6 - ...Principi projektovanja softvera

Saša Malkov

Odlomak iz knjige Razvoj softvera
(u pripremi)

6.1 Pojam i motivacija

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

Savremeni razvoj softvera voma često pretpostavlja agilni razvoj i primenu OO metodologija. Zbog toga se i problem projektovanja softvera obično posmatra u kontekstu agilnog razvoja i OO metodologija. Skupovi principa projektovanja

softvera koje ćemo predstaviti u ovom poglavlju takođe se odnose prvenstveno na agilne i OO metodologije, ali su u velikoj meri formulisani u odnosu na opšte pojmove (kao što su elementi programa, celine i delovi, zavisnosti i slično) pa bez mnogo prilagođavanja mogu da se primenjuju i na druge razvojne metodologije.

Principi projektovanja softvera obično imaju oblik *pravila* ili *preporuka*. Osnovni cilj svih principa je da nam u određenim situacijama pomognu da izaberemo one kriterijume koji su značajniji i da odlučimo kako da strukturiramo elemente programa koji projektujemo. Kada razmatramo neki manji skup principa, možemo da primetimo da nam navedeni principi nisu dovoljni da razrešimo sve slučajeve sa kojima se susrećemo u praksi, ali sa druge strane, ako razmatramo više skupova principa, onda možemo da dođemo u situaciju da je primena nekih principa čak i delimično suprotstavljena. Upravo zbog toga principe valja prihvatiti kao *savete* ili *lekcije*, a ne kao ultimativna pravila.

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

6.2 Ključni principi OO dizajna

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

Skup principa koji ćemo predstaviti se obično pripisuje Robertu Martinu. On nije lično osmislio sve pojedinačne principe, ali je uticao na njihovo formulisanje u obliku u kome su danas poznati [Martin2003]. Ovaj skup principa se često označava engleskom skraćenicom *SOLID*, na osnovu početnih slova njihovih engleskih naziva.

Princip jedinstvene odgovornosti (SRP)

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

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

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

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

Ovo je jedan od osnovnih principa projektovanja. Njegov značaj se ogleda u težnji čistoj i jasnoj modularizaciji softverskog sistema. Iako deluje vrlo jasno i razumljivo, čak do te mere da se stiče utisak da se može i intuitivno primenjivati, u praksi stvari stoje potpuno drugačije i vrlo često se nailazi na primere projekata u kojima ovaj principi nije dobro (ili čak nije uopšte) primenjen.

Radi ilustracije, razmotrimo softverski projekat u kome se pojavljuje apstraktno sintaksno drvo izraza, gde čvorovi tog drveta predstavljaju objekte koji pripadaju klasama hijerarhije koja modelira različite vrste literala, operatora i drugih sintaksnih konstrukcija. Sintaksno drvo nam služi za pravljenje interne reprezentacije izraza i gradi se parsiranjem izraza. U zavisnosti od sintakse jezika, hijerarhija može da sadrži prilično veliki broj klasa.

Nakon izgradnje izraza može da bude potrebno da se on prevede u neki drugi oblik (možda i na mašinski jezik), ili da se možda ispiše u nekom drugom obliku, ili da se optimizuje, a možda i da se izračuna. Jedan od uobičajenih načina implementiranja ovakvih zahteva je dodavanje novog ponašanja i novih metoda čvorovima hijerarhije izraza, tako da oni mogu da ispišu, izračunaju ili prevedu odgovarajući podizraz. Međutim, iako to izgleda prirodno i relativno jednostavno, takvim pristupom bismo prekršili princip jedinstvene ogovornosti (a kao što ćemo videti, čak i neke druge principe projektovanja).

Klase apstraktnog drveta izraza već imaju jednu odgovornost, a to je izgradnja interne reprezentacije izraza. Dodavanjem bilo koje druge odgovornosti, kao što je na primer izračunavanje vrednosti izraza, mi bismo svakoj od klasa povećali broj odgovornosti, a time i broj mogućih razloga za menjanje. Iako u slučaju samo jedne od operacija to može da izgleda kao dobar pristup, problem se u punoj meri ispoljava ako moramo da svakoj klasi dodamo sve navedene aspekte ponašanja – i pisanje, i prevođenje i optimizovanje, i izračunavanje. Održavanje takvog programa postaje veoma nezahvalan posao. Umesto toga, možemo da primenimo obrazac za projektovanje *Posetilac* i problem rešimo na mnogo čistiji način, bez nagomilavanja odgovornosti, njihovim raspoređivanjem u više različitih klasa posetilaca (*7.5 – Primer – Obrazac Posetilac*, na str. 128).

