8 - Agilni razvoj softvera

Saša Malkov

Odlomak iz knjige Razvoj softvera

(u pripremi)

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*) [AA] 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 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 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 odovorimo na pitanje: *Šta nije agilni razvoj softvera*?

Agilni razvoj softvera nije:

- odsustvo sistematičnost i strukturnog pristupa;
- anarhično ili svojevoljno ponašanje članova tima;
- potpuno odsustvo dokumentacije;
- selektivna primena samo nekih od principa agilnog razvoja softvera;
- 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 površne, iako 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 na zanimljiv način objavljivanjem Manifesta agilnog razvoja softvera [AM]:

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
Reagovanje na promene – od pridržavanja plana

Odnosno, iako cenimo vrednosti na desnoj strani,
smatramo za vrednije one koje su na levoj.

Veoma je važno da se ne previdi napomena na kraju manifesta – tradicionalne vrednosti iz starijih metodologija razvoja softvera se *ne odbacuju*, već opstaju i dalje,

ali se prepoznaju neke druge vrednosti kao još značajnije. Agilne metodologije su izgrađene na svim tim vrednostima, uz stavljanje akcenta na *one leve*.

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, teško da posao može da bude uspešan. Jednako je teško doći do cilja i ako su u timu sposobni pojedinci, ako 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 alatima, to može biti još gore od potpunog odsustva alata, zato što može da stavi ljude u drugi plan. Izgradnja tima mora uvek da bude važnija od izgradnje okruženja.

Pretpostavka da je *softver koji radi* važniji od *iscrpne dokumentacije* podseća da je cilj razvoja softver, a ne dokumentacija. Mnogo je projekata imalo izvrsnu dokumentaciju, a da softver nije radio – takva dokumentacija na kraju priče nije imala nikakvu vrednost. Bila bi velika greška ako bi neko ovu pretpostavku razumeo kao 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 isključivo 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 vreme njenog pisanja. 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 obim dokumentacije, zato što se obimna dokumentacija veoma teško sinhronizuje sa naknadnim izmenama programa.

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 softverska industrija razvija, složenost problema koji se rešavaju raste mnogo brže. Posledica je da je u mnogim slučajevima veoma teško da se pre započinjanja razvoja naprave tačne procene obima poslova, troškova i rokova isporuke. Treća pretpostavka agilnog razvoja softvera ističe da je *saradnja sa klijentom* mnogo važnija od *pregovaranja oko ugovora*.

Naravno, pregovaranje i ugovaranje je neophodno – to niko ne dovodi u pitanje. Međutim, kod svih agilnih metodologija se teži da se pri sklapanju posla ugovorom prvenstveno 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 one naredne sagledavaju samo okvirno.

Agilni razvoj softvera prihvata da je i poslovno okruženje klijenta vrlo verovatno agilno, što za posledicu ima veliku verovatnoću da se neki već iskazani stavovi klijenta promene tokom razvoja softvera. 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 zbog nekih promena moraju da se menjaju planovi, agilne metodologije nam kažu da to neodložno treba 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.

8.3 Principi agilnog razvoja softvera

Agilni razvoj softvera počiva na dvanaest principa, koji su posledica pretpostavki izrečenih u manifestu. Taj skup principa je zajednički za sve agilne metodologije. 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.

Principi agilnog razvoja softvera su:

- Najviši prioritet je zadovoljiti klijenta kroz brzo i neprekidno isporučivanje vrednog softvera.
- Uvek otvoreno prihvatati promene, čak i u kasnim fazama razvoja. Agilni razvoj softvera uvažava promene kao sredstvo postizanja kvaliteta za klijenta.
- Isporučivati softver koji radi, što češće, sa intervalom od par nedelja do par meseci, pri čemu se kraći intervali više cene.
- Poslovni ljudi i razvijaoci moraju svakodnevno da sarađuju na projektu.

- Zasnivati projekat na motivisanim pojedincima. Pružiti im okruženje i potrebnu podršku i imati poverenja u njih da će obaviti posao.
- Najefikasniji način za prenošenje informacija timu i unutar tima je razgovor licem u lice.
- Softver koji radi je osnovno merilo napretka.
- Agilni razvoj softvera promoviše uzdržani razvoj. Sponzori, razvijaoci i korisnici bi trebalo da budu u stanju da neograničeno dugo održavaju ujednačen ritam.
- Neprekidno posvećivanje pažnje tehničkoj doteranosti i dobrom dizajnu podiže agilnost.
- Jednostavnost, kao umetnost maksimizovanja količine posla koji se ne obavlja, je od suštinskog značaja.
- Najbolje arhitekture, zahtevi i projekti potiču iz samoorganizovanih timova.
- U redovnim intervalima tim mora da sagledava svoj rad i mogućnosti unapređivanja svog ponašanja i efikasnosti.

