4 - Uvod u projektovanje softvera

Čitava istorija softverskog inženjerstva
je istorija podizanja nivoa apstrakcije
Grejdi Buč

4.1 Projekat softvera

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

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

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

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

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

4.2 Elementi projektovanja softvera

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

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

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

Proces projektovanja i odnos prema pojedinačnim postupcima u tom procesu su se menjali tokom vremena i značajno se razlikuju od metodologije do metodologije. Neke od metodoloških razlika ćemo da sagledamo u narednom odeljku (*Pristupi projektovanju softvera*, na strani 49).

Istraživanje domena

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

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

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

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

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

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

slika onog dela fakulteta koji ima neposrednog dodira sa procesom upisivanja novih studenata. Istraživanje domena bi trebalo da obuhvati, između ostalog:

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

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

Analiziranje domena

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

Strukturni model domena, na primer, može da ima formu dijagrama klasa domena, dok funkcionalni model domena može da ima formu dijagrama aktivnosti, dijagrama BPMN, ili da obuhvata različite tablice odlučivanja ili korelacija[12]. Sa porastom složenosti domena, povećava se i broj različitih modela domena, koje je neophodno da napravimo da bi stanje (tj. struktura i ponašanje) tog domena bilo precizno opisano.

Model domena mora da jasno opisuje strukturu domena i funkcionalnosti pojedinačnih elemenata te strukture. Obično se domen najpre posmatra kao jedinstvena celina, pa se postepeno raspoznaju delovi te celine koji u njoj imaju neku zaokruženu i jasno prepoznatu ulogu. Preduzima se tzv. funkcionalna dekompozicija celovitog sistema, kako bi se prepoznale i razdvojile pojedinačne komponente i njihove funkcije. Pri pravljenu modela domena osnovni cilj je da se *što vernije predstavi domen*. Zato se tom prilikom obično ne preduzimaju veliki koraci apstrahovanja, kao što će to biti slučaj u kasnijim fazama projektovanja[13].

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

Definisanje zahteva

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

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

---

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

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

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

Projektovanje implementacije

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

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

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

Za razliku od komponenti, koje predstavljaju izdvojene funkcionalne celine, paketi predstavljaju logički grupisane kolekcije klasa (funkcija, potprograma i sl.), koje koristimo zajedno radi obavljanja nekog posla ili koje čine neku strukturnu celinu softvera. Na primer, ako bismo pravili interpretator za neki programski jezik,

onda bismo u jedan paket mogli da stavimo hijerarhiju klasa koje predstavljaju apstraktno sintaksno drvo – sve klase te hijerarhije čine jednu logičku celinu i među njima postoji visok stepen međuzavisnosti, ali sa druge strane one ne čine komponentu, koja ima jasno prepoznatu funkcionalnost i neki jednostavno upotrebljiv interfejs. Paketi predstavljaju važne strukturne elemente softvera, ali za razliku od komponenti ne predstavljaju elemente funkcionalne dekompozicije.

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

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

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

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

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

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

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

4.3 Pristupi projektovanju softvera

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

Projektovanje softvera u klasičnim metodologijama

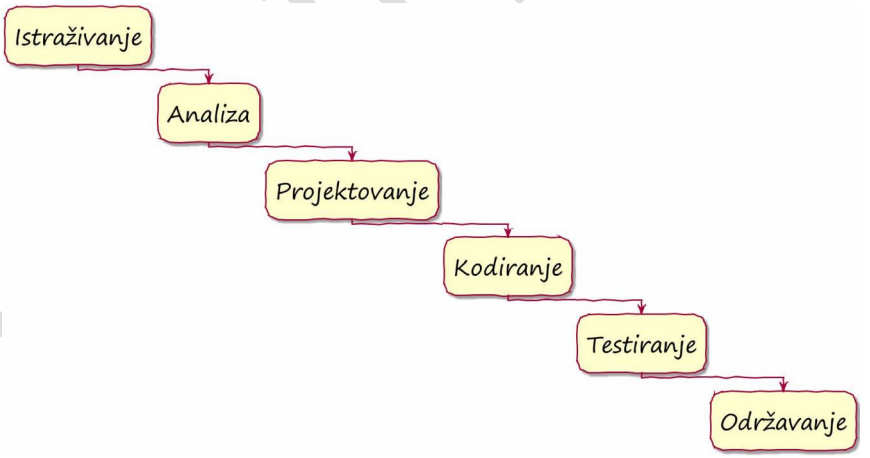
Klasične razvojne metodologije su na različite načine pokušavale da uvedu red u razvojni proces, ali nećemo mnogo pogrešiti ako primetimo da su praktično sve stavljale fokus na (1) oblikovanje toka životnog ciklusa projekta kroz striktno određivanje redosleda obavljanja određenih poslova, (2) opisivanje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