Princip otvorenosti i zatvorenosti (OCP)

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

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

Otvorenost za proširivanje podrazumeva da bi ponašanje strukturnog elementa trebalo da bude proširivo, ako se za proširivanjem ukaže potreba. Na taj način se obezbeđuje fleksibilnost projekta, posebno u kontekstu agilnog razvoja gde promene i dopune specifikacija prirodno dovode i do potrebe za novim (tj. proširenim) ponašanjem.

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

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

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

Osvrnućemo se ponovo na pomenuti primer sa hijerarhijom klasa apstraktnog drveta izraza. Pri prvom dodavanju novog ponašanja u formi posetioca, morali bismo da dopunimo hijerarhiju čvorova izraza tako da se omogući prihvatanje posetilaca. Pri dodavanju svakog sledećeg ponašanja, biće dovoljno da napravimo novog posetioca (novu klasu) koji će posećivati drvo izraza pomoću već postojećeg mehanizma.

Princip zamenljivosti (LSP)

„Podtipovi moraju da mogu da zamene bazne tipove."

Princip zamenljivosti (engl. *LSP – The Liskov Substitution Principle*) je dobio ime po Barbari Liskov, koja ga je izvorno definisala u nešto opširnijem obliku još 1988.

godine: *Ovde govorimo o nečemu poput sledećeg svojstva zamenljivosti: ako za svaki objekat o_1 tipa S postoji neki objekat o_2 tipa T takav da za sve programe P, u kojima se pojavljuje tip T, važi da se ponašanje programa P ne menja kada se objekat o_1 zameni objektom o_2, onda S predstavlja podtip tipa T*. Primetimo da u iskazanom obliku ovaj princip predstavlja definiciju pojma podtipa, kao tipa koji ispunjava uslov zamenljivosti.

Ovaj princip predstavlja osnovu hijerarhijskog polimorfizma – ako on ne bi važio onda ni hijerarhijski polimorfizam ne bi imao smisla. Međutim, iako može da izgleda da je zbog toga ovaj princip *suvišan* ili *redundantan*, zato što se čini da on automatski važi, stvari zapravo stoje sasvim drugačije – pri projektovanju klasa je često potrebno da se interfejs klase oblikuje veoma pažljivo i da se još pažljivije definiše semantika metoda klase, da ne bi došlo do narušavanja ovog principa. I pored toga, njegovo narušavanje nekada ne može da se izbegne.

Dobra ilustracija problematičnog usaglašavanja sa ovim principom je predstavljena u poglavlju o OO metodologijama, u odeljku *3.4 – Slabosti objektno orijentisanih koncepata*, na stranici 32. Podsetimo se, u baznoj klasi `Pravougaonik` je definisan metod `postaviSirinu`, koji se zatim predefiniše za klasu `Kvadrat`, tako da menja ne samo širinu nego i visinu, da bi objekat zadržao osnovnu osobinu kvadrata da su mu sve stranice jednake. Obratimo pažnju na naredni primer koda:

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

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

Kao što je već ranije naglašeno, ne postoji dobro rešenje ovog problema. Štaviše, na osnovu ovakvih problema možemo da zaključimo da se ispravnost modela ne sme proveravati izolovano, posmatranjem pojedinačnih klasa, već samo posmatranjem čitavog programa. Ovde je uzrok problema u tome što je klasa `Kvadrat` napisana tako da narušava pretpostavke o ponašanju metoda klase `Pravougaonik`.

Da bi bio poštovan princip zamenljivosti, neophodno je da svaki put, kada u izvedenoj klasi predefinišemo neki metod bazne klase, pri tome važi da su

(1) preduslovi tog metoda u izvedenoj klasi isti ili blaži nego u baznoj klasi, a da su (2) postuslovi tog metoda u izvedenoj klasi isti ili jači nego u baznoj klasi.

