Oblikovanje vođeno domenom

Kosanović, Alen

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Alen Kosanović

OBLIKOVANJE VOĐENO DOMENOM

Diplomski rad

Voditelj rada:
Prof. dr. Sc. Robert Manger

Zagreb, travanj, 2016.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Unutar svakog softverskog projekta pojavljuju se razne zapreke: birokracija, nejasni ciljevi, manjak resursa itd. Pristup oblikovanju je taj koji utječe na to hoće li sustav koji stvaramo biti kompleksan ili ne. Kada kompleksnost izmakne kontroli, članovi razvojnog tima više ne mogu razumjeti sustav, niti ga s lakoćom izmijeniti. Postoje tehnički aspekti kao što su baze podataka ili mreže računala koje sustav mogu učiniti vrlo kompleksnim. Ipak, najznačajniji uzroci kompleksnosti nisu tehnički naravi, već se nalaze u aktivnostima poduzeća ili korisnika - u domeni.

Oblikovanje vođeno domenom je način razmišljanja i postavljanja prioriteta kojim želimo ubrzati razvoj softvera koji u pozadini ima kompleksnu domenu. Takav pristup oblikovanju razvio je Eric Evans u svojoj knjizi istoimenog naslova. Glavne premise su da je **fokus razvoja softvera domena** i da se **oblikovanje kompleksne domene mora bazirati na modelu domene**. Pri tome se nećemo vezati uz određenu metodu razvoja softvera, ali će nam biti iznimno bitno da je **razvoj iterativan** i da su **stručnjaci domene u stalnom kontaktu s razvojnim inženjerima**.

U ovom radu bit će iznesene tehnike oblikovanja i razvoja koje pomažu izgraditi sustav koji zadovoljava gornje premise. Te tehnike će biti ilustrirane na primjeru sustava za generiranje rasporeda sati na fakultetu. Bitno je naglasiti da će fokus biti **oblikovanje sustava**, a ne sama izvedba algoritma. Primjerice, navest ćemo genetski algoritam i njegove komponente, a nećemo objašnjavati kako on točno radi. Kôdovi će biti napisani u programskom jeziku `Java`.

Studijski primjer. Riječ je o sustavu koji služi za oblikovanje i praćenje rasporeda sati na fakultetu. Sustav pamti sve potrebne podatke o nastavnicima, predmetima, satnici predmeta, studentima i dvoranama, te bilježi veze između tih entiteta (na primjer koji nastavnik predaje i koji student sluša određeni predmet). Osim što pretpostavlja očigledna ograničenja (da isti student ili nastavnik ili dvorana ne mogu imati dva različita sata nastave u isto vrijeme), sustav nastoji poštivati dodatna pravila koja nisu obavezna ali ih treba nastojati poštivati (na primjer izbjegavanje "rupa" u studentovom rasporedu). Sustav stvara polaznu verziju rasporeda i iscrtava je u obliku tablica za nastavnika, dvoranu ili studenta. Satničar može premještati pojedine nastavne sate, no sustav to dozvoljava samo ako se time ne krše ograničenja.

Poglavlje 1

Model domene

U ovom poglavlju uvode se centralni pojmovi za *oblikovanje vođeno domenom*, objasnit ćemo kako dolazimo do modela koji sadrže duboko znanje o domeni, definirat ćemo *sveprisutni jezik* i objasniti njegove ulogu u *oblikovanju vođenom domenom* te ćemo objasniti povezanost modela i implementacije.

1.1 Domena i model

Svaki softver izvodi neku aktivnosti ili predstavlja neku korist svojem korisniku. To područje rada kojim se program bavi zove se **domena** softvera. Za razvoj aplikacije moramo biti upućeni u dijelove poslovne domene, koja je često kompleksna i o kojoj u početku ne znamo niša. Jedan od glavnih izvora znanja o domeni su nam ljudi čije je područje rada domena naše aplikacije. Te ljude nazivamo **stručnjacima domene**. Oni predstavljaju glavni (ali ne i jedini) izvor znanja potreban za razvijanje aplikacije.

Znanje o domeni samo po sebi nije dovoljno za razvoj sustava. Domena je često veoma kompleksna i duboka te sadrži mnoštvo informacija koje nisu potrebne. Kako bismo strukturirali to znanje, koristimo se modeliranjem. **Model** općenito predstavlja pojednostavljen i strukturiran skup znanja o nekom području. U svijetu softverskog oblikovanja **model** označava sustav apstrakcija koje opisuju odabrane aspekte domene koje se koristi za rješavanje povezanih problema domene.[1] Odnos između modela i domene je isti kao između karte i teritorija - karta je pojednostavljenje teritorija. U svijetu softverskog oblikovanja 'karta' je skup dijagrama, dokumenata, kodova ili pseudokôdova kojima ilustriramo model.

[1] U nekim literarnim referencama termin model odnosi se na pojednostavljenu apstrakciju sustava. Kod nas se ne modelira sustav, već domena.

Model predstavlja jedan od centralnih pojmova *oblikovanja vođenog domenom* i bitno je da u potpunosti razumijemo njegovu važnost.

1. *Model sadrži strukturirano znanje o domeni.*
2. *Temelj je zajedničkog jezika u timu.*
3. *Diktira oblikovanje sustava.*

Bitno je napomenuti da **model nije skup dijagrama**. Problem koji se javlja kad ljudi žele čitav model i dizajn izraziti dijagramima je taj da su ti dijagrami precjeloviti. Objekti sadrže previše informacija i od silnih detalja teško je uočiti bit dijagrama. Dijagrami nam pomažu u lakšoj vizualizaciji dijela modela, ali nisu jedini alati koji služe toj svrsi. Ponekad je i običan kôd dovoljan.

1.2 Procesiranje znanja

Domena je često golema i kompleksna. Stručnjaci domene koriste razne vrste internog žargona i termina koji nam otežavaju shvaćanje domene. Umijeće jednog projektanta je iz tog znanja, nizom razgovora i 'brainstorming' tehnika izvući i istaknuti bitno, a izostaviti nebitno. Takav proces **procesiranja znanja** nije posao za jednu osobu već za niz ljudi, od stručnjaka za domenu, razvojnih inženjera pa sve do korisnika.

U zastarjelom *modelu vodopada*, razvoj aplikacije odvija se u nekoliko faza u kojoj kraj jedne faze predstavlja početak iduće. Sav teret modeliranja sustava leži na analitičarima te modelu nedostaje povratna informacija od razvojnog tima. S druge strane, ako razvijamo softver samo da zadovolji funkcionalnost trenutnog inkrementa (*model inkrementalnog razvoja*), akumuliranog znanja nema i ne možemo se nositi s kompleksnošću domene.

Pravilan tijek modeliranja kreće jednostavnim razgovorom sa stručnjacima domene u kojem se identificiraju osnovni termini koji su uglavnom subjekti i predikati. S određenom količinom znanja možemo formulirati prve skice i dijagrame kako bismo dobili povratnu informaciju od stručnjaka domene jesmo li na 'plodnom tlu'. Prve skice su grube i neprecizne, model vjerojatno nije kao što bi ih stručnjak zamislio, ali svejedno predstavljaju nekakvo strukturirano znanje i obogaćuju naše znanje i naš poslovni jezik. Bitno je slušati stručnjake i vidjeti kako možemo produbiti znanje i obogatiti model.

