SI Vježbe 8

Dizajn softvera

1. Sistemski dijagrami sekvenci 1
1.1 Šta su sistemski dijagrami sekvenci? 3
1.2 Imenovanje sistemskih događaja i operacija 4
1.3 SSD modeliranje i eksterni sistemi 4
2. Ugovori operacija 5
3. Logička arhitektura 9
3.1 Uvod 9
3.2 Arhitektura 10
3.3 UML paketi 10
3.4 Dizajniranje u slojevima 11
3.5 Domain sloj i Domain model 12
3.6 Princip odvajanja modela od UI sloja 12
4. Dizajn objekata 13
4.1 UML kao alat za crtanje 13
4.2 Statički i dinamički dijagrami 14
4.3 Modeliranje ponašanja sistema (dinamički model) 15
4.4 Ulazi i izlazi procesa dizajniranja objekata 15
4.5 Dizajn objekata zasnovan na dodjeli odgovornosti (Responsibility-Driven Design) 16
4.6 GRASP: Metodološki pristup osnovnom OO dizajnu 16

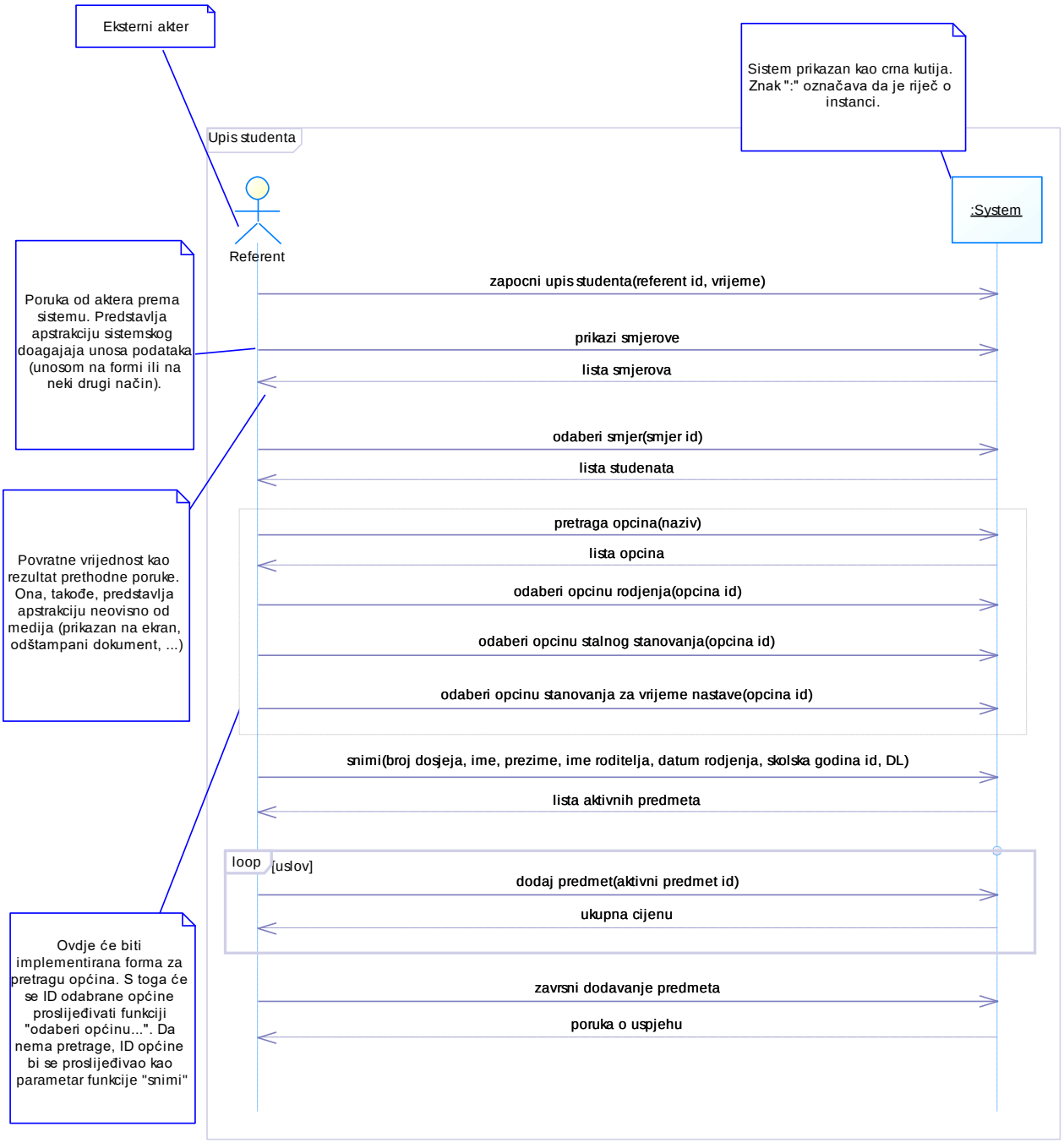
1. Sistemski dijagrami sekvenci

Sistemski dijagram sekvenci (System Sequence Diagram – SSD) je artifakt koji nije teško kreirati, a prikazuje interakciju eksternih aktera sa sistemom koji se razmatra. SSD se kreiraju na osnovu opisa *use-case* dijagrama. Nakon toga, SSD se koriste kao ulazni artifakt za kreiranje ugovora operacija (koji su objašnjeni u drugom dijelu ovog dokumenta).

SSD pokazuje određeni (sekvencu) događaja u okviru jednog "use-case"-a i aktere koji imaju direktnu interakciju sa sistemom. Sistem se pri tome posmatra kao "crna kutija", tj. prikazuju se događaji koje generiše akter i rezultati koje daje sistem, bez detalja o tome kako je sistem došao do tih rezultata. Vrijeme na dijagramu teče odozgo prema dole, tj. prvi događaj je na vrhu dijagrama, a događaji koji slijede nakon njega su prikazani ispod i to po redoslijedu dešavanja.

Slijedeća ilustracija daje primjer jednog sistemskog dijagrama sekvenci.