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

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

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

Princip razdvajanja interfejsa (ISP)

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

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

Pod *razdvajanjem interfejsa* se podrazumeva da složene interfejse, koji služe za obavljanje više poslova, valja podeliti na više manjih interfejsa. Ovaj princip može da se naruši na različite načine, od kojih su najčešći (1) pravljenje klasa ili hijerarhija sa višestrukim odgovornostima i (2) kada celoj hijerarhiji dodajemo nove elemente interfejsa zato što su potrebni samo jednom delu te hijerarhije (jedna grana). Narušavanje ovog principa obično ukazuje na loše raspodeljene odgovornosti pojedinačnih klasa ili preplitanje odgovornosti između delova hijerarhije ili čak više različitih hijerarhija klasa.

U prvom slučaju nije uvek jednostavno prepoznati da li je složen interfejs posledica višestrukuih odgovornosti ili je obrnuto, pa to zahteva pažljivu analizu. Međutim, u oba slučaja je rešenje relativno slično i prilično očigledno – potrebno je podeliti klasu ili komponentu na više manjih, tako da svaka ima jasno definisane odgovornosti, a time i jasno definisan i sužen interfejs.

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

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

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

Vratimo se ponovo na primer sa apstraktnim drvetom izraza. Ako bismo implementirali ranije pominjana ponašanja u svim klasama hijerarhije izraza, onda bismo narušili i princip razdvajanja interfejsa, zato što bismo proširili interfejs tih klasa ponašanjima koja većini korisnika nisu potrebna. Sa druge strane, implementacija pomoću Posetioca je u skladu sa ovim principom, zato što i klase

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

Princip inverzne zavisnosti (DIP)

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

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

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

Pri uvođenju u osnovne pojmove projektovanja naglasili smo važnost apstrahovanja u postupku projektovanja i istakli da se apstrahovanje vrši posmatranjem pojedinačnih slučajeva i oblikovanjem karakteristika opšteg rešenja na osnovu uočenih zajedničkih karakteristika tih pojedinačnih slučajeva. Otuda navedena formulacija principa inverzne zavisnosti može da nam izgleda suprotstavljeno postupku apstrahovanja – sa jedne strane apstrahujemo na osnovu pojedinačnih slučajeva, a sa druge zahtevamo da zavisnost ide u suprotnom smeru, od apstrakcija prema konkretnim slučajevima. Zaista, u nekim starijim metodologijama se težilo da zavisnosti idu upravo u smeru od pojedinačnih slučajeva ka opštim rešenjima. Međutim, danas je drugačije – agilne i OO metodologije zahtevaju upravo suprotno. Otuda i potiče naziv ovog principa – *inverzna zavisnost*.

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

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

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

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

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

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

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

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

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

kao i odgovarajuće konkretne klase formata:

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

```
...
};
...
```

posle čega bi smo umesto oznake formata metodu `napraviIzvestaj` predavali objekat odgovarajućeg formata. U implementaciji metoda bismo koristili interfejs tog objekta:

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

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

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

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

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

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

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

Alternativa uvođenju novog interfejsa je primena parametarskog polimorfizma, u kom slučaju bismo metod `napraviIzvestaj` mogli da napišemo kao šablonski metod, gde je parametar šablona objekat za formatiranje. U konkretnom slučaju ovakav pristup nema neke značajne prednosti, zato što svejedno moramo da definišemo interfejs koji formati moraju da zadovolje. Štaviše, zbog formalnog

definisanja interfejsa u vidu apstraktne klase, prvo rešenje je pouzdanije i bolje. Sličnu ulogu bi mogli da odigraju koncepti u novijim verzijama programskog jezika C++. U nekim drugim slučajevima, a posebno ako nije moguće ili poželjno sve te konkretne objekte stavljati u istu hijerarhiju (a to je u praksi relativno čest slučaj, na primer ako oni već pripadaju različitim hijerarhijama), upotreba parametarskog polimorfizma može da bude bolje rešenje.

