softverskih projekata.

13.4.2 Optimizacija unazad

Strategija *optimizacija unazad* odlaže najveći deo staranja o performansama za sam kraj razvoja. Umesto da se sve vreme tokom razvoja mnogo vremena i energije posvećuje iznalaženju što boljih rešenja, umesto toga se tokom većeg dela razvoja akcenat stavlja na dobro strukturiranje i ispravnost programa. Tek na kraju, kada su razvijene sve funkcije i kada je provereno da sve radi ispravno, onda se proverava da li softver radi dovoljno efikasno ili je možda potrebno da se dodatno podigne efikasnost nekog njegovog dela. Ovakav pristup razvoju se ponekad naziva i metodom *funkcionalnost pre performansi*.

Ova strategija počiva na dve osnovne pretpostavke:

- programski kod se tokom razvoja relativno često menja i
- najveći deo performansi zavisi od vrlo ograničenog dela softvera.

Ako znamo da se kod često menja i da je menjanje optimizovanog koda značajno otežano, onda ćemo da težimo da optimizujemo što je moguće manje delove programa i da to radimo što je moguće kasnije. Ako pri tome znamo da je za dobre performanse najčešće dovoljno da se optimizuje relativno mali deo programa, pa to još povežemo sa prvom pretpostavkom, onda ćemo tim pre biti motivisani da primenjujemo ovu strategiju.

Jedan od glavnih kvaliteta ovakvog pristupa je izbegavanje suvišnog, nepotrebnog ili preuranjenog optimizovanja. Takvim pristupom se ostvaruju velike uštede tokom razvoja, kako zbog toga što se ne posvećuje vreme optimizaciji, tako i zbog toga što razvijen kod ima bolje strukturiran i često jednostavniji dizajn, pa se lakše i piše i debaguje i kasnije proširuje ili modifikuje.

Performanse softvera mogu da se objektivno izmere tek kada je razvoj softvera pri kraju, kada su prisutne sve funkcionalnosti i kada je pri kraju testiranje ispravnosti programa. Sve do tada se radi samo o procenjivanju, a tek tada je u pitanju egzaktno merenje performansi. Samim tim, tek tada možemo da donesemo ispravan sud o tome koji je deo softvera i do koje granice neophodno da se dodatno optimizuje. Pre objektivnog merenja i donošenja odgovarajućeg egzaktnog suda, postoji značajan rizik da se optimizacija preduzme na pogrešnom delu koda ili da ima neprimerenu širinu ili dubinu. Zato je optimizacija unazad najčešće poželjnija od optimizacije unapred.

Najveći problem sa ovakvim pristupom je što je neke stvari veoma teško popraviti na samom kraju, kada je već napisan najveći deo programa. Ako se ispostavi da je potrebno da se zameni neki algoritam ili čak da se promeni arhitektura softvera, onda to može da zahteva preduzimanje veoma obimnih izmena, pa čak i ponovno implementiranje nekih delova softvera. Cena takvih zahvata može da bude veoma visoka.

13.4.3 Odmerena optimizacija

Odmerenost (ili lokalizovanost) optimizacije predstavlja zadržavanje procesa optimizacije softvera u okvirima koji su određeni odgovorima na pitanja *šta* i *koliko* optimizujemo. Strategija odmerene optimizacije nalaže precizno određivanje predmeta i dubine optimizacije pre njenog preduzimanja.

Određivanje predmeta optimizacije

Kao što smo već istakli u prehodnim odeljcima, najveći deo performansi softvera zavisi od njegovog relativno malog dela. Od presudnog značaja za uspeh optimizovanja softvera je da se vrlo precizno odredi koji je deo softvera potrebno da se optimizuje.

Programeri troše enormne količine vremena
na razmišljanje i brigu o brzini nekritičnih delova programa,
a takvi pokušaji postizanja efikasnosti zapravo imaju negativan uticaj
kada se uzmu u obzir debagovanje i održavanje.

Potrebno je da zanemarimo sitne dobitke u efikasnosti
u recimo 97% slučajeva:
preuranjena optimizacija je osnov svakog zla.

Sa druge strane, ne smemo da propustimo priliku
za optimizaciju preostalih kritičnih 3%.

Donald Knut

Veoma je važno da se razume da najveći deo programskog koda ne utiče kritično na performanse. Sa druge strane, nije nam posebno bitno koliko je precizna procena koju je izneo Donald Knut – u nekim slučajevima će se optimizovati i više od 10% programskog koda, ali u nekim drugim i manje od 0,1%. Možda neki deo programa može da se napiše i tako da radi 10 puta brže, ali ako to znači da će odziv korisničkog interfejsa biti 1ms umesto 10ms, a pri tome nema značajnog uticaja na ukupnu efikasnost softvera, onda eventualna optimizacija tog dela koda ne donosi praktično nikakvu korist.

Određivanje predmeta optimizacije počinje na vrlo grubom nivou, od uočavanja poslova koji nisu dovoljno efikasno implementirani. Precizno lokalizovanje optimizacije ima sličnosti sa postupkom lokalizovanja bagova – pažljivim posmatranjem i analiziranjem programa se trudimo da što preciznije odredimo deo programskog koda koji je potrebno da se optimizuje. To je nekada relativno lako, ali često ne može da se uradi dovoljno dobro bez primene odgovarajućih alata.

Kada tačno prepoznamo deo softvera koji je potrebno da optimizujemo, onda u sledećem koraku moramo da odredimo tačnu meru potrebne optimizacije. Stepen optimizacije nekog izabranog dela softvera nazivamo i *dubinom optimizacije*. Kažemo da je optimizacija dublja ili dalja, ako se njom više približavamo pretpostavljenom idealizovanom optimalnom rešenju.

Određivanje dubine optimizacije

Potrebna dubina optimizacije se određuje kroz definisanje kriterijuma performansi. Kriterijumi performansi mogu da budu izraženi na različite načine, a obično imaju oblik *minimalnih* performansi i *planiranih* ili *ciljanih* performansi. Minimalne performanse predstavljaju zahtev koji mora da se ispuni. Ako se ne ispuni, onda se smatra da softver nije dovršen. Minimalne performanse moraju da se dostignu, čak i po cenu povećavanja trajanja i troškova razvoja. Za razliku od minimalnih, planirane performanse predstavljaju cilj kome se teži, ali koji uglavnom ne zaslužuje produžavanje rokova ili produžavanje troškova razvoja. Pri samom

kraju razvoja softvera se često vrši naknadna revizija ovih kriterijuma i njihovo usklađivanje sa rokovima i troškovima.

Proces određivanja kriterijuma performansi se razlikuje u zavisnosti od vrste softvera koji se razvija i planiranih načina njegove upotrebe. U tom smislu se najveća razlika pravi između načina ocenjivanja i određivanja kriterijuma performansi interaktivnog i neinteraktivnog softvera.

Kriterijumi performansi interaktivnog softvera

Kada se radi o softveru kojim korisnik interaktivno rukuje tokom obavljanja nekog posla, tada se pod dovoljnom efikasnošću podrazumeva da je softver dovoljno efikasan da bude upotrebljiv, tj. da ne izaziva neprijatnosti ili stres kod korisnika. Upotrebljivost korisničkog interfejsa zavisi od načina upotrebe softvera, odnosno od načina komunikacije korisnika i softvera. U kontekstu razmatranja efikasnosti softvera upotrebljivost se obično ocenjuje na osnovu tri kriterijuma [*Card 1991*][86]:

- perceptivna obrada, do 0,1s;
- neposredan odgovor, do 1s;
- jednostavan zadatak, do 10s.

Ova tri slučaja mogu da se povežu sa interaktivnom komunikacijom među ljudima i sa normativima na koje smo naviknuti u toj komunikaciji.

Perceptivna obrada podrazumeva slučajeve kada je uobičajeno da se na akciju odgovara bez čekanja na razmišljanje, tj. instiktivno ili na osnovu osećaja. To bismo mogli da poredimo sa saradnjom sportista u nekom brzom timskom sportu. U kontekstu softvera, to su elementarne aktivnosti kao pritisak tastera na tastaturi ili mišu, pomeranje olovke po tabli ili izvođenje pokreta na ekranu osetljivom na dodir. U takvim slučajevima, koji obično predstavljaju samo delove neke složenije interakcije, očekujemo da nam računar odgovara dovoljno brzo da nam čekanje na odgovor ne remeti tok aktivnosti. Obično se smatra da je gornja granica dopuštenog čekanja u takvim slučajevima 0,1s, ali ćemo često očekivati da reakcija bude i malo brža. Na primer, da bi putanja olovke ili miša bila verno opisana, često je potrebno da se detektuje više nego 10 promena u jednoj sekundi. Takođe, kod akcionih video

[86] Ove koncepte je prethodno mnogo detaljnije razradio Robert Miler [*Miller 1968*]. On je opisao 19 različitih vrsta aktivnosti i prodiskutovao dopustive opsege čekanja na reakciju računara. Od tada su se značajno promenili načini interakcije korisnika i računara, ali te procene su ipak uglavnom relevantne i danas. Upotrebljivosti softvera je kasnije temeljnije razmatrana u [*Nielsen 1993*], gde su kao najvažniji kriterijumi upotrebljivosti interaktivnog softvera istaknuti isti ovi granični kriterijumi.

igara vreme reakcije softvera mora da bude nešto kraće, zato što vrhunski igrači mogu da izdaju i do 600 komandi za minut.

Neposredan odgovor podrazumeva odgovaranje sagovornika na jednostavno pitanje. Na primer, ako pitamo sagovornika „Da li je veći mali slon ili veliki miš?“, on mora da razume pitanje, sasvim kratko razmisli o njemu i odgovori. U slučaju interaktivnog softvera, pod neposrednim odgovorom podrazumevamo brzo reagovanje na zahtev korisnika, koji očekuje odgovor da bi mogao da nastavi neku složeniju aktivnost. U kontekstu interaktivnog rada, smatra se da se na zahtev korisnika odgovara neposredno ako se u periodu od postavljanja pitanja do isporučivanja odgovora korisniku ne prikazuje nikakva dodatna povratna informacija (engl. *feedback*). Kao gornja granica dopuštenog čekanja na nesporedan odgovor obično se smatra 1s.

Treća vrsta interaktivnih poslova su jednostavni zadaci. Jednostavnim zadacima se obično smatraju svi oni zadaci kod kojih se na rezultat rada čeka duže nego na neposredne odgovore ali dovoljno kratko da komunikacija i dalje može da se nazove interaktivnom. I dalje pretpostavljamo da će korisnik na osnovu dobijenog odgovora želeti da postavi neki drugi zahtev (to je pretpostavka interaktivnosti), ali i da je on svestan složenosti zadatka i da može malo da sačeka. Ako je čekanje na odgovor duže od 1s (tj. iznad praga neposrednog odgovora), onda je neophodno da se korisniku pruže neka informacija o tome da je obrada u toku, kao i približna procena koliko će da potraje. U suprotnom, korisnik može da doživi neprijatan osećaj neizvesnosti – da li će odgovor uopšte da stigne, ili je nešto krenulo naopako?

Sve poslove na koje se čeka duže od 10s trebalo bi raditi u neinteraktivnom režimu. Na primer, njihovo izračunavanje može da se nastavi u pozadini, a da se korisniku omogući da zadaje nove zadatke.

Imajući u vidu navedena vremena reakcije koja određuju upotrebljivost interaktivnog softvera, možemo da okvirno prepoznamo i ciljeve optimizovanja pojedinih delova softvera. Za svaki interaktivan posao prvo moramo da prepoznamo da li mora da pruži utisak perceptivne obrade, ili je dovoljno da radi poput neposrednog odgovora ili može da spada u jednostavne zadatke. Zatim na osnovu prepoznatog nivoa interaktivnosti ustanovljavamo ciljno vreme odziva. Na taj način određujemo i koliko daleko taj posao moramo da optimizujemo.

Kriterijumi performansi neinteraktivnog softvera

U slučaju softvera koji ne radi interaktivno, cilj optimizacije se ne ustanovljava na osnovu nekih globalnih objektivnih merila, već na osnovu nefunkcionalnih zahteva ili nekih sistemskih ograničenja.

Na primer, u nekom sistemu za podršku rada lanca prodavnica mogao bi da se postavi poslovni zahtev da rukovodstvo svakog dana dobija izveštaj o dnevnom prometu i to u roku od 15 minuta posle zatvaranja svih prodavnica. U takvom

slučaju cilj optimizacije je sasvim precizno određen konkretnim potrebama korisnika i softver mora da se napravi tako da se te potrebe zadovolje.

Veoma često cilj optimizacije zavisi od nekih složenijih faktora. U slučaju složenih sistema, opterećenost neke od komponenti, koja radi veliki broj poslova ili opslužuje veliki broj korisnika, može u nekim periodima radne nedelje ili radnog vremena da predstavlja usko grlo čitavog sistema. Tada je neophodno da se radi na tome da se ta komponenta rastereti ili da se njeni poslovi optimizuju do mere koja garantuje ispravno funkcionisanje sistema. Na primer, neka je u onlajn prodavnici oglašen popust na neke proizvode, koji prestaje da važi u nekom trenutku. Može da se očekuje da će pred istek važenja popusta doći do povećanog opterećenja sistema za naručivanje i plaćanje. Da bismo mogli da rešimo takav problem potrebno je da napravimo procenu opterećenja i da probamo da softver optimizujemo tako da može da ga podnese.

Postoje i slučajevi kada nema nikakvih spolja nametnutih ciljeva. Na primer, ako pravimo softver koji mora da izvede složenu obradu podataka radi nekog naučnog istraživanja, onda nam limiti često nisu strogo definisani. Nekada ćemo biti u prilici da limite odredimo na osnovu toga da li je očekivano vreme prihvatljivo ili ne (na primer, ako procenimo da će obrada da traje više od 20 dana, to nekada može da bude prihvatljivo, ali nekada nije), ali nekada će trajanje biti prihvatljivo, a da ipak imamo utisak da bi bilo dobro da bude kraće (na primer, procenjeno trajanje obrade je 20 dana i to nam je u načelu prihvatljivo, ali bi bilo dobro da ga skratimo na 10 dana, da bismo mogli da stignemo da u istom roku obradimo još jedan skup podataka). Tada je potrebno da sami odlučimo da li nam je potrebna optimizacija, pa ako jeste onda i da odredimo kriterijume i ciljeve.

13.4.4 Savremena praksa optimizovanja softvera

Savremenu praksu optimizovanja softvera možemo da predstavimo pomoću nekoliko osnovnih principa, koji se međusobno nadopunjuju:

- Procenjivanje performansi;
- Lenja optimizacija;
- Lokalizovanje optimizacije i
- Eksperimentisanje.

Ovde izloženi principi su donekle zasnovani na radu [*Auer 1996*], u kome su Ken Aur i Kent Bek opisali agilni pristup optimizaciji softvera kroz predstavljanje skupa od 16 *obrazaca*. Njihovi obrasci optimizovanja su izloženi na primeru programskog jezika *Smalltalk*, ali uglavnom mogu da se odnose i na druge programske jezike. Neki od tih obrazaca predstavljaju specifične tehnike optimizacije, ali neki drugi, koji su

nama interesantniji, su vrlo uopšteni i ostvarili su veliki uticaj na oblikovanje agilne strategije optimizovanja softvera.

Procenjivanje performansi

Procenjivanje performansi dela softvera se preduzima pre početka njegovog projektovanja i implementiranja. Osnovna namena procenjivanja performansi je da se blagovremeno uspostave merila, koja bi trebalo da nam pomognu da se odaberu ili oblikuju dovoljno dobra arhitektura i adekvatni algoritmi, koji omogućavaju dostizanje uspostavljenih kriterijuma performansi. Dobijene procene ne bi trebalo da se koriste kao osnov za forsiranje kontinualne optimizacije.

Procenjivanje performansi smanjuje rizik nastajanja ozbiljnijih problema pri optimizaciji unazad. Ima ulogu prevencije tzv. nedostižnih performansi, tj. slučajeva kada pri optimizaciji unazad nisu dovoljne samo optimizacije niskog nivoa, već je neophodno da se preduzimaju i skupe naknadne optimizacije visokog nivoa.