U jednom trenutku ćemo početi s oblikovanjem i implementacijom. Razvojni tim će koristiti znanje akumulirano u modelu pri razvijanju prvih funkcionalnosti. U razvoju će se vjerojatno pojaviti poteškoće s postojećim modelom zbog čega će započeti novi krug razgovora sa stručnjakom kako bi se razjasnile nejasnoće. U novom krugu razgovora će doći do novih spoznaja. Sukladno tome model će se izmijeniti tako da reflektira novo znanje, nakon čega započinjemo s refaktoriranjem kôda.

Takvim naizmjeničnim razvojem i razgovorom sa stručnjacima neprestano učimo o domeni te razvijamo kvalitetniji model dubljeg znanja. Sukladno tome, naš dizajn, koji reflektira model, postaje sve bolji i robusniji. Razumijevanje sustava je bolje, izmjene nad sustavom su jednostavnije te je razvoj brži!

1.3 Sveprisutni jezik

U komunikaciji između stručnjaka domene i programera lagano je ustanoviti da obojica pričaju dva *različita jezika*. Dok stručnjak domene koristi termine i žargone domene, programer koristi tehničke termine tehnologije razvoja. Čak i dva programera mogu koristiti isti termin za dva različita pojma ili različite termine za isti pojam. Takve razlike u jeziku mogu dovesti do nerazumijevanja i nesporazuma.

Kako bismo izbjegli nesporazume, bitno je razviti zajednički jezik, tzv. **sveprisutni jezik**, koji bi se trebao koristiti između stručnjaka domene, projektanta i programera. Razvojni tim dužan je usvojiti taj jezik te ga koristiti u modelima, dijagramima, tekstovima, pa čak i u usmenom govoru! Vježbajući sveprisutni jezik lakše je uočiti potencijalne greške u modelu. Prisjetimo se da model treba biti temelj sveprisutnog jezika. Ako se jezikom koji model nalaže ne može nešto izraziti, onda vrlo vjerojatno imamo grešku u modelu. Potrebno je igrati se s alternativnim nazivima i vidjeti koji nam daje lakšu mogućnost izražavanja. Nakon što učinimo promjenu u sveprisutnom jeziku, potrebno je tu promjenu propagirati i u model.

S druge strane, treba u komunikaciji sa stručnjacima domene vidjeti postoje li nekonzistentnosti u našem modelu. Primjerice, ako stručnjak domene uporno koristi jedan izraz umjesto onoga kojeg smo mi usvojili ili ako u našem modelu ili jeziku uopće ne postoji neki termin, lako je moguće da nismo uočili bitan koncept domene.

U kasnijim poglavljima ćemo se neprestano vraćati na koncept *sveprisutnog jezika*. Svi gradivni elementi modela će postati dio sveprisutnog jezika. Poteškoće korištenja jezika će nam ukazivati na mogućnosti za refaktoriranje. Svi slojevi koje uvedemo i druge odluke strateškog dizajna postat će dio sveprisutnog jezika.

1.4 Povezivanje modela i implementacije

Jedna od glavnih osobitosti *oblikovanja vođenog domenom* jest da su model domene i dizajn sustava usko povezani. Takav pristup oblikovanju zove se **oblikovanje vođeno modelom**. Želimo imati dizajn sustava koji *vjerno* odgovara domeni tj. modelu domene. Ako *dizajn* odgovara *modelu*, vjerojatnije je da gradimo korektan sustav. Dizajn koji ne sadrži osnovne koncepte domene obavlja korisne stvari, a da ne sadrži znanje *kako* ih obavlja.

Neke metode razvoja softvera sadrže koncept modela domene koji nastaje kao produkt analize, tzv. *model analize*. On je jednostavan, daje globalnu sliku sustava i strukturiran je u terminima dotične poslovne domene kako bi bio razumljiv korisnicima. U oblikovanju se taj model profinjuje do tzv. *modela sustava* koji sadrži implementacijske detalje. Pogreška je na analizu i oblikovanje gledati kao na dvije odvojene aktivnosti, na kojima rade dvije različite skupine ljudi. Naime, velik dio otkrivanja i učenje događa se upravo tijekom oblikovanja i implementacije sustava, tako da je potrebno napraviti ponovnu analizu. Ta se druga analiza odvija na manje discipliniran način jer u nju nisu uključeni isti ljudi (npr. stručnjaci domene) koji su bili uključeni u analizi. Kao posljedicu toga dobivamo dizajn sustava koji ne odgovara domeni, pa više ne možemo biti sigurni niti u korektnost svog softvera! S druge strane, ako odlučimo ostati vjerni modelu analize, bit ćemo prisiljeni raditi kompleksna preslikavanja između modela analize i sustava, preslikavanja koja su teška za shvaćanje, a još teža za održavanje prilikom promjene dizajna.

Oblikovanje vođeno domenom odbacuje dihotomiju analize i oblikovanja kako bismo došli do jednog modela kojeg koristimo u obje svrhe. Zato prvo treba dizajnirati dio sustava tako da odražava model na doslovan način. Ako nam takav dizajna zadaje poteškoće prilikom implementacije, potrebno je izmijeniti model tako da se može prirodnije implementirati kao softver. Time kôd koji pišemo postaje odraz domene. Izmjena u kôdu može postati i izmjena u modelu. U kasnijim poglavljima ćemo prikazati gradivne elemente i tehnike kojima pospješujemo odražavanje domene u našem kôdu.

U ovom poglavlju smo upravo istaknuli srž *oblikovanja vođenog domenom*. U idućim poglavljima ćemo uglavnom govoriti o gradivnim elementima i tehnikama kojima ćemo dobiti takav kôd koji vjerno odražava model tj. domenu. S obzirom na to da će gradivni elementi i tehnike biti usko povezani s programskom paradigmom koju izaberemo, bitno je prodiskutirati ulogu programske paradigme u odnosu modela i dizajna.

Odabir programske paradigme

Kako bismo omogućili usku korespodenciju između modela i implementacije, moramo koristiti paradgimu koja nam omogućava stvaranje direktnih analogona između modela i dizajna. U ovom radu to će biti **objektno-orijentirana paradigma (OOP)**. Glavni razlozi za odabir *objektno-orijentirane paradigme* su:

- *Objekti su sveprisutni u poslovnoj domeni.* U poslovnom svijetu uglavnom baratamo sa stvarima koje možemo fizički zamisliti. Korištenjem npr. proceduralne paradigme bilo bi teško modelirati predavanje, profesora, studenta i definirati njihove odnose. Objektima možemo definirati sve entitete, relacije među njima ćemo realizirati pomoću asocijacija, a operacije koje obavljamo nad njima pomoću servisa.