Slijedi primjer sa pretragom općine:



Slika 1.1. Sistemski dijagrami sekvenca bez pretrage

Slijedi primjer bez pretrage općine:

Slika 1.2. Sistemski dijagrami sekvenca sa pretragom

Prethodni dijagram nastao je na osnovu "use-case" opisa za upis studenata. Tekst opisa "use-case"-a upućuje na to da postoje slijedeći sistemski događaji koje generiše eksterni akter: zapocni unos studenta (iniciranje scenarija), odaberi smjer, ... zavrsi dodavanje predmeta (potvrda unosa).

Obično se na sistemskom dijagramu sekvenci prikazuje samo glavni tok izvršavanja "use-case"-a. Ukoliko je neki od alternativnih tokova također interesantan, može se prikazati na posebnom dijagramu.

1.1 Šta su sistemski dijagrami sekvenci?

Use-case opisuje način na koji eksterni akteri komuniciraju sa softverskim sistemom koji želimo kreirati. Za vrijeme te interakcije akter generiše **sistemske događaje** prema sistemu (što zahtjeva da neka **sistemska operacija** obradi taj sistemski događaj, ili drugim riječima, sistemski događaj uzrokuje izvršavanje odgovarajuće sistemske operacije).

Sistemski dijagram sekvenci je "slika" koja za jedan scenarij (glavni tok) prikazuje sistemske događaje koje generiše

akter, redoslijed kojim se ti događaji izvršavaju, kao i događaje između sistema. Svi sistemi se tretiraju kao "zatvorene kutije" jer se žele istaknuti događaji koji prelaze granice sistema, od aktera prema sistemu.

Postavlja se pitanje zašto crtati SSD? Jako je bitno precizno specifikovati koji vanjski događaji utiču na sistem. Krajnji cilj je dizajniranje sistema koji obrađuje te vanjske događaje i generiše izlaze kao rezultat tih događaja.

Uopšteno, softverski sistem izvršava akcije uslijed:

- eksternih događaja koje generiše akter
- vremenskih događaja (generišu se u određeno vrijeme, periodično, itd.)
- pogreške

Prije detaljnog dizajna softverske aplikacije potebno je specifikovati šta sistem treba da radi, a ne kako to radi (ponašanje sistema). Crtanjem sistemskih dijagrama sekvenci upravo se fokusiramo na to pitanje: Koje eksterne događaje sistem treba da obradi?".

Treba naglasiti da UML ne definiše pojam "sistemskih dijagrama sekvenci", nego jednostavno *dijagrama sekvenci* ("sequence diagrams"). Kada dijagram sekvenci koristimo da opišemo interakciju između korisnika i sistema, tada govorimo o sistemskim dijagramima sekvenci. Kasnije ćemo dijagram sekvenci koristi da opišemo interakciju softverskih objekata, tj. koristićemo isti alat koji nam UML stavlja na raspolaganje, ali u drugom kontekstu.

1.2 Imenovanje sistemskih događaja i operacija

Sistemski događaji bi trebali izraženi u opštem obliku, bez naznačavanja konkretnih fizičkih ulaznih uređaja. Na primjer "unosPodataka" je primjerenije on "utipkavanjePodataka" jer se ovim drugim implicira korištenje tipkovnice za unos iako bi se unos mogao riješiti i na drugi način (npr. "barcode" skeniranje broja dosijea ili slično). Pored toga, kada je sistemski događaj izražen u opštem obliku, fokusira se na namjeru događaja, a ne na tehniku generisanja događaja.

Također, radi poboljšanja jasnoće, sistemski događaji bi trebali počinjati glagolom (dodaj..., unesi..., kreiraj...) čime se naglašava da se u stvari radi komandama upućenim sistemu.

1.3 SSD modeliranje i eksterni sistemi

Sistemski dijagrami sekvenci se mogu koristiti da bi se ilustrovale interakcije između različitih sistema. Na primjer, sistem studenske službe bi mogao vršiti upite na računovodstveni sistem univerziteta radi provjere izvršnih uplata (recimo kod prijave ispita, ako se želi onemogućiti prijava ispita ukoliko student nije izvršio sve obaveze plaćanja).

2. Ugovori operacija

Use-case model je osnovni način na koji se opisuje ponašanje sistema (u okviru UP-a). To je obično dovoljno. Ponekad je ipak potreban detaljniji ili precizniji opis ponašanja softverskog sistema. U tu svrhu se koriste *ugovori operacija* ("operation contracts"). Ugovori operacija opisuju promjene na objektima domain modela kao rezultat izvršavanja jedne sistemske operacije.

Domain model je najčešći rezultat objektno orijentisane analize, ali ugovori operacija i modeli stanja (koji će biti opisani kasnije) također mogu biti koristi dokumenti vezani za OOA.

Može se smatrati da su ugovori operacija dio use-case modela UP-a zato što pružaju više analitičkih detalja o efektima sistemskih operacija na koje upućuju pojedini slučajevi korištenja ("use-case").

Primarni "input" pri kreiranju ugovora operacija su sistemske operacije identifikovane u sistemskim dijagramima sekvenci (npr. *zapocni_upis_studenta*), domain model, ali i pojašnjenja domain modela koja mogu pružiti eksperti iz domene problema (npr. referenti studentske službe). Operacijski ugovori zatim mogu poslužiti kao "input" pri dizajnu objekata jer opisuju promjene koje se zahtijevaju nad objektima ili bazom podataka.

U nastavku je dat primjer operacijskog ugovora za sistemsku operaciju "*zapocni_upis_studenta*". Najbitniji dio je "rezultati", dok su ostali dijelovi manje važni.

CO1: Ugovor operacije *zapocni_upis_studenta*

Operacija:	*zapocni_upis_studenta* (referent_id, vrijeme)**
Povezan sa:	Use-case: Upis studenta
Preduslovi:	Upis studenta je u toku
Rezultati:	• kreirana instanca „**student**“ *(kreiranje instance)* • kreirana instanca „**upisana godina**“ *(kreiranje instance)* • postavljene vrijednosti atributa „**referent**“ i „**datumUpisanLjetni**“ instance „**upisana godina**“ *(modifikacija atributa)* • instanca "**upisana godina**" povezana sa instancom "**student**" *(formiranje veze)* • Instanca „**upisana godina“** se čuva dok referent ne završi upis studenta ili dok ne odustane *(Ovdje nije potrebno naglasiti da se instanca „student“ mora čuvati. Ona se automatski čuva jer je ona atribut instance „upisana godina“. Vidjeti domain model).*

** atributi *referent_id* i *vrijeme* su poznati u momentu početka upisa studenta, tako da se oni prosljeđuju pri pozivu prve operacije

Značajne pojedinih dijelova operacijskog ugovora može se zaključiti na osnovu primjera:

Operacija – naziv operacije i parametri.