Uobičajeno je da rezultat procene ima oblik izveštaja o sagledanim minimalnim i planiranim performansama i uočenim potencijalnim problemima u realizaciji. U okviru pregleda potencijalnih problema bi trebalo da budu prepoznata i opisana moguća uska grla ili kritični resursi, i to kako u delu softvera koji će se razvijati, tako i u njegovom okruženju.

Procenjivanje performansi bi trebalo da uzme relativno malo vremena, ali praksa pokazuje da potrebno vreme i obim izveštaja zavise od veličine i složenosti dela softvera čijem se razvoju pristupa, kao i od zahteva koji su postavljeni u odnosu na performanse. Ako se radi na nekom relativno jednostavnom slučaju upotrebe, koji opisuje, na primer, neku izdvojenu transakciju u informacionom sistemu, onda su obim i složenost problema relativno niski, a i sama priroda analize potencijalinh problema verovatno veoma liči na neke druge već razvijene transakcije. Zato procenjivanje performansi u takvom slučaju često zateva sasvim kratko vreme.

Međutim, u mnogim slučajevima stvari stoje sasvim drugačije. Ako nam je zadatak da postavimo osnovu za buduću arhitekturu dela sistema, ili da izaberemo algoritam koji je najprimereniji problemu, u smislu da je rešenje dovoljno efikasno i istovremeno što jednostavnije za implementaciju, onda ni analiza ni donošenje odluke nisu nimalo jednostavni. U takvim slučajevima moramo pažljivo da izvagamo moguća rešenja i da donesemo odluke, koje mogu da imaju dalekosežne posledice.

Procenjivanje performansi se izvodi na početku razvoja, unapred, pa zbog toga obično nije na raspolaganju dovoljno informacija da bi mogla da se ostvari visoka preciznost procene. Radi dobijanja dodatnih informacija često se sprovode različiti eksperimenti (princip *Eksperimentisanje*). Problem ograničene preciznosti može donekle da se prevaziđe naknadnim revizijama. Revizije procene efikasnosti se preduzimaju u slučajevima kada dostignut stadijum razvoja ili neke izmenjene

okolnosti pružaju osnov za tačnije ili realnije sagledavanje i procenjivanje performansi. Na primer, ako se bližimo prekoračenju vremenskih i finansijskih okvira, onda bi trebalo da se napravi revizija procene performansi. Ona bi mogla da prilagodi kriterijume minimalnih ili planiranih performansi na osnovu realno dostižnih okvira razvoja.

Lenja optimizacija

Videli smo da i optimizacija unapred i optimizacija unazad imaju svoje prednosti i slabosti. Zbog toga savremena praksa obično počiva na kombinovanju ovih strategija, uz uzimanje u obzir specifičnosti savremenog razvoja softvera i posebno agilnih metodologija. Pri tome je izbor načina ili trenutka započinjanja procesa optimizacije korelisan sa odnosom projektovanja arhitekture i dizajna softvera. Kao što skupe odluke o arhitekturi ima smisla doneti što ranije (i naravno što opreznije), tako i neke elemente optimizacije visokog nivoa ima smisla preduzimati unapred. Sa druge strane, optimizacija nižeg nivoa je najbolje da se ostavi za završni period razvoja softvera.

Lenja optimizacija podrazumeva da se svi vidovi optimizacija niskog nivoa preduzimaju tek pri kraju razvoja softvera, nakon dostizanja njegove planirane funkcionalnosti. Nasuprot tome, samo neki elementi optimizacije visokog nivoa će se preduzimati unapred, i to samo u minimalnom obimu, koji pomaže da se arhitektura i algoritmi oblikuju tako da potonjim razvojem i optimizacijom unazad mogu da se dostignu kriterijumi koji su ustanovljeni procenjivanjem performansi.

Lenja optimizacija je u potpunosti u duhu principa agilnog razvoja softvera „neće biti potrebno“ – ona odlaže optimizaciju sve do trenutka kada je ona zaista i potrebna. Na taj način se štedi vreme, zato što se ne optimizuju delovi programa koje nije neophodno da optimizujemo, a uz to nam pomaže da se što duže očuva dobar dizajn softvera, te da se on „kvari“ optimizacijom tek na samom kraju razvoja.

Uobičajeno je da se na samom kraju razvojnog ciklusa predvidi jedan period u kome će se razvojni tim moći da se posveti optimizaciji. Poslovi optimizacije se često odvijaju paralelno sa poslovima na doterivanju korisničkog interfejsa. U zavisnosti od vrste problema, korišćenih programskih jezika i drugih alata, stepena stručnosti tima i drugih faktora, taj period može da obuhvata od 5% do 35% predviđenog vremenskog okvira.

Već smo istakli da pri optimizaciji unazad postoji opasnost da dođemo u situaciju da moramo da naknadno preduzimamo ozbiljne optimizacije visokog nivoa (na primer paralelizaciju ili distribuiranje izračunavanja), što je veoma skupo. Da bismo to izbegli, ili bar da bismo smanjili rizik a time i broj i cenu takvih slučajeva, oslanjamo se na princip *Procenjivanje performansi*.

Lokalizovanje optimizacije

Princip *Lokalizovanje optimizacije* se odnosi na primenu strategije odmerene optimizacije. U osnovi ovog principa je što preciznije odgovaranje na pitanja *šta* i *koliko* se optimizuje. Princip lokalizovanja optimizacije se primenjuje u skladu sa merilima utvrđenim pri procenjivanju performansi. Suština ovog principa može da se rezimira kroz tri jednostavne preporuke:

- Budi odmeren pri optimizovanju.
- Prvo postavi ciljeve, pa tek onda optimizuj.
- Ne optimizuj dalje od postavljenih ciljeva.

Ken Aur i Kent Bek su izdvojili četiri obrasca [*Auer 1996*], koji odgovaraju ovom principu:

- Kriterijumi performansi – Pre početka razvoja je neophodno da se uz pomoć klijenta odrede kriterijumi performansi. Oni mogu da se kasnije revidiraju.
- Prag prihvatljivosti – Prag prihvatljivosti se određuje na osnovu kriterijuma performansi. Ne sme da se ide dalje od praga, čak ni kada izgleda da je dodatno povećanje performansi lako dostižno.
- Merenje performansi – Neophodno je da precizno merimo performanse delova programa za koje su definisani kriterijumi performansi i pragovi prihvatljijvosti.
- Ključna mesta – Pre preduzimanja optimizovanja je potrebno da se jasno utvrdi koja su to ključna mesta u softveru na čijim performansama moramo da radimo. To će biti mesta koja imaju veze sa kriterijumima performansi i pragovima prihvatljivosti i na kojima merenje performansi ukazuje na određene probleme.

Ova četiri obrasca mogu da predstavljaju vid uputstva za sprovođenje lokalizovane optimizacije. Obrasci *Merenje performansi* i *Ključna mesta* se odnose prvenstveno na prostorno lokalizovanje optimizacije, a obrasci *Kriterijumi performansi* i *Prag prihvatljivosti* nam pomažu da odredimo dubinu optimizacije.

Priprema za lokalizovanje

Lokalizovanje optimizacije počinje od pripreme. Dobra priprema je ključna za uspešno lokalizovanje optimizacije, pa i za njeno kasnije sprovođenje. Priprema se oslanja na prethodno ustanovljene kriterijume performansi i ciljeve optimizacije. Ona se sastoji od širokog i iscrpnog izučavanja problema sa kojim se suočavamo i okolnosti u kojima se optimizacija preduzima.

Pod izučavanjem problema obično podrazumevamo dobro poznavanje zadataka koje softver mora da reši, primenjenih algoritama i izvedene implementacije. Svaka optimizacija predstavlja neki oblik kompromisa, pa zato moramo dobro da znamo kakve kompromise eventualno smemo da pravimo i koliko daleko u tome možemo da idemo u prostoru konkretnih zadataka i algoritama.

Poznavanje implementacije nam omogućava da steknemo uvid u to šta eventualno možemo da menjamo u postojećoj implementaciji i koliko toga možemo da unapredimo bez modifikovanja ili zamenjivanja algoritma.

Poznavanje i razumevanje okolnosti se odnosi na alate koje koristimo, kontekst projekta i različite oblike prisutnih ograničenja. Alati obuhvataju sve ono što koristimo u razvoju i što je vezano za deo softvera koji pokušavamo da optimizujemo. Tu spadaju programski jezici, prevodioci, arhitektura procesora, mašinski jezik procesora, ali i različiti alati za merenje performansi. Optimizacija se često svodi na korišćenje nekih karakteristika alata, koje se relativno retko neposredno upotrebljavaju. Zbog toga nam iscrpno poznavanje alata otvara širi prostor za unapređenja i veći izbor potencijalno korisnih tehnika optimizacije.

Pri optimizovanju moramo da dobro poznajemo i poštujemo razna ograničenja sa kojima se suočavamo. Neka od ograničenja su implicirana poslovnim zahtevima (kako funkcionalnim tako i nefunkcionalnim), neka druga su posledica arhitekture procesora ili računara, a neka ograničenja nastaju usled prethodno načinjenih izbora tokom razvoja, kao što su izbor arhitekture softvera, izbor tehnologije za povezivanje komponenti ili izbor nekih infrastrukturnih elemenata (na primer, sistem za upravljanje bazom podataka ili sistem za deljenje fajlova i drugo).

Eksperimentisanje

Svaka pojedinačna optimizacija softvera je uvek samo *pokušaj* optimizovanja. Nakon što *probamo* neku optimizaciju, moramo uvek da proverimo da li ona zaista donosi očekivan pozitivan efekat. Ako eksperimentalno potvrdimo da optimizacija donosi povećanje performansi, onda moramo da procenimo i da li je taj dobitak vredan odgovarajućeg kvarenja programskog koda ili nije. Tek ako procenimo da jeste, onda optimizaciju odobravamo i usvajamo kao implementiranu. U svim ostalim slučajevima bi trebalo da je obrišemo iz programa, tj. da vratimo programski kod u stanje koje je prethodilo tom pokušaju optimizacije, ali i da dokumentujemo da smo konkretnu optimizaciju probali i da nam nije odgovarala.

Eksperimentisanje sa optimizacijama niskog nivoa je relativno jednostavno, zato što su pojedinačne optimizacije najčešće dobro lokalizovane. Sa druge strane, eksperimentisanje sa optimizacijama visokog nivoa je složeno, obimno i skupo, ali nam je i pored toga neophodno i korisno, zato što je cena eksperimentisanja ipak daleko manja nego što bi bila cena razvijanja dela softvera koji počiva na arhitekturi ili algoritmima koji ne mogu da pruže neophodne performanse.

13.5 Tehnike optimizacije

Optimizacija softvera se u praksi svodi na analizu i merenje performansi i odabir i primenu određenih tehnika, kojima se teži da se rastereti uočeno usko grlo. Skup mogućih tehnika je u osnovi veoma veliki. Na njihov broj utiče činjenica da optimizacije mogu da se izvode na više nivoa, kao i da možemo da imamo za cilj rasterećenje različitih uskih grla.

Tehnike optimizacije mogu da se podele na opšte i specifične. Opšte tehnike su one čija primena nije čvrsto vezana za programski jezik i mogu da se primenjuju na svim ili bar na većem broju različitih programskih jezika. Nasuprot njima, specifične tehnike optimizacije su one koje mogu da se primenjuju na jednom konkretnom programskom jeziku ili na manjem broju srodnih jezika. Neke specifične tehnike se čak odnose na konkretne prevodioce.

Najpre ćemo da razmotrimo tehnike optimizacije prema nivoima, a zatim ćemo da predstavimo neke opšte tehnike i neke tehnike specifične za C++. Pregled tehnika ćemo da završimo ukazivanjem na neke od najčešćih grešaka pri optimizaciji.

13.5.1 Tehnike optimizacije visokog nivoa

Kao što se oblikovanje arhitekture i algoritama odvija na apstraktnijem nivou od pisanja delova programa, tako se i optimizacija na nivou arhitekture ili algoritama izvodi na apstraktnijem nivou od optimizacije programskog koda. Ali ta apstraktnost se odnosi pre svega na principe i ciljeve, a ne na konkretne tehnike optimizacije.

Optimizacije višeg nivoa se zapravo svode na revidiranje projekta. U tom smislu teško možemo da govorimo o velikom broju konkretnih tehnika, već je pre reč o širokom prostoru za menjanje postojećih ili pravljenje novih elemenata projekta. Što je optimizacija višeg nivoa, to je ona istovremeno i manje univerzalna i više zavisi od konkretnog problema, arhitekture i algoritama.

Na primer, ako imamo algoritme za pretraživanje kolekcije tekstova i algoritme za analizu astronomskih fotografija, u oba slučaja ćemo raditi sa velikim količinama podataka, ali ne možemo da na isti način smanjujemo opterećenje radne memorije. Možemo da imamo sličan cilj, ili da pri razmatranju optimizacije primenjujemo neke slične principe, ali sama konkretna tehnika promene algoritma je potpuno drugačija, zato što se i algoritmi značajno razlikuju.

Ako ustanovimo da arhitektura ili projekat ne mogu da pruže neophodne performanse, to znači da imamo neke nove informacije. Po potrebi možemo da izvedemo i dodatne eksperimente i merenja i obezbedimo još dodatnih informacija. Sve prikupljene dodatne informacije predstavljaju dopunu skupa informacija o domenu i problemu, koje su već ranije bile prikupljene prilikom istraživanja sistema. Prikupljanjem dodatnih informacija smo napravili reviziju istraživanja sistema.

Na osnovu novih ili izmenjenih informacija, moramo da ponovimo neke elemente projektovanja onih delova softvera na koje te informacije imaju uticaj. Na osnovu rezultata revizije istraživanja sistema, najpre se pravi revizija analize problema, pa zatim i revizija svih aspekata modela domena i softvera.

Drugim rečima, optimizacija visokog nivoa predstavlja vid naknadne razrade projekta i samim tim je daleko sličnija projektovanju softvera nego njegovoj implementaciji. Posledica je i da se tehnike optimizacije visokog nivoa praktično svode na tehnike projektovanja softvera. Zbog toga ćemo pri razmatranju konkretnih tehnika obraditi svega nekoliko tehnika koje spadaju u optimizacije visokog nivoa, a videćemo da će i one uglavnom biti obrađene na relativno apstraktnom nivou.

Ako se optimizacije visokog nivoa primenjuju nakon implementacije softvera, onda one zahtevaju mnogo promena, pa i ponavljanje pisanja programskog koda, što ima za rezultat njihovu visoku cenu. Zato se teži, kao što smo već naglasili, da se optimizacije visokog nivoa primenjuju unapred, pre implementiranja softvera.

13.5.2 Tehnike optimizacije niskog nivoa

Što je optimizacija nižeg nivoa, to se ona odnosi na manji deo konkretnog programskog koda. Optimizacije niskog nivoa su relativno bliske hardveru računara i najviše utiču na rad procesora i njegovih komponenti. Zato se optimizacije niskog nivoa odnose pre svega na rasterećenje procesora ili delova procesora, kao što su procesorska jezgra, keš memorija, procesorska (memorijska) magistrala i slično. One za osnovni cilj imaju povećavanje efikasnosti rada procesora, u smislu povećavanja brzine rada programa.

Optimizacijama niskog nivoa najčešće može da se ostvari samo ograničeno rasterećenje ostalih resursa, kao što su radna memorija, mreža, diskovi i drugo. Što je neki resurs fizički udaljeniji od procesora, to je za njegovo rasterećivanje neophodno da se optimizacija preduzima na višem nivou: opterećenje radne memorije zavisi najviše od odabranog algoritma; opterećenje diskova zavisi i od algoritma i od arhitekture; opterećenje mreže zavisi najviše od arhitekture i slično.

U nastavku ovog poglavlja se uglavnom fokusiramo na optimizacije niskog nivoa, koje se tiču rasterećenja procesora. Mnoge od njih počivaju na prilagođavanju programskog koda specifičnostima računarskog sistema, arhitekture procesora, magistrale podataka i drugo. Među njima su izdvajaju optimizacije koje ističu upotrebu keš memorije i one kojima je cilj smanjivanje broja skokova i grananja.