- *Objekte si mogu s lakoćom predočiti i programeri i stručnjaci domene.* U pozadini se sve što napišemo odvija kao niz instrukcija prema računalu. Zamišljanje našeg sustava na takav način neće nam pomoći da s lakoćom razumijemo što naš sustav radi. S obzirom na to da nam je komunikacija sa stručnjacima domene bitna, moramo koristiti modele koje i oni mogu razumjeti. Objekti su vrlo intuitivni i razvojnim inženjerima i stručnjacima tako da se postavljaju kao logičan izbor u modeliranju.

- *Oko OOP postoji razvijena zajednica programera, programskih alata i sakupljenog znanja.* Ako se odlučimo razvijati sustav u jeziku koji podržava OOP, na poznatom smo terenu! OOP se koristi već godinama, postoji mnoštvo oblikovnih obrazaca i rješenja za česte probleme te postoje brojni alati bazirani na OOP koji nam olakšavaju i ubrzavaju razvoj softvera.

Za neke matematičke probleme funkcijska, logička ili relacijska paradigma može biti pogodnija. Pravila *oblikovanja vođenog domenom* vrijede i za tu paradigmu. I dalje je potrebno imati dobar model koji vjerno opisuje domenu i koji je čvrsto povezan uz implementaciju. Ali s obzirom na to da ćemo za većinu projekata koristiti OOP, u ostatku rada ćemo koristiti *objektno-orijentiranu paradigmu* kao polazište za modeliranje i oblikovanje.

Poglavlje 2

Gradivni blokovi

Sada kad smo se opredijelili za *objektno-orijentiranu paradigmu*, možemo početi navoditi osnovne gradivne blokove s kojima postižemo usklađenost modela i implementacije. Osnovni gradivni objekti su:

- *Entiteti.* Objekti koji su određeni svojim identitetom.
- *Vrijednosni objekti.* Objekti koji nemaju konceptualni identitet već su određeni skupom svojih atributa.
- *Servisi.* Operacije dostupne putem sučelja koje nemaju enkapsulirano stanje.
- *Moduli (paketi).* Logičke cjeline koje se sastoje od entiteta, vrijednosnih objekata i servisa. Odražavaju dio poslovne domene.
- *Agregati.* Nakupine asociranih objekata koji predstavljaju konzistentnu cjelinu kojoj pristupamo preko jednog objekta unutar agregata - *korijena*.
- *Tvornice.* Mehanizmi koji enkapsuliraju stvaranje kompleksnih objekata i agregata.
- *Repozitoriji.* Mehanizmi koji enkapsuliraju spremanje, dohvat i izmjene objekata ili agregata.

2.1 Izoliranje domene

Prilikom izrade softvera, domena, o kojoj čitavo vrijeme govorimo, obično je samo malen, ali izuzetno bitan dio našeg softvera. Naš softver se obično sastoji i od korisničkog grafičkog sučelja, baze podataka, mrežne komunikacije itd. Česta praksa programera je razbiti kompliciran sustav u niz podsustava koji su organizirani u slojeve gdje viši slojevi koriste funkcionalnosti nižih slojeva. U ovom radu nam nije cilj favorizirati jednu

specifičnu n-slojnu arhitekturu, već istaknuti važnost slojevite arhitekture, tj. važnost da imamo jedan sloj koji je posvećen isključivo domeni.

Najjednostavniji pristup je upite prema bazi podataka 'ugurati' u objekte domene, a ponašanje domene umetnuti u elemente grafičkog sučelja. Iako je to najbrži način implementiranja neke funkcionalnosti, što projekt postaje veći, to nam postaje sve teže dokučiti što naš sustav zapravo radi. Segmenti poslovne domene su ispremiješani s tehničkim pojedinostima sustava. Moramo se spuštati kroz niz pravila poslovne logike pa sve do upita prema bazi kako bi izmijenili samo jedan element grafičkog sučelja. Uz sve tehnologije i svu poslovnu logiku s kojom se moramo nositi, želimo da naš program bude što jednostavniji.

Upravo zato želimo sustav oblikovati u slojeve između kojih postoje **slabe veze**, a unutar kojih postoji **jaka kohezija**. Samo viši slojevi smiju koristiti funkcionalnosti nižih slojeva. Veza između slojeva je jednosmjerna - prema nižim slojevima tj. samo viši slojevi smiju imati reference prema nižim slojevima. Kada niži sloj želi komunicirati s višim slojevima, to činimo mehanizmima kao što su *povratni pozivi* ili *promatrači*. Oslanjanjem na jednosmjernu komunikaciju smanjujemo međuovisnosti slojeva i pojednostavljujemo praćenje toka jedne operacije. U idealnom svijetu želimo da se jednim slojem koristi samo sloj koji je neposredno iznad njega. Što manje ovisnosti sloj ima nad sobom, to ga je jednostavnije izmijeniti poslije.

Autor knjige *Domain driven design* [1] predlaže korištenje 4-slojne arhitekture koja se sastoji od *prezentacijskog sloja*, *sloja primjene*, *sloja domene* i *infrastrukturnog sloja*. Principi *oblikovanja vođenog domenom* mogu se ostvariti i s drugim višeslojnim arhitekturama sve dok postoji *sloj domene* u kojem su koncepti domene odvojeni od ostalih dijelova aplikacije.

Prezentacijski sloj (grafičko sučelje)	Zadužen za prikaz informacija korisniku i za interpretaciju njegovih naredbi.
Sloj primjene	Zaprima podatke i događaje iz prezentacijskog sloja te delegira *sloju domene*. U njemu se ne nalazi poslovna logika domene, ali može imati i stanja o napretku zadatka koje sloj domene obavlja.
Sloj domene (sloj modela)	Odgovoran za predstavljanje koncepata poslovne domene, poslovnih pravila i trenutne poslovne situacije. **Ovaj sloj objedinjuje srž posla softvera**.
Infrastrukturni sloj	Pruža tehničke mogućnosti višim slojevima kao što su slanje poruka, perzistencija podataka itd.

Tablica 2.1: Četveroslojna arhitektura sustava.

2.2 Objekti domene izraženi softverom

U ovom odjeljku bavit ćemo se detaljnije individualnim elementima modela i kako ih oblikovati kako bi podržavale aktivnosti u kasnijim poglavljima. Počet ćemo s *asocijacijama* među objektima te ćemo kasnije reći nešto o tri elementa kojima izražavamo naš model: *entiteti*, *vrijednosni objekti* i *servisi*. *Agregati*, *tvornice* i *repozitoriji* su gradivni objekti kojima ne opisujemo objekte domene već mehanizmi kojima reguliramo njihovo stvaranje i integritet, pa ćemo ih opisati odvojeno u odjeljku 2.3.

Asocijacije

Prije nego što krenemo na gradivne objekte, prokomentirat ćemo veze između objekata tj. *asocijacije*. Poteškoće između usklađivanja modeliranja i implementacije javljaju se kod asocijacija među objektima. Asocijacija između *nastavnika* i njegovog *predavanja* označava dvije stvari. S jedne strane označava bitan odnos dvaju koncepta domene, a s druge strane označava referenciranje (npr. u Javi) između dva objekta. Pravi izazov je odabrati takvu implementaciju asocijacije koja najviše govori o odnosu objekata u domeni. Primjerice, jedno predavanje predaje jedan predavač, a u jednoj dvorani se odvija mnoštvo predavanja. Hoćemo li objekt *predavanja* držati referencu na jednu dvoranu ili će objekt *dvorana* sadržavati kolekciju predavanja?