Povezan sa – Slučajevi korištenja pri čijem izvršavanju se može generisati ova operacija.

Preduslovi – bitne pretpostavke o stanju sistema ili objekata iz domain modela prije izvršavanja operacije.

Rezultati – Najvažniji dio koji opisuje stanje objekata domain modela nakon izvršenja operacije.

Može se primijetiti da su na prethodnom primjeru rezultati (kao pomoć učenju) kategorisani na kreiranje instance, formiranje veze itd.

Rezultati na operacijskim dijagramima opisuju promjene na objektima modela. Promjene uključuju:

- kreiranje instanci
- formiranje i raskidanje veza
- izmjena vrijednosti atributa

Rezultati nisu akcije koje operacija mora izvršiti nego stanje objekata modela nakon izvršenja operacija.

Slijedi primjer ugovora operacija za use-case "*Upis studenta*"

COx: Ugovor operacije *prikazi_smjerove***

Operacija:	*Prikazi_smjerove*
Povezan sa:	Use-case: Upis studenta
Preduslovi:	Upis studenta je u toku
Rezultati:	• učitane sve instance tipa „Smjer“ i vraćene kao rezultat funkcije (povratna vrijednost)

** s obzirom da ovakve operacije ne vrši izmjene objekata u domain modelu, nećemo ih opisivati

CO2: Ugovor operacije *odaberi_smjer*

Operacija:	*odaberi_smjer(smjer_id)*
Povezan sa:	Use-case: Upis studenta
Preduslovi:	Upis studenta je u toku i započet upis studenta
Rezultati:	• postavljena vrijednost atributa „**smjer**“ instance „**student**“

CO3: Ugovor operacije *odaberi_opcinu_rodjenja*

Operacija:	*odaberi_opcinu_rodjenja(opcina_id)*
Povezan sa:	Use-case: Upis studenta
Preduslovi:	Upis studenta je u toku i započet upis studenta
Rezultati:	• postavljena vrijednost atributa „**rodjen**“ instance „**student**“

CO4: Ugovor operacije *odaberi_opcinu_stalnog_prebivalista*

Operacija:	*odaberi_opcinu_stalnog_prebivalista(opcina_id)*
Povezan sa:	Use-case: Upis studenta
Preduslovi:	Upis studenta je u toku i započet upis studenta
Rezultati:	• postavljena vrijednost atributa „**stalno_prebivaliste**“ instance „**student**“

CO5: Ugovor operacije *odaberi_opcinu_stanovanja_za_vrijeme_nastave*

Operacija:	*odaberi_opcinu_stanovanja_za_vrijeme_nastave(opcina_id)*
Povezan sa:	Use-case: Upis studenta
Preduslovi:	Upis studenta je u toku i započet upis studenta
Rezultati:	• postavljena vrijednost atributa „**privremeno_prebivaliste**“ instance „**student**“

CO6: Ugovor operacije *snimi*

Operacija:	*snimi(broj dosjeja, ime, prezime, ime roditelja, datum_rodjenja, skolska godina id, DL)*
Povezan sa:	Use-case: Upis studenta
Preduslovi:	Upis studenta je u toku i započet upis studenta
Rezultati:	• postavljene ostale vrijednosti atributa instance „**student**“ i instance „**upisana godina**“ • instanca **„student“**, a zatim instanca **„upisana godina“** se snimaju u bazu podataka *(Pratite na redoslijed. Snimanje u bazu je, takođe, moguće vršiti pri pozivu operacije zavrsni_dodavanje_predmeta)*

CO7: Ugovor operacije *dodaj_predmet*

Operacija:	*dodaj_predmet (aktivni_predmet_id)*
Povezan sa:	Use-case: Upis studenta
Preduslovi:	Upis studenta je u toku i započeto dodavanje predmeta
Rezultati:	• kreirana instanca **„upisan_kurs“** • instanca „**upisan_kurs**“ povezana sa instancom **„aktivni predmet“** i sa instancom „**upisana godina**“ • postavljen atribut „**cijena**“ instance „**upisan kurs**“ (na osnovu statusa studenta, DL ili redovan) vrijednost cijena iz tabele predmet se kopira u ovaj atribut) • instanca „**upisan_kurs**“ se pamti dok referent ne potvrdi unos predmeta ili dok ne odustane (ovo se pamti u niz ili neki drugi kontejner jer je moguće upisati više kurseva pri jednom upisu godine)

CO8: Ugovor operacije *zavrsni_dodavanje_predmeta*

Operacija:	*zavrsni_dodavanje_predmet*
Povezan sa:	Use-case: Upis studenta
Preduslovi:	Upis studenta je u toku i započeto dodavanje predmeta
Rezultati:	• sve instance „**upisani kurs**“ se zapisuju u bazu podataka

Korisno je još jednom pojasniti ulogu operacijskih ugovora u cjelokupnom procesu. Slijedeći "mini" dijagram to lijepo opisuje:

Use-case -> Sistemski dijagrami sekvenci -> Operacijski ugovori -> Dizajn objekata Code

Prethodni dijagram ne opisuje slijed kreiranja pojedinih dokumenata, nego samo opisuje koji dokument predstavlja ulazne informacije za druge dokumente. Još treba napomenuti da operacijski ugovori nisu neophodni, ali su dobar alat za učenje i uvođenje studenata u dizajn objekata.

3. Logička arhitektura

Do sada, u okviru "case study" primjera, fokus je bio na analizi zahtjeva i analizi objekata. Ako se prate upute UP-a, do sada bi 10% zahtjeva bilo potpuno istraženo (u "inception" fazi). U prvoj iteraciji faze "elaboracije" započinje malo dublje istraživanje. U nastavku se postepeno iz analize (istraživanje problema) prelazi na dizajn (pronalaženje rješenja u obliku objekata koji međusobno sarađuju).