Optimizacije koje ističu upotrebu keš memorije obično se odnose na menjanje redosleda upotrebe podataka, tako da se poveća verovatnoća da je naredni upotrebljen podatak već keširan. Povećanje efikasnosti usled intenziviranja upotrebe keš memorije može da bude zaista dramatično, čak i reda više od 100 puta. Dobitak je prvenstveno u tome što je rad sa keširanim podacima višestruko brži, ali i u tome što se izbegava ponovljeno punjenje keš memorije istim podacima.

Skokovi i grananja imaju veoma visoku cenu pri izvršavanju programa. Savremeni procesori dele obradu instrukcija u korake i istovremeno obrađuju, u različitim fazama, po nekoliko uzastopnih instrukcija. Skokovi remete sekvencijalno izračunavanje niza instrukcija i poništavaju značaj (ali ne i cenu) već urađenog posla na izvršavanju nekoliko narednih instrukcija. U slučaju petlji i uslovnih naredbi, procesori pokušavaju da predvide tok izvršavanja, ali to često nije uspešno. Zbog toga smanjivanjem broja skokova i grananja možemo da doprinesemo boljoj iskorišćenosti procesorskog vremena. Dodatno, poželjno je da se izbegavaju skokovi na udaljene delove koda, zato što je manja verovatnoća da takvi delovi koda budu keširani.

13.5.3 Opšte tehnike optimizacije

Odbacivanje nepotrebne preciznosti

Što su podaci sa kojima radimo precizniji, to je njihov zapis veći i izračunavanje operacija je složenije. Zbog toga bismo mogli da očekujemo da smanjivanjem preciznosti možemo da dobijemo na efikasnosti, zato što se tako pojednostavljuju zadaci koje postavljamo pred procesor. Odbacivaje nepotrebne preciznosti može da se primenjuje kao smanjivanje preciznosti u radu sa realnim brojevima (tj. brojevima u zapisu sa pokretnom zapetom) i kao smanjivanje opsega celih brojeva. Oblik ove optimizacije je i zamenjivanje rada sa realnim brojevima radom sa celim brojevima.

Pri ovoj tehnici optimizacije potrebno je da imamo u vidu da se njena primena različito odražava na različite vrste operacija i da razmatramo efekte optimizacije u odnosu na svaku vrstu operacija posebno.

- Efekat ove optimizacije na operacije prepisivanja može da bude minimalan, pa čak i negativan, ako je ciljna veličina zapisa različita od veličine reči procesora.
- Cena konverzija nije lako predvidiva i efekti moraju da se izmere.
- Efikasnost izračunavanja često zavisi od procesora.

Danas se ova optimizacija relativno retko primenjuje na realne brojeve, zato što na većini savremenih procesora praktično ne postoji razlika u radu sa tipovima `double` i `long double`, a rad sa tipom `float` može čak da bude i sporiji nego rad sa većim tipovima. Razlog za to je što se svi ovi tipovi pri izračunavanjima kovertuju u osnovni tip koji koristi jedinica za rad sa realnim brojevima – to je danas uglavnom neki oblik 64-bitnog zapisa.

Upotreba osnovnog celobrojnog tipa

Procesori obično najbrže rade sa celobrojim podacima čija veličina odgovara veličini procesorske reči. Operacije na većim podacima se uglavnom implementiraju

pomoću više operacija na manjim podacima, pa su zato manje efikasne. Operacije na manjim celobrojnim podacima su u većini slučajevima jednako efikasne, ali u nekim slučajevima mogu da zahtevaju implictne konverzije, koje mogu da dovedu do smanjivanja efikasnosti.

Upotreba umetnutih funkcija i metoda

Pozivanje posebno implementirane funkcije obuhvata prenos argumenata, pozivanje funkcije i povratak iz nje, a često i čuvanje i restauriranje vrednosti nekih registara. U slučaju relativno jednostavnih funkcija, cena samog mehanizma pozivanja može da bude veća nego cena izvršavanja tela funkcije. U takvim slučajevima može da se ostvari značajno povećavanje efikasnosti tako što se pozivanje funkcije zameni ugradnjom njenog tela na mestu pozivanja. To se naziva *umetanje* funkcija i metoda (engl. *inlining*).

Ova tehnika može da se primenjuje manuelno (eksplicitnim prepisivanjem tela funkcije), ili pomoću odgovarajuće tehnike programskog jezika ili u okviru optimizacije koju preduzima prevodilac. Na primer, osnovna tehnika za umetanje funkcija u programskom jeziku *C* je bila upotreba makroa, a u programskom jeziku *C++* se koristi posebna deklaracija `inline` kao sugestija prevodiocu da funkciju prevodi kao umetnutu[87].

Na primer, funkcija `swap(a,b)`, koja razmenjuje vrednosti dva cela broja, radi 10 do 15 puta brže ako se prevede kao umetnuta.

Integracija petlji

Slično kao što pozivanje funkcije ima svoju cenu, tako svoju cenu ima i implementacija petlje. Ako je telo petlje relativno jednostavno, onda cena mehanizma ponavljanja može da bude značajna u odnosu na cenu posla koji se radi u petlji. Ako imamo više petlji, takvih da svaka radi neki jednostavan posao, onda njihovo spajanje u jednu petlju sa složenijim korakom može da nam donese značajne uštede procesorskog vremena. Integracija petlji može da uštedi i do 50% vremena.

Ova optimizacija je suprotna pravilima pisanja čistog i razumljivog koda, čak dotle da imamo i refaktorisanja koja rade upravo suprotno. Pri tome je često i sasvim suvišna, zato što će neki prevodioci sami da je primene.

---

[87] Zbog karakteristike programskog jezika *C++* da se svi metodi definisani u okviru definicije klase tretiraju kao da su označeni da treba da budu umetnuti, u praksi nastaje prava poplava umetnutih metoda. Zato savremeni prevodioci sve više ignorišu odgovarajuće sugestije i automatski procenjuju da li bolje da se neka funkcija ili metod prevodi kao umetnuta ili ne.

Izmeštanje invarijanti van petlje

Sve ono što se računa u petlji, a ima uvek isti rezultat, potrebno je da se izmesti iz petlje i izračuna pre nje. Na primer, ako imamo petlju poput:

```
for( int i=0; i<limit(a,b,c); i++ )...
```

gde se promenljive `a`, `b`, `c`, i vrednost funkcije `limit(a,b,c)` ne menjaju tokom izračunavanja, onda je bolje da to zamenimo sa:

```
int lim = limit(a,b,c);
for( int i=0; i<lim; i++ )...
```

Ako je telo petlje relativno jednostavno, ili je izraz koji izračunava invarijantu relativno složen, ova optimizacija može višestruko da ubrza petlju.

Na primer, izmeštanje računanja veličine vektora van petlje može da ubrza petlju sa jednostavnim telom čak i više od 3 puta (tj. za više od 65%):

```
auto sz = v.size();
for( size_t i=0; i<sz; i++ ){...}
```

Zamenjivanje dinamičkog uslova statičkim

Ako opseg broja ponavljanja nije fiksan, ili se obrada ne vrši za sve elemente kojima pristupamo, onda nekada može biti efikasnije da se posao ipak uradi za sve podatke nego da se stalno proveravaju granice ili uslovi. Zamenjivanje dinamičkog uslova statičkim može da ima sličnosti sa izmeštanjem invarijanti, ali ima dodatni efekat – osim što se iz petlje izbacuje računanje uslova ili dela uslova, čime se neposredno štedi procesorsko vreme, ovde postoji i dodatna ušteda zbog toga što se eliminišu neka grananja.

Na primer, ako imamo neki niz i želimo da anuliramo vrednosti elemenata koji nisu već jednaki nuli:

```
for( size_t i=0; i<sz; i++ )
    if( niz[i] ) niz[i] = 0;
```

onda može da bude mnogo efikasnije da bezuslovno postavimo sve elemente na 0 nego da proveravamo njihove vrednosti:

```
for( size_t i=0; i<sz; i++ )
    niz[i] = 0;
```

U predstavljenom primeru razlika u brzini može biti i 5 do 10 puta.

Razmotavanje petlji

Jedan način da se smanji broj grananja je da se smanji broj ponavljanja u petlji, a da se umesto toga telo petlje eksplicitno navede više puta uzastopno. Na primer ako u petlji u $8N$ prolaza obrađujemo po jedan podatak, onda umesto toga možemo da napravimo petlju kroz koju prolazimo N puta ali tako da u svakom koraku sekvencijalno obradimo po 8 podataka.

Na ovaj način se neposredno narušava princip izbegavanja ponavljanja koda, ali može da se dobije i više od 50% efikasnije izračunavanje.

Ova optimizacija ne radi uvek. Ako je telo petlje već relativno složeno, onda njegovim multipliciranjem može da se dobije programski kod koji se slabije kešira, pa zbog toga rezultat može da bude i sporiji nego originalni kod. Takođe, prevodilac može da odluči da neke od operacija u petlji optimizuje na neki drugi način, pa onda razmotavanje može čak i da uspori rad programa, umesto da ga ubrza.

Tablice unapred izračunatih vrednosti

Ako se neke funkcije više puta izračunavaju u petlji, a za ograničen broj različitih argumenata, onda može da bude efikasnije da se umesto izračunavanja konsultuju tablice, koje sadrže već izračunate vrednosti. Prvi doprinos je što se tako izbegavaju složena računanja i grananja. Drugi doprinos je što su takve tablice obično relativno male, pa efikasno koriste keš memoriju. Ako je izračunavanje relativno složeno ili sadrži granjanja, onda ova optimizacija može da donese veliko unapređenje efikasnosti.

Na primer, neka niz sadrži cele brojeve u rasponu od 0 do 20 i u petlji želimo da izbrojimo koliko brojeva je deljivo sa 5. Uobičajen način je da računamo neposredno:

```
for...
    if( niz[i] % 5 != 0 )
        n++;
```

Ali ako napravimo tablicu, koja za brojeve deljive sa 5 sadrži 1 a za ostale 0, onda pomoću te tablice možemo da ubrzamo petlju i do 10 puta:

```
for...
    n += tablica[ niz[i] ];
```

Kao i u slučaju razmotavanja petlji, može da se desi da prevodilac sam izvede neke optimizacije pa da dobitak bude svega 30-50%, a ne očekivanih 90%.

Eliminacija grananja i petlji

Ovu tehniku možemo da posmatramo kao uopštenje ili kombinaciju nekih od prethodnih tehnika, koje ostvaruju doprinos performansama tako što smanjuju broj grananja i petlji.

Odličan primer je poznati slučaj brojanja bitova u 32-bitnom broju pomoću tablice izračunatih vrednosti i razmotavanja petlji:

```
const short tablica[] = {0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4};
short brojBitova(int x)
{
   return tablica[(x) & 0xF] +
          tablica[(x >> 4) & 0xF] +
          tablica[(x >> 8) & 0xF] +
          tablica[(x >> 12) & 0xF] +
          tablica[(x >> 16) & 0xF] +
          tablica[(x >> 20) & 0xF] +
          tablica[(x >> 24) & 0xF] +
          tablica[(x >> 28)];
}
```

Dobijena implementacija nije sasvim laka za razumevanje i održavanje, ali zato i do 10 puta skraćuje vreme izvršavanja u odnosu na pojedinačno brojanje.

Smanjivanje broja argumenata funkcije

Prenošenje argumenata i rezultata potprograma se uobičajeno odvija pomoću registara i računskog steka. Upotreba registara procesora je daleko efikasnija, ali broj registara koji mogu da se koriste za te namene je ograničen. Znači, što je više argumenata, ili što su argumenti veći, to je manje verovatno da njihovo prenošenje može da se izvede pomoću registara i dolazi do intenzivnije upotrebe steka. Cilj smanjivanja broja argumenata funkcije je da se smanji upotreba steka i da se tako podigne efikasnost mehanizma za prenošenje argumenata, a sa tim i čitavog pozivanja funkcije.

Doprinos ove optimizacije zavisi od mnogo faktora – broja argumenata, broja registara procesora, veličine keša, načina i učestalosti pozivanja funkcije, načina upotrebe argumenata u funkciji i drugo, pa da ga je zato relativno teško predvideti. Zbog toga moramo da merimo i poredimo performanse pre i posle optimizacije, da bismo znali da li je optimizacija isplativa.

Problem je u tome što smanjivanje broja i veličine argumenata često mora da se nadoknadi na drugi način. Na primer, umesto da se prenosi pet argumenata, oni mogu da se zapišu u pomoćnoj strukturi, pa da se onda prenese samo adresa te strukture. Međutim, tako samo menjamo način prenošenja – umesto da prepisujemo argumente u registre ili na stek, mi ćemo ih prepisati u pomoćnu strukturu, koja će verovatno opet biti na steku (ako je lokalna).

Na značaj ove optimizacije utiču i savremene arhitekture procesora. One obezbeđuju veći broj registara, koji mogu da se koriste za prenošenje argumenata. Pored toga, moramo da imamo u vidu i da je rad sa računskim stekom relativno brz, zato što se stek odlično kešira.

Izbegavanje globalnih promenljivih

Upotreba lokalnih promenljivih je često efikasnija od upotrebe globalnih promenljivih. Za to ima nekoliko razloga:

- lokalne promenljive se zapisuju na računskom steku, koji se dobro i efikasno kešira, pa zato predstavlja verovatno najefikasniji deo radne memorije;
- ako program radi u više niti, onda globalne promenljive mogu da se dele između niti i moramo da se staramo o njihovom deljenju, što komplikuje implementaciju i usporava rad;
- ako se lokalna promenljiva često koristi u nekoj funkciji, onda prevodilac može da optimizuje prevod na mašinski jezik tako da ta promenljiva bude smeštena u registru a ne u memoriji[88]; odgovarajuća optimizacija za globalne promenljive je dovoljno složena da se retko preduzima
- i drugo.

Postoje i slučajevi kada je upotreba globalne promenljive bolji izbor. Na primer, ako se neki podatak koristi na veoma velikom broju mesta, onda lokalizacija njegove upotrebe podrazumeva da se on prenosi kao argument velikom broju funkcija, pa se stalno plaća cena njegovog prepisivanja (ili bar prepisivanja njegove adrese). U takvim slučajevima je obično potrebno da se eksperimentisanjem i merenjem proveri da li je efikasnije koristiti lokalne ili globalne promenljive.

Lokalnost upotrebe podataka

Savremeni procesori raspolažu višeslojnom i veoma efikasnom keš-memorijom. Međutim, da bismo iskoristili njene mogućnosti moramo da pazimo šta radimo. Efikasnost keša počiva na prostornoj i vremenskoj lokalnosti upotrebe podataka i programskog koda. Vremenska lokalnost podrazumeva ponovljenu upotrebu istih podataka u kratkim vremenskim intervalima, a prostorna lokalnost podrazumeva uzastopnu upotrebu podataka koji su zapisani jedni blizu drugih u memoriji.

Na primer, uobičajeno je da se dobro keširaju računski stek, lokalne promenljive (koje se uobičajeno nalaze na računskom steku) ili nizovi koji se sekvencijalno obrađuju. Dinamičke strukture podataka, kao što su povezane liste i stabla, se daleko teže keširaju, što utiče na njihovu efikasnost. Takođe, dobro se kešira programski

[88] U nekim programskim jezicima postoje deklaracije kojima se prevodiocima sugeriše da neku promenljivu čuvaju u registru. Na primer, u C/C++-u za to služi ključna reč `register`. Međutim, savremeni prevodioci uglavnom ignorišu takve sugestije i sami pokušavaju da procene da li je isplativije da se koristi registar ili memorija.

kod koji se često ponavlja (telo petlje), a lošije kod koji divergira (velike funkcije sa mnogo grananja, kod koji nema duže sekvencijalne delove i slično).