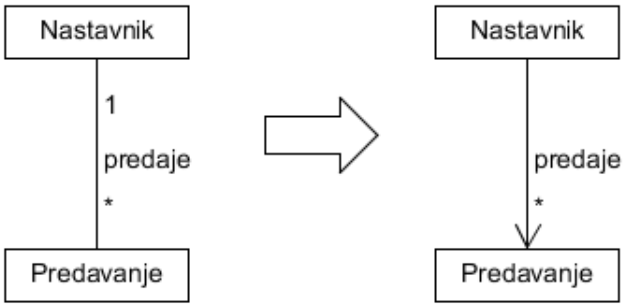
Postoje tri načina na koje možemo asocijaciju učiniti prohodnijom.

1. Uvođenjem smjera prolaženja.
2. Dodavanjem atributa.
3. Eliminiranjem nepotrebnih asocijacija.

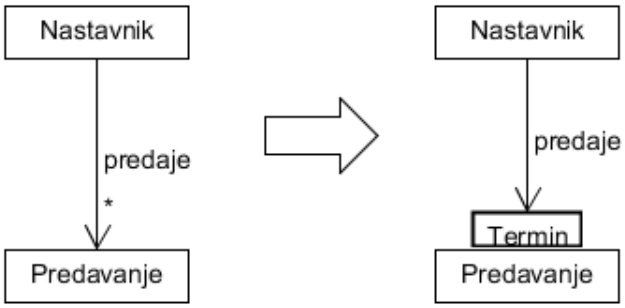
Na primjeru nastavnika i predavanja bi se trebali zapitati koji nam je smjer logičniji. Naša aplikacija bi trebala omogućiti korisniku da generira raspored za nastavnike, tako da se čini logičnijim preko nastavnika doći do predavanja koja predaje. Uz to, treba provjeravati da nastavnik nema dva predavanja u isto vrijeme. Smjer u kojem iz nastavnika dolazimo do predavanja se (u ovom trenutku) čini logičnijim.

Iako smo ustanovili da je smjer prolaženja od nastavnika do predavanja logičniji, nismo uračunali jedan bitan koncept domene, a to je *vrijeme* održavanje predavanja tj, *termin*. Kada bismo predavanju pridružili atribut *termin* kad se predavanje održava, dobili bismo 1-na-1 odnos između predavanja s atributom termina i nastavnika koji ga predaje. Takav odnos vrijedi uz pretpostavku da jedan nastavnik ne može biti na dva mjesta u isto vrijeme (dosta logična pretpostavka).

Ovo i dalje nije prava slika modela rasporeda sati. Ovakve prijelaze možemo tretirati kao razvijanje dubljeg modela sakupljanjem novih spoznaja.



Slika 2.1: Uvođenje smjera prolaženja.



Slika 2.2: Dodavanje atributa.

Objekt *termin* će biti ključan objekt u našoj aplikaciji. On će predstavljati osnovni gradivni element rasporeda sati. *Termin* možemo zamisliti kao 'kućicu' na rasporedu sati u kojoj piše 'Matematička analiza 1 (P) [A-LJ], Ivan Horvat, 003' i koja se pruža kroz neki vremenski interval.

Unatoč svemu, krajnje pojednostavljenje asocijacija jest eliminirati ih općenito ako uočimo da njihova veza nije nužna.

Entiteti

Neki objekti našeg modela nisu određeni primarno skupom svojih atributa, već oni samo opisuju stanje objekta u nekom trenutku njegovog života. Za takav objekt je bitno da posjeduje jedinstven identitet koji ga razlikuje od drugog objekta čak iako imaju iste atribute. Takve objekte, koji su određeni primarno svojim identitetom nazivamo **entitetima**.

Objektno-orijentirani jezici danas posjeduju vlastiti mehanizam raspoznavanja identiteta u kojem su dva objekta ista ako se nalaze na istom mjestu u memoriji. Takav mehanizam nije dovoljno prikladan u svrhu modeliranja. Pohranom i dohvatom objekta iz baze podataka ili slanjem preko mreže taj se identitet gubi te više nemamo mehanizam razlikovanja. U tu svrhu potrebno je definirati **jedinstveni ključ** objekta kao kombinaciju nekih njegovih atributa koji ga **jednoznačno** određuju. Često nije moguće jednoznačno odrediti objekt nekim njegovim atributima. U tom slučaju, svakoj instanci objekta pridružujemo jedinstven simbol - ID.

U našem primjeru, ime i prezime nisu dovoljan identifikator nekog studenta. Čak i uz informaciju o adresi i dalje ne možemo biti sigurni u jedinstvenost nekog studenta po tim parametrima. Identifikator u kojeg možemo biti sigurno jest JMBAG ili OIB studenta. Bitno je dodati da identifikator za jednu aplikaciju ne mora biti identifikator za drugu! Primjerice, bilo bi suludo studenta identificirati po njegovom telefonskom broju, dok u aplikaciji za jednu telefonsku kompaniju isti telefonski broj može biti dobar kandidat za identifikator!

Vrijednosni objekti

Za razliku od entiteta, neki objekti su u potpunosti određeni svojim atributima. Takvi objekti, koji nam služe u svrhu opisivanja aspekta domene, bez konceptualnog identiteta nazivamo **vrijednosnim objektima**. Za njih je specifično da se predaju kao parametri između objekata, prijelazni su, stvoreni su za operaciju te se nakon toga odbacuju. Mogu referencirati objekt, ali su najčešće atributi entiteta. U tom slučaju, dobra je praksa *vrijednosne objekte* tretirati kao konceptualnu cjelinu. Primjer vrijednosnog objekt je adresa neke osobe. Ona je isključivo određena svojim atributima: ulica, kućni broj, poštanski broj, grad i država. Dvije adrese koje imaju iste atribute zaista jesu iste adrese! Ako jednoj adresi promijenimo jedan od atributa, npr. kućni broj, nije više riječ o istoj adresi! Bilo kakav dodatan identitet adresi je suvišan.

Zašto je korisno vrijednosne objekte tretirati kao konceptualnu cjelinu? Zašto ne bismo mogli sve atribute adrese pridružiti objektu osobe? Adresa je jedan zamisliv, konkretan koncept domene koji može imati svoju ulogu i u drugim dijelovima aplikacije. Osim što bi objekt osobe bio znatno kompliciraniji s dodatnim atributima, upravljanje adresom osobe smo izbacili iz samog objekta osobe. Razmislimo, kada mijenjamo adresu, rijetko kada se to radi samo o jednom podatku npr. o kućnom broju kojeg ćemo promijeniti s `person.setHomeNumber(newHomeNumber)` metodom, već o skupu podataka koje mijenjamo odjedanput metodom `person.setAdress(newAddress)`.