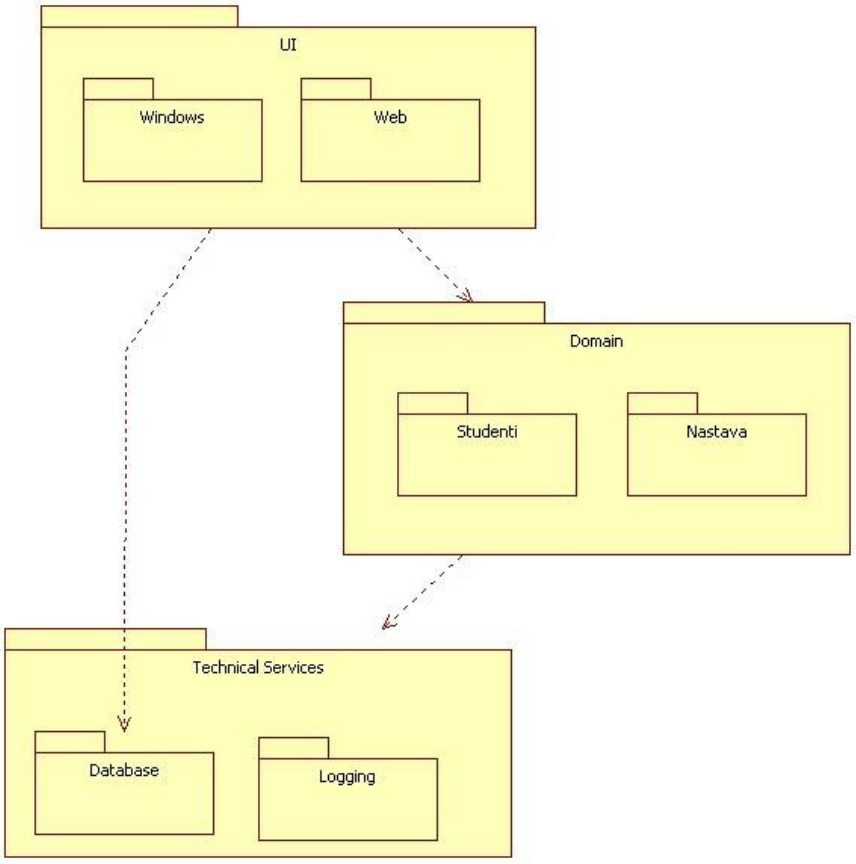
3.1 Uvod

Ovdje će biti prikazan vrlo kratak uvod u logičku arhitekturu, jer je to prilično opširna tema.

Obično se dizajn tipičnog OO sistema zasniva na nekoliko arhitekturalnih slojeva, kao npr. sloj korisničkog interfejsa (UI sloj), sloj aplikacijske logike itd. Osnovni "inputi" za kreiranje logičke arhitekture obično se nalaze se u dodatnoj specifikaciji, gdje se nalaze nefunkcionalni zahtjevi, kao npr. detalji o korisničkom interfejsu, poslovna pravila, zahtjevi o tehničkim servisima (baza podataka i slično), od čega u velikoj mjeri zavisi logička arhitektura sistema.

Logička arhitektura može biti prikazana *UML dijagramima paketa*, kao dio dizajn modela. Sumarni pregled arhitekture može biti dat i u posebnom dokumentu softverske arhitekture.

Slijedeći dijagram predstavlja primjer logičke arhitekture sistema za studentske službe.



Slika 3.1. Logička arhitekturna studentske službe

Na dijagramu su prikazani UML paketi. Paket "UI" sadrži pakete "Windows" i "Web". Ako bi bile implementirane dvije

verzije aplikacije, jedna web orijentisana, a druga kao windows desktop aplikacija, vjerovatno bi klase različitih korisničkih interfejsa razdvojili kao na gornjem dijagramu.

Isprekidana strelica označava "dependency" vezu, tj. ovisnost. Klase iz "UI" paketa ovise o klasama (koriste ih) iz "Domain" paketa. To znači da ako nešto izmijenimo u "Domain" paketu, to može uticati na klase iz UI paketa. Obrnuto ne važi, jer klase iz domain sloja nemaju informaciju o postojanju UI sloja. Tako, ako izmijenimo UI sloj, klase iz "Domain" sloja se ne moraju mijenjati.

3.2 Arhitektura

Logička arhitektura je organizacija softverskih klasa u pakete (ili imenske prostore), podsisteme i slojeve. Ta organizacija označava se kao logička arhitektura jer ne donosi odluke o tome kako će pojedini elementi biti raspoređeni po procesima operativnog sistema, niti po fizičkim računarima u mreži. Takve odluke su dio "deployment" arhitekture (arhitektura razmještaja).

Sloj je grupa klasa, paketa ili podsistema koje imaju logički povezana zaduženja za jedan veći aspekt sistema. Često se koristi termin "kohezivna" zaduženja da bi se naglasila jaka međusobna povezanost tih zaduženja. Slojevi se organizuju tako da viši slojevi (npr. UI sloj) koriste usluge nižih slojeva. Obično slojevi OO sistema uključuju:

- korisničko okruženje ("User Interface" – UI);
- logika aplikacije i softverski objekti koji predstavljaju koncepte iz domene (npr. softverska klasa Student) koji ispunjavaju zahtjeve aplikacije;
- tehnički servisi, klase opšte namjene i podsistemi koji pružaju tehničke servise kao npr. komunikacija sa bazom podataka, log pogrešaka i slično (ovi servisi obično ne zavise od aplikacije i mogu se koristiti u više sistema).

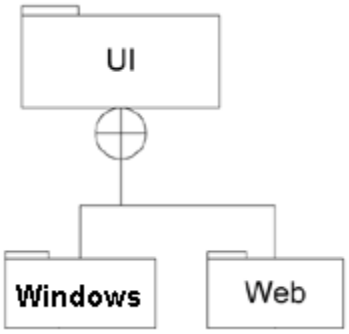
U "strogoj" slojevitoj arhitekturi jedan sloj smije komunicirati samo sa slojem direktno ispod sebe. Iako je takva arhitektura pogodna za dizajn mrežnih protokola, u softverskim sistemima je češća "opuštenija" verzija, gdje jedan sloj može komunicirati i sa donjim slojevima koji nisu direktno "ispod" (na prethodnom dijagramu klase iz UI sloja koriste klase iz paketa "Database").