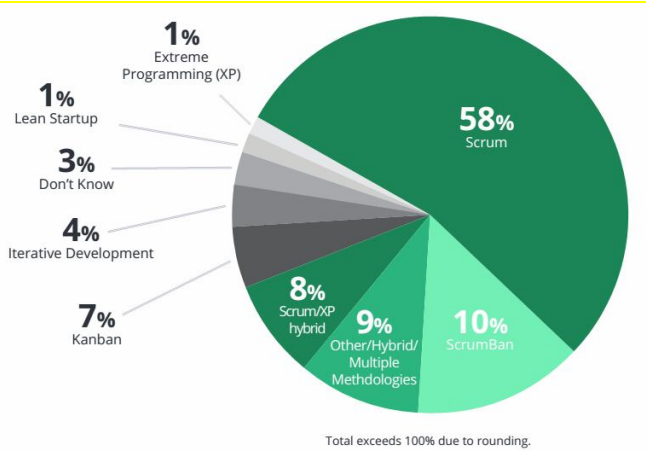
Navedene principe nećemo detaljnije objašnjavati na ovom mestu. Umesto toga ćemo nešto više pažnje posvetiti konkretnim tehnikama ekstremnog programiranja, koje se primenjuju radi ostvarivanja navedenih principa. Tada ćemo razmotriti i motivaciju za konkretne tehnike i principe koji stoje iza njih.

8.4 Primeri agilnih metodologija

Agilni razvoj softvera je osnova za više različitih metodologija. Mnoge od njih su prisutne u razvojnim timovima širom sveta. Neke od najpoznatijih su:

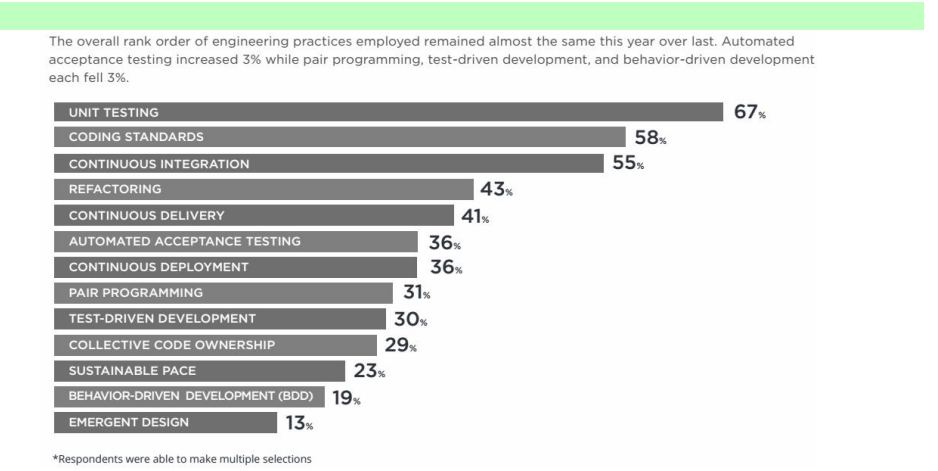
- Ekstremno programiranje (engl. *XP – Extreme Programming*);
- Skram (engl. *Scrum*);
- Agilno modeliranje;
- Agilan objedinjen proces (engl. *AUP – Agile Unified Process*) i
- Otvoren objedinjen proces (engl. *OpenUP – Open Unified Process*).

Prostor nam ne dopušta da opisujemo više agilnih metodologija, niti da im se temeljno posvetimo, pa ćemo u nastavku ovog poglavlja da se ograničimo na upoznavanje osnovnih elemenata i karakteristika metodologija *Ekstremno programiranje* i *Skram*.



Slika 23 – Pregled zastupljenosti agilnih metodologija [...REF...]

Nakon toga ćemo upoznati i Skram, kao agilnu metodologiju koja je, po mnogim analizama, najzastupljenija u savremenoj praksi. Za razliku od Esktremnog programiranja, koje predstavlja relativno kompletnu metodologiju, Skram bi pre mogao da se nazove metodološkim okvirom, koji je otvoren za primene tehnika iz drugih metodologija. Takva struktura Skrama dodatno doprinosi značaju prethodnog upoznavanja Ekstremnog programiranja.



Slika 24 – Pregled zastupljenosti agilnih praksi [...REF...]

8.5 Ekstremno programiranje

Ekstremno programiranje je predstavio Kent Bek 1999. godine u nekoliko članaka i izlaganja na konferencijama [Back1999]. Motivacija i ciljevi se praktično poklapaju sa onim što je kasnije formulisano kao Manifest agilnog razvoja.

Priroda metodologije se najneposrednije sagledava razmatranjem osnovnih pojmova sa kojima se srećemo pri razmatranju razvojnog ciklusa Ekstremnog programiranja. Ovi osnovni pojmovi se nazivaju i strukturnim elementima Ekstremnog programiranja i predstavićemo ih narednom odeljku.

Kao jedan od navažnijih elemenata metodologije se izdvajaju tzv. *prakse* Ekstremnog programiranja. One predstavljaju tehnike, metode i principe ove metodologije. Zbog značaja i obima, opisaćemo ih u posebnom odeljku, nakon predstavljanja strukturnih elemenata.