Netko se može zapitati zašto vrijednosnim objektima ne pridružimo identifikator. Takva odluka bi imala neželjene posljedice. Osim što unosimo zbrku u model tretirajući sve

objekte isto, ovakva odluka može naštetiti performansama sustava. Nepotrebno imamo nekolicinu objekata koji su konceptualno identični, koje bismo inače mogli koristiti na više mjesta, npr. dvije osobe mogu imati istu adresu.

Pri dijeljenju vrijednosnih objekata moramo paziti na njihov integritet. Ako dvije osobe imaju iste adrese, promjena adrese jedne osobe ne bi smjela promijeniti adresu druge osobe. Rješenje je da prilikom dohvata / mijenjanja vrijednosnog objekta kopiramo vrijednosni objekt ili da vrijednosni objekt učinimo nepromjenjivim (*immutable*) tj. da nije moguće promijeniti atribute objekta nakon što se jednom taj objekt stvori.

Servisi

Postoje važne akcije i aktivnosti domene koje ne možemo pridružiti niti entitetu, niti vrijednosnom objektu, a da pri tome ne deformiramo usku povezanost modela i objekta. U našem primjeru nalazimo aktivnosti kao što je generiranje rasporeda sati iz postojećih podataka. Tko generira taj raspored? Koristimo objektno-orijentiranu paradigmu, tako da nekom objektu moramo dodijeliti tu dužnost. U aplikaciji ćemo očigledno imati objekt *RasporedSati*, hoće li on imati dužnost da stvori primjerak rasporeda sati? Ako da, onda ćemo unutar objekta *RasporedSati* morati donositi i odluku koji algoritam koristimo, s kojim parametrima, kada se zaustavlja, morat ćemo dohvaćati ulazne podatke (informacije o upisu na predavanja, zauzetosti nastavnika...) te spremati izlazni podatak (raspored sati). To je previše odgovornosti za jedan objekt koji konceptualno samo sadrži informaciju o tome kada i gdje se održava neko predavanje!

Rješenje je to ponašanje držati u jednom objektu kojeg ćemo zvati *servis*. **Servis** je operacija koja nam stoji na raspolaganju kao samostojeće sučelje koje ne enkapsulira nikakva stanja. Pod samostojećim mislimo da ne pripada niti jednom entitetu i objektu, već je objekt sam za sebe koji koristi druge objekte i entitete. Operacija servisa treba biti bez stanja tako da povijest korištenja servisa ne utječe na njegovo buduće ponašanje.

Servisi kao koncept koriste se i u dijelovima sustava koji ne pripadaju sloju domene kao što su sloj primjene i sloj infrastrukture. Servisi unutar sloja primjene su zaduženi za pružanje usluga prezentacijskom sloju, dok *servisi infrastrukture* obavljaju 'tehničke servise' kao što je slanje poruka, e-maila, obavljanje transakcija. *Servisi domene* su upravo posebni po tome što obavljaju akcije i operacije domene i kao takvi trebaju biti dio *sveprisutnog jezika*.

Moduli (paketi)

Kako bi izašli na kraj s kompleksnošću sustava, često trebamo na naš sustav gledati kao na skup manjih dijelova koji čine cjelinu. Ranije smo naveli promatranje arhitekture kao niz slojeva. Sljedeći postupak je dekompozicija tih slojeva na manje konceptualne cjelinu koje

nazivamo **moduli** ili **paketi**. Uz pomoć modula možemo promatrati jednu komponentu sustava odvojeno od ostalih komponenti, zato je bitno da je povezanost između modula slaba. S druge strane, nepovezani fragmenti jedne cjeline mogu izazvati zbunjenost, stoga moduli unutar sebe moraju biti čvrsto povezani.

Osim što moduli, s tehničke strane, predstavljaju grupaciju objekata u cjelinu, bitno je primijetiti da određeni modul treba odgovarati jednoj konceptualnoj cjelini iz domene same i trebali bi biti nazvani tako da daju uvid u odgovarajući aspekt domene. Ukoliko nailazimo na nepodudaranja između modula i modela, potrebno je preraditi model tako da u isto vrijeme, na vjeran način odgovara i domeni i modulu. U slučaju kad moramo raditi kompromise, bolje je imati model sa konceptualnom jasnoćom, čak iako nam kao posljedica toga moduli nisu više labavo povezani.

Korištenjem višeslojne arhitekture često dolazimo do fragmentacije objekata modela. Primjerice, neki poznati aplikacijski okviri podupiru višeslojnu arhitekturu, ali pri tome razdvajaju odgovornosti jednog objekta domene kroz više objekata u raznim slojevima. Obično je riječ o posebnoj vrsti objekata za pristup podacima te o posebnoj vrsti objekata za poslovnu logiku. Česta greška je ta dva objekta, koji odgovaraju istom konceptualnom objektu, smjestiti u dva različita paketa. Tim smo razdvajanjem otežali mogućnost poimanja tih objekata kao jedne cjeline pa tako gubimo povezanost modela i dizajna. U svakom slučaju, trebamo težiti da jedan konceptualni objekt domene bude implementiran u istom modulu.

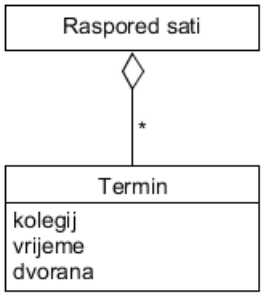
2.3 Životni ciklus objekta domene

Do sada smo obradili ključne objekte kao što su entiteti i vrijednosni objekti. Neki od njih su jednostavni, stvaramo ih konstruktorom, žive kratko u memoriji te ih nakon nekog vremena uništava sakupljač smeća (*Garbage Collector*). Mnogi objekti nisu toliko jednostavni, često s njima asociramo još mnoštvo drugih objekata koji imaju složene međuovisnosti. U ovom poglavlju ćemo se više fokusirati na upravljanje životnim ciklusom takvih kompleksnih objekata te kako paziti na njihov integritet, a da pri tome ne pregazimo elegantnost modela s kompleksnim upravljanjem životnih ciklusa.

Agregati

Problem čuvanja integriteta se pojavljuje kod objekata koji imaju složene asocijacije nad kojima postoje pravila koja stalno trebaju biti ispoštovana - *invarijante*. Kako bi ilustrirali jedan od takvih slučajeva, vratit ćemo se na asocijacije koje smo spomenuli u pododjeljku asocijacija. Spomenuli smo koncept *termina* i rekli da će nam biti od velike važnosti u našem modelu. *Raspored sati* je zapravo skup termina pri čemu termin sadrži informaciju

o tome kada i gdje se održava neko predavanje. Po ovakvoj definiciji, raspored sati za studenta ili za nastavnika postaje podskup *generalnog rasporeda sati*[1].



Slika 2.3: Prikaz rasporeda kao skup termina.

Prisjetimo se da naša aplikacija mora korisniku nuditi mogućnost ručne izmjene rasporeda. Pogledajmo sljedeći primjer u kojem izmjenu rasporeda činimo na najjednostavniji način.

```
public void manuallyChangeTimetable(Integer classId, Time newTime, Integer newClassroomId) {
  Timetable latestTimetable = repository.getLatestTimeTable();
  for (Term term : latestTimetable.getTerms()) {
    if (term.getClassId().equals(classId)) {
      term.setTime(newTime);
      term.setClassroomId(newClassroomId);
    }
  }
  repository.updateTimetable(latestTimetable);
}
```

Kôd 2.1: Jednostavna implementacija zamjene u rasporedu.