Razmotrimo, na primer, naredni isečak programskog koda, koji računa sumu elemenata matrice:

```
const unsigned len = 10000;
int niz[len][len];
...
int sum = 0;
for( unsigned i=0; i<len; i++ )
    for( unsigned j=0; j<len; j++ )
        sum += niz[j][i];
```

Ovaj programski kod je ispravan, ali ne koristi lokalnost podataka pri obradi elemenata niza. Nakon što se sumi doda element `niz[j][i]`, naredni element koji će se dodati je `niz[j+1][i]`. Ova dva elementa se nalaze u memoriji međusobno razdvojeni jedan od drugog sa 40000 bajtova i malo je verovatno da će oba da budu istovremeno u keš memoriji, a čak i da jesu, sledeći element će biti još 40000 bajtova dalje, pa sledeći još 40000 itd.

Očigledno je da ova implementacija ne poštuje lokalnost upotrebe podataka. Ako bismo je popravili tako da se posle elementa `niz[j][i]` obrađuje `niz[j][i+1]`, a ne `niz[j+1][i]`:

```
for( unsigned i=0; i<len; i++ )
    for( unsigned j=0; j<len; j++ )
        sum += niz[i][j];
```

onda bi uzastopno obrađivani podaci bili neposredno susedni u memoriji i bio bi poštovan princip lokalnosti upotrebe podataka. Mogli bismo sa pravom da očekujemo da će izmenjen programski kod biti efikasniji.

Zaista, ova mala izmena može da napravi veliku razliku u performansama. U zavisnosti od arhitekture procesora i veličine keša, ali i od veličine matrice koja se obrađuje, povećanje efikasnosti može da bude čak i do 100 puta.

Lokalnost upotrebe programskog koda

Kao što mogu da se keširaju upotrebljavani podaci, tako se kešira i programski kod. Zbog toga je važno da se program piše tako da se njegovi delovi bolje keširaju. Ova tehnika optimizacije počiva na izbegavanju nekoliko osnovnih oblika slabog keširanja programskog koda:

- ako imamo relativno veliku funkciju sa mnogo grananja, onda je povećana verovatnoća da će neki skokovi unutar te funkcije biti skokovi na delove koda koji nisu keširani;

- ako koristimo dinamičko vezivanje funkcija, onda konkretne pozivane funkcije često neće biti keširane;
- ako iz jedne funkcije često pozivamo brojne druge funkcije, onda ni one verovatno neće biti dobro keširane
- i drugo.

Neki od vidova unapređivanja lokalnosti koda su:

- skraćivanje funkcija;
- umetanje funkcija koje se najčešće pozivaju;
- eliminacija grananja;
- pažljivo određivanje redosleda proveravanja uslova;
- upotreba statičkog umesto dinamičkog vezivanja funkcija
- i drugo.

Pažljivo određivanje redosleda proveravanja uslova

Cilj ove optimizacije je da se smanje broj poređenja i broj skokova. Osnovna ideja je da se u slučajevima gde se proverava više uslova i bira jedna od više grana, najpre proveravaju oni uslovi za koje se očekuje da će češće biti ispunjeni, da bi se tako smanjio i broj proveravanih uslova i broj skokova.

Na primer, ako imamo uslovne izraze `uslov1`, `uslov2` i `uslov3`, takve da je najčešće ispunjen `uslov1`, pa onda `uslov2`, pa `uslov3`, a najređe nije ispunjen nijedan od njih, onda bi njihovo proveravanje trebalo da ide tim redom:

```
if( uslov1 ){...}
else if( uslov2 ){...}
else if( uslov3 ){...}
else {...}
```

Jedna od varijanti ove optimizacije je i hijerarhijska organizacija granjanja, tako da se umesto sekvencijalnog proveravanja velikog broja uslova oni podele u hijerarhiju približno ravnomerne dubine:

```
if( A ){
    if( uslov1 ) {...}
    else if( uslov2 ) {...}
    else {...}
}
else {
    if( uslov3 ) {...}
    else if( uslov4 ) {...}
    else {...}
```

```
}
```

Ova optimizacija može da se primeni i na višestruka grananja, tj. na naredbu `switch`.

Varijanta ove optimizacije je da se prevodiocu sugeriše koja uslovna grana će češće da se bira, da bi mogao da optimizuje mašinski kod tako da se linearno izvršava ona grana koja je verovatnija i da se manje vremena gubi na uslovnim skokovima. U programskom jeziku *C++*, od standarda *C++20*, postoje atributi `likely` i `unlikely`. Oni služe da se označe grane za koje očekujemo da predstavljaju verovatniji odnosno manje verovatan smer uslovnog grananja. Na primer, ako znamo da funkcija `skoroUvekTacna` najčešće vraća `true`, a sasvim retko `false`, onda možemo da napišemo:

```
if( skoroUvekTacna(...)) [[likely]] {   ...}
else {...}
```

ili:

```
if( skoroUvekTacna(...)) {...}
else [[unlikely]] {...}
```

Snižavanje složenosti operacije

Neke operacije mogu da se zamene efikasnijim operacijama. Na primer, umesto `a*8` možemo da napišemo `a<<3`. Takve optimizacije su nekada bile veoma cenjene, ali danas obično očekujemo da ih prevodioci sami izvode, čak i u složenijim slučajevima, i to bolje nego što bi prosečni programeri mogli da urade. Zato se takve optimizacije danas relativno retko primenjuju.

Snižavanje složenosti algoritma

Jedna od osnovnih opštih tehnika optimizacije je zamenjivanje neefikasnog algoritma nekim drugim algoritmom koji ima nižu složenost izračunavanja. Ova optimizacija ima nivo algoritma i zato je njena primena nešto složenija. Zato što spada u optimizacije visokog nivoa, uglavnom je poželjno da se primenjuje unapred.

Osnovna ideja je da se algoritam zameni alternativnim algoritmom, koji ima nižu složenost. Neposredna posledica je da će efikasnost novog algoritma sporije da se smanjuje sa porastom obima posla, u odnosu na prethodni algoritam.

Iako je konceptualno sasvim jednostavna, ova tehnika je među najsloženijim tehnikama optimizacije. Ne postoji opšte pravilo kako se pravi algoritam sa nižom složenošću, pa čak najčešće ne možemo unapred da znamo ni da li takav algoritam postoji.

Čak i kada znamo da postoji algoritam koji ima nižu složenost, ipak moramo da budemo oprezni pri njegovoj primeni. Problem je u tome što algoritmi sa nižom složenošću često imaju skuplje korake. Zbog toga može da se desi da za neke manje skupove podataka ovakvi algoritmi budu manje efikasni. Isplativost algoritama mora da se procenjuje na osnovu njihove složenosti, cene jednog koraka i karakteristika problema.

Izbor rešenja prema najčešćem slučaju

Neki algoritmi su za neke slučajeve efikasniji a za neke druge manje efikasni. Zbog toga je potrebno da se izbor algoritma pravi u skladu sa očekivanim slučajevima upotrebe. Na primer, za uređivanje kratkih nizova primitivan algoritam sortiranja *bubble-sort* može da bude efikasniji od naprednijeg algoritma *quick-sort*.

Može se reći da optimizacija „Pažljivo određivanje redoseda proveravanja uslova“ predstavlja specijalan slučaj ove optimizacije, ali i da ova optimizacija predstavlja specijalan slučaj optimizacije „Snižavanje složenosti algoritma“.

Pisanje zatvorenih funkcija

Funkcija (ili drugi potprogram) je *zatvorena* ako ne poziva nijednu drugu funkciju. Nešto šira definicija dopušta da zatvorena funkcija poziva zatvorene umetnute funkcije. Značajna karakteristika zatvorenih funkcija je da prevodiocu za njihovo prevođenje i optimizovanje nisu potrebne nikakve druge informacije o programu osim tela konkretne funkcije i informacija o podacima kojima ona rukuje.

Takve funkcije se bolje optimizuju od strane prevodilaca, pa se zato podstiče njihovo pisanje. Prevodioci često mogu da naprave efikasniji izvršni kod ako se koristi jedna složenija zatvorena funkcija nego ako je odgovarajuće ponašanje podeljeno na nekoliko manjih funkcija.

I ova optimizacija je suprotstavljena pravilima dobrog dizajna i refaktorisanja. Osim što proizvodi veće funkcije sa potencijalno grupisanim odgovornostima, ova tehnika posredno dovodi do ponavljanja delova koda, pa se njenom primenom otežava održavanje programa.

Uvođenje konkurentnog ili distribuiranog izračunavanja

Uvođenje konkurentnog ili distribuiranog izračunavanja se uobičajeno naziva *paralelizacijom* softvera.

Savremeni procesori imaju više izvršnih jezgara, što omogućava istovremeno izvršavanje više niti. To je posebno važno kada se radi o programima koji se pišu za servere i radne stanice, zato što takvi računari mogu da imaju po više procesora sa po više od 100 jezgara.

Naredni korak je uvođenje distribuiranog izračunavanja, tako da se izvršavanje programa podeli ne samo na više niti ili procesa već i da se ti procesi pokreću na više čvorova (tj. odvojenih računara). Uvođenje konkurentnog i distribuiranog izrača-

navanja može da donese veoma značajno ubrzanje, koje može da napravi razliku između upotrebljivosti i neupotrebljivosti softvera.

Paralelizovanje izračunavanja je često veoma složeno, a može da pruži najviše linearan dobitak u performansama u odnosu na broj jezgara procesora. Na primer, ako program pokrenemo na 10 računara sa po 20 jezgara, onda će on raditi najviše 200 puta brže, ali u praksi je ubrzanje najčešće značajno niže, zato što se gubi vreme na razmenjivanju podataka i sinhronizaciji.

Imajući to u vidu, uvek je važnije da prvo odaberemo algoritam što niže složenosti, pa tek onda da se bavimo njegovom paralelizacijom. U suprotnom možemo da dobijemo softver koji radi veoma brzo za male količine podataka ali nije upotrebljiv za velike.

Veoma je teško naknadno paralelizovati algoritme koji su po svojoj prirodi sekvencijalni. Na primer, ako algoritam u nekom koraku zahteva da su svi prethodni koraci dovršeni i koristi njihove rezultate, onda se takav problem teško paralelizuje. U takvim slučajevima je potrebno mnogo sinhronizacije, što otežava programiranje i usporava izvršavanje.

Implementacija paralelizacije može da se ostavi za kasnije faze razvoja, ali razmišljanje o paralelizaciji i njeno planiranje je poželjno da se urade već u ranijim fazama razvoja softvera, tj. pri određivanju arhitekture softvera i odabiru odgovarajućih algoritama. Zato se paralelizacija obično svrstava u optimizacije visokog nivoa.

Upotreba jedinica za masivno izračunavanje

Savremeni računari često imaju posebne jedinice za obradu vektorskih, matričnih ili tenzorskih podataka (engl. *vector processing unit – VPU*, *matrix processing unit – MPU, tensor processing unit TPU*). Za takve jedinice je zajednička podrška za instrukcije koje se izvršavaju istovremeno nad većim brojem elementarnih podataka, što se uobičajeno naziva arhitekturom „jedna instrukcija više podataka“ (engl. *Single Instruction Multiple Data – SIMD*). Ove jedinice se objedinjeno nazivaju *jedinice za masivna izračunavanja*, a njihova primena se naziva *masivno paralelno izračunavanje*. Primena ovakvih jedinica može da višestruko smanji trajanje nekih vrsta izračunavanja. Zbog toga ona predstavlja veoma važnu klasu optimizacija.

Vektorske jedinice se često implementiraju kao vid proširenja skupa instrukcija centralnog procesora. Na primer, prvi skup takvih instrukcija je u arhitekturu *x86* uveden još 1996. godine pod imenom *MMX*. Kasnije je dodavano više generacija *SSE* instrukcija (engl. *Streaming SIMD Extensions*), zaključno sa *SSE4* iz 2008. godine, kao i nekoliko generacija *AVX* instrukcija (engl. *Advanced Vector Extension*), od kojih je poslednja *AVX512* iz 2016. godine. Iako neke od tih instrukcija mogu da se koriste i kao deo operacija nad matricama, one su u osnovi projektovane za rad sa vektorima.

Integrisanje novih instrukcija u centralni procesor je obično praćeno i dodavanjem novih posebnih registara. Nove instrukcije i registri se u osnovi upotrebljavaju na isti način kao i ostale, pa se relativno lako upotrebljavaju na nivou mašinskog koda. Zato se optimizacija zasnovana na primeni takvih integrisanih jedinica obično svrstava u optimizacije nižeg nivoa i može da se koristi podjednako i za optimizaciju unapred i za optimizaciju unazad.

Problem sa integrisanjem jedinica za masivna izračunavanja u centralni procesor je u tome što je njihova implementacija prilično složena i raste sa njihovom snagom. Ako posmatramo broj tranzistora u procesorskim jedinicama, onda je složenost jedinica za masivna izračunavanja uporediva sa centralnim procesorima. Zbog toga je najčešće praktičnije da se moćne jedinice za masivno izračunavanje imlementiraju posebno, obično kao grafički procesori ili procesori za mašinsko učenje[89]. Ključna razlika u odnosu na integrisane skupove instrukcija za rad sa vektorima je u tome što ove jedinice ne izvršavaju isti program kao centralna jedinica, već svaka od njih radi svoj posao i pri tome moraju da komuniciraju da bi sarađivale.

Upotreba dodatnih jedinica za masivna izračunavanja zahteva da se podaci razmenjuju između više jedinica (na primer između centralnog i grafičkog procesora), što zahteva dodatne protokole za prenos podataka ili deljenje memorije. Ti protokoli moraju istovremeno da budu i efikasni i bezbedni, pa su zato i relativno složeni, što dodatno otežava pisanje programa koji koriste i centralne i dodatne jedinice. Pored toga, i arhitektura takvih jedinica i način njihovog programiranja se značajno razlikuju od uobičajene arhitekture centralnih jedinica. Da bi se to delimično olakšalo razvijaju se posebne programske biblioteke, kao što su *CUDA*, *OpenCL*, *HIP* i druge.

Usled složenosti njihovog programiranja, optimizacija primenom jedinica za masivno paralelno izračunavanje deli osnovne karakteristike sa optimizacijom uvođenjem paralelnog ili distribuiranog izračunavanja. I ova optimizacija se često svrstava u optimizacije visokog nivoa i primenjuje se ili unapred ili uz preduzimanje ozbiljnih promena u programskom kodu.

[89] Veliki broj centralnih procesora danas ima integrisane grafičke procesore ili jedinice za mašinsko učenje. Međutim, nivo njihove integracije nije ni približno isti kao što je slučaj sa jedinicama za vektorska izračunavanja. U većini savremenih slučajeva integracija se svodi na deljenje radne memorije i radne magistrale, što definitivno olakšava neke od problema, ali takve jedinice se ipak programiraju odvojeno od centralnih procesora. Pri tome je snaga takvih integrisanih jedinica obično daleko niža nego u slučaju odvojenih implementacija.

Odabir pogodnijeg algoritma ili strukture podataka

Jedan od najvažnijh činilaca razvoja efikasnog softvera predstavlja odabir najpogodnijeg algoritma i odgovarajućih struktura podataka. Neke aspekte značaja algoritama i struktura podataka smo već obradili kroz priču o projektovanju, ali i kroz priču o optimizacijama visokog nivoa, ranije u ovom poglavlju. Odabiranje boljeg algoritma i pogodnijih struktura podataka spada u optimizacije visokog nivoa, pa se ne preporučuje da se vrši kao vid optimizacije unazad. Naprotiv, uobičajeno je da se algoritmima i strukturama podataka bavimo u okviru planiranja i projektovanja, kako pre tako i u toku implementacije softvera.

Bavljenje algoritmima i strukturama podataka u okviru optimizacije unazad se obično svodi na dva postupka: (1) konstatovanje i po potrebi eksperimentalno potvrđivanje da je upravo implementirani algoritam ili struktura podataka faktor koji ograničava performanse i (2) pravljenje odnosno odabiranje boljeg algoritma ili strukture podataka. Prvi postupak se najčešće svodi na testiranje u različitim uslovima i potvrđivanje da izabrano rešenje ne može da prati porast složenosti problema koji bi trebalo da rešava. Drugi se u osnovi ne razlikuje od odgovarajućeg postupka u fazi planiranja i projektovanja softvera.