8.5.1 Strukturni elementi Ekstremnog programiranja

Struktura metodologije se može opisati kroz njene osnovne elemente, koji se tiču načina organizovanja posla i uređivanja razvojnog ciklusa. Među osnovne strukturne elemente metodologije se obično ubrajaju:

- Razvojni ciklus;
- Korisničke celine;
- Izdanja;
- Iteracije;
- Zadaci i
- Testovi.

Razvojni ciklus

Jedan od osnovnih doprinosa Ekstremnog programiranja (EP) je drastično skraćivanje razvojnog ciklusa. Tu nema značajnog odstupanja od opšteg pristupa koji imaju i druge agilne metodologije. Specifičnost je podela na dve osnovne vrste ciklusa: 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 iteracije, a po njegovom završetku se započinje novo izdanje ili se projekat završava.

O planiranju korisničkih celina, izdanjima i itracijama će biti više reči u narednim odeljcima, kao i u okviru opisa prakse Igra planiranja.

Korisničke celine

Korisničke celine (engl. *user stories* ili često samo *stories*) imaju sličnu ulogu kao *slučajevi upotrebe* u OO metodologijama, ali se u mnogo čemu od njih razlikuju.

Korisničke celine predstavljaju okvirnu specifikaciju zahteva. Piše ih klijent, pa obično nemaju tehničku preciznost kao slučajevi upotrebe. Naprotiv, često su opisi 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, korisnička celina mora da bude jasna i proverljiva – tj. mora da bude razumljiv i nedvosmislen poslovni cilj koji bi korisničkom celinom trebalo da se ostvari, kao i kriterijumi kojima će moći da se ustanovi da li je ona ostvarena 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 jer 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, skup izabranih korisničkih celina se obično bira taka da predstavlja jednu veću zaokruženu celinu koja može klijentu da pruži veoma jasnu i prepoznatljivu korist. Kada je kod moguće, teži se da izdanje bude ne samo jedinca 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 šta će od izabranih celina da ide u koju iteraciju, ali se detaljno planira samo prva iteracija.

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 u slučaju izdanja, ali često razvojnog tima ima nešto veći uticaj na 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 razvojem, 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 do 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. Rezultat tog sastanka bi trebalo da bude početni kratak ali sadržajan spisak testova koji bi morali da rade da bi se zadatak smatrao završenim (prakse Testovi prihvatljivosti i Razvoj vođen testovima). Nakon podele korisničke celine na zadake može da se desi da neki zadaci imaju strogo tehnički karakter i nemaju neposrednog dodira sa klijentom. U takvom slučaju nema potrebe da sastanku prisustvuje klijent, ali se umesto njega obično uključuje vođa tima.

Cilj rada na jednom zadaku je ispunjavanje definisanih testova. Tokom rada se obično prave i proveravaju i novi detaljniji testovi (praksa Razvoj vođen testovima).

Testovi

Pored specifičnog oblika razvonog ciklusa moglo bi se reći da su testovi ključni element Ekstremnog programiranja. Testiranje softvera je bilo uvek prisutno, 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.

U kontekstu priče o testiranju svi testovi se obično dele na one koje pravi (ako ne tehnički a onda bar konceptualno) klijent i koji služe da se proveri da li je završen proizvod (ili deo proizvoda) u skladu sa očekivanjima. Takve testove obično nazivamo testovima prihvatljivosti i oni služe da provere ispravnost i kvalitet proizvoda ili njrgove komponente kao složene celine. Razmotrićemo ih kroz praksu Testovi prihvatljivosti.

Drugu grupu testova čine testovi koje prave programeri i čiji je cilj da testiraju pojedinačne strukturne elemente softvera. Na najnižem nivou imamo testove jedinaca 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 kroz praksu 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 praktično ostvaruju principi agilnog razvoja. Može se reći da prakse EP predstavljaju *sredstvo* ostvarivanja principa agilnog razvoja. Većina praksi nije potpuno nova, ni osmišljena 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 onoga 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 strukturne elemente metodologije (neke ili sve) svrstavaju u prakse, dok se neke prakse kod nekih autora spuštaju na nivo preporuke.

Prakse se obično klasifikuju u tri grupe prema aspektu razvojnog procesa na koji se pretežno odnose, pri čemu ima i preklapanja. Na primer, Vejk [Wake2000] predlaže sledeći spisak klasifikovanih praksi (sa ponavljanjem nekih praksi):

- 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 cikljusi;
- Programerske prakse:
  - Jednostavan dizajn;
  - Testiranje;
  - Refaktorisanje i
  - Standardi kodiranja.

Unastavku 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[22].

[22] 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, odlučeno je se predstavi u vidu prakse.

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 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 uključenje 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, onako kako nastaju, a time i da se informacije sagledavaju samo parcijalno, a ne u dovoljno širokom obimu. Drugi potencijalno problematičan aspekt je možda preterano olakšavanje klijentu da saopštava želje za izmenama, što može da proizvede neumereno veliki broj zahteva za izmenama, tako da nastupi lavina izmena koja može da uspori, pa čak i zaustavi razvoj. Jedan od načina sprečavanja ovog problema je, na primer, da se u završnom delu perioda razvoja (na primer, tokom poslednjih nekoliko dana ili poslednje 2-3 nedelje, a zavisno od dužine 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 i 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 bi trebalo da se ponavljaju svaki put pri izgradnji sistema (i

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 kroz nekoliko iteracija isporučene verzije isprobavaju samo 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.