Neke od klasičnih metodologija su se više fokusirale na samo neke od navedenih aspekata, dok su neke druge pokušavale da im daju relativno ujednačen značaj. Tako je opisivanje poslovnih procesa, na način koji je po mnogo čemu sličan opisivanju algoritama, bio ključni element projekata u tzv. *procesnim* metodologijama. Takav pristup je bio posledica pretpostavke da se softverski sistem izrađuje da bi podržao obavljanje nekih poslovnih procesa u poslovnom domenu. Sa povećanjem složenosti

rešavanih problema povećavala se i kompleksnost strukture softvera. Zato su se pojavile *strukturne* metodologije, koje su fokus prebacile na strukturu, kao i *kombinovane* metodologije, koje su pokušavale da fokus ravnomerno rasporede na strukturu i procese.

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

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



Slika 1 – Životni ciklus projekta po modelu vodopada

Slika 1 predstavlja jedan primer životnog ciklusa po modelu vodopada. Iz primera možemo da vidimo osnovnu ideju pristupa i osnovne razvojne celine. Naravno, mogli bismo da podelimo neke korake na više manjih, ili možda čak da neke objedinimo većim koracima, ali osnovni princip se time ne menja.

Takav pristup životnom ciklusu projekta je bio tesno povezan sa pretpostavkom da razvijaoci softvera imaju relativno uske specijalnosti i da zbog toga u različitim

poslovima i različitim fazama životnog ciklusa učestvuju različiti profili razvijalaca. Tako su se, na primer, prikupljanjem informacija o domenu i analizom domena bavili *sistem-analitičari*, projektovanjem *programeri analitičari* ili *sistemski programeri*, dok su se kodiranjem bavili *programeri*.

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

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

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

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

Projektovanje softvera u OO metodologijama

Sa prelaskom na objektno-orijentisane metodologije, sve veći značaj se pridaje povezivanju strukture i ponašanja softvera. OO razvojne metodologije teže da OO način posmatranja i modeliranja prenesu sa nivoa kodiranja programa na nivo

modeliranja problema, pa čak i samog razvojnog procesa. Zbog toga se postepeno razvijaju novi pristupi posmatranju domena i softvera, a sa njima i nove tehnike modeliranja, što je na kraju dovelo i do pravljenja Objedinjenog jezika za modeliranje (*UML*)[14].

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

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

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

Iako su OO metodologije značajno promenile pristup problemu projektovanja, sam životni ciklus projekta nije nužno bio izmenjen. U početnim fazama razvoja OO metodologija faze projektovanja i implementacije su još uvek bile jasno razdvojene, kako vremenski tako i prema specijalnostima razvijalaca. I dalje je životni ciklus obično imao odlike (unapređenog) modela vodopada.

[14] Zbog njegovog izuzetno velikog značaja, *UML*-u je posvećeno posebno poglavlje *5 - UML*, na strani 83.

Projektovanje softvera u agilnim metodologijama

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

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

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

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

4.4 Arhitektura i dizajn

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

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

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

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

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

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

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

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

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

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

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

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

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

4.5 Apstrahovanje i dekomponovanje

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

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

Apstrahovanje može da ima nekoliko različitih vidova, među kojima se kao najvažniji izdvajaju: (1) klasifikacija, (2) generalizacija, (3) oblikovanje servisa i (4) komponovanje.

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

primer klasifikovanje komponenti, klasifikovanje procesa, klasifikovanje slučajeva upotrebe i drugo.

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

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

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

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

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

Komponovanje je vid apstrahovanja kojim se prepoznaje da neki skup objekata može da se posmatra kao celina iz „spoljnog" sveta. Podrazumeva grupisanje objekata u celinu i definisanje granica te celine prema spoljnom svetu. Nakon definisanja granica obično je neophodno primeniti oblikovanje servisa. Komponovanje može da se odnosi na više klasa, koje sadrađuju u obavljanju nekog posla, kojom prilikom se pravi tzv. *fasada*, koja predstavlja interfejs celine i zaklanja ostale klase koje tu celinu implementiraju. Slično, komponovanje može da se odnosi i na komponente ili servise, kada se oblikovanjem nove komponente definiše opštiji ili suženi interfejs prema nekim složenijim upotrebama manjih servisa, da bi se njihova upotreba učinila jednostavnijom.