6.3 Principi dodeljivanja odgovornosti

Kreg Larman je u svojoj knjizi [Larman2002][12] predstavio skup principa projektovanja, koje je prvenstveno povezao sa problemom raspoređivanja (dodeljivanja) odgovornosti. Predstavljen skup principa je nazvao *opštim softverskim obrascima dodeljivanja odgovornosti* (engl. *GRASP – General Responsibility Assignment Software Patterns*) i opisao ih je nalik na obrasce za projektovanje, kako bi oni bili od pomoći pri učenju i razumevanju problema projektovanja softvera. Iako se prvenstveno odnose na OO metodologije, oni u značajnoj meri mogu da se primene i na druge metodologije. Većina ovih principa nam pomaže da oblikujemo strukturu programa tako da se smanji broj i složenost zavisnosti između različitih delova programa, što nam zatim omogućava da lakše pišemo i održavamo program.

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

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

Informacioni ekspert

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

Osnovna ideja principa Informacioni ekspert (engl. *Information Expert*)[13] je da bi bilo najbolje da ponašanje (tj. metodi koji obavljaju neki posao) i odgovarajuće znanje

[12] Prvo izdanje je iz 1997. godine.

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

(tj. informacije potrebne za obavljanje tog posla) budu locirani na istom mestu. To je u skladu sa težnjom da u programu imamo što manje zavisnosti između različitih strukturnih elemenata programa.

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

Ako pri rešavanju nekog problema ne poštujemo ovaj princip, vrlo je verovatno da ćemo u rezultatu imati neke najviše umereno poželjne vrste kohezije ili relativno nepoželjne nivoe spregnutosti. Na primer, ako podelimo znanje i ponašanje na različite klase jedne komponente, onda ćemo verovatno dobiti komunikacionu koheziju delova komponente. Sa druge strane, ako posmatramo problem na nivou klasa te komponente, onda ćemo među njima verovatno imati spregnutost po sadržaju, preko zajedničkih delova, spoljašnju spregnutost ili spregnutost preko kontrole – što su sve nepoželjniji nivou spregnutosti.

...primer?

Stvaralac

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

- **instanca klase B predstavlja kompoziciju instanci klase A;**
- **instanca klase B referiše instance klase A;**
- **instanca klase B blisko koristi instance klase A;**
- **instanca klasa B ima informacije za inicijalizaciju instanci klase A i prenosi ih pri pravljenju.“**

I ovde je motivacija za primenu principa u težnji da se smanji broj zavisnosti. Osnovna ideja principa Stvaralac (engl. *Creator*) je da se odgovornost za pravljenje instanci neke klase locira na mestu na kome se ta klasa već upotrebljava i na kome već postoji zavisnost od te klase. Na taj način se ne dodaje nova zavisnost, mada može doći do proširenja već postojeće zavisnosti.

U opisu principa su navedena četiri takva slučaja, u kojima je dobar izbor da instanca klase B pravi instance klase A.

Ovaj princip predstavlja primenu dekomponovanja prema promenljivosti. Ako je već neophodno da izaberemo neku klasu X, koja će da pravi instance klase A, onda je dobro da uočimo da je takva sprega relativno jaka i da predstavlja jednu osu promenljivosti. Ako već postoje neke druge zavisnosti klase B od klase A, odnosno

ose promenljivosti koje idu od klase A prema klasi B, onda je poželjno da pokušamo da grupišemo veći broj osa promenljivosti od A tako da idu prema istoj klasi, pa je zato dobro da za X biramo baš klasu B.

Kao primer primene ovog principa može da nam posluži obrazac za projektovanje Apstraktna fabrika [Gamma1995]. Kada je potrebno da se u zavisnosti od okolnosti prave i koriste objekti različitih familija klasa, onda se odgovornost za pravljenje takvih objekata prepušta apstraktnoj fabrici, koja će na osnovu konteksta, argumenata i drugih raspoloživih informacija umeti da odabere odgovarajuću konkretnu klasu čiji će objekat da napravi i vrati pozivaocu. Na primer, ako bi program podržavao više različitih vrsta korisničkih interfejsa, onda bi klasa `FabrikaInterfejsa` mogla da ima metode `napraviProzor`, `napraviDugme` i druge, a koji bi u zavisnosti od konteksta pravili objekat koji odgovara potrebnoj vrsti interfejsa. Gde god da u programu zatreba novi prozor ili novo dugme, takvi objekti se ne bi pravili neposredno već korišćenjem usluga apstraktne fabrike. Ovaj primer predstavlja ilustraciju poslednjeg od četiri slučaja previđena principom Stvaralac.