U metodologiji Ekstremnog programiranja je jasno istaknuta podela odgovornosti između klijenta i razvojnog tima u procesu planiranja. Klijenti odlučuju o značajnosti različitih karakteristika razvijanog softvera. Oni određuju šta je potrebno, 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, 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 (u smislu broja radnih sati) okvira iteracije. Okvire iteracije određuje razvojni tim, pri čemu na termin završetka eventualno može da utiče neka značajna okolnost poslovanja klijenta. Na taj način razvijaoci praktično saopštavaju klijentu koliko posla mogu da urade u narednom koraku. Zatim klijent bira celine koje će se implementirati u okviru planirane iteracije. 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.

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* 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.

Izdanje se planira 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 slične *poslovnim slučajevima* (engl. *business case*), a poslovi (zadaci) na koje se one dele su slični *slučajevima upotrebe* (engl. *use case*) ili njihovim delovima. U skladu sa time mogu da se koriste i slične tehnike opisivanja, kao što su *UML* dijagrami aktivnosti, stanja i 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* 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 se pratiti i meriti i drugi elementi projekta ili njihove karakteristike, kao na primer:

- broj programskih datoteka;
- broj klasa;
- broj linija koda;
- broj testova
- i druge.

Problemu merenja u razvoju softvera ćemo posvetiti više pažnje u poglavlju *Metrike softvera*.

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 koriničkog 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 razvijali.

Neprekidna integracija

Kolektivno vlasništvo nad kodom ima za posledicu da se relativno često dešava da više parova radi nad delovima istog koda. Zbog toga razvojni timovi moraju da koriste sisteme za kontrolu verzija koda, 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, Ekstremno programiranje zastupa *neprekidnu integraciju*. Neprekidna integracija 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.

Postupak pisanja koda 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 proverava se 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.

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 kompletan izvodni kod projekta, prevode ga i izgrađuju od početka i to sa različitim opcijama i za sve ciljne platforme, izvršavaju automatski sve raspoložive testove i pripremaju izveštaje o obavljenom testiranju, izgrađuju distribucione i instalacione pakete, pa čak po potrebi i pokreću instaliranje izgrađnog softvera na platformi za testiranje i izvršavaju odgovarajuće testove prihvatljivosti.

Osim pojednostavljivanja razrešavanja konflikata, glavni doprinos neprekidne integracije je i stalna raspoloživost kompletnog izgrađenog softvera, što omogućava tačan uvid u trenutno stanje razvoja. Sekundarni značaj je i u olakšavanju debagovanja.

Metafora

Metafora je idealizovana velika slika softvera koji se razvija. Ako imamo u vidu da čitava metodologija usmerava programere da gledaju najviše jednu iteraciju daleko, onda ova praksa lako pada u oči kao nešto potpuno suprotno. Zaista, čak i neki zastupnici Ekstremnog programiranja smatraju da ovoj praksi nije tu mesto.

Međutim, neki od uticajnijih autora smatraju da je ona veoma značajna za 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 identifikuju 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 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.

Primarna motivacija za uzdržavanje od preteranog rada je rasterećivanje članova tima i održavanje kvalitetnih odnosa u timu. Posledica rasterećenosti je manji broj grešaka i veća iskorišćenost radnog vremena, pa i ukupno veća produktivnost.

Razvoj softvera nije sprint, nego maraton.
Sprint je dopustiv samo pred ciljem.

Robert Martin

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 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 održava 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 timpovima 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 pojedinca, kao i da je broj grešaka u napisanom kodu daleko manji [Cockburn2001]. 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 počeo „specijalista".

Otvoren radni prostor

Praksa *Otvoren radni prostor* propisuje da bi razvojni tim, u kome se primenjuje Ekstremno programiranje, 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 i da je to neophodno 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 a 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. Čak i pre epidemije Covid-19 primećeno je da razvijaoci koji rade u otvorenom radnom prostoru značajno češće oboljevaju i odsustvuju sa posla zbog bolesti. Međutim, pada u oči da jedan broj takvih istraživanja nije razmatrao programiranje u paru, već samo otvoren radni prostor kao izolovanu praksu prostorne organizacije.

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 i fleksibilnost, performanse i drugo.

Pravilo „Neće biti potrebno" sugeriše da za sve one potencijalne elemente softvera, koji će biti potrebni *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. 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 neke druge načine. Čak i kada su delovi koda samo *veoma slični* preporučuje se njihovo apstrahovanje. Redundantnost u kodu je neophodno izbegavati 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 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 preko vrlo ozbiljnog rada, koji obuhvata dobro upoznavanje domena, pažljivu analize problema, pažljivo projektovanje i druge poslove.