Iako se OO tehnologija može primijeniti na sve nivoe, u okviru materijala glavni fokus je na logici aplikacije ili "domain" sloju (iako će biti uvodnih diskusija i o drugim slojevima).

3.3 UML paketi

Obično se na UML dijagramima elementi logičke arhitekture (slojevi, podsistemi, imenski prostori...) ilustruju paketima.

Jedan paket može sadržati (grupisati) klase, druge pakete, "use-case" itd. Uobičajeno je da jedan paket sadrži druge (ugniježđene pakete). Takva ugniježđena struktura može biti prikazana kao na prethodnoj slici, kao jedan paket unutar drugog ili korištenjem slijedeće notacije:



Slika 3.2. UML paketi

Kružnica sa znakom X označava da paket "UI" sadrži "Windows" i "Web" pakete.

Uobičajeno je da se na dijagramu paketa prikažu međusobne ovisnosti korištenjem "dependency" veza. Time se na globalnom pregledu može vidjeti povezanost paketa, kao i koji dio sistema zavisi od kog dijela sistema.

UML paket predstavlja imenski prostor, tako da neka klasa npr. Datum može biti definisana u dva paketa. Puno ime klase sadrži imena svih paketa koji je sadrže, odvojeno znakom "::". Na primjer, ako postoji klasa "FrmMeni" u paketu "Windows", puno ime klase je:

UI::Windows::FrmMeni

3.4 Dizajniranje u slojevima

Osnovna ideja korištenja slojeva je jednostavna:

- Logički povezani elementi (klase, paketi...) se organizuju u pojedine slojeve sa jasno odvojenim zaduženjima tako da se niži slojevi brinu o generalnim servisima, a viši slojevi se brinu o pitanjima poslovne logike.
- Objekti iz viših slojeva šalju poruke objektima iz nižih slojeva (obrnuto se izbjegava).

Korištenje slojevite arhitekture pomaže u rješavanju nekoliko problema:

- Sprječava da izmjene u kodu zahtijevaju izmjene koda u mnogim drugim dijelovima sistema
- Sprječava miješanje logike aplikacije sa korisničkim okruženjem, čime se onemogućava korištenje drugog okruženja ili izmještanje logike aplikacije na poseban računar.
- Sprječava miješanje tehničkih servisa sa poslovnom logikom ili logikom aplikacije (što umanjuje mogućnost ponovnog korištenja koda)
- Omogućava odvajanje dijelova aplikacije po zaduženjima što olakšava podjelu posla na različite programere.

Pored toga, prednosti podjele u slojeve su slijedeće:

- Kompleksan problem se može podijeliti na manje komponente
- Pojedini slojevi se mogu potpuno zamijeniti novom implementacijom (često to nije moguće za niže slojeve, ali jeste za UI sloj i sloj logike aplikacije.
- Donji slojevi sadrže funkcije koje se mogu ponovo iskoristiti
- Neki slojevi mogu biti u distribuiranom obliku (domain i tehnički servisi)
- Logička segmentacija pospješuje mogućnost razvoja u timovima

Zaduženja objekata u okviru jednog sloja bi trebala biti veoma povezana. Ne bi trebalo miješati zaduženja iz drugih slojeva. Na primjer, objekti iz UI sloja bi se trebali fokusirati na pitanja korisničkog okruženja (kreiranje prozora, klikovi miša, događaji sa tastature). Objekti iz sloja aplikacijske logike (ili "domain" sloja) bi se trebali fokusirati na aplikacijsku logiku (ako se radi o sistemu studentske službe ta zaduženja bi bila upisi semestara, prijave ispita i slično).

Objekti iz UI sloja se ne bi trebali baviti pitanjima logike aplikacije. Na primjer, objekat klase Form ne bi trebao sadržavati logiku za upis semestra ili prijavu ispita. Sa druge strane, objekti klase Student ne bi trebali voditi računa o obradi događaja kao što su klikovi miša ili događaji na tastaturi. Time bi se narušilo jasno razdvajanje zaduženja i kompaktnost slojeva, a to su osnovni principi slojevite arhitekture.

Većina popularnih objektno orijentisanih programskih jezika imaju podršku za pakete. U C# i C++ programskom jeziku paketi se nazivaju imenski prostori ("namespace"). Java programski jezik koristi naziv "paket".

3.5 Domain sloj i Domain model

Softverski sistem obično posjeduje UI logiku (korisnički interfejs, forme, kućice za unos teksta, komandna dugmad) i logiku aplikacije (upis semestra, ovjera semestra itd.). Ključno pitanje je:

Kako dizajnirati logiku aplikacije korištenjem objekata?

Mogli bi kreirati jednu klasu XYZ i staviti sve metode, za kompletnu zahtijevanu logiku u tu jednu klasu. Tehnički gledano, to bi i moglo funkcionisati, ali to nije preporučeni pristup OO programiranju jer se time gubi osnovna prednost OO programiranja – smanjivanje reprezentacijskog jaza (a i održavanje takvog sistema bi bila prava noćna mora).

Šta je onda preporučeni pristup? Preporučuje se kreiranje softverskih objekata sa nazivima i informacijama koje nose po uzoru na stvarni svijet i zatim dodjela zaduženja tim objektima. Na primjer, u realnom okruženju studentske službe postoje studenti, nastavno osoblje, predmeti, ocjene... U softveru, po OO principu, kreiraju se klase Student, Nastavnik, Predmet... Takvi objekti, kreirani po uzoru na koncepte iz stvarnog svijeta, nazivaju se "domain" objekti. Oni predstavljaju stvari iz domene problema, i posjeduju odgovarajući logiku aplikacije (ili poslovnu logiku). Na primjer, objekat klase Student ima mogućnost upisa i ovjere semestra.

Dizajniranje objekata na takav način vodi kreiranju grupe objekata koju nazivamo "domain" sloj (u slojevitoj arhitekturi). Taj sloj sadrži "domain" objekte koji su zaduženi za obavljanje ključnih poslova vezanih za logiku aplikacije.