Čini se da kôd radi svoj posao, zar ne? Da, samo što ne provjeravamo postoje li preklapanja u terminima. Ako postoje, upravo smo prekršili jednu bitnu *invarijantu* rasporeda sati i spremili nekorektan raspored u bazu! Sljedeće najbrže rješenje je definirati nekakvu vrstu kontrole korektnosti rasporeda prije njegovog spremanja u bazu podataka. Ovakvim pristupom kontrolu integriteta rasporeda sati delegiramo **svim klijentima koji koriste objekt raspored sati**. To znači da će svatko tko koristi objekt raspored sati morati pogledati njegovu implementaciju kako bi znao napisati svoj dio kôda. Takvim pristupom ćemo češće raditi greške i teže će biti predvidjeti posljedice našeg kôda.

Iako se ovaj problem na površini čini tehničkim, njegov korijen se krije u modelu. Rješenje koje se postigne na razini modela će učiniti model razumljivijim te će nas navoditi

[1] U ovom trenutku ćemo u *sveprisutni jezik* ubaciti termin *generalnog rasporeda sati* koji označava raspored sati koji sadrži informacije o svim terminima predavanja na jednom fakultetu.

na potrebne promjene u implementaciji. U modelu trebamo uočiti te nakupine objekata između kojih imamo nekakve invarijante koje moramo poštovati. Takve objekte nazivamo **agregatima**. Svaki agregat sadržava *granicu* i *korijen*. **Granica** definira što se sve nalazi unutar agregata, a **korijen** je dio agregata koji definira pristupnu točku prema agregatu. Jedino on smije biti referenciran izvan agregata, dok objekti unutar granice agregata smiju imati reference jedni prema drugima. Referenca korijena je globalno jedinstvena, dok ostali entiteti u agregatu imaju identitet koji je jedinstven samo na lokalnoj razini, unutar agregata. Ako korisnik želi pristupiti jednom od objekata unutar agregata, to je moguće samo preko korijena i korijen je dužan vratiti samo *kopiju* tog objekta ili objekt koji je nepromjenjiv. Vraćanje reference na taj objekt dalo bi vanjskom korisniku mogućnost narušavanja invarijante. Sukladno tome, samo se korijeni agregata mogu dohvatiti iz baze podataka, a do ostalih podataka možemo doći samo preko korijena.

Uvođenjem agregata osigurali smo integritet naših podataka kroz njihov životni ciklus, ali smo zato dodatno povećali kompleksnost stvaranja potrebnih objekata, odnosno, povećali smo kompleksnost njihovog dohvata iz baze podataka. Objekt sam po sebi može biti kompliciran za stvaranje, a sada dodatno moramo paziti i na pravilno stvaranje objekata unutar agregata. U tom slučaju, potrebno je na neki način odgovornost za stvaranje objekata, odnosno agregata, delegirati nekom drugom objektu koji će od klijenta sakriti njihovu unutarnju strukturu.

Ovaj problem je čest u svijetu programiranja i postoji niz upotrebljivih rješenja u obliku oblikovnih obrazaca stvaranja. Za sada ćemo se fokusirati na *tvornicu*. Razlog tome je što ne želimo ući dublje u temu oblikovnih obrazaca već samo reći o njihovoj važnosti u kontekstu oblikovanja vođenog domenom. *Tvornicu* smo izabrali jer je najkorišteniji oblikovni obrazac za stvaranje te ujedno i najpogodniji za stvaranje agregata koje smo upravo spomenuli. Za neke kompleksne objekte postoje i pogodniji oblikovni obrasci, a neke od njih ćemo spomenuti i kasnije.

Tvornica

Tvornica je mehanizam za enkapsuliranje kompleksne logike stvaranja i mehanizam za apstrahiranje tipa objekta kojeg je kreirao klijent. Naziv tvornica je posuđen iz stvarnog svijeta. Klijenta, koji naručuje auto iz tvornice, zanima samo model kojeg naručuje, a ne proces njegovog stvaranja. U svijetu programiranja, klijenta zanima samo objekt kojeg traži, a ne kako se on stvara niti kojeg je konkretnog tipa.

Ako odgovornost stvaranja prebacimo na klijenta, onda on mora posjedovati znanje i o unutarnjoj strukturi objekta te invarijantama agregata. Time upravo gubimo dobro svojstvo agregata da enkapsuliraju svoje unutarnje objekte. Ako je klijent neki objekt domene, tada on dobiva jednu nepotrebnu odgovornost te ga čvrsto vežemo uz jedan drugi objekt domene ili njegovu konkretnu implementaciju. Kao posljedicu toga dobivamo nezgrapan dizajn. U

slučaju da klijent nije objekt sloja domene, već sloja primjene, situacija postaje još gora time što odgovornosti sloja domene prebacujemo na sloj primjene!

Postoje dvije vrste tvornica: *tvornice entiteta* i *tvornice vrijednosnih objekata*. **Tvornica vrijednosnih objekata** stvara vrijednosne objekte, tako da joj je potrebno pružiti potpun opis kako bi mogla konstruirati sve atribute vrijednosnog objekta. **Tvornica entiteta** stvara entitet, tako da joj je potrebno pružiti samo nužne atribute, kao što su ključevi objekata koji su potrebni za stvaranje.

Postoje dva bitna zahtjeva na tvornicu.

1. *Stvaranje mora biti atomarno.* Tvornica može stvoriti samo objekt ili agregat u kojem su sve invarijante zadovoljene. Ne smije se stvoriti objekt ili agregat u nekonzistentnom stanju. Za tvornicu entiteta to se odnosi na stvaranje čitavog agregata sa svim invarijantama zadovoljenim. Za tvornicu vrijednosnih objekata to znači da su svi atributi postavljeni na svoje konačne vrijednosti.

2. *Tvornica bi trebala podržavati sve željene produkte*. Po potrebi ju treba učiniti apstraktnom tako da nije vezana samo uz jednu vrstu produkta.

Tvornica nam omogućava jednostavnije stvaranje kompleksnih objekata, ali u dosta scenarija nam je dovoljan i konstruktor. Običan javni konstruktor nam je dovoljan u sljedećim uvjetima.

- *Klasa je konkretan tip.* Nije član duboke hijerarhije i nije izvedena klasa kod koje želimo koristiti pogodnosti polimorfizma.

- *Klijent želi imati kontrolu nad implementacijom,* kao način odabira strategije (u našem slučaju algoritma).

- *Klijent već ima pristup svim atributima*.

- *Konstruktor nije kompliciran.*

- *Stvaranje je atomarno.* Kao i kod *tvornice*, ne bi smjelo biti moguće stvoriti objekt koji narušava neku invarijantu.

Repozitorij

Preostao nam je posljednji scenarij stvaranja objekata kada trebamo dohvatiti postojeći objekt koji nam se nalazi u bazi podataka. Konceptualno, riječ je o objektu koji se nalazi u sredini svog životnog ciklusa tako da u ovom slučaju govorimo o *rekonstrukciji*.