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, ideja je upravo obrnuta – 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 veoma brzo smenjuju, 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 relativno kompletna 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 izmena, pomaže se *refaktorisanje* i vrši se dodatni pritisak da se razdvajaju jedinice koda, čime se dobija bolji dizajn projekta.

Razvoj vođen testovima je iscrpnije predstavljen u poglavlju *9 - ...Razvoj vođen testovima*, na strani 180.

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 trasformacije se pomoću testova jedinica koda proverava da nije slučajno izmenjeno ponašanje. Transformacije se ponavljaju sve dok se ne dobije ponovo čist i dobro dizajirann 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 205.

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

dokumentacje 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.

8.5.3 Ekstremno programiranje danas

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 takvih izveštaja moglo zaključiti. Ako već pominjemo istraživanja i statistike, onda moramo da istaknemo da su rezultati istraživanja o primeni metodologija često veoma neprecizni i diskutabilni. Njihovom analizom bi moglo da se ustanovi da razvojni timovi danas najčešće ne primenjuju dosledno *nijednu* metodologiju (iako tvrde drugačije), već samo primenjuju izabrane tehnike, za koje je u okviru tima procenjeno da najviše odgovaraju konkretnom projektu i konkretnom organizacionom modelu po kome tim funkcioniše. Kada se pogleda koje su to tehnike, vidi se da je najveći broj njih ili potekao iz EP ili bar imao značajno mesto u toj metodologiji.

Kao i svaka druga metodologija i EP 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 u realnim primenama to najčešće nije slučaj i da razvojni timovi biraju i primenjuju samo neke od praksi, za koje sami procene da bi mogle da najviše doprinesu razvojnom procesu.

Imajući u vidu prethodno navedeno, neophodno je da naglasimo da predstavljanje Ekstremnog programiranja u ovoj knjizi ne predstavlja istovremeno i preporuku za primenu, već je pre sugestija da je najbolje da se izučavanje agilnih metodologija započne upravo od upoznavanja Ekstremnog programiranja, kao jedne od najkompletnijih i najuticajnijih agilnih metodologija.

8.6 Skram

Skram je agilna metodologija koja su razvili Ken Švaber i Džef Saterlend u prvoj polovini 1990-ih, a prvi put je javno predstavljena 1995. godine. Za razliku od Ekstremnog programiranja, koje predstavlja 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 organizovanje toka i strukture razvojnog procesa, a tehnike i alate *pozajmljuje* iz drugih metodologija. Imajući u vidu agilnu prirodu Skrama, njegova primena obično počiva na primeni nekog izabranog podskupa praksi Ekstremnog programiranja.

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 preciznija uputstva nisu potrebna) i da 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 predstavljanju najviše osloniti na priručnik koji su napisali i održavaju autori metodologije [Schwaber2020] i na knjigu Keneta Rubina [Rubin2013].

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 je da pažljivo prati napredak svih članova tima, da im ukazuje na greške i 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 tima 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 svih neobavljenih poslova i definiše kriterijuma koje implementacija mora da zadovolji (praktično testovi prihvatljivosti). Vlasnik proizvoda može deo svog posla delegirati 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*. 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 obučeni 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 iznalaženju dobrih rešenja, da poštuju svoje kolege i da budu odvažni u svom radu.

8.6.2 Razvojni proces

Skram propisuje iterativni razvojni proces, kao i sve agilne metodologije. Ono što je specifično 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 ili manje od toga. 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 praviti promene koje dovode u pitanje ispunjavanje cilja sprinta. Ako se u toku sprinta ispostavi da je cilj nedostižan ili da je 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.
- Kako će izabrani poslovi da se urade?
  Za svaki izabrani posao se planira kako će da se uradi tako da se zadovolje kriterijumi dovršavanja.

Nakon planiranja sprinta započinje njegova realizacija. Pri tome 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.

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 pireg 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, u retrospektivi učestvuju samo članovi tima. Retrospektiva uobičajeno traje do tri sata i služi da tim sagleda kvalitete i slabosti ispoljene u toku sprinta. To je prilika da se izvrši samokritični uvid u tok sprinta, da se skrene pažnja na dobro urađene stvari i na eventualne greške, da se prepozna da li je potrebno da se radi na popravljanju međusobnih odnosa, na dodatnom usavršavanju članova tima, na zameni ili unapređenju nekih alata i slično. 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 mnogobrojne različit vrste rezultata rada na projektu.

Umesto toga svega, 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.

Pored cilja i poslova, spisak nedovršenih poslova može da obuhvata i okvirni plan raspoređivanja poslova po iteracijama, ili bar okvirni redosled po kome se očekuje izvođenje poslova. Može da obuhvata i oznake međuzavisnosti poslova.

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 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 sprinta i poslove koji su izabrani da se dovrše u okviru 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. Ovaj spisak je dobro da bude što sadržajniji, da bi mogao da se koristi za ustanovljavanje 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 preponatljiv 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. U okviru jednog sprinta mora da bude ostvaren bar jedan doprinos, a često se ostvaruje i više njih. Ostvareni doprinosi se sagledavaju u okviru razmatranja sprinta, ali mogu da se isporuče klijentu i rnije u toku sprinta.