Visoka kohezija

„Dodeljivati odgovornosti tako da kohezija ostane visoka.“

Kao što je već istaknuto (*5.7 – ...Kohezija i spregnutost*), kohezija (engl. *cohesion*) je stepen međusobne povezanosti elemenata koji čine jednu strukturnu celinu programa (klasu, modul, komponentu, funkciju i sl.). Zajedno sa spregnutošću predstavlja jednu od najvažnijih karakteristika softverskih celina.

Ukratko, težimo da pišemo programe tako da u svakoj strukturnoj celini programa kohezija bude visoka. Svaka celina bi trebalo da predstavlja skup čvrsto međusobno povezanih elemenata.

...primer...

Niska spregnutost

„Dodeljivati odgovornosti tako da spregnutost ostane niska.“

Kao što je već istaknuto (*5.7 – ...Kohezija i spregnutost*), spregnutost (engl. *coupling*) je stepen međusobne povezanosti elemenata različitih strukturnih elemenata programa (klasa, modula, komponenti, funkcija i sl.). Zajedno sa kohezijom predstavlja jednu od najvažnijih karakteristika softverskih celina.

Ukratko, težimo da pišemo programe tako da spregnutost bude što je moguće niža, tj. da različiti strukturni elementi programa budu što manje međuzavisni (tj. da budu što manje povezani i da pri tome budu povezani što je moguće slabijim vezama).

...primer...

Kontroler

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

- **predstavlja ceo sistem, uređaj ili podsistem (tzv. fasadni kontroler), ili**
- **predstavlja scenario slučaja upotrebe u kome se pojavljuju sistemski događaji (i pri tome obično nosi naziv poput `<NazivSlučaja>Rukovalac`, `<NazivSlučaja>Koordinator`, `<NazivSlučaja>Sesija` i slično, tj. predstavlja kontroler sesije ili slučaja upotrebe).“**

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

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

Na primer, ako slučajem upotrebe upravlja korisnik, onda akcija korisnika (tj. korisničko upravljanje slučajem) mora nekako da se prenese na programsko upravljanje tokom implementiranog slučaja upotrebe. Nećemo praviti klasu koja predstavlja korisnika, ali ćemo da napravimo klasu koja će biti odgovorna da prihvata informacije o događajima i da reaguje na njih pokretanjem odgovarajućih aktivnosti.

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

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

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

...primer...?

Polimorfizam

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

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

Osnovna motivacija za primenu ovog principa je u tome da se na mestu korišćenja onih aspekata ponašanja koji se razlikuju za različite tipove ne pravi zavisnost prema svakom od tih tipova, već samo prema njihovoj zajedničkoj (uopštenoj) apstrakciji. U tom pogledu princip primene polimorfizma je zapravo samo varijacija već predstavljenog principa inverzne zavisnosti.

Tipičan primer primene ovog principa je refaktorisanje „Zamena uslova polimorfizmom" [Fowler1999]. Ako na nekom mestu u programu imamo višestruko grananje (na primer naredbu `switch`), koje zavisi od nekog parametra ili neke oznake vrste slučaja, onda je poželjno uvesti hijerarhiju klasa, takvu da svaki poseban slučaj odgovara posebnoj klasi, a zatim se grananje eliminiše pozivanjem metoda koji će u zavisnosti od stvarne klase objekta, a ne u zavisnosti od nekog eksplicitnog parametra) uradi odgovarajući posao.

Izmišljotina

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

Princip Izmišljotina (engl. *Fabrication*)[14] sugeriše da u nekim slučajevima projektant ne mora da tačno sledi strukture iz domena koji se modelira, već može da *izmišlja* nove koncepte, kako bi lakše i bolje modelirao stvarni svet. Izmišljanje novih koncepata se obično preporučuje u slučajevima kada dosledno praćenje postojećih elemenata i koncepata iz prostora domena ima za rezultat problematične zavisnosti, koje se obično iskazuju u obliku niske kohezije, visoke spregnutosti ili dvosmernih ili cirkularnih zavisnosti. Može da bude od koristi i u slučajevima kada dosledno praćenje domena dovodi do pravljenja velikog broja jednostavnih klasa ili raspoređivanje višestrukih odgovornosti u jednu klasu ili komponentu.