Algoritmi i strukture podataka su veoma zahtevna oblast, koja izlazi iz okvira ove knjige. Čitaocima preporučujemo da svoja znanja o algoritmima unaprede, na primer, pomoću knjige „Algoritmi“ [*Živković 2000*].

13.5.4 Primeri tehnika specifičnih za C++

Svaki programski jezik ima neke specifične karakteristike koje otvaraju prostor za specifične tehnike optimizacije. Ovde ćemo predstaviti neke tehnike optimizacije koje su specifične za programski jezik C++.

Neke od tih tehnika nisu izvorno zamišljene kao tehnike optimizacije koda, ali mogu da doprinesu performansama ili da olakšaju optimizaciju i održavanje programskog koda. Zato se podjednako često razmatraju i kao tehnike pisanja efikasnog koda, onda kada se on po prvi put piše, i kao tehnike za naknadnu optimizaciju.

Koristiti standardnu biblioteku

Standardna biblioteka programskog jezika C++ je projektovana i implementirana tako da obezbedi visoke performanse. Ona u velikom broju slučajeva pruža visok nivo apstrakcije, koja se prevashodno zasniva na upotrebi parametarskog polimorfizma. Za razliku od hijerarhijskog polimorfizma, koji podrazumeva upotrebu dinamičkog vezivanja metoda i nešto nižu efikasnost, parametarski polimorfizam omogućava da se pri prevođenju programa u potpunosti iskoriste mogućnosti strogog proveravanja tipova, statičkog vezivanja, pa i automatske lokalne optimizacije prema konkretnim slučajevima upotrebe.

Veoma je teško napisati programski kod koji radi posao efikasnije nego odgovarajući elementi standardne biblioteke, koji su godinama razvijani, debagovani i optimizovani[90]. Čak i kada je to moguće, razumno je postavi pitanje isplativosti, zato što je dobitak u performansama obično relativno mali, a cena koju plaćamo je dobijanje nefleksibilnijeg koda.

Zbog toga se korišćenje standardne biblioteke svrstava u osnovna pravila pisanja efikasnih programa na programskom jeziku C++. Naravno, to je pre svega jedna od osnovnih tehnika programiranja na C++-u, a ne tehnika optimizacije, ali standardna biblioteka se sa svakom novom verzijom standarda i sa svakom novom verzijom implementacije prevodilaca i biblioteke povećava i unapređuje, što nam pruža priliku da upotrebu standardne biblioteke iskoristimo i kao vid optimizacije.

Relativno često se dešava da su neki delovi programa implementirani eksplicitno, bez upotrebe standardne biblioteke, zato što u trenutku njihovog pisanja odgovarajuća funkcionalnost nije bila obuhvaćena standardnom bibliotekom. Nakon što je odgovarajuća funkcionalnost uvrštena u standardnu biblioteku, ima smisla da se razmotri zamenjivanje postojeće implementacije upotrebom tih novih elemenata biblioteke. Može da se očekuje i dobitak u performansama i dobitak u preglednosti programskog koda.

Upotrebljavati reference za prenošenje argumenata

Prenošenje objekata po vrednosti može da bude relativno skupo, kako zbog veličine tako i zbog eventualne složenosti unutrašnje strukture objekata. Prepisivanje objekata najčešće podrazumeva i primenu konstruktora kopije ili operatora dodeljivanja, koji sprečavaju da novi i stari primerak objekta nastave da nekontrolisano dele neke elemente unutrašnje strukture.

Umesto prenošenja velikih objekata po vrednosti mnogo je bolje upotrebiti prenošenje po referenci na konstantan objekat. U tom slučaju se objekat prenosi po imenu (adresi), ali tako da ne može da se menja. Prenošenje putem referenci je veoma slično prenošenju putem pokazivača, ali je sintaksno čistije i onemogućavaju se neke greške poput promene vrednosti pokazivača, praznih pokazivača i drugo.

Reference se relativno retko upotrebljavaju za vraćanje rezultata. Vraćanje reference mora da se koristi veoma oprezno, zato što takva referenca mora da se odnosi na objekat koji će preživeti povratak iz funkcije, tj. ne sme da se odnosi na lokalnu promenljivu ili drugi objekat koji će biti obrisan pri povratku iz funkcije. Obično se

[90] Bilo bi dosadno da je sve crno-belo, pa i ovde imamo izuzetke. Neki elementi standardne biblioteke su izrazito neefikasni, kao na primer `std::regex` i donekle `std::unordered_map`. U oba slučaja se performansama plaća opštost principa i rešenja.

upotrebljava samo za pristupanje nekim delovima veće strukture, na primer elementima niza i slično.

Korišćenje referenci se koristi pretežno kao tehnika pisanja programa, ali može da se upotrebi i kao tehnika optimizacije.

Uvoditi lokalne promenljive što bliže mestu upotrebe

U programskom jeziku *C++* pravljenje objekta pretpostavlja i njegovu inicijalizaciju, a brisanje objekta njegovu deinicijalizaciju. To praktično znači da svako pravljenje nekog privremenog objekta u okviru funkcije ima svoju cenu, koja može da bude relativno značajna, posebno ako se često poziva. Zbog toga je poželjno da se lokalne promenljive ne uvode na početku tela funkcije, već neposredno pre nego što budu potrebne. Ako je to u nekom ugneždenom bloku, na primer u jednoj od grana uslovne naredbe, onda će taj lokalni objekat biti napravljen (a kasnije i obrisan) samo ako se ta grana izvrši. Na taj način se dobija na performansama, a i programski kod postaje razumljiviji.

Ova optimizacija nije specifična samo za *C++*, ali u njemu ima značajnije posledice nego u većini drugih jezika. U mnogim drugim jezicima ovo nije optimizacija, već samo pravilo za preglednije pisanje programskog koda. Na primer, u jezicima kao što su Java ili *C#*, objekti se koriste putem referenci, pa ni deklarisanje ni oslobađanje promenljive ne podrazumevaju značajan dodatni posao, a pravljenje i brisanje mogu da se rade dublje u telu. U programskom jeziku *C* ne postoje automatska konstrukcija i destrukcija, pa se uvođenje lokalne promenljive na početku funkcije reflektuje samo na njenu alokaciju, a to se i inače radi odjedanput na početku funkcije za sve lokalne promenljive definisane u funkciji.

Odložena inicijalizacija objekata

Odložena inicijalizacija objekata je u suprotnosti sa principom *RAII* (pravljenje resursa je njegova inicijalizacija, engl. *resource acquisition is initialization*), koji se svrstava u najvažnije principe programiranja na programskom jeziku *C++*. Osnovna ideja ovog principa je da je svaki napravljen objekat u potpunosti inicijalizovan i spreman za upotrebu.

Nasuprot tome, ova tehnika optimizacije počiva na pretpostavci da potpuna inicijalizacija objekata može da bude relativno složena i skupa, a da nama često nisu neophodni svi aspekti takve složene inicijalizacije. Ako su neki aspekti inicijalizacije potrebni samo povremeno i relativno retko, možda ima smisla da pri pravljenju objekata izvedemo samo najosnovniji deo inicijalizacije, a da ostatak posla radimo tek ako nam bude potreban.

Na taj način mogu da se ubrzaju neke jednostavnije operacije sa objektima, ali može i da se uspori rad u nekim kompleksnijim slučajevima, zato što ćemo morati prvo da proveravamo da li je odgovarajući aspekt inicijalizacije obavljen, pa da ga obavimo ako nije, pa tek onda da nastavimo posao.

Ova optimizacija mora da se izvodi vrlo oprezno i da se dobro dokumentuje. Potencijalni problemi nastaju ako se pri nekom narednom menjanju programa pretpostavi da je neki objekat pri konstrukciji potpuno inicijalizovan, a da to nije slučaj. Takve greške mogu da budu prikrivene i da se relativno teško pronalaze i ispravljaju.

Koristiti liste inicijalizacija članova i baznih objekata

Pri konstrukciji objekata u programskom jeziku C++, inicijalizacija članova objekta i nasleđenih delova objekta može da se obavlja na dva načina – u telu konstruktora i u okviru liste inicijalizacija. Značajno je jednostavnije i efikasnije upotrebljavati listu inicijalizacija.

Jedna od osnovnih pretpostavki pri pisanju konstruktora je da su u trenutku kada započinje izvršavanje tela konstruktora već konstruisani i upotrebljivi svi elementi koji čine objekat. Znači, svi članovi objekta, kao i svi nasleđeni delovi su već konstruisani. U telu konstruktora možemo da ih menjamo i da usklađujemo njihov sadržaj prema potrebnim ciljevima, ali sve te promene se obavljaju nad objektima i podacima koji su već inicijalizovani. To znači da se postavljanjem vrednosti u telu konstruktora u izvesnoj meri poništava ono što je već urađeno pri konstrukciji odgovarajućih delove, tj. da se njihov sadržaj dva puta postavlja – najpre pri konstrukciji, pa onda pri menjanju.

Namena liste inicijalizacija je da se pomoću nje opiše kako će koji član i koji nasleđeni deo klase da se inicijalizuje, tj. koji konstruktori i sa kojim parametrima će biti za to upotrebljeni. Ako podatak ili nasleđeni deo nisu navedeni u listi inicijalizacija, onda će se za njihovu inicijalizaciju upotrebiti podrazumevani konstruktor. Zbog toga je mnogo efikasnije da navedemo kako će već pri konstrukciji ti elementi dobiti odgovarajući sadržaj, umesto da ih prvo pravimo i inicijalizujemo pomoću podrazumevanih konstruktora a zatim da tako određeni sadržaj menjamo.

Koristiti konstrukciju objekata a ne dodeljivanje

Motivacija za ovu tehniku je slična kao i za upotrebu liste inicijalizacija. Umesto da objekat prvo napravimo, pa mu onda dodelimo vrednost, bolje je da ga odmah napravimo na odgovarajući način.

Znači, umesto da pišemo nešto poput:

```
A a;
a = 5;
```

ili nešto kao:

```
A a;
a = (A)5;   // Ovo je isto kao: a = A(5);
```

bolje je da pišemo nešto poput:

```
A a = 5;   // Ovo je isto kao: A a { 5 };
```

U prvom slučaju, prvo pravimo objekat `a`, pomoću konstruktora bez argumenata, pa mu onda dodeljujemo vrednost 5.

U drugom slučaju prvo pravimo objekat `a`, pomoću konstruktora bez argumenata, pa mu onda dodeljujemo objekat konstruisan konstruktorom koji je dobio parametar 5. Ako ne postoji eksplicitno napisana operacija dodeljivanja celog broja objektu klase `A`, onda se prvi slučaj prevodi potpuno isto kao i drugi.

U trećem slučaju pravimo novi objekat `a` pomoću konstruktora koji ima celobrojni parametar 5. Samo u ovom slučaju imamo samo jednu operaciju, dok u prethodna dva slučaja prvo pravimo objekat pa ga onda menjamo, a pri tome se čak prave i brišu i dodatni privremeni objekti.

Iz delova koda koji se optimizuju izbaciti rukovanje izuzecima

Ova optimizacija se relativno često predlaže, ali je zapravo diskutabilna. Njena motivacija je relativno jasna – ako u delu programskog koda nema mogućnosti da se pojavi izuzetak, zato što se u njemu koriste samo operacije i funkcije koje ne mogu da proizvedu izuzetak, onda nema ni razloga da se u njemu staramo o izuzecima. Ako znamo da staranje o izuzecima ima cenu, onda možemo da probamo da napravimo uštedu tako što ćemo da izbegnemo taj deo posla.

Problemi sa ovako optimizovanim kodom nastaju pri njegovom menjanju. Ako označimo da funkcija ne proizvodi izuzetke, a pri njenom menjanju iskoristimo neku operaciju ili funkciju koja može da proizvede izuzetak, onda ćemo imati posledice i po pouzdanost i po efikasnost programa.

Ako nastupi izuzetak, a nismo u stanju da ga obradimo, biće prekinut rad programa. Ipak, ako smo sigurni da upotrebljene operacije, koje načelno mogu da izazovu izuzetke, u konkretnom slučaju koristimo tako da ih one sigurno neće izazvati, onda nas prethodni problem ne brine mnogo. Bar ne u trenutku pisanja, ali mogu da nastupe problemu kada dođe do menjanja programa.

Specifičan problem može da nastane zbog toga što prevodioci često ugrađuju poseban deo koda za obezbeđivanje prelaza iz potprograma koji rade sa izuzecima u one koji ne rade sa njima, tako da u nekim slučajevima ovakav zahvat može čak da izazove *pad performansi*.

Savremene tehnike za podršku rada sa izuzecima su oblikovane tako da se u redovnim okolnostima, tj. kada nema izuzetaka, plaća sasvim mala cena i efikasnost programskog koda se praktično ne umanjuje. Sa druge strane, kada nastupi izuzetak, onda njegova obrada interno nije jednostavna i može da ima visoku cenu, ali tu se radi o vanrednim okolnostima i stoga ne predstavlja problem.

Ovu tehniku je važnije razumeti kao uputstvo da se izuzeci ne koriste za stvari za koje nisu namenjeni, na primer za upravljenje kontrolom toka programa, kao „napredni“ oblik naredbe *GO-TO*. Obrada izuzetaka je složena i relativno spora, što nam nije posebno važno u slučejevima za koje su oni namenjeni (prepoznavanje i obrađivanje neočekivanih specijalnih slučajeva, problema i grešaka), ali je veoma skupo za druge vidove upotrebe.

Upotreba constexpr izraza i uslova

Od verzije *C*++11 dodata je ključna reč `constexpr`, koja služi da se deklarišu promenljive, izrazi i uslovi koji mogu da se jednokratno izračunavaju u fazi prevođenja programa. Ova tehnika je preporučljivija nego upotreba makroa, zato što je čistija i lakša za debagovanje.

Naravno postoje određena ograničenja, prvenstveno u odnosu na tip vrednosti izraza ili funkcije. Ukratko (ali ne i sasvim precizno), `constexpr` može da se koristi za sve tipove koji imaju trivijalnu inicijalizaciju kopije i deinicijalizaciju.

Upotreba meta-programiranja

Primenom šablona i parametarskog polimorfizma smo se bavili u poglavlju *11 - Polimorfizam.* Videli smo da je parametarski polimorfizam vid statičkog polimorfizma, koji se u potpunosti razrešava u fazi prevođenja, pa zato primena šablona funkcija i klasa može da se veoma dobro optimizuje.

Poseban aspekt šablona je *meta-programiranje*, odnosno pisanje programskog koda koji se izračunava u fazi prevođenja programa. To je značajno uopštenje u odnosu na izraze sa `constexpr`. Na taj način čak neki složeniji aspekti izračunavanja mogu da se ubrzaju tako što se neće ponavljati svaki put pri izvršavanju programa već samo jedanput pri njegovom prevođenju. Meta-programiranje može da se upotrebi za pravljenje različitih tablica, uslova i slično. Meta-programiranje pomoću šablona je Tjuring-kompletno, tj. sve što može da se izračuna u *C*++-u, načelno može da se izračuna i primenom meta-programiranja u fazi prevođenja. U praksi nije sve baš tako lepo, zato što prevodioci rade relativno neefikasno sa meta-programima, a i uvode različita ograničenja. Pored toga, meta-programi se relativno teško debaguju.

Radi se o relativno složenoj tehnici za čije opisivanje nemamo dovoljno prostora. Zainteresovanima preporučujemo da pročitaju neku od knjiga [*Alexandrescu 2001*; *Abrahams 2004*].

Zamenjivanje dinamičkog vezivanja statičkim

Upotreba hijerarhijskog polimorfizma je neposredno vezana za dinamičko vezivanje metoda. Odlučivanje o verziji metoda koji će se u nekom trenutku pozvati zavisi od klase konkretnog objekta na kome je metod pozvan, pa zato mora da se odvija u fazi izvršavanja programa. Programski jezik C++ je jedan od retkih OOPJ koji podržava i statičko i dinamičko vezivanje metoda. Štaviše, podrazumevano

vezivanje je statičko, a metode koji želimo da se vezuju dinamički moramo eksplicitno da označavamo pomoću ključne reči `virtual`.