Doprinos može da obuhvati samo poslove koji su zadovoljili kriterijume dovršenosti. U tom smislu kriterijumi dovršenosti su čvrsto povezani sa poslovima i doprinosima sprinta. Kriterijumi dovršenosti su koncept koji uglavnom odgovara testovima prihvatljivosti. Definišu se formalno a proveravaju bilo automatski ili manuelno, zavisno od konkretnog slučaja.

8.6.4 Skram i druge metodologije

Imajući u vidu da Skram ne propisuje sve uobičajene metodološke elemente, Skram se prirodno kombinuje sa drugim metodologijama. Skram ničim nije vezan za OO tehnike i metodologije, ali ni za bilo koje druge. Praktično je ortogonalan u odnosu na sve raspoložive tehnike i može da se koristi u praktično svim razvojnim projektima.

Skram propisuje organizacione elemente i tehnike upravljanja razvojnim ciklusom, a iz drugih metodologija se preuzimaju konkretne razvojne tehnike. 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 proizvoljno odabiru tehnika i praksi koje će se upotrebljavati. Kao što smo istakli kada smo predstavljali Ekstremno programiranje, tehnike i prakse koje čine neku metodologiju nisu slučajno oblikovane i grupisane na odgovarajući način, već zato što se pokazuje da kroz međusobno kombinovanje više i bolje ispoljavaju svoje pozitivne karakteristike. Proizvoljnim i delimičnim odabirom tehnika dolazimo u priliku da ostvarimo neke benefite koje nam njihova primena pruža ali i da propustimo da ostvarimo neke dodatne benefite zato što ih ne primenjujemo u kombinaciji sa tehnikama sa kojima se one prirodno nadopunjuju.

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 kasnije može skupo da nas košta. Međutim, do sada se nismo bavili problemom primene ovog principa 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 i odgovarajućoj fazi razvoja, 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 ne samo modifikovanje programskog koda, već i strukture programa a time i projekta. Zbog toga je u agilnom procesu uobičajeno da se projekat menja tokom implementirana 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 imeplementacije? 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 sklopu 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 i mogu da imaju 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 manjih ili većih 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 pravi projekat i zapisuje na programskom jeziku umesto dijagramima. Čest je slučaj da problem može da se modelira i implementira na više različitih načina. Izbor načina se obično određuje pri oblikovanju modela, tj. pri projektovanju, a onda programer mora da nametnut način rešavanja implementira. Prepuštanjem

projektovanja programeru dajemo mu veću slobodu i mogućnost da, u slučajevima kada postoji izbor, 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 na projektu i trajanje razvoja. Uz to, dobija se i brže započinjanje sa implementacijom, 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*, kada će i kako tim *A* saopštiti timu *B* 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 se taj interfejs neće menjati? 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 imati zahteve u odnosu na njih?

Odgovori na ova pitanja nisu laki, 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 i postavlja pitanje da li će eventualne izmene na nekoj komponenti zahtevati da se sve 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.

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.

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 viziju 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 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 više informacija nego što pojedinačni razvojni timovi imaju na raspolaganju tokom implementacije.

Poseban kvalitet koji nam donosi projektovanje pre imeplementacije 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, on nema potrebu da nekome „šalje poruku“, pa samim tim nema ni potrebu da iscrpno dokumentuje napravljeni dizajn, već je programski kod obično sasvi 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, koja mora da sačeka da se najpre dovrši projektovanje. 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.

8.7.3 Projektovanje do nivoa komponenti...

Kada sagledamo dobre i loše strane opisanih pristupa projektovanju, možemo da vidimo da nijedan nije savršen, ali i da oba imaju i izrazito dobre i izrazito loše strane. Projektovanje kroz implementiranje ispoljava svoje kvalitete kada se bavimo projekovanjem 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, koji će se često značajno menjati u odnosu na originalni projekat tokom implementiranja (i kasnije), 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 se u iole većim projektima uobičajeno 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. Ovde možemo da se prepozna isti kriterijum za 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.

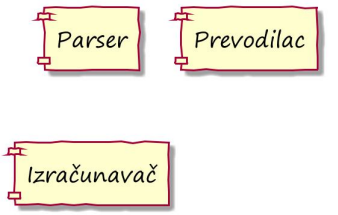
8.7.4 Primer

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 25).

Slika 25 – Početak projektovanja – ceo softver je jedna komponenta

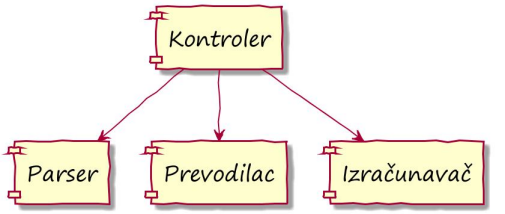
Pretpostavimo da je za naše skriptove najbolje da se obrađuju u tri koraka, koje poveravamo različitim komponentama: najpre će *Parser* da pročita skript, zatim će

Prevodilac da napravi graf izračunavanja koji odgovara pročitanom skriptu i na kraju će *Izračunavač* da izračuna grafom predstavljen program (Slika 26).