Cilj dizajna vođenog domenom jest poboljšati kvalitetu softvera prebacivanjem fokusa na model domene umjesto na tehnologiju koju koristimo. Ako klijentu u sloju domene prepustimo sastavljanje SQL upita, njegovo prosljeđivanje u niže slojeve aplikacije, dohvat rezultata upita te rekonstrukciju objekata, fokus našeg oblikovanja svodi se na tehnologiju, a ne na model. Štoviše, ovakav pristup omogućava klijentu da zaobilazi svojstva modela kao što su agregati i enkapsulacija. Kao posljedica toga, pravila domene počinju se skrivati unutar SQL upita prema bazi ili se kompletno izgube.

Iz tog razloga potrebno je odgovornost pristupa prema bazi prebaciti na globalno vidljiv objekt **repozitorij** koji će od klijenta skrivati implementacijske detalje baze i davati **privid** klijentu **kao da pristupa objektima koji su u memoriji**. Moguće je birati, dodavati, uređivati i brisati objekte iz repozitorija. Klijent pristupa repozitoriju preko upita u kojem su navedeni neki kriteriji, najčešće po vrijednosti njegovih atributa. Osim objektima nekog tipa, klijent može zatražiti razne izračune kao što je količina objekata koji zadovoljavaju neki kriterij.

Svaki objekt prema kojem trebamo globalni pristup mora imati svoj *repozitorij* koji će pružati klijentu privid da pristupa kolekciji objekata u memoriji. *Repozitorij* treba klijentu biti ponuđen kao globalno vidljivo sučelje. Preko njega, klijent može dohvatiti, izmijeniti, dodati i brisati objekte bez potrebe da ima konkretno znanje kako se to izvršava u bazi podataka. Klijenta interesira vrsta operacije koju treba izvršiti i kriterij nad kojim objektom ju treba izvršiti. Za agregate, klijentu treba omogućiti dohvaćanje samo korijena agregata!

Korištenja repozitorija ima nekoliko prednosti.

- Pružaju klijentima jednostavan model za dohvat spremljenih objekata i upravljanje njihovim životnim ciklusima.
- Odvajaju klijenta od implementacije baze podataka.
- Podržavaju jednostavnu zamjenu repozitorija s lažnim tzv. *'dummy'* implementacijama radi potrebe testiranja.

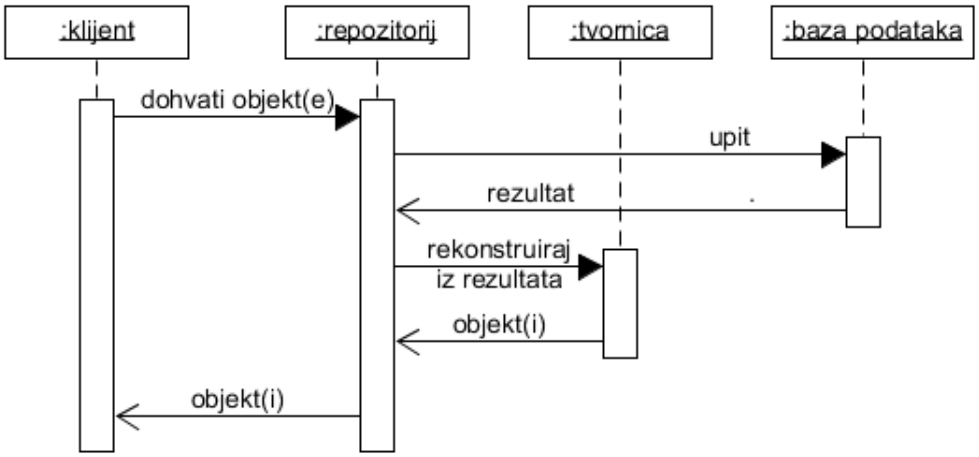
Karakteristike dobrog repozitorija su:

- *Odvojenost klijenta od izvora i implementacije baze podatka*. Moguć je dohvat iz raznih izvora podataka kao što su tekstualne datoteke, relacijske ili druge vrste baza podataka. Datoteke mogu biti zapisane u običnom tekstu *(plain text)*, CSV-u, XML-u ili nečem drugom. Baze podataka, kao što su relacijske, imaju nekoliko vrsta implementacija. Jedan dobar repozitorij bi trebao klijenta odvojiti od takvih detalja kako bi se na jednostavan način mogao promijeniti izvor, odnosno, implementacija baze podataka bez potrebe mijenjanja samog klijenta.
- *Podržava dohvat apstraktnog tipa*. Repozitorij sadrži sve instance određenog tipa, ali to ne znači da za svaku instancu treba postojati poseban repozitorij. Treba omogućiti

dohvat podataka i za određen apstraktni tip unatoč tome što su u bazi te instance definirane kao odvojene tablice. Bitno je voditi računa o tome da postoje razna ograničenja zbog kojih neće biti jednostavno podatke iz tih tablica umetnuti u isti kalup.

- *Kontrola nad transakcijama DBMS-a je prepuštena klijentu*. Iako je moguće automatski izvršiti svaku izmjenu nad bazom podatka, ponekad želimo tu kontrolu prepustiti objektu domene. Primjerice, doista postoji transakcija (npr. bankovna) kao element domene.

Implementacija repozitorija nije jedinstvena. Odluku o tome kako ćemo implementirati repozitorij donosimo ovisno o tehnologiji koju koristimo i zahtjevima koje nam model nalaže. Implementacijski detalj oko kojeg se možemo složiti je da repozitorij bude ponuđen klijentu kao sučelje. Time se omogućuje odvojenost klijenta od implementacije. Nakon što se odlučimo za implementaciju, dužni smo u njoj specificirati upite prema bazi te stvoriti objekte koje dohvaćamo iz rezultata upita. S obzirom na to da smo opet suočeni s problemom stvaranja složenih objekata, možemo se poslužiti i ranije spomenutim tvornicama. Klijent od repozitorija traži određeni objekt, repozitorij šalje *upit* prema bazi podataka, na što dobije *rezultat upita*. On se potom prosljeđuje prema tvornici koja konstruira objekt i prosljeđuje ga repozitoriju, koji ga potom vraća klijentu.



Slika 2.4: Repozitorij koji se koristi tvornicom.

Tvornica je samo jedan od mehanizama kojim stvaramo objekt iz rezultata upita. Neki aplikacijski okviri daju korisniku na raspolaganje već prirađen skup klasa koje su zadužene za preslikavanje rezultata upita u objekte. Danas je sve popularnije objektno-relacijsko

preslikavanje (*Object-relational mapping* - **ORM**) u kojem više niti ne trebamo pisati upite prema bazi već samo trebamo definirati preslikavanje objekata u tablice i varijabli objekata u kolone tablica.