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

Neki od jednostavnijih primera izmišljotina su obrasci Fasada ili Adapter. U slučaju obrasca Fasada, ideja je da se složenost nekog sistema zakloni objektom koji predstavlja fasadu i koji stoji između sistema i njegovih korisnika. Obrazac Adapter ne zaklanja neki veći sistem već jednu klasu – on prilagođava interfejs neke klase specifičnom kontekstu u kome je potrebno da se ona koristi. U ovom slučajevima rezultat izmišljanaj nije neki novi složeni koncept već samo neki vid enkapsuliranja složenog ili drugačijeg ponašanja u neke razumljivije okvire (interfejse).

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

Indirekcija

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

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

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

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

Različiti obrasci za projektovanje premenjuju princip indirekcije i različitim kontekstima i na različite načine. Na primer, obrazac Apstraktna fabrika predstavlja posrednika koji od onoga kome je potreban nov objekat neke klase sakriva kako se prave novi objekti, a u mnogim slučajevima čak i klasu čiji se objekti prave. Sličnu ulogu imaju i drugi gradivni obrasci. Sa druge strane, strukturni obrasci poput Adaptera, Mosta, Fasade sakrivaju od korisnika specifičnosti implementacije objekata koji se koriste. Većina obrazaca ponašanja imaju za cilj da sakriju složenost nekog ponašanja uvođenjem nekog vida indirekcije.

Izolovane promenljivosti

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

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

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

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

6.4 Principi oblikovanja celina

Kao što prethodno opisani principi *GRASP* teže da opišu na koji način je potrebno dodeljivati odgovornosti strukturnim elementima programa (najčešće klasama), tako principi oblikovanja celina pokušavaju da ukažu na ispravne načine grupisanja strukturnih elemenata u složenije celine (na primer, kako se grupišu klase u pakete ili komponente). Principi iz ove grupe su tesno povezani sa ostvarivanjem kohezije i spregnutosti, kao i sa dekomponovanjem prema promenljivosti, pa se može reći i da predstavljaju dalje preciziranje prethodno opisanih principa *Kohezija*, *Spregnutost* i *Izolovane promenljivosti*. Mogu se naći u donekle različitim oblicima, a ovde ćemo ih predstaviti u obliku u kome ih je naveo Robert Martin [Martin2003].

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

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

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

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

Princip ekvivalentnosti izdanja i ponovljive upotrebe (REP)

„Granula ponovljive upotrebe je granula objavljivanja.“

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

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

Ovaj princip implicira da zbog (izvesne ili očekivane) ponovljive upotrebe, svaka objavljena celina mora da ima što stabilniji interfejs i što bolju enkapsulaciju. Koliko god da je implementacija neke celine složena, ona bi morala da sakrije od korisnika sve što oni ne moraju da znaju (tj. detalje implementacije) i da im ponudi jasan i čist interfejs (tj. funkcionalnost). Princip *REP* je jedan od osnovnih faktora za funkcionalno grupisanje elemenata u celine, što vodi tzv. *funkcionalnoj koheziji*, kao jednom od najpoželjnijih vidova kohezije.

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

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

Princip zajedničke upotrebe (CRP)

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

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

Primetimo da imamo dva osnovna vida zajedničkog korišćenja klasa. Prvi je da su te klase međusobno funkcionalno povezane i da implementacije nekih od tih klasa zavise od interfejsa ili implementacije drugih klasa. Ako koristimo klase koje su međuzavisne, onda posredno koristimo i klase od kojih one zavise. Takva

povezanost je zapravo već razmotrena u okviru principa *REP* i ovde nam nije od primarnog značaja.

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

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

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

Princip *CRP* nam pomaže i da se odlučimo da neke elemente ne grupišemo u istu celinu. Ako imamo elemente strukture softvera koji nisu međusobno funkcionalno zavisni i pri tome se još i nezavisno upotrebljavaju, onda nema osnova da primenimo ni *REP* ni *CRP*, pa je verovatno bolje da ih ne stavljamo u istu celinu.