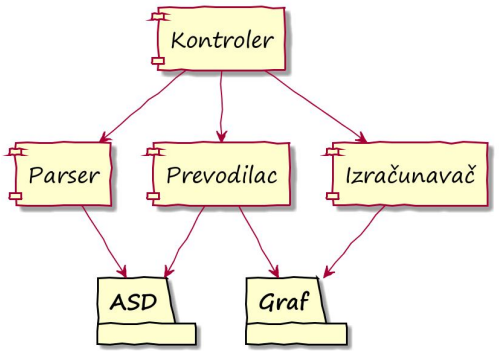
Slika 26 – Prepoznate su glavne komponente

Primetimo da su komponente potpuno razdvojene. Da bismo ih povezali, dodajemo komponentu *Kontroler*, koja će upravljati celim poslom koristeći preostale komponente (Slika 27).



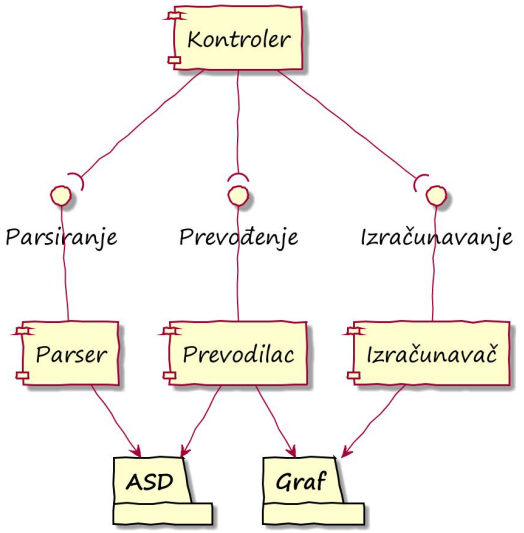
Slika 27 – Prepoznata je potreba da kontroler upravlja poslom

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 u prethodnom koraku, ovde imamo logičku i strukturnu a ne funkcionalnu dekompoziciju – želimo da odgovornost o internoj strukturi apstraktnog sintaksnog drveta i grafa izračunavanja izmestimo iz prepoznatih komponenti i grupišemo u posebne celine. Zbog toga pravimo pakete, a ne komponente (Slika 28).



Slika 28 – Dodajemo pakete *ASD* i *Graf*

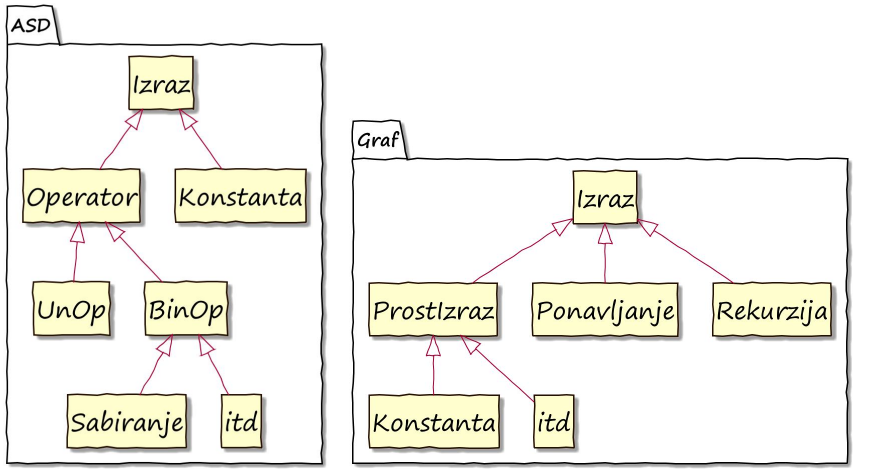
Svaka komponenta mora da ima jasno prepoznat interfejs. On može, ali ne mora da se predstavlja na dijagramu. Ako komponenta ima više različitih interfejsa, onda oni moraju da se navedu. Ako ima samo jedan onda to nije neophodno, ali je dobro da bi se eksplicitnije razlikovale od paketa (Slika 29).



Slika 29 – Eksplicitno predstavljanje interfejsa

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 što preciznije i što ranije odredi njihova struktura, kako bi razvoj komponenti *Parser* i *Prevodilac* mogao da se paralelno odvija. Počeli bismo od prepoznavanja ključnih klasa (Slika 24) a zatim bismo nastavili sa definisanjem njihovog interfejsa. (U primeru ćemo preskočiti definisanje interfejsa, zato što je to

već nešto specifičnije i zahtevalo bi bolje poznavanje problema, a samim tim i značajno više prostora.)



Slika 30 – Početak modeliranja paketa ASD i Graf

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 poput optimizatora, keširanja međurezultata, koordinatora paralelizacije i slično.