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

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

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

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

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

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

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

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

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

Dobra strana ovakvog pristupa je što pruža potencijal da se kao rezultat dobije čvršća arhitektura, koja je veoma otporna na promene. Međutim, ni takav pristup nije bez potencijalnih slabosti. Jedna od slabosti je da je za utvrđivanje tačaka i osa promenljivosti potrebno da se makar započne detaljnije projektovanje, pa je potrebno ili veliko iskustvo i dobar osećaj projektanata ili učestalo smenjivanje opštijeg i

detaljnijeg projektovanja. Druga slabost je u tome što tačke i ose promenljivosti mogu da se prepoznaju praktično bilo gde i bilo kad, pri čemu procenjivanje njihovog značaja nije uvek jednostavno. Ako istaknemo previše tačaka i osa promenljivosti, među kojima neke možda imaju sasvim ograničen značaj, dobićemo mnogo komplikovaniji model nego što je potrebno. Sa druge strane, ako propustimo da istaknemo neku značajnu tačku ili osu promenljivosti, onda će model biti osetljiviji na promene nego što bi trebalo da bude.

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

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

Logička dekompozicija predstavlja podelu celine na *pakete* (ili druge strukturne celine) koji grupišu delove implementacije koji se zajedno razvijaju ili imaju izražene unutrašnje međuzavisnosti strukture ili ponašanja. Delovi implementacije koji čine jedan paket ne moraju da budu funkcionalno povezani, mada je to čest slučaj. Takođe, ne moraju da predstavljaju ni potpuno zaokruženu funkcionalnu celinu. Uobičajena praksa je da se elementi paketa povezani logički a ne funkcionalno.

Primer paketa logički povezanih elemenata je paket klasa koje implementiraju hijerarhiju čvorova apstraktnog drveta izraza. Sve te klase su strukturno povezane, imaju jaku logičku vezu i koriste se zajedno, ali među njima ne postoji jaka funkcionalna veza, osim što su sve klase hijerarhije funkcionalno zavisne od bazne klase.

Osim funkcionalne i logičke povezanosti, paketi relativno često mogu da sadrže i delove različitih komponenti koji se na sličan način implementiraju. Možemo reći da se tu radi o vidu strukturne povezanosti elemenata paketa. To je svakako manje poželjno od funkcionalne i logičke povezanosti ali može da bude opravdano. Na rimer, paket može da sadrži nekoliko osnovnih klasa hijerarhije koja se koristi za obezbeđivanje komunikacije među komponentama.

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

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

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

4.6 Procenjivanje kvaliteta projekta

Kvalitet projekta možemo da procenjujemo kvalitativno i kvantitativno. Kvalitativno procenjivanje počiva na analiziranju projekta i razvijenog softvera i razmatranju njihovih karakteristika. Neke od karakteristika softvera prepoznajemo

kao dobre ili poželjne, dok neke druge smatramo za loše ili nepoželjne. Sagledavanjem poželjnih i nepoželjnih karakteristika nekog projekta i njihovim upoređivanjem sa drugim projektima ili nekim prethodno utvrđenim planom, možemo da izvedemo kvalitativnu procenu projekta.

Kvantitativno procenjivanje počiva na definisanju i izračunavanju različitih numeričkih ocena i njihovom analiziranju. Pri kvalitativnom i kvantitativnom procenjivanju projekta posmatraju se uglavnom iste karakteristike, ali na različite načine – u jednom slučaju ih opisujemo i upoređujemo prema načinu i obliku ostvarivanja, a u drugom ih predstavljamo numerički i poredimo dobijene brojeve.

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

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

- ispravnost i
- efikasnost

Ispravnost softvera podrazumeva da se softver ponaša u potpunosti u skladu sa funkcionalnom specifikacijom, tj. da sve funkcije softvera rade onako kako je dokumentovano, bez neispravnih rezultata ili otkazivanja. *Efikasnost* podrazumeva da softver radi uz planirano zauzeće resursa, uključujući ne samo procesorsko vreme nego i radnu memoriju, komunikacione linije, skladišni prostor i sve druge resurse računarskog sistema na kome se izvršava.

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

sagledaju od strane korisnika, što se često svodi na ispravnost i efikasnost[15]. Iako to nije uvek očigledno, ipak moramo da imamo u vidu da nije jedina namena razvijenog softvera da nešto radi ispravno i efikasno, pa zbog toga moramo da razmatramo i neke druge karakteristike softvera.