Dinamičko vezivanje metoda je sporije. Razlika u brzini nije velika, ali postoji. Za složenije metode ona obično nije značajna, ali za jednostavnije metode može da bude veoma važna. Na efikasnost pozivanja statički vezanih metoda dodatno utiču mogućnost bolje predikcije skokova i činjenica da jednostavni statički metodi mogu da budu umetnuti na mestu pozivanja (engl. *inline*). Dinamički metodi ne smeju da se umeću, zato što u fazi prevođenja još uvek ne znamo koja verzija metoda će biti potrebna u kom slučaju.

U slučaju kada je neki metod deklarisan kao dinamički (tj. kao virtualan), a zapravo ne postoji više verzija tog metoda (tj. nije izmenjeno njegovo ponašanje u izvedenim klasama), onda ima smisla da se vezivanje tog metoda promeni tako da ne bude dinamičko nego statičko. Ovakva optimizacija posebno ima smisla ako se radi o jednostavnom metodu koji bi mogao da bude umetnut.

Ako se pri nekom menjanju programa naknadno pojavi izvedena klasa koja bi trebalo da ima izmenjeno ponašanje statički vezanog metoda, obično je relativno jednostavno da se metod (ponovo) proglasi za virtualan i da se tako obezbedi njegovo dinamičko vezivanje.

Implementirati operatore alokacije i dealokacije

Za dinamičko alociranje i dealociranje objekata u programskom jeziku *C++* se upotrebljavaju operatori `new` i `delete`. Svaka klasa može da ima svoje verzije ovih operatora. Ako oni nisu eksplicitno napisani, onda se koriste globalne univerzalne varijante. Globalne univerzalne operacije se koriste i pri alokaciji i dealokaciji nizova, a možemo i sami eksplicitno da ih upotrebimo, ako ih referišemo sa `::new` i `::delete`.

Ako imamo neku klasu čiji se objekti stalno iznova dinamički prave i brišu, onda može da bude korisno da implementiramo odgovarajuće verzije metoda `new` i `delete` u toj klasi. To je posebno korisno ako se radi o malim objektima, zato što globalni ugrađeni operatori rade za sve različite veličine objekata, pa zbog svoje univerzalnosti nisu posebno optimizovani za neke specifične namene. Optimizovanjem ovih metoda za našu klasu, možemo da značajno podignemo performanse pravljenja i brisanja objekata. Dodatna korist može da bude eventualno ostvarivanje lokalnosti takvih objekata u prostoru.

13.5.5 Optimizacije u hodu

Već smo naglasili da su programski jezik *C++* i njegova standardna biblioteka posebno posvećeni pisanju efikasnih programa. Posledica takvog pristupa je da i proces programiranja na *C++*-u počiva na značajnom broju tehnika koje se odnose na

staranje o performansama. Pridržavanje takvih tehnika u toku pisanja programa, kao vid *optimizacije unapred*, je uobičajeno među programerima koji pišu na C++-u.

Pri predstavljaju specifičnih tehnika optimizacije za C++ smo nagovestili, a ponegde i eksplicitno naglasili da se neke tehnike koriste ne samo kao tehnike optimizacije već i kao uobičajene tehnike pri pisanju programa. I neke opšte tehnike optimizacije se upotrebljavaju na sličan način.

Takve tehnike se ponekad nazivaju *optimizacijama u hodu*. U prethodnim odeljcima smo već istakli neke od njih pa im se nećemo ponovo vraćati:

- upotreba umetnutih funkcija i metoda;
- izbegavanje globalnih promenljivih;
- lokalnost upotrebe podataka;
- lokalnost upotrebe programskog koda;
- upotreba standardne biblioteke;
- prenošenje objekata po referenci;
- definisanje privremenih promenljivih što dublje u kodu;
- upotreba liste inicijalizacija;
- upotreba konstrukcije objekata a ne dodeljivanja;
- upotreba izraza i uslova `constexp` i
- upotreba meta-programiranja.

13.5.6 Prepuštanje optimizacije prevodiocu

Savremeni prevodioci mogu da se pohvale automatizacijom velikog broja različitih optimizacija niskog nivoa. Konkretne raspoložive tehnike zavise od verzije prevodica. Prevodioci obično nude i preporučene pakete optimizacija, koji obuhvataju one optimizacije za koje se procenjuje da ostvaruju najveći doprinos.

Pri primeni optimizacija na nivou prevodilaca moramo da imamo u vidu da se tehnike optimizacije stalno unapređuju i inoviraju, pa se zbog toga u njihovoj implementaciji relativno često pronalaze bagovi. Preporučeni paketi opcija su uglavnom bezbedni za upotrebu, zato što se sastoje od optimizacija koje su relativno dobro testirane, pa je verovatnoća da sadrže bagove uglavnom relativno niska. Sa druge strane, uključivanje dodatnih pojedinačnih optimizacija zahteva posebnu pažnju.

Pregled konkretnih opcija prevodilaca izlazi van okvira ove knjige. Čitaoce upućujemo da potraže opis podržanih optimizacija i odgovarajućih opcija u dokumentaciji za svoje prevodioce.

13.5.7 Česte greške

Pregled tehnika optimizacije ćemo da dovršimo podsećanjem na neke od grešaka koje se često prave pri optimizovanju softvera. Većina ovih grešaka je u nekom obliku već pomenuta u ovom poglavlju, ali zbog njihovog velikog značaja ne smemo da propustimo priliku da im se vratimo još jedanput.

Pogrešne pretpostavke

Nikada ne smemo da optimizujemo na osnovu pretpostavke da je neko rešenje efikasnije nego neko drugo. Da budemo do kraja precizni, možemo da pokušamo i trebalo bi da pokušamo sa primenom alternativnog rešenja, ali svakako uvek moramo da proverimo da li je promena donela povećanje performansi ili nije. Slično kao i u slučaju debagovanja, ako promena nije donela ono što smo od nje očekivali, onda bi trebalo da se poništi. Ne smemo da kvarimo dizajn programa zato što nešto *možda* radi efikasnije. Ako se nekada odlučimo da pravimo kompromis sa dobrim dizajnom, onda bar moramo da budemo sigurni da od toga imamo i neku korist, a ne samo štetu.

Kao što razvijaoci softvera teže da ubrzaju konkretan softver na kome rade, tako se projektanti procesora i razvijaoci prevodilaca trude da obezbede bolje tehnike implementacije, koje više doprinose performansama. Sa novim verzijama procesora nam dolaze i bolje performanse, a sa novim verzijama prevodilaca nam dolaze napredniji optimizatori. Pri tome je uobičajeno da se najviše pažnje posvećuje onim elementima za koje se pokazalo da su relativno neefikasni.

Šta se dešava ako mi optimizujemo program, a u međuvremenu se procesori ili prevodioci unaprede tako da se podignu performanse neoptimizovanog programa? Optimizacija softvera je stalna igra između lepo i dobro strukturiranog programa i različitih „trikova“ koji kvare dizajn ali koriste specifičnosti prevodilaca ili arhitekture računara da bi se neki posao uradio brže. Ali kada se te specifičnosti promene, onda mogu da se promene i posledice odgovarajućih trikova, pa se tako dešava i da neoptimizovani kod postane efikasniji od optimizovanog.

Pre nekoliko decenija se čak moglo okarakterisati kao neprofesionalno ponašanje ako bi neko u kodu napisao `a*8`, a ne `a<<3`, a danas o takvim stvarima praktično više nema razloga da razmišljamo. Moramo da budemo spremni da nešto što je juče bila optimizacija danas može da bude suvišno kvarenje koda, zato što više ne doprinosi performansama. Štaviše, u nekim slučajevima može da dođe i do pada efikasnosti, zato što dodatnim komplikovanjem otežavamo prevodiocu ili procesoru da iskoriste svoje tehnike optimizacije.

Možda bismo mogli da zamislimo neku idealnu budućnost u kojoj optimizovanje nije potrebno, već procesori i prevodioci sve što smo napisali sami modifikuju tako da i dalje radi tačno, ali najefikasnije što je moguće? Izvesno je da ćemo se u

narednim godinama (ili pre decenijama) postepeno približavati takvom idealu, ali je jednako izvesno da će još dugo biti potrebe za dodatnim manuelnim optimizacijama.

Smanjivanje koda nije uvek i njegovo optimizovanje

Iako intuicija može da nas navede da pomislimo da je manji kod istovremeno i efikasniji, na primeru razmotavanja petlji smo videli da nije uvek tako. Umesto da mnogo puta prolazimo kroz petlju i ponavljamo proveravanje uslova i skačemo na početak, videli smo da je nekada efikasnije da se telo petlje eksplicitno napiše više puta i da se smanji broj ponavljanja.

Ali kako se to slaže sa keširanjem koda? Zar ne bi trebalo da dovede do usporenja? Ispada da treba imati dobru meru. Ako preteramo pri razmotavanju petlji, lako može da se dogodi da telo petlje preraste veličinu keš-memorije i da efikasnost počne da pada.

Kada kažemo da je cilj optimizacije rasterećenje procesora, to ne znači da je dovoljno da se smanji broj instrukcija koje procesor mora da izvrši. Procesor je složena kolekcija resursa i rasterećenje procesora često znači dobro upravljanje svim tim resursima, a ne puko smanjivanje broja instrukcija:

- da, bolje je da imamo manje instrukcija;
- ali neke instrukcije su *mnogo* skuplje od drugih;
- skokovi (granjanja, ponavljanja) su među najskupljim operacijama;
- čitanje instrukcije iz memorije je mnogo skuplje nego čitanje iz keša;
- izvršavanje uzastopnih instrukcija na međusobno nezavisnim podacima u nekim slučajevima može da se odvija istovremeno
- i drugo.

Vidimo da smo od vrlo jednostavnih pretpostavki, koje je lako primeniti na pisanje programa, došli do nekih čija primena nije tako jednostavna. Na osnovu toga možemo da uopštimo zaključak ovog odeljka – pojednostavljivanje zaključivanja o optimizovanju često može da nas odvede na pogrešnu stranu. Optimizovanje softvera je složeno i ta složenost ne sme da se zanemaruje. Smanjivanje koda je samo jedan od takvih primera.

Optimizovanje tokom inicijalnog kodiranja

Istakli smo da preuranjena optimizacija može da donese mnogo više problema nego koristi. A onda smo opisali tehnike optimizacije koje se u hodu preduzimaju pri programiranju u *C++*-u. Zar to nije protivrečno?

Optimizacija i pisanje dobro dizajniranog programskog koda su u nekim jezicima jasno i drastično razdvojeni, ali u nekim drugim jezicima (pre svih *C* i *C++*) nisu, zato

što je jedan od osnovnih principa programiranja u tim jezicima upravo pisanje efikasnih programa.

Ako pogledamo tehnike za koje smo istakli da mogu da se primenjuju već pri pisanju koda, možemo da primetimo da te tehnike uglavnom ne kvare dizajn. Naprotiv, u njima se koriste specifičnosti programskog jezika koje omogućavaju da se zadrži čist i jasan dizajn, a da pri tome kod bude efikasan (na primer, liste inicijalizacija, prenošenje podataka po referenci, umetnute funkcije, `constexpr`) ili tehnike koje čak doprinose preglednosti i razumljivosti programa (na primer, upotreba lokalnih promenljivih, definisanje privremenih promenljivih što bliže mestu upotrebe, upotreba standarne biblioteke, konstrukcija umesto dodeljivanja).

Ako nam je cilj da pišemo efikasne programe, onda ćemo programski kod *odmah* da pišemo tako da bude efikasan. Ali pri tome bi pricipi agilnog razvoja i težnja dobrom dizajnu ipak trebalo da nam budu u prvom planu. Tokom inicijalnog kodiranja je prihvatljivo, pa čak i poželjno da se koriste one tehnike pisanja efikasnog koda, koje ne kvare dizajn.

Sa druge strane, upotreba tehnika koje kvare dizajn i otežavaju dalji razvoj i održavanje programa bi trebalo da se zaobilazi u širokom luku sve do dostizanja funkcionalnosti doftvera, zato što predstavlja vid preuranjenog optimizovanja.

Preuranjena ili preterana optimizacija

Preuranjena optimizacija je optimizacija preduzeta pre nego što je ustanovljeno šta je potrebno da se optimizuje. Kao što je Knut napisao, „preuranjena optimizacija je osnov svakog zla“. Ona se sukobljava sa principom agilnog razvoja „neće biti potrebno“. Ako taj princip agilnog razvoja prevedemo na domen optimizacije, onda dobijamo preporuku „optimizuj kasnije“. Osnovna ideja ove preporuke je da sve dok možemo da odlažemo optimizaciju, to znači da ona nije neophodna i da je zato dobro da je i dalje odlažemo.

Preterana optimizacija je optimizacija koja je obavljana dublje nego što je potrebno. Nastaje ako nismo postavili jasne ciljeve optimizacije i kriterijume performansi. Optimizovanje programskog koda je vrlo izazovan posao. Kada se u njega jednom uđe, onda može vrlo lako da proguta naše resurse kao neka programerska crna rupa. Veoma često se dešava da se nepotrebno gubi vreme na optimizacije koje nisu neophodne, umesto da se to vreme posveti nekom drugom, korisnijem problemu.

Preuranjena i preterana optimizacija su veoma česte greške u razvoju softvera. Nastaju kao posledica nepridržavanja principa savremene prakse optimizovanja softvera. Predstavljaju neposrednu posledicu propuštanja da se pažljivo lokalizuje predmet optimizovanja i odmere neophodne performanse, a posredno mogu da budu i posledica lošeg procenjivanja potrebnih performansi.

Posvećivanje više pažnje performansama nego korektnosti

Nekada efikasnost softvera koji pišemo spada u osnovne zahteve koji su nam postavljeni. U takvim slučajevima je posebno velika opasnost od preuranjene optimizacije. Ako trpimo pritisak da se staramo o efikasnosti (bilo u obliku pozitivne motivacije da ispunimo zahteve ili stalnog insistiranja nekog od naručilaca ili rukovodilaca), onda možemo da dođemo u iskušenje da već pri planiranju i implementaciji softvera posvetimo mnogo više pažnje performansama nego što je poželjno. Posledica takvog rada može da bude stavljanje ispravnosti softvera u drugi plan, a zatim i vrlo verovatno preduzimanje preuranjene ili preterane optimizacije.

To je nedopustivo. Koliko god da su nam važne performanse, ispravnost mora uvek da nam bude važnija. Čak i kada je program neefikasan, on će možda biti upotrebljiv za neke manje skupove podataka, ali ako je neispravan, onda je svaka njegova upotreba rizična.

Neispravno merenje performansi

Ako se merenje performansi softvera sprovodi u uslovima koji nisu identični uslovima njegove produkcione upotrebe, onda postoji opasnost da se merenjem dobiju neispravni rezultati, na osnovu kojih kasnije mogu da se donesu neke pogrešne odluke.

Jedna od čestih grešaka pri merenju performansi je merenje u izolovanom okruženju. Ako posmatramo izolovan deo koda (na primer jednu funkciju) i ustanovimo da je jedna implementacija efikasnija nego druga, to ne mora da važi kada se taj deo koda ugradi u veću celinu.

Na primer, neka imamo dve funkcije `f1` i `f2` koje su funkcionalno ekvivalentne, ali su različito implementirane. Ako merenje efikasnosti pokaže da je `f1` efikasnija od `f2`, da li to znači da je svakako bolje da koristimo `f1` u našem softveru? Ne. To može da nam ukaže da je vrlo verovatno da je bolje da koristimo `f1`, ali ne i da je to sigurno. Postoje mnoge situacije u kojima može da se pokaže da je efikasnije koristiti `f2`, na primer:

- ako `f2` može da se umetne a `f1` ne može;
- ako se pored `f1` i `f2` poziva i neka funkcija `g` koja se poziva iz `f2` a ne iz `f1`, pa njeno keširanje doprinosi efikasnosti;
- ako se iz `f1` i `f2` pozivaju dinamički vezani metodi, onda konkretne instance klasa koje se koriste mogu da značajno utiču na merenje;
- i drugo.