Princip zajedničke zatvorenosti (CCP)

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

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

Princip *CCP* se obično odnosi na pakete, koji sadrže skupove klasa ili komponenti, koji nisu neophodno funkcionalno povezani niti se često koriste zajedno. Zbog toga se u formulaciji principa često umesto pojma *celina* koristi neposredno pojam *paket*. Princip *CCP* može da se odnosi i na komponente i klase, pri čemu takve komponente i klase obično imaju veoma širok interfejs, ali i neki oblik objedinjenog interfejsa (npr. obrazac *Fasada*), da bi se bar delimično zaklonila

složenost implementacije. Takav interfejs često ne pruža sve funkcionalnosti koje delovi celine podržavaju, već samo one koje se najčešće koriste.

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

Princip stabilne zavisnosti (SDP)

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

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

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

Princip stabilne zavisnosti nam sugeriše da bi zavisnosti između elemenata programa trebalo da idu od manje stabilnih prema stabilnijim elementima, a ne u suprotnom smeru. Motivacija je sasvim jednostavna – ako je neki element programa nestabilan, to znači da će se on relativno često menjati; a ako od njega zavise drugi elementi, onda je moguće da će usled njegovih promena i oni morati da se menjaju. To znači da u slučaju nepoklapanja smera zavisnosti i porasta stabilnosti postoji mogućnost da dođe do tzv. *lavine promena*, kada promene u jednom elementu indukuju promene u drugim zavisnim elementima i tako redom.

Može da izgleda da je poštovanje ovog principa jednostavno, zato što zavisnost jedne celine od mnogo drugih celina izaziva njenu nestabilnost, pa se onda čini da imamo i poklapanje smerova zavisnosti i porasta stabilnosti. Čest oblik problema potencijalno nastaje kada neka celina istovremeno zavisi od mnogo drugih i kada od nje zavise mnoge druge celine. Takvi slučajevi su relativno česti u praksi, na primer kada pravimo fasadu ispred nekog složenog skupa celina – fasada istovremeno i koristi više drugih celina i ima više korisnika. Rešenje problema je u primeni principa Izolovanih promenljivosti, tj. u definisanju što opštijeg i apstraktnijeg interfejsa fasade, čime se ostvaruje njena stabilnost i poštovanje principa stabilne zavisnosti.

Ovaj princip se u praksi često primenjuje kao detektor, tj. samo ukazuje na postojanje problema, koji se zatim rešava primenom nekih drugih principa, na primer principa inverzne zavisnosti. Primer takvog slučaja može da bude hijerarhija klasa kroz koju se implementira veći broj odgovornosti. Svako dodavanje ili menjanje odgovornosti hijerarhije menja i njen interfejs (baznu klasu) i utiče na njene korisnike, pa je jasno da se tako remeti i ovaj princip – bazna klasa hijerarhije je nestabilna a od nje zavisi veći broj korisnika. Jedno od uobičajenih rešenja ovakvog problema je primena prinicipa inverzne zavisnosti u vidu obrasca Posetilac.

Princip stabilne apstrakcije (SAP)

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

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

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

Dobar primer primene ovog principa predstavlja hijerarhijski polimorfizam. Svaka hijerarhija klasa, ako posmatramo zbirno sve njene elemente, implementira neko složeno ponašanje. To ponašanje obuhvata veći broj različitih slučajeva, koje opisujemo različitim klasama hijerarhije. Sa druge strane, imamo korisnike te hijerarhije za koje ne želimo da zavise od složenih aspekata ponašanja koje je ugrađeno u hijerarhiju, pa ni da znaju za sve te različite slučajeve. Problem rešavamo pravljenjem abstraktne bazne klase hijerarhije, koja će da predstavlja jedinstveni interfejs za sve elemente hijerarhije i sve različite slučajeve ponašanja. Apstraktni interfejs predstavlja istovremeno i primenu Principa inverzne zavisnosti, zato što sve klase hijerarhije zavise od konkretnog ustanovljenog interfejsa bazne klase, dok interfejs bazne klase zavisi samo od sveukupnog načelnog ciljnog ponašanja hijerarhije. Smer zavisnosti je time preokrenut tako da klase hijerarhije zavise od interfejsa, a ne obrnuto. Time što je napravljena kao apstraktna, bazna klasa je postala i stabilna.