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

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

- fleksibilnost;
- proširivost i
- modularnost.

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

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

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

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

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

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

Jasna razdvojenost odgovornosti komponenti podrazumeva da se odgovornosti pojedinačnih komponenti ne preklapaju, odnosno da jedna komponenta potpuno samostalno obavlja posao za koji je zadužena[16]. Eventualni izostanak razdvojenosti odgovornosti može da ima za posledicu da neki deo posla ne obavlja jedna komponenta samostalno, već više komponenti zajedno. To može da značajno otežа kako razvoj tih pojedinačnih komponenti (zato što funkcionalnost i implementacija jedne komponente potencijalno značajno utiču na rad druge) tako i njihovo povezivanje u celinu.

Razdvajanje odgovornosti je obično veoma tesno povezano sa definisanjem preciznih i čistih interfejsa komponenti. Interfejs neke komponente predstavlja sredstvo putem koga druge komponente mogu da koriste funkcionalnosti te komponente. Interfejs može da se posmatra i kao deklaracija ponašanja (tj.

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

deklaracija funkcionalnosti) jedne komponente. Ako su odgovornosti dobro razdvojene, onda interfejsi mogu da se definišu tako da budu uski, precizni i jasni.

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

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

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

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

4.7 Kohezija i spregnutost

Kao što smo već naglasili, kvalitet projekta i uspešnost implementacije često presudno zavise od uspešnosti ostvarivanja ravnoteže između razdvajanja i povezivanja modula. Da bismo mogli da ocenimo kvalitet dekompozicije, neophodno je da posmatramo i da na neki način merimo i procenjujemo kvalitet povezanosti odnosno razdvojenost modula. Dve osnovne mere koje pri tome uvodimo su *kohezija* i *spregnutost*:

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

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

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

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

4.7.1 Kohezija

Visoka kohezija jednog modula znači da su svi delovi tog modula međusobno snažno povezani. Ako posmatramo jedan modul kao deo neke šire celine, koji okuplja neke srodne elemente, onda je očekivano da ti elementi budu nekako povezani, bilo da je ta povezanost funkcionalna (na primer, delovi jedne

komponente sarađuju radi ostvarivanja funkcije komponente) ili logička (na primer, delovi jednog paketa su implementaciono međuzavisni).

Sa druge strane, niska kohezija u nekom modulu znači da pojedini delovi tog modula nisu međusobno povezani ili su vrlo slabo međusobno povezani. Niska kohezija nam obično ukazuje na to da dekompozicija možda nije izvedena dovoljno dobro i da granice modula nisu ispravno određene. Na primer, ako u modulu možemo da prepoznamo manje grupe njegovih elemenata koji su međusobno snažno povezani, a pri tome su relativno slabo povezani sa ostatkom modula, onda bi verovatno trebalo nastaviti dekomponovanje, tj. podeliti modul na više manjih modula. Niska kohezija može da bude posledica dodeljivanja više funkcija ili odgovornosti jednom modulu.

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

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

Funkcionalna kohezija

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

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

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

Sekvencijalna kohezija

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

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

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

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

Komunikaciona kohezija

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

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

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

Proceduralna kohezija

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

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

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

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

Vremenska kohezija

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

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

Vremenska kohezija ne spada u poželjne vrste kohezije.

Logička kohezija

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

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

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

Koincidentna kohezija

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

Koincidentna kohezija je dobila ime po tome što ona nastaje kada su elementi modula praktično *slučajno* okupljeni u isti modul. Presudan faktor za takvo grupisanje može da bude to što su možda pisani istog dana, ili istim alatom, ili što oni koriste neke slične interfejse i sl. Za ovu vrstu kohezije je karakteristično da među elementima modula ne postoji nikakva ili tek veoma slaba funkcionalna zavisnost. Odgovornosti uopšte nisu razdvojene, a često ne mogu ni da se jasno raspoznaju, zato što delovi modula rade potpuno raznorodne poslove.

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

4.7.2 Spregnutost

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

Visoka spregnutost predstavlja snažnu povezanost nekog modula sa drugim modulima. Ako jedan modul intenzivno koristi usluge nekog drugog modula, ali kroz veoma čist i jednostavan interfejs, to obično ne predstavlja problem. Međutim,

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

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

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

Aksiome spregnutosti

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

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

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

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

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

Vrste spregnutosti

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

Sprega logike

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

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

Na primer, ako dve komponente A i B obavljaju međusobno komplementarne poslove (recimo da A obavlja kodiranje a B dekodiranje po istom algoritmu, ili da A obavlja zapisivanje a B čitanje podataka u istom formatu), onda je implementacija delova modula A čvrsto povezana sa implementacijom delova modula B. Ova dva modula *dele pretpostavke* o tome kako se neki posao obavlja u drugom modulu.