Druga česta greška je da se efekti neke optimizacije mere u odnosu na neku operaciju koja nema istu složenost kao operacija na koju se ta optimizacija na kraju zaista primenjuje. Na primer, ako merimo efikasnost razmotavanja neke petlje,

rezultati mogu drastično da se promene ako se neka od operacija u petlji zameni složenijom operacijom. Prva posledica je da efekat ubrzavanja može da se izgubi, zato što relativna cena ponavljanja (proveravanja uslova i grananja) postaje značajno manja nego što je cena tela petlje. Druga posledica je da čak može da dođe i do značajnog usporavanja, zato što je telo kratke petlje možda moglo da se dobro kešira, a nakon razmotavanja to više nije slučaj.

Merenje performansi je presudno za ocenu uspešnosti optimizacije. Već smo istakli da ne smemo da se oslanjamo na pretpostavke da je nešto efikasnije već da to mora da se proveri u praksi, a sada vidimo da je veoma lako i da se pogreši pri merenju performansi. Zato je neophodno da se merenje performansi preduzima u realnim uslovima, koji su praktično isti kao uslovi produkcione primene softvera. Svako drugačije merenje predstavlja samo procenu, a ne i pouzdanu meru efikasnosti.

Optimizovanje verzije za debagovanje

Optimizovanje verzije za debagovanje je specifičan slučaj neispravnog merenja performansi.

Kada se program prevede u režimu za debagovanje, onda prevodilac po pravilu ne primenjuje brojne interne optimizacije, već ostavlja prevod programa na mašinski kod u obliku koji je pregledniji i lakši za čitanje i analiziranje tokom debagovanja. Takav prevod po pravilu ima ugrađene i dodatne elemente koji olakšavaju praćenje toka izvršavanja programa ili menjanja njegovog stanja. Sa druge strane, kada se program prevede za objavljivanje, onda se iz prevoda sklanjaju svi viškovi i mašinski kod se dodatno optimizuje.

Uobičajeno je da se program preveden (i izgrađen) za debagovanje izvršava daleko sporije nego program koji je preveden (i izgrađen) za objavljivanje. Zato u slučaju optimizovanja verzije za debagovanje, a zbog oslanjanja na pogrešne rezultate merenja performansi, možemo da napravimo različite greške pri optimizovanju.

Zbog velike razlike u brzini različito prevedenih verzija programa (pa čak i u zauzeću radne memorije), može da se desi da ocenimo da program ne radi dovoljno efikasno, a da on zapravo radi odlično kada se prevede za produkciju. U takvim slučajevima se preduzima nepotrebno duboka optimizacija, zato što se insistira na nepotrebno visokim performansama verzije za debagovanje.

Drugi problem je što verzija za debagovanje i verzija za produkciju mogu da imaju potpuno različita uska grla. Ako procenjujemo usko grlo na osnovu verzije za debagovanje, onda možemo da odredimo pogrešan predmet optimizacije i da zatim nepotrebno gubimo vreme na optimizovanje pogrešno odabranog dela softvera.

13.6 Profajleri

Profajleri (engl. *profiler*) su programi koji prate i analiziraju upotrebu resursa tokom rada programa. Osnovna namena profajlera je da olakšaju uočavanje delova programa koji predstavljaju usko grlo u pogledu zauzeća nekog od resursa. Imaju nezamenljivu ulogu pri prepoznavanju predmeta optimizacije, pa predstavljaju jedan od najvažnijih alata koje koristimo pri optimizovanju softvera.

Prema vrsti resursa koji se prati, profajleri mogu da se podele na procesorske profajlere, memorijske profajlere i drugo. Pri optimizacijama niskog nivoa se najčešće koriste procesorski profajleri, pa se često pod nazivom profajler podrazumeva da se radi o procesorskom profajleru. I ovde ćemo se u daljem tekstu baviti prvenstveno procesorskim profajlerima. Pored njih se često koriste i memorisjki profajleri. Pri merenju performansi složenijih aplikacija koriste se profajleri koji mere zauzeće mreže, diskova i drugih resursa.

Procesorski profajleri prate zauzeće procesora i procesorskih resursa. Neki od osnovnih parametara koji se prate su brojanje koliko puta je izvršena pojedina funkcija ili metod, merenje vremena provedenog u funkciji ili metodu, merenje upotrebe keš memorije, brojanje promašaja stranice virtualne memorije i drugo. Prema tome da li mere vreme provedeno u funkcijama i metodima ili vreme provedeno u određenim linijama programskog koda, profajleri mogu da budu funkcijski ili linijski. Dobri savremeni profajleri obično omogućavaju da se prati i jedno i drugo.

Prema načinu na koji prikupljaju informacije o izvršavanju programa, procesorske profajlere delimo na (1) profajlere zasnovane na praćenju događaja, (2) profajlere sa ugradnjom brojača i (3) statističke profajlere.

Profajleri zasnovani na praćenjenju događaja (engl. *event-based*) prate i evidentiraju različite događaje o kojima ih obaveštava izvršno okruženje ili operativni sistem tokom izvršavanja programa. Na ovaj način mogu da se prate različiti sistemski događaji ili resursi, kao što su operacije sa kešom ili virtualnom memorijom, različiti hardverski događaji koje proizvode procesori, kao i specifični događaji koje proizvode izvršna okruženja. Na primer, profajleri *perf* i *OProfile* za *Linux* su delimično zasnovani na praćenju događaja o kojima ih izveštava jezgro operativnog sistema [*de Melo 2010*]. Ovakav način rada je uobičajen i za profajlere za interpretirane programske jezike ili virtualne mašine, kao što su *JVM* ili *CLR*.

Drugi način prikupljanja informacija je ugradnja brojača u sam programski kod (engl. *instrumentation-based*) [*Graham 1982*]. Na početku i na kraju svakog potprograma se ugrađuje programski kod koji obaveštava profajler o ulasku i izlasku iz potprograma. Ovakvi profajleri imaju visoku, čak apsolutnu tačnost brojanja događaja. Posledica je da se profajleru veoma često dostavlja veoma velika količina informacija, što zahteva dosta vremena, pa se pod ovakvim profajlerima programi

najčešće izvršavaju značajno sporije. Samim tim, merenje vremena može da izgubi na tačnosti, pa čak može da opadne i tačnost merenja relativnog vremena provedenog u određenim delovima koda.

Savremeni procesorski profajleri za kompilirane jezike najčešće rade pomoću povremenog uzimanja uzoraka (engl. *sampling-based*) [*Arnold 2000*]. Nazivaju se i statističkim profajlerima. Pomoću sistema prekida ili pomoću zamki procesora, rad procesora se prekida u otprilike jednakim vremenskim intervalima i profajler se obaveštava o tome koja se mašinska naredba trenutno izvršava i u kom kontekstu. Uobičajeno je da se program prekida između 100 i 10.000 puta u sekundi, zavisno od mogućnosti procesora i profajlera. Svaki put kada se izvršavanje programa zaustavi, beleži se vrednost brojača instrukcija, ali i stanje steka, da bi moglo da se ustanovi kako se došlo do tog mesta u programu. Zbog beleženja stanja steka, pojedinačna obrada je složenija nego u slučaju ugradnje u kod, ali se evidentira daleko manji broj događaja, pa ova vrsta profajlera nešto manje usporava izvršavanje programa. Posledica ovakvog rada je da brojanje upotrebe potprograma nije sasvim tačno, ali se ispostavlja da je relativno merenje vremena čak i nešto tačnije nego u slučaju ugradnje brojača.

Za interpretirane programske jezike, profajleri se obično prave kao dodatni moduli, koji se pri pokretanju programa povezuju sa izvršnim okruženjem i prate zauzeće resursa. Na primer, za interpretator za Pajton postoji više profajlerskih modula, među kojima su *cProfile* (funkcijski procesorski profajler zasnovan na praćenju događaja), *line_profiler*, (linijski procesorski profajler zasnovan na praćenju događaja), *statprof* (funkcijski statistički procesorski profajler) i drugi.

U slučaju programskih jezika koji se prevode, profajleri se obično prave kao posebni programi. Oni pri izvršavanju nemaju neposrednog dodira sa prevodiocem, osim u slučaju kada se koriste brojači ugrađeni u programski kod. U tom slučaju programi moraju da budu prevedeni na odgovarajući način i profajler mora da bude kompatibilan sa konkretnom tehnikom ugradnje brojača koju koristi prevodilac. Na primer, *GNU* profajler *gprof* je procesorski profajler zasnovan na ugrađenim brojačima [*Graham 1982*]), a profajler integrisan u vizualno razvojno okruženje *Visual Studio* je procesorski statistički profajler. Slični profajleri postoje i za neke interpretirane jezike, na primer *py-spy* ili *vprof* za Pajton.

Za profajliranje može da se koristi i *Valgrind*. Iako je izvorno namenjen za druge stvari, njegov modul *cachegrind* je nadgrađen modulom *callgrind*, koji omogućava praćenje pozivanja funkcija i može da se upotrebi za profajliranje. Program *kcachegrind* je koristan vizualni alat za analizu izveštaja koje prave moduli *cachegrind* i *callgrind*.

Primer optimizacije pomoću profajlera gprof

Upotrebu profajlera ćemo ilustrovati na jednostavnom primeru programa `primer.cpp`, koji popunjava matricu jedinicama i zatim je sumira:

```
#include <iostream>
using namespace std;

int constexpr len = 10000;
int niz[len][len];

void init()
{
    for( int i=0; i<len; i++ )
        for( int j=0; j<len; j++ )
            if( niz[i][j] != 1 )
                niz[i][j] = 1;
}

int sumline( int n )
{
    int sum = 0;
    for( int j=0; j<len; j++ )
        sum += niz[j][n];
    return sum;
}

int sum()
{
    int sum = 0;
    for( int i=0; i<len; i++ )
        sum += sumline(i);
    return sum;
}

int main()
{
    init();
    cout << sum() << endl;
    return 0;
}
```

Ovaj primer je sasvim jednostavan i trebalo bi da je očigledno gde su problemi. Za merenje performansi ćemo da upotrebimo profajler *gprof,* koji počiva na ugradnji brojača u programski kod. Pri prevođenju programa je potrebno je da se uključe odgovarajuće opcije, čime će se prevodiocu reći da u svaku funkciju ugradi odgovarajući kod za merenje broja prolazaka kroz funkciju i vremena provedenog u njoj. Postoji više opcija, ali osnovne su:

- `-pg`, u prevod se automatski ugrađuju brojači za profjaliranje pomoću profajlera *gprof*;
- `-g`, prevod uključuje i informacije za debagovanje, što može da bude potrebno ako se performanse mere na nivou linija koda, a ne samo na nivou potprograma.

Ako se samo neki moduli prevode sa opcijama za profajliranje, evidentiranje informacija će i dalje raditi, ali samo za odgovarajuće delove programa, tj. neće biti na raspolaganju informacije za ceo program. Iste opcije moraju da se koriste i pri povezivanju.

Naš primer ćemo da prevedemo sa opisanim opcijama za profajliranje i paketom opcija za optimizaciju `-O3`. Dodatnom opcijom `-fno-inline-small-functions` ćemo zahtevati da se jednostavne funkcije ne umeću na mestu pozivanja, zato što bi inače ceo naš program stao u jednu funkciju:

```
g++ -g -pg -O3 -fno-inline-small-functions primer.cpp -o primer
```

Sledeći korak je pokretanje programa. Program se pokreće i izvršava na sličan način kao da je preveden bez dodatnih opcija, ali će zbog uključenih opcija za profajliranje napraviti datoteku `gmon.out`, u kojoj će zapisati prikupljene podatke o toku izvršavanja programa:

```
$ ./primer
100000000
```

Sada nam preostaje da pokrenemo program `gprof`. Kao parametar se navodi program koji se profajlira, da bi iz njega mogli da se prikupe podaci o programskom kodu. Opciono se navodi i naziv datoteke sa podacima o izvršavanju, a ako se ne navede onda se podrazumeva da se koristi `gmon.out`. Navešćemo i dodatni parametar `-b`, da ne bi bila ispisivana legenda o podacima. Kao rezultat ćemo dobiti sledeći izveštaj:

```
$ gprof -b ./primer
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 87.07      1.28     1.28    10000     0.00     0.00  sumline(int)
 12.93      1.47     0.19        1     0.19     0.19  init()
  0.00      1.47     0.00        1     0.00     1.28  sum()
```

```
            Call graph

granularity: each sample hit covers 4 byte(s) for 0.68% of 1.47
seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00    1.47                 main [1]
                0.00    1.28       1/1           sum() [3]
                0.19    0.00       1/1           init() [4]
-----------------------------------------------
                1.28    0.00   10000/10000       sum() [3]
[2]     87.1    1.28    0.00   10000         sumline(int) [2]
-----------------------------------------------
                0.00    1.28       1/1           main [1]
[3]     87.1    0.00    1.28       1         sum() [3]
                1.28    0.00   10000/10000       sumline(int) [2]
-----------------------------------------------
                0.19    0.00       1/1           main [1]
[4]     12.9    0.19    0.00       1         init() [4]
-----------------------------------------------

Index by function name

   [3] sum()              [4] init()             [2] sumline(int)
```

Prvi deo izveštaja obuhvata podatke o funkcijama čije je izvršavanje zahtevalo najviše vremena. Vidimo da je ukupno trajanje izvršavanja programa bilo 1,47*s* i da je najviše vremena (1,28*s*) utrošeno u funkciji `sumline`, koja je pozvana 10.000 puta.

U drugom delu izveštaja se vidi koja je funkcija odakle pozivana i koliko je vremena utrošeno u kom njenom delu. Svaka sekcija opisuje po jednu funkciju:

- U prvoj sekciji se vidi da je funkcija `main` pozvana od strane sistema („*<spontaneous>*“) i da je ona utrošila 100% vremena, tj. 1,47*s*. Međutim, vidimo i da je od toga 1,28*s* utrošeno pri pozivanju funkcije `sum`, a još 0,19*s* pri pozivanju funkcije `init`;
- Funkcija `sumline` je pozvana ukupno 10.000 puta i to svaki put iz funkcije `sum`. Pri tom je utrošila 1,28*s*, što je 87% ukupnog vremena i sve je to utrošeno u njenom telu, tj. nije pozivala druge funkcije;
- Funkcija `sum` je pozvana jednaput iz funkcije `main` i potrošila je 1,28*s*, ali vidimo da je sve to utrošeno prilikom 10.000 pozivanja funkcije `sumline`;
- Funkcija `init` je pozvana jedanput iz funkcije `main` i potrošila je 0,19*s* ili 12,9% vremena.

Treći deo izveštaja služi kao indeks, za lakše snalaženje u drugom delu izveštaja. Veoma je koristan u slučaju većih programa sa velikim brojem funkcija, zato što tada drugi deo izveštaja može da bude prilično obiman.

Analizom dobijenih podataka vidimo da je najviše vremena uzelo sumiranje, što je i očekivano. Znači, predmet eventualne optimizacije bi trebalo da bude, pre svega, funkcija `sumline`. Lako se uočava da je uzrok neefikasnosti ove funkcije u tome što ne poštuje princip lokalnosti. Znači, možemo da probamo da optimizujemo program tako što bismo elemente sumirali red po red, a ne vrstu po vrstu:

```
int sumline( int n )
{
    int sum = 0;
    for( int j=0; j<len; j++ )
        sum += niz[n][j];
    return sum;
}
```

Kada ponovimo prevođenje, izvršavanje i pokretanje profajlera, dobićemo izmenjene podatke:

```
Flat profile:
...
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 86.96      0.20     0.20        1   200.00   200.00  init()
 13.04      0.23     0.03    10000     0.00     0.00  sumline(int)
  0.00      0.23     0.00        1     0.00    30.00  sum()

            Call graph
...
-----------------------------------------------
                0.20    0.00       1/1           main [1]
[2]     87.0    0.20    0.00       1         init() [2]
-----------------------------------------------
...
```

Vidimo da je sada funkcija `sumline` daleko efikasnija i da troši samo 0,03*s* umesto 1,28*s*, što je oko 42 puta manje vremena. Ukupno izvršavanje programa je sada svedeno na 0,23*s* umesto početnih 1,47*s*, pa je program ubrzan za oko 82%.