Bitno je razgraničiti pojmove "domain" model i "domain" sloj. "Domain" model je produkt nastao u analizi objekata i služi kao inspiracija za davanje imena klasama iz "domain" sloja pri dizajnu objekata. "Domain" sloj je dio softvera, a "domain" model je dio analize iz konceptualne perspektive. Kreiranjem "domain" sloja na način da se inspiracija traži u "domain" modelu postiže se smanjivanje *reprezentacijskog procjepa* između stvarnog svijeta i dizajna softvera.

3.6 Princip odvajanja modela od UI sloja

Postavlja se pitanje kakav odnos ostali paketi imaju sa UI slojem.

Princip odvajanja modela od UI sloja (Model-View principle) se izražava kroz slijedeće dvije tačke:

1. Ne bi se trebali direktno povezivati objekti koji ne pripadaju UI sloju sa UI objektima. Na primjer objekat Student ne bi trebao imati referencu na Form objekat ili TextBox. Zašto? Zato što su Form ili TextBox objekti vezani za

konkretnu aplikaciju, a u idealnom slučaju bi se objekti iz "domain" sloja mogli koristiti u više aplikacija sa različitim korisničkim interfejsom.

2. UI objekti ne bi trebali imati zaduženja vezana za poslovnu logiku. UI objekti bi jedino trebali incijalirati druge UI elemente, obrađivati UI događaje i prosljeđivati zahtjeve za izvršavanjem zadataka iz poslovne logike objektima iz domain sloja.

Gore izneseni princip predstavlja osnovu za "Model-View-Controller" (MVC) arhitekturalni pristup. Model je "domain" sloj, "View" je UI sloj, a "Controller" predstavljaju objekti koji primaju sistemske događaje od UI sloja i koordiniraju izmjene na modelu. Obično kontrolere smještamo u zaseban, aplikacijski sloj.

Motivacija za Model-View razdvajanje uključuje:

- kreiranje kohezivnog modela koji se fokusira na procese iz domene problema, a ne na korisnički interfejs;
- omogućavanje razvijanja modela neovisno od UI sloja;
- minimiziranje uticaja promjena zahtjeva vezanim za interfejs na "domain" sloj;
- omogućavanje razvoja novih korisničkih okruženja koji se lako mogu spojiti na postojeći "domain" sloj (bez izmjena na "domain" sloju);
- omogućavanje istovremenih pogleda na isti objekat iz domain sloja (npr. grafikon i tabelarni prikaz isti podataka);
- dozvoljavanje izvršavanja "domain" sloja neovisno o korisničkom interfejsu, npr. u "message-processing" sistemu.
- jednostavno prebacivanje "domain" sloja u drugo korisničko okruženje.

4. Dizajn objekata

Nakon izrade sistemskih dijagrama sekvenci i operacijskih ugovora pristupa se dizajniranju objekata. Objekti dizajnirani u okviru ove aktivnosti direktno se mogu pretvoriti u izvorni kod u nekom od objektno orijentiranih programskih jezika.

Na koji način se dizajniraju objekti? Moguća su najmanje tri načina:

1. **Kodiranje**. Dizajn objekata se kreira za vrijeme kodiranja, eventualno uz tehnike kao što su "refactoring". Može se reći da se na ovaj način "mentalni" model pretvara u kod.
2. **Crtanje, zatim kodiranje**. Prvo se crta jedan dio UML dijagrama na pločama ili nekim od UML CASE alata, zatim se u nekom integrisanom razvojnom okruženju (IDE) kao što su VisualStudio ili Eclipse piše izvorni kod na osnovu tih dijagrama
3. **Samo crtanje**. Prvo se crtaju UML dijagrami u nekom UML CASE alatu, a zatim alat na neki način generiše kompletan kod na osnovu UML dijagrama. Međutim, ne postoji alat koji može generisati kompletnu logiku aplikacije samo na osnovu UML crteža. Postoje alati u kojima je moguće "prikačiti" dijelove programskog koda uz UML dijagrame, tako da "samo crtanje" i nije posve korektan način za ovu tehniku.

Najpopularniji način dizajniranja objekata je "crtanje, zatim kodiranje". Pri tome "crtanje" mora biti vrijedno truda, tj. UML dijagrami ne bi smjeli biti svrha sami sebi. U nastavku će biti data uputstva upravo za ovaj način dizajniranja objekata.

4.1 UML kao alat za crtanje

Postoje tri načina na koji ljudi koriste UML

- **UML kao skica** - neformalni nekompletni dijagrami, često rukom skicirani na ploči, koji se kreiraju da bi se istražio teži problem ili dizajniralo rješenje problema;
- **UML kao detaljni nacrt** – relativno detaljni UML dijagrami koji mogu služiti za: 1. detaljno razumijevanje postojećeg sistema ili dijela sistema (inženjering unazad, "reverse-engineering"), 2. za generisanje izvornog koda (inženjering unaprijed, "forward-engineering");
- **UML kao programski jezik** – generisanje kompletne izvršne specifikacije sistema korištenjem UML dijagrama, na osnovu kojih se, automatski, može generisati kompletan izvorni kod (ovakav pristup je još u razvoju, tj. ne postoji alat koji može generisati kompletan kompleksniji sistem).

U okviru prethodnih poglavlja bilo je govora o principima "agilnog" modeliranja. Jedan od principa je reduciranje crtanja dijagrama i modela. Ideja je da dijagrami i model služe kao alat za komunikaciju i razumijevanje, a ne za dokumentovanje (mada je dokumentovanje jednostavno korištenjem digitalnih foto aparata, ako se dijagrami rade na tablama za crtanje). Može se reći da "agilno" modeliranje preferira princip kreiranja UML dijagrame kao skica. Međutim, ako je dostupan alat kojim se može raditi reverzni inženjering, moguće je dobiti detaljne nacrte koda koji nastane nakon UML skica:

UML skice -> Izvorni kod -> Reverzni inženjering u UML nacrte.

Također, agilno modeliranje uključuje paralelno kreiranje nekoliko modela. Na primjer, pet minuta na interakcijskim dijagramima, zatim pet minuta na odgovarajućim dijagramima klasa.