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

Sprega tipova

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

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

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

Sprega specifikacije

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

Sprega specifikacija se u praksi ostvaruje primenom nekog od vidova polimorfizma. U slučaju hijerarhijskog polimorfizma se modul koji koristi druge tipove implementira tako da koristi apstraktnu klasu[17]. Vid sprege specifikacija je i kada jedan modul definiše implementaciju interfejsa koji je definisan u drugom modulu[18].

U slučaju drugih vidova polimorfizma se modul-korisnik implementira nad apstraktnim parametrizovanim tipovima ili čak bez navođenja tipova. U slučaju programskog jezika C++ to se ostvaruje implementiranjem šablona klasa ili šablona funkcija. U tom slučaju su specifikacije praktično implicitne, zato što ne postoji njihovo eksplicitno navođenje i povezivanje sa implementacijom. Od verzije C++20 programerima su na raspolaganju *koncepti* kao sredstvo za bliže opisivanje dopuštenih parametarskih tipova.

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

Nivoi spregnutosti

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

---

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

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

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

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

Spregnutost po sadržaju

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

Pojednostavljen primer:

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

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

Spregnutost po sadržaju je najsnažnija i najnepoželjnija vrsta spregnutosti.

Spregnutost preko zajedničkih delova

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

Pojednostavljen primer:

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

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

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

Spoljašnja spregnutost

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

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

```
... A::metod1 {
    ...external::oper1(...);
    ...external::oper2(...);
};
... B::metod2(...){
    ...external::oper1(...);
    ...external::oper2(...);
}
... C::metod3(...){
    ...external::oper3(...);
    ...external::oper4(...);
}
```

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

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

složenosti upotrebe i stavila na raspolaganje samo vrlo uzak interfejs, čime se praktično dobija spregnutost preko kontrole.

Spregnutost preko kontrole

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

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

Pojednostavljen primer:

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

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

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

Drugi potencijalan problem je sličan kao i u slučaju spoljašnje spregnutosti – ako jednu kontrolisanu komponentu kontroliše veći broj kontrolera, onda u slučaju promene interfejsa kontrolisane komponente može da se zahteva i veći broj izmena u kontrolerima. Zbog toga ovaj nivo spregnutosti obično ima smisla samo u slučaju kada se ne uvodi veći broj kontrolera.

Spregnutost preko markera

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

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

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

Pojednostavljen primer:

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

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

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

Spregnutost preko podataka

Spregnutost preko podataka je kada jedna komponenta koristi interfejs druge komponente, putem koga mu prenosi pojedinačne podatke.

Ako su podaci koji se prenose relativno složeni, onda može da bude reč o spregnutosti preko zajedničkih delova.

Pojednostavljen primer:

```
... simulacija::promenaSmera(...)
{
    ...
```

```
        automobil->skreniLevo( 5 );
        ...
    }
```

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

Spregnutost preko poruka

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

Pojednostavljen primer:

```
... A::metod1(...) {
        ...
        transakcija->izvrsi();
        ...
    }
```

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

Smer spregnutosti

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

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

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

Širina sprege

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

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

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

Način ostvarivanja sprege

Spregnutost između različitih modula može da se ostvaruje statički i dinamički

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

- klasa `B` nasleđuje klasu `A`;
- podatak `b` klase `A` je objekat klase `B`;
- argument metoda klase `A` je objekat klase `B`;
- u metodima klase `A` se pravi ili upotrebljava objekat klase `B`.

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

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

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

Intenzitet spregnutosti

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

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

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

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

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

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

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

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

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

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

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

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

4.8 Umesto zaključka

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

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

Među prvima koji su javno zastupali stav da programiranje predstavlja čin projektovanja softvera je bio Džek Rivs, koji je objavio tekst o tome u časopisu *C++ Journal* [*Reeves 1992*]. Taj tekst je izazvao burne reakcije, ali je ostvario veliki uticaj na nadolazeći talas razvoja agilnih metodologija. Sa druge strane, stav o važnosti stabilne i robusne arhitekture možda najjače promoviše metodologija *RUP* [*Kruchten 2003*]. Odnos arhitekture i dizajna je do danas ostao relativno neprecizan, o čemu na veoma zanimljiv način piše Martin Fauler [*Fowler 2003b*].