Kao drugi primer može da nam posluži fasada, pomenuta u prethodnom odeljku. Na osnovu ovog principa nedvosmisleno zaključujemo da fasada mora da ima apstraktan interfejs, kako bi se ostvarila njena stabilnost.

Princip acikličnih zavisnosti (ADP)

„Ne dopuštati ciklične zavisnosti paketa."

Princip acikličnih zavisnosti (engl. *ADP – The Acyclic-Dependencies Principle*) se odnosi prvenstveno na pakete. Može da se primenjuje i na funkcionalne elemente programa, pre svega na komponente i klase, ali tada moramo imati u vidu da ciklične funkcionalne zavisnosti ponekad ne mogu da se potpuno eliminišu, pa se tada samo trudimo da ih modifikujemo tako da imaju što manji uticaj na stabilnost delova programa.

Osnovni problem sa cikličnim zavisnostima je što značajno otežavaju lokalizovanje promena u programu. Ako imamo neke pakete koji su međusobno ciklično zavisni, pa se dogodi da moramo da promenimo neki od paketa, onda postoji mogućnost da ta promena mora da se propagira na naredni povezan paket, pa tako u krug. Da bismo sprečili eventualni dugačak niz promena, moramo pri menjanju prvog paketa odmah imati u vidu sve aspekte ostalih ciklično uvezanih paketa, čime se i rezonovanje i programiranje značajno otežavaju. Logična posledica toga je upravo ovaj princip – ne želimo da imamo ciklične zavisnosti paketa.

Cikličnost se obično „razbija" tako što se promeni smer zavisnosti između neka dva elementa, koji učestvuju u cikličnoj zavisnosti. Tu obično ima mesta za primenu Principa inverznih zavisnosti ili Principa izmišljotina. Smer zavisnosti se obično menja uvođenjem novog apstraktnog koncepta, koji često ne izgleda prirodno u posmatranom domenu, već predstavlja vid izmišljotine.

Kao primer može da nam posluži često primenjivana arhitektura korisničkog interfejsa „Model-pogled-kontroler". Podsetimo se, ideja ove arhitekture je da model opisuje objekte, pogled prikazuje njihovo stanje korisniku, a kontroler upravlja izmenama. To je u osnovi ciklična arhitektura, zato što kontroler upravlja izmenama stanja modela, model promene mora da prosledi pogledu da bi se prikazale, a korisnik preko pogleda zadaje naredbe kontroleru. Jedan vid prevazilaženja cikličnosti u ovom slučaju je uvođenje obrasca Posmatrač u odnos pogleda i modela, čime se taj odnos apstrahuje tako da i model i pogled zavise od apstraktnog interfejsa koji propisuje taj obrazac, a ne neposredno jedan od drugog. Na sličan način i odnos između pogleda i kontrolera može da se apstrahuje upotrebom principa razvoja vođenog događajima, za koji možemo da kažemo da predstavlja uopštenje i unapređenje obrasca Posmatrač.

6.5 ...Umesto zaključka

U ovom poglavlju su predstavljeni neki od najznačajnijih savremenih principa projektovanja softvera, koji su nastali kao rezultat rada istraživača i razvijalaca u oblasti OO programiranja i agilnog razvoja softvera. Principi projektovanja ne predstavljaju univerzalno rešenje za sve probleme u projektovanju softvera, ali su od velike pomoći zato što nam ukazuju na potencijalne slabosti u dizajnu softvera i usmeravaju nas prema boljim rešenjima.

Principi projektovanja se pominju u donekle različitim oblicima, a ovde su navedeni u skladu sa načinom njihovog predstavljanja u [Martin2003, Larman2002]. Alternativa neposrednom izučavanju principa projektovanja je izučavanje praktičnih primera njihove primene, a teško da ima boljih primera nego što su obrasci projektovanja [Gamma1995] i refaktorisanja [Fowler1999].

...