Korištenje UML CASE alata kao što su "Rational Rose", "Power Designer", "Enterprise Architect" itd. može pomoći, naročito ako se UML koristi za kreiranje nacrta, generisanje koda ili reverzni inženjering. Postoji veliki broj UML CASE alata, kako komercijalnih, tako i "open source" alata. Pri izboru alata može se voditi slijedećim uputstvima:

- Dobro je izabrati UML CASE alat koji se integriše u IDE okruženje u kom se radi i kodiranje (npr. alat koji se integriše u Visual Studio ili Eclipse).
- Dobro je izabrati alat koji može uraditi reverzni inžinjering, ali ne samo dijagrama klasa nego i interakcijskih dijagrama (ova osobina je rijetka, ali može biti veoma korisna za analizu strukture programa).
- Dobro je izabrati alat koji može reverznim inžinjeringom ažurirati postojeći model, a isto tako, pri generisanju koda, da može ažurirati datoteke sa izvornim kodom, bez gubljenja dijela koda koje je napisao programer ("round-trip" inžinjering, također jedna od osobina koju imaju samo rijetki CASE alati).

Postavlja se pitanje: Koliko vremena potrošiti na izradi UML dijagrama? Ukoliko se razvoj radi u iteracijama koje traju 3 sedmice, može se provesti par sati, najviše jedan dan, na početku iteracije, crtajući dijagrame na tabli ili korištenjem UML CASE alata i to za kompleksne dijelove objektnog dizajna. Nakon toga je dobro stati i uslikati digitalnim fotoaparatom crteže (ako se radi na tabli) odnosno, odštampati dijagrame iz UML CASE alata i preći na kodiranje. Ako se koristi UML CASE alat može se generisati izvorni kod koji će služiti kao polazna tačka pri daljem programiranju. Inače, UML dijagrami služe kao polazna tačka i inspiracija za kreiranje izvornog koda. Konačni dizajn će sigurno odstupati od UML dijagrama jer će biti poboljšan i modifikovan pri kodiranju. Zato, ako se želi UML koristiti kao nacrt i za dokumentovanje sistema, nakon izrade koda potrebno je ažurirati UML dijagrame. Idealno bi bilo da CASE alat to može automatski uraditi ("round-trip" inženjering). Ukoliko se UML koristi samo za skiciranje, onda takvo "povratno" ažuriranje UML dijagrama na osnovu izmjena dizajna u kodu nije neophodno.

4.2 Statički i dinamički dijagrami

Postoje dvije vrste UML dijagrama: statički i dinamički. Preciznije, to su dijagrami koji opisuju statičku strukturu sistema (npr. dijagram klasa) i dijagrami koji opisuju dinamiku, ponašanje sistema (npr. dijagrami sekvenci i

kolaboracijski dijagrami). Statički dijagrami pomažu pri kreiranju paketa, kostura klasa (klasa sa metodama bez implementacije). Dijagrami koji prikazuju dinamiku sistema su interesantniji, ali ih je teže kreirati i pomažu pri implementaciji metoda, tj. pisanju tijela metoda.

Postoji veza između dinamičkih i statičkih dijagrama, pa se obično, u agilnom modeliranju, statički i dinamički modeli kreiraju paralelno. Obično se jedno (kratko) vrijeme potroši na kreiranje interakcijskih dijagrama (dinamičkog modela), a na osnovu njih se zatim kreiraju odgovarajući dijagrami klasa (statička struktura). Proces se ponavlja za pojedine operacije koje se modeliraju.

4.3 Modeliranje ponašanja sistema (dinamički model)

Osobe koje nemaju previše iskustva u UML-u obično misle da je najbitniji statički pogled na sistem (dijagrami klasa). U stvari, najinteresantniji rad, sa najviše izazova je upravo rad na kreiranju dinamičkog modela. Za vrijeme modeliranja ponašanja sistema rješava se ključna stvar, a to je način na koji objekti "surađuju" preko poruka i metoda.

Za vrijeme dinamičkog modeliranja primjenjuje se tehnika "dizajn vođen odgovornostima" (responsibility-driven design, RDD). Najvažnija "vještina" pri modeliranju je upravo primjena RDD-a za dizajniranje objekata.

Crtanje UML dijagrama je samo način na koji se bilježe donesene odluke vezane za dizajn. Tačno poznavanje svih notacijskih finesa UML-a nije esencijalna vještina.

Objektni dizajn, minimalno, zahtijeva poznavanje:

- principa dodjele odgovornosti (RDD-a)

- obrazaca za dizajniranje (design "patterns").

4.4 Ulazi i izlazi procesa dizajniranja objekata

Slijedeća lista sumarno prikazuje uticaj koji imaju prethodno kreirani modeli na dizajn objekata.

- Tekstualni opis slučajeva korištenja ("uce-cases") definišu vidljivo ponašanje koje softverski objekti moraju podržavati. Objekti se dizajniraju tako da realizuju (implementiraju) slučajeve korištenja. U okviru UP-a, OO dizajn se često naziva realizacija slučajeva korištenja ("use-case" realization).
- Dodatna specifikacija definiše ne-funkcionalne ciljeve (npr. internacionalizacija) koje objekti moraju ispuniti.
- Sistemski dijagrami sekvenci identifikuju sistemske poruke koje predstavljaju početne poruke u budućim interakcijskim dijagrama. Te poruke inicijaliziraju kolaboraciju objekata.
- Rječnik pojmova objašnjava detalje o podacima (parametrima sistemskih poruka) koji se prosljeđuju iz UI sloja, validacijske zahtjeve, dozvoljene vrijednosti i sl.
- Ugovori operacija mogu biti dodatak tekstu slučajeva korištenja i pomažu pri razjašnjavanju toga šta objekti moraju postići kao rezultat sistemske operacije.
- Domain model sugeriše neka od imena atributa i softverskih objekata u "domain" sloju softverske arhitekture.

Naravno, nisu svi ovi dokumenti neophodni (u okviru UP-a svi "artifakti" su opcionalni). Kreiraju se samo oni koji su potrebni za otklanjanje rizika i pojašnjavanje problema ili rješenja problema.