2.4 Primjena gradivnih blokova na studijskom primjeru

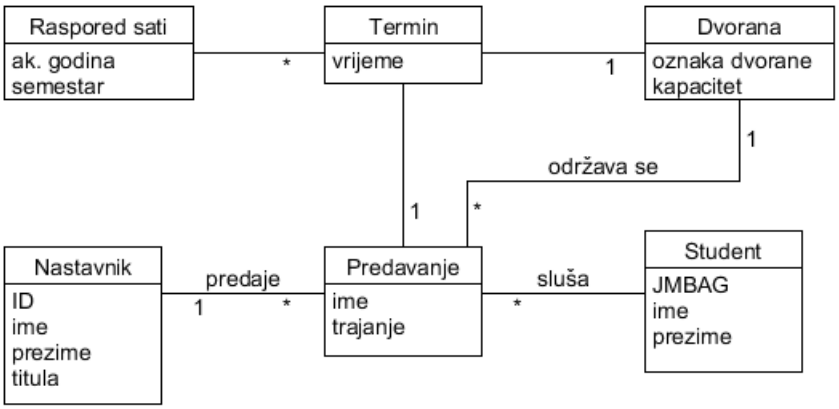
Kada zamišljamo **raspored sati**, većina nas će ga vjerojatno zamisliti kao tablicu gdje iznad stupaca pišu dani, prije redaka vrijeme, a u tablici se nalaze 'kućice' u kojima stoje informacije o predavanju, mjestu i vremenu. Tu 'kućicu' ćemo mi nazivati **termin**. S takvom slikom u glavi, jednostavno je zamisliti premještanje termina nekog predavanja kao premještaj kućice iz jednog dijela tablice u drugi. Isto vrijedi i u kontekstu algoritma - jedno rješenje će biti jedan razmještaj tih *termina* u tablici. Pogled na raspored kao na skup *termina* je pogodan za razumijevanje domene, za algoritam i za prikaz na ekranu, tako da ćemo ga kao takvog prikazati u modelu i koristiti u *sveprisutnom jeziku*.

U *terminu* ćemo držati informaciju o kojem predavanju je riječ, kada *se održava* i u kojoj **dvorani**. **Predavanje** će biti objekt koji sadrži informaciju o nazivu predavanja, predavaču i trajanju. Na predavanja *Mat. analiza 1 [A-LJ] (P)* i *Mat. analiza 1 [A-G] (V)* ćemo gledati kao na dva različita predavanja iako je riječ o istom kolegiju. Takvo pojednostavljenje nam omogućava da predavanje sadrži informaciju o svom (jednom) *predavaču* tj. **nastavniku**. Informacija o predavanju bit će potrebna i **studentu** koji ga *sluša*.

Entiteti i vrijednosni objekti

Do sada nabrojene elemente svrstati ćemo u entitete i vrijednosne objekte.

- **Raspored sati** je *entitet* koji je određen *akademskom godinom* i *semestrom* u kojem je na snazi. Promjenom nekog od termina samo mijenjamo stanje rasporeda sati, ali je i dalje riječ o istom rasporedu! Raspored sati može biti *generalni*, koji sadrži informaciju o svim terminima ili *konkretni* koji sadrži informaciju samo za jednog predavača / dvoranu ili studenta. *Konkretni* raspored je samo podskup *generalnog*.
- **Termin** je *vrijednosni objekt* koji je potpuno određen svim svojim atributima (predavanje, dvorana, vrijeme). Ako izmijenimo bilo koji atribut, dobili smo novi termin.
- **Student** je *entitet* određen svojim JMBAG-om.
- **Nastavnik** je *entitet* određen svojim ID-em npr. OIB-om.
- **Dvorana** je *entitet* određen svojom oznakom npr. "A001".



Slika 2.5: *Class* dijagram koji predstavlja model rasporeda sati.

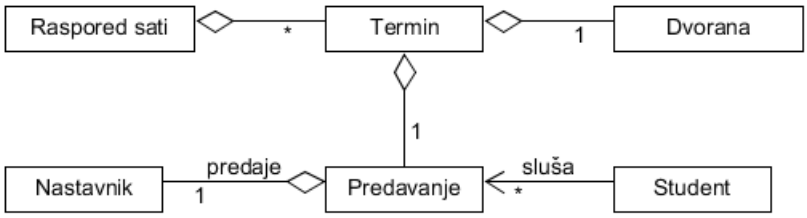
Oblikovanje asocijacija

Apsolvirali smo da se raspored sati sastoji od više termina (*agregacija*) i da se jedan termin sastoji od *dvorane* i *predavanja* (također *agregacija*). Ostaje nam još oblikovati asocijacije između nastavnika i predavanja (veza *predaje*), predavanja i studenta (veza *sluša*) te dvorane i predavanja (veza *održava se*).

- **Veza *predaje***. Nastavnik predaje nekoliko predavanja, ali i jedno predavanje predaje *nastavnik*. Za potrebe evaluacije rasporeda (prilikom izvršavanja algoritma) htjet ćemo za svakog nastavnika provjeriti koliko mu je 'dobar' određeni raspored tako da se logičan smjer čini preko nastavnika doći do predavanja koje predaje. S druge strane, kako bi saznali tko predaje određeno predavanje, moramo proći sve nastavnike sve dok ne dođemo do onog koji predaje to predavanje. Takvo rješenje je vrlo nezgrapno i ne odgovara situaciji u domeni, te ga kao takvog odbacujemo.

- **Veza *sluša***. Informacija koji sve studenti slušaju određeno predavanje nam niti u jednom scenariju naše aplikacije nije potrebna, dok ćemo, s druge strane, htjeti znati koja predavanja pojedini student sluša. Zato ćemo preko studenta dolaziti do predavanja koje sluša.

- **Veza *održava se***. S obzirom na to da preko termina možemo saznati sve o održavanju predavanja u dvoranama, ova veza nam nije potrebna te je kao takvu eliminiramo.

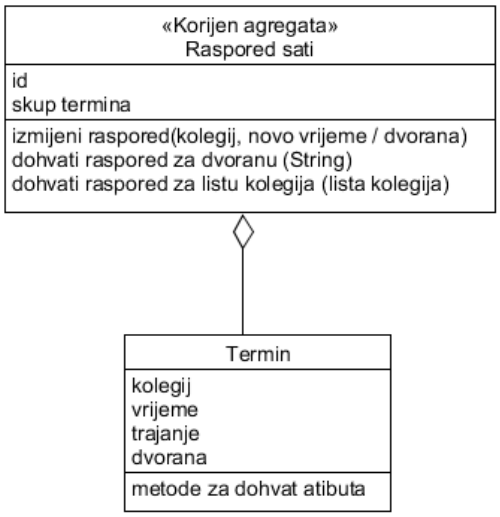


Slika 2.6: *Class* dijagram modela s profinjenim vezama.

Agregati

Agregat čine objekt rasporeda sati i skup termina koji ga sačinjavaju. Jedini način izmjene pojedinog termina je preko objekta rasporeda sati. Ako se pri tome prekrši neko od strogih ograničenja, metoda neće prihvatiti izmjenu već će dojaviti korisniku da krši stroga ograničenja rasporeda. Ako želimo dohvatiti *generalni raspored* ili neki njegov podskup, vratit ćemo kopije kolekcije termina.