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 prvo razviti pakete *ASD* i *Graf*, pa zatim komponente koje ih koriste i na kraju komponentu *Kontroler*. Štaviše, ako su poznati interfejsi komponenti i struktura paketa, onda razvoj može da ide i paralelno. Naravno, detaljnija analiza komponenti i paketa mogla bi 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 dobili arhitekturu (uz pretpostavku da smo oblikovali interfejse komponenti i hijerarhija *ASD* i *Graf*).

8.8 Kritički pogled na agilni razvoj

Ništa nije savršeno, pa ni agilni razvoj softvera i predstavljene agilne metodologije. Dok neki autori ističu kvalitete ovakvog pristupa problemu razvoja softvera, neki drugi autori, 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 u konkretnim metodologijama 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 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 značajno po tom pitanju.

Poseban problem predstavlja pretpostavljen relativno visok nivo stručnosti članova tima. Zbog toga se često dešava da se početnici relativno sporo uklapaju u agilne timove i 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 ipak teško može od početnika da se 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), na koju novi članovi obično moraju da se navikavaju.

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 i potencijalno nerealno visok profit razvijalaca. To ponekad može da značajno otežа 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.

Negativna posledica iterativnog razvoja može da bude zapostavljanje korisničkih celina koje ne donose neposredan profit i iz ugla klijenta zato imaju nizak prioritet, a mogu da značajno utiču na ukupnu upotrebljivost razvijanog softvera. Takve celine se mogu postepeno odlagati za kasnije iteracije, a na kraju često neke od njih 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 incijalno nisu ni bile planirane, pa je zbog povećavanja obima radova sasvim prirodno da je 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.

Dosledna težnja jednostavnim rešenjima i detaljnom planiranju samo tekuće iteracije može da u nekim slučajevima ima veoma neugodne posledice. Postoje sistemi i softveri 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 prakse ekstremnog programiranja, onda se pri implementaciji jedne celine ne uzima u obzir da će naredna celina *morati* da se implementira, pa se zato koristi neprimereno jednostavan dizajn, koji se zatim u više narednih iteracija *značajno menja*. Temeljnost izmena koje su u takvim slučajevima neophodne može da ima za posledicu drastično smanjivanje efikasnosti razvojnog tima.

Još je teže u slučajevima kada *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. Najveći broj problema može da se podeli na manje celine i prilagodi agilnom razvoju, ali 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), a prakse agilnih metodologija njenu izradu često odlažu i nepotrebno cepkaju na parčiće.

Jedna od čestih zamerki se upućuje na ime odsustva sistematično uređene dokumentacije. Agilni timovi dokumentaciju razvijaju 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 ne obavezno i nekome ko će kasnije morati da održava ili dograđuje 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 za to obično više nema vremena. Zato može 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 često 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. Na primer, kod ekstremnog programiranja je praksa da zahtevi i izmene mogu da se podnose samo tokom planiranja iteracije, ali nedovoljno formalna priroda planiranja iteracije i oslanjanje na prisutnost klijenta i tokom implementacije mogu da imaju za posledicu i izmene tokom implementacije.

Relativno često se ističe primedba da postoji gornja granica složenosti softvera koji se može efikasno razvijati 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 [Cumm2008]. 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 [Schiel2009].

Iz predstavljenog kratkog pregleda kritika na račun agilnih metodologija možemo da steknemo utisak da su neke primedbe opravdane, ali i da su neke od njih 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 zato što nisu primenjivane neke druge prakse.

8.9 Umesto zaključka

Kao što se OO programiranje izdvojilo kao vodeća paradigma programiranja, tako su se agilne metodologije [AA, AM] izdvojile i postale zastupljene 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 [Martin2003].

Danas je najzastupljenija agilna metodologija Skram [Schwaber2020, Rubin2013]. 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 [Back1999, Wake2000].

Kada je reč o primeni metodologija, važno je a istaknemo da se ispostavlja 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.

Pregled referenci

[AA]
Agile Alliance, `https://www.agilealliance.org/`

[AM]
Agile manifesto, `http://www.agilemanifesto.org/`

[Back1999]
Kent Back, **Embracing Change With Extreme Programming**, *IEEE Computer*, 32(10), 1999.

[Cockburn2001]
Alistair Cockburn, Laurie Williams, **The Costs and Benefits of Pair Programming**, *in Extreme Programming Examined*, *Addison-Wesley*, 2001.

[Cumm2008]
Fred A. Cummins, **Building the Agile Enterprise**, *Morgan Kaufmann*, 2008.

[Martin2003]
Robert C. Martin, **Agile Software Development: Principles, Patterns, and Practices**, *Prentice Hall*, 2003.

[Rubin2013]
Kenneth Rubin, **Essential Scrum**, *Addison-Wesley*, 2013.

[Schiel2009]
James Schiel, **Enterprise-Scale Agile Software Development**, *CRC Press*, 2009.

[Schwaber2020]
Ken Schwaber, Jeff Sutherland, **The Scrum Guide – The Definitive Guide to Scrum: The Rules of the Game**, 2020, `https://scrumguides.org`

[Wake2001]
William Wake, **Extreme Programming Explored**, *Addison-Wesley*, 2001.