Rezultat dizajniranja može biti:

- UML interakcijski dijagrami, dijagrami klasa i paketa za kompleksnije dijelove sistema
- UI skice i prototipi
- Modeli baza podataka (koji se također mogu kreirati korištenjem UML-a, tj. UML profila za modeliranje podataka).
- Skice i prototipi izvještaja

4.5 Dizajn objekata zasnovan na dodjeli odgovornosti (Responsibility-Driven Design)

Uobičajeno je da se o dizajnu softverskih objekata razmišlja u terminima odgovornosti, uloga i saradnje objekata. Dizajn zasnovan na odgovornostima je apstrakcija objekata kojima se dodjeljuju zaduženja za obavljanje određenih radnji. Odgovornost ("responsability") je ugovor ili obaveza neke klase. U osnovi, postoje dva tipa odgovornosti:

- izvršavanje neke radnje (doing)
- poznavanje nekog podatka (knowing)

Odgovornosti objekta vezane za izvršavanje neke radnje uključuju:

- izvršavanje radnji direktno od strane objekta
- iniciranje akcija koje izvršavaju drugi objekti
- kontrolisanje i koordiniranje aktivnosti drugih objekata

Odgovornosti objekta vezane za poznavanje nekog podatka uključuju:

- poznavanje privatnih enkapsuliranih podataka
- poznavanje drugih objekata sa kojim je dati objekat u vezi
- poznavanje stvari koje objekat može izračunari ili izvesti

Odgovornosti se dodjeljuju klasama za vrijeme objektnog dizajna. Na primjer, može se odrediti da je "Student" zadužen za kreiranje "Ocjena" (obavljanje radnje), ili da je "Student" odgovaran da zna prosječu ocjenu u toku studiranja (poznavanje podatka).

Odgovornosti vezane za poznavanje podataka obično su inspirisane domain modelom.

Odgovornost obavljanja neke radnje nije isto što i metoda, ili funkcija članica neke klase. To može biti istinito, ali ne mora. Npr. neka klasa može imati odgovornost za komunikaciju sa bazom podataka, a tu odgovornost realizuje većim brojem funkcija.

Dizajn zasnovan na dodjeli odgovornosti uključuje i *kolaboraciju*, ili saradnju između objekata. Neki objekat svoje zaduženje može ispuniti samostalno, ili može komunicirati sa drugim objektima u cilju ispunjavanja tog zaduženja.

RDD (Responsibility-Driven Design) je metafora vezana za dizajn OO softvera. O softverskim objektima se može razmišljati kao o osobama, koji komuniciraju sa drugim osobama da bi obavili neki posao. RDD je pogled na OO dizajn kao na društvo odgovornih objekata koji međusobno sarađuju.

4.6 GRASP: Metodološki pristup osnovnom OO dizajnu

GRASP principi ili obrasci su alat za učenje koji pomaže u razumijevanju osnova objektnog dizajna. GRASP je skraćenica od "General Responsibility Assignment Software Patterns" (Opšti softverski obrasci za dodjelu odgovornosti).

Šta su to softverski obrasci ("software patterns")? Iskusni OO programeri i dizajneri kroz rad formiraju opšte principe i rješenja kojim se vode pri kreiranju softvera. Ti principi, ako se predstave u struktuiranom formatu opisuju problem i rješenje, i ako im se da ime mogu predstavljati softverski obrazac.

Slijedi pojednostavljen primjer jednog softverskog obrasca:

Naziv obrasca:	Information Expert
Problem:	Koji je osnovni princip za dodjelu odgovornosti objektima?
Rješenje:	Određenu odgovornost dodijelite klasi koja ima informacije za ispunjavanje te odgovornosti.

Jednostano rečeno, softverski obrazac je imenovan, uobičajeni par problem-rješenje, koji se može primijeniti u novim situacijama. Softverski obrazac uključuje i savjete o načinu primjenjivanja obrasca, diskusiju o pozitivnim i negativnim efektima primjene obrasca, varijacije itd.

Jedno od najutjecajnih izdanja sa ovom problematikom je svakako knjiga: "*Design Patterns, Elements of Reusable Object-Oriented Software*" autora *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides*. Knjiga se smatra "biblijom" za softverske obrasce. Često se za obrasce iz ove knjige koristi termin "GoF obrasci" (GoF – Gang of Four). Međutim, to nije knjiga za uvod u OO dizajn. Podrazumijeva se da čitalac vlada OO programiranjem, a i većina primjera je u C++ programskom jeziku.

GRASP principi se koriste za učenje. A za detaljnu diskusiju o softverskim obrascima čitalac se upućuje na GoF knjigu.

Postoji devet GRASP principa. Slijedeća lista daje pet osnovnih principa. Ostali obrasci će biti prikazani nakon Case Study primjera i ovladavanja ovim obrascima.

Naziv:	**Creator**
Problem:	Ko je odgovoran za kreiranje objekata neke klase A?
Rješenje (savjet):	Rješenje (savjet): Klasi B dodijelite odgovornost za kreiranje intanci klase A ako važi jedno od slijedećeg: - B sadrži objekte klase A - B zapisuje objekte klase A - B intenzivno koristi A - B posjeduje inicijalizacijske podatke za A

Naziv:	**Information Expert**
Problem:	Koji je osnovni princip za dodjelu odgovornosti objektima?
Rješenje (savjet):	Određenu odgovornost dodijelite klasi koja ima informacije za ispunjavanje te odgovornosti.

Naziv:	**Low Coupling**

Problem:	Na koji način umanjiti uticaj izmjena na sistem?
Rješenje (savjet):	Odgovornosti dodjeljujte tako da (nepotrebno) povezivanje objekata bude što manje. Ovaj princip koristite pri razmatranju više mogućih alternativa.

Naziv:	**Controller**
Problem:	Koji prvi objekat iza UI sloja prima i koordinira (kontroliše) sistemske operacije?
Rješenje (savjet):	Odgovornosti dodjeljujte tako da (nepotrebno) povezivanje objekata bude što manje. Ovaj princip koristite pri razmatranju više mogućih alternativa.

Naziv:	**High Cohesion**
Problem:	Kako objekte održati fokusiranim, razumljivim, upravljivim a da podržavaju "low coupling"
Rješenje (savjet):	Odgovornosti dodjeljujte tako da se održi kompaktnost objekt, tj. da se objekat brine samo o logički povezanim zadacima. Ovaj princip se također može koristiti pri razmatranju alternativa.