Sada je najneefikasnija funkcija `init`, na koju odlazi skoro 7 puta više vremena nego na sumiranje. Potencijalna optimizacija je da iz nje izbacimo proveru uslova, tako da se vrednosti matrice bezuslovno postavljaju na 1:

```
void init()
{
    for( int i=0; i<len; i++ )
        for( int j=0; j<len; j++ )
```

```
            niz[i][j] = 1;
}
```

Kada ponovimo prevođenje, izvršavanje i pokretanje profajlera, dobijamo sledeće podatke:

```
Flat profile:
...
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 75.00      0.09     0.09        1    90.00    90.00  init()
 25.00      0.12     0.03    10000     0.00     0.00  sumline(int)
  0.00      0.12     0.00        1     0.00    30.00  sum()
...
```

Sada je funkcija `init` potrošila svega 0,09*s*, što je 55% efikasnije nego ranije. Izvršavanje programa je trajalo ukupno 0,12*s*, što je 48% manje nego ranije, a oko 92% manje nego na početku.

13.7 Umesto zaključka

Jedan od osnovnih razloga što su računari postali sveprisutni je njihova visoka efikasnost. Mi koristimo računare i pišemo programe za njih baš zbog toga što su efikasni. To onda ima za posledicu da pri pisanju programa prirodno težimo da tu efikasnost ne dovedemo u pitanje svojom lošom implementacijom, već da nasuprot tome, pokušamo da je još više istaknemo i iskoristimo. Kada god primetimo da imamo problem sa performansama, pokušavaćemo da taj problem prevaziđemo, nekada ispravljanjem eventualno napravljenih grešaka, a nekada traženjem efikasnijih načina da se uradi neki posao. To je ono što nazivamo optimizacijom softvera.

Kada se početnici prvi put susretnu sa nekom optimizacijom, ona može da im izgleda kao magija – neko dođe i promeni nešto u kodu, čak na način koji izgleda besmisleno ili trivijalno, a program posle toga počne da radi mnogo brže!? Naravno, nema tu nikakve magije. U pitanju je vrlo egzaktna oblast, koja zahteva iscrpno poznavanje alata koje koje koristimo. Taj „magijski“ doživljaj nastaje kao posledica nerazumevanja nekog od aspekata rada programa, programskog jezika ili čitavog računarskog sistema.

Svima koje ova oblast zanima preporučujemo da ne idu preko reda, već da prvo nauče dobro i lepo dizajniranje softvera, zatim debagovanje, pa da dodatno upoznaju konkretne alate koje koriste i da se tek onda posvete tehnikama optimizacije.

Ima mnogo izvora iz kojih se mogu izučavati principi i tehnike optimizacije. Za upoznavanje sa opštim principima optimizovanja programa u kontekstu dobrog dizajniranja i agilnog razvoja preporučujemo članak Kena Aura i Kenta Beka [*Auer*

1996]. Za temeljnije upoznavanje sa tehnikama optimizacije na programskom jeziku *C++* predlažemo [*Guntheroth 2016*] ili [*Fog 2022*]. Knjiga [*Gerber 2006*] se bavi karakteristikama arhitekture Intelove familije procesora *IA*-32 i specifičnim tehnikama optimizacije. Iako je u osnovi posvećena arhitekturi *IA*-32, najveći deo sadržaja te knjige može da se primeni i na druge savremene procesore.

Literatura

Abouzahra, Mohamed. 2011. **Causes of failure in Healthcare IT projects**. *International Conference on Advanced Management Science*, Singapore

Abrahams, David, Aleksey Gurtovoy. 2004. **C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond**. *Addison-Wesley Professional*.

Agile Alliance. `https://www.agilealliance.org/`

Agile Manifesto. `https://agilemanifesto.org/`

Agans, David J. 2006. **Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems**. *Amacom*.

Al-Ahmad, W., K.Al-Fagih, K.Khanfar, K.Alsamara, S.Abuleil, and H.Abu-Salem. 2009. **A taxonomy of an IT project failure: root causes.** *International Management Review*, 5(1), 93-104.

Alexander, Christopher, Sara Ishikawa and Murray Silverstein. 1977. **A Pattern Language: Towns, Buildings, Construction**. *Oxford University Press*.

Alexandrescu, Andrei. 2010. **The D Programming Language.** *Addison-Wesley Professional*.

Alexandrescu, Andrei. 2001. **Modern C++ Design: Generic Programming and Design Patterns Applied**. *Addison-Wesley Professional*.

Ambler, Scott, Mark Lines. 2012. **Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise**. *IBM Press*.

Arnold, Matthew, Peter Sweeney. 2000. **Approximating the Calling Context Tree Via Sampling**. *Research report, IBM Watson Research Center*.

Auer, Ken, Kent Beck. 1996. **Lazy Optimization: Patterns for Efficient Smalltalk Programming**. U John Vlissides, James Coplien, Norman Kerth, **Pattern Languages of Program Design 2**. *Addison-Wesley Professional*.

Avison, David, Guy Fitzgerald. 2003. **Information Systems Development.** 3.ed, *McGraw Hill*.

Back, Kent. 1999. **Embracing Change With Extreme Programming**. *IEEE Computer*, 32(10).

Back, Kent. 2002. **Test Driven Development**. *Addison-Wesley Professional*.

Back, Kent. 2004. **Extreme Programming Explained.** 2.ed. *Addison-Wesley*.

Bernardy, Jean-Philippe, P.J.Jansson, M.Zalewski, S.Schupp, A.Priesnitz. 2008. **A comparison of C++ concepts and Haskell type classes**. *WGP '08: Proc. ACM SIGPLAN Workshop on Generic programming, 37–48*.

Boehm, Barry W. 1991. **Software Risk Management: Principles and Practices**. *IEEE Software* 8(1).

Booch, Grady. 1990. **Object oriented design with applications**. 1.ed. *Benjamin/Cummings*.

Booch, Grady, James Rumbaugh, Ivar Jacobson. 1999. **The Unified Modeling Language User Guide.** *Addison Wesley*. Izdanje na srpskom jeziku: **UML – Vodič za korisnike**, *CET Computer Equipment and Trade*.

Booch, Grady, James Rumbaugh, Ivar Jacobson. 2005. **The Unified Modeling Language User Guide.** 2nd ed. *Addison Wesley*.

Charette, Robert N. 2005. **Why Software Fails**. *IEEE Spectrum*.

Card, S.K., G.G. Robertson, J.D. Mackinlay. 1991. **The information visualizer: An information workspace**. *Proc. ACM CHI'91 Conf.*, 181-188.

Cardelli, Luca, Peter Wegner. 1985. **On Understanding Types, Data Abstraction, and Polymorphism**. *ACM Computing Surveys,* 17(4), 471-523.

Catch2. `https://github.com/catchorg/Catch2`

Church, Alonzo. 1936. **An unsolvable problem of elementary number theory**. *American Journal of Mathematics*, 58, 354-363.

Coad, Peter, Edward Yourdon. 1991. **Object Oriented Analysis.** 2.ed. *Prentice Hall*.

Cockburn, Alistair, Laurie Williams. 2001. **The Costs and Benefits of Pair Programming**. U **Extreme Programming Examined**. *Addison-Wesley*.

Cohn, Michael. 2009. **Succeding with Agile**. *Addison Wesley*.

Cook, William R. 2009. **On Understanding Data Abstraction, Revisited**. *Proceedings of the 24th ACM SIGPLAN conference on OOPSLA*, 557-572.

Cummins, Fred A. 2008. **Building the Agile Enterprise**. *Morgan Kaufmann*.

Čukić, Ivan. 2018. **Functional Programming in C++**. *Manning Publications*.

D – Programming Language D. veb. `https://dlang.org`

Date, C.J., Hugh Darwen. 1995. **The Third Manifesto**. *SIGMOD Records* 24.

Date, C.J., Hugh Darwen. 2006. **Databases, Types and the Relational Model**. 3.ed. *Addison Wesley*.

Dijkstra, Edsger W. 1976. **A Discipline of Programming**. *Prentice-Hall*.

Doxygen. `https://www.doxygen.nl/index.html`

Fenton, Norman, James Bieman. 2014. **Software Metrics: A Rigorous and Practical Approach.** 3.ed. *CRC Press*.

Fog, Agner. (2004-)2022. **Optimizing software in C++ - An optimization guide for Windows, Linux and Mac platforms**. Technical University of Denmark. `https://www.agner.org/optimize/optimizing_cpp.pdf`

Fortune, Peters. 2005. **Information Systems - Achieving Success by Avoiding Failure**. *John Wiley & Sons*.

Fowler, Martin. 1999. **Refactoring: Improving the Design of Existing Code**. *Addison-Wesley*. Izdanje na srpskom jeziku: **Refaktorisanje: Poboljšanje dizajna postojećeg koda**, *CET Computer Equipment and Trade*.

Fowler, Martin. 2002. **Patterns of Enterprise Application Architecture**. *Addison-Wesley*.

Fowler, Martin. 2003a. **UML Distilled.** 3.ed. *Addison-Wesley*. Izdanje na srpskom jeziku: **UML ukratko**, *Mikro knjiga*.

Fowler, Martin. 2003b. **Who Needs an Architect?** *IEEE Software, 20/5*.

Fowler, Martin. veb. **Refactoring.Com**. `https://refactoring.com/`

Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides. 1995. **Design Patterns: Elements of Reusable Object-Oriented Software.** *Addison-Wesley*. Izdanje na srpskom jeziku: **Gotova rešenja: Elementi objektno-orijentisanog softvera**, *CET Computer Equipment and Trade*, 2002.

Gerber, Richard, Aart Bik, Kevin Smith, Xinmin Tian. 2006. **The Software Optimization Cookbook**. 2.ed. *Intel Press*.

Graham, Susan, Peter Kessler, Marshall McKusick. 1982. **gprof: A call graph execution profiler**. *Proc. ACM SIGPLAN Symposium on Compiler Construction, ACM Press.*

Grotker, Thorsten, Ulrich Holtmann, Holger Keding, Markus Wloka. 2008. **The Developers Guide to Debugging.** *Springer*.

Guntheroth, Kurt. 2016. **Optimized C++.** *O'Reilly Media.*

Huston, Vince. veb. `http://www.vincehuston.org/dp/`

IEEE. 2017. ISO/IEC/IEEE 24765:2017(E), **ISO/IEC/IEEE International Standard: Systems and Software Engineering – Vocabulary**. *IEEE*.

Knaster, Richard, Dean Leffingwell. 2020. **SAFe 5.0 Distilled: Achieving Business Agility with the Scaled Agile Framework**. *Addison-Wesley Pro*.

Kohnfelder, Loren. 2021. **Designing Secure Software: A Guide for Developers**. *No Starch Press*.

Kruchten, Philippe. 2003. **The Rational Unified Process: An Introduction**. *Addison-Wesley*.

Larman, Craig. 2004. **Applying UML and Patterns.** 3.ed. *Pearson*.

Larman, Craig, Bas Vodde. 2016. **Large-Scale Scrum: More with LeSS**. *Addison-Wesley*.

Lipovača, Miran. 2011. **Learn You a Haskell for a Great Good!** *No Starch Press.*

Llopis, Noel. 2010. **Exploring the C++ Unit Testing Framework Jungle**. `https://gamesfromwithin.com/exploring-the-c-unit-testing-framework-jungle`

Lowy, Juval. 2019. **Righting Software**. *Addison-Wesley Professional*.

Malkov, Saša. 2007. **Objektno-orijentisano programiranje: C++ kroz primere**. *Matematički fakultet*.

Malkov, Saša. 2010. **Customizing a Functional Programming Language for Web Development**. *Computer Languages, Systems & Structures*, 36(4):345-351.

Martin, James, James Odell. 1992. **Object Oriented Analysis and Design**. *Prentice Hall.*

Martin, Robert C. 2003. **Agile Software Development: Principles, Patterns, and Practices**. *Prentice Hall.*

de Melo, Arnaldo Carvalho. 2010. **The New Linux 'perf' tools**. *Linux Kongress*.

Metzger, Charles. 2004. **Debugging By Thinking: A Multidisciplinary Approach**. *Digital Press*.

Miller, Robert B. 1968. **Response time in man-computer conversational transactions**. *Proc. AFIPS Fall Joint Computer Conference Vol.* 33, 267-277.

Nielsen, Jakob. 1993. **Usability Engineering**. *Academic Press*.

OMG – Object Management Group. `http://www.omg.org`

Pereira, Rui. 2017. **Energy efficiency across programming languages: how do energy, time, and memory relate?** *Proc. 10th ACM SIGPLAN Int. Conf. on Software Language Engineering, SLE 2017, 256-267*.

Pfleeger, Shari Lawrence, Joanne M. Atlee. 2006. **Software Engineering – Theory and Practice**. 3.ed. *Pearson Education.* Izdanje na srpskom jeziku: **Softversko inženjerstvo – Teorija i praksa**, *CET Computer Equipment and Trade.*

Popović, Jovan. 2019. O**snove softverskog inženjerstva**. *CET Computer Equipment and Trade.*

Pressman, Roger, Bruce Maxim. 2020. **Software Engineering: A Practitioner's Approach, 9.ed**. *McGraw Hill.*

Qt – Cross-Platform Software Development Framework. `https://doc.qt.io/`

Reeves, Jack. 1992. **What is Software Design?** *C++ Journal.* `http://user.it.uu.se/~carle/softcraft/notes/Reeve_SourceCodeIsTheDesign.pdf`

Rodriguez, Daniel, Miguel Ángel Sicilia, Elena García-Barriocanal, Rachel Harrison. 2012. **Empirical findings on team size and productivity in software development.** *Journal of Systems and Software, 85/3, 562-570*.

Rubin, Kenneth. 2013. **Essential Scrum**. *Addison-Wesley*.

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen. 1992. **Object-Oriented Modeling and Design**. *Prentice-Hall*.

Schiel, James. 2009. **Enterprise-Scale Agile Software Development**. *CRC Press*.

Schwaber, Ken, Jeff Sutherland. 2020. **The Scrum Guide – The Definitive Guide to Scrum: The Rules of the Game**. `https://scrumguides.org`

Seacord, Robert. 2013. **Secure Coding in C and C++**. *Pearson Education*.

Sommerville, Ian. 2016. **Software Engeneering.** *Pearson Education Ltd.*

Standish Group. 1994. **The Chaos Report 1994**.

Standish Group. 2015. **The Chaos Report 2015**.

State of Agile. 2020. **14th Annual State of Agile Report.** `https://stateofagile.com/`

State of Agile. 2022. **16th Annual State of Agile Report.** `https://stateofagile.com/`

Stepanov, Alexander, Meng Lee. 1994. **The Standard Template Library**. *HP Laboratories Technical Report 94-34(R.1)*.

UML – Unified Modeling Language. `http://www.uml.org`

Wake, William. 2001. **Extreme Programming Explored**. *Addison-Wesley*.

Wang, Yingxu. 2007. **Software Engineering Foundations: A Software Science Perspective**. *Auerbach Publications*.

Wirfs-Brock, Rebecca, Alan McKean. 2003. **Object Design: Roles, Responsibilities, and Collaborations**. *Addison-Wesley Professional*.

Zeller, Andreas. 2006. **Why Programs Fail – A Guide to Systematic Debugging**. *Morgan Kaufmann Publishers*.

Živković, Miodrag. 2000. **Algoritmi**. *Matematički fakultet*.

Indeks

CIP - Каталогизација у публикацији Народна библиотека Србије, Београд

004.41(075.8)

МАЛКОВ, Саша, 1970-
Razvoj softvera : [univerzitetski udžbenik] / Saša Malkov. - 1. izd. - Beograd : Univerzitet, Matematički fakultet, 2024 (Grocka : Donat Graf). - XXIV, 481 str. : ilustr. ; 25 cm

Tiraž 500. - Bibliografija: str. 469-473. - Registar.

ISBN 978-86-7589-196-3

а) Софтвер -- Пројектовање

COBISS.SR-ID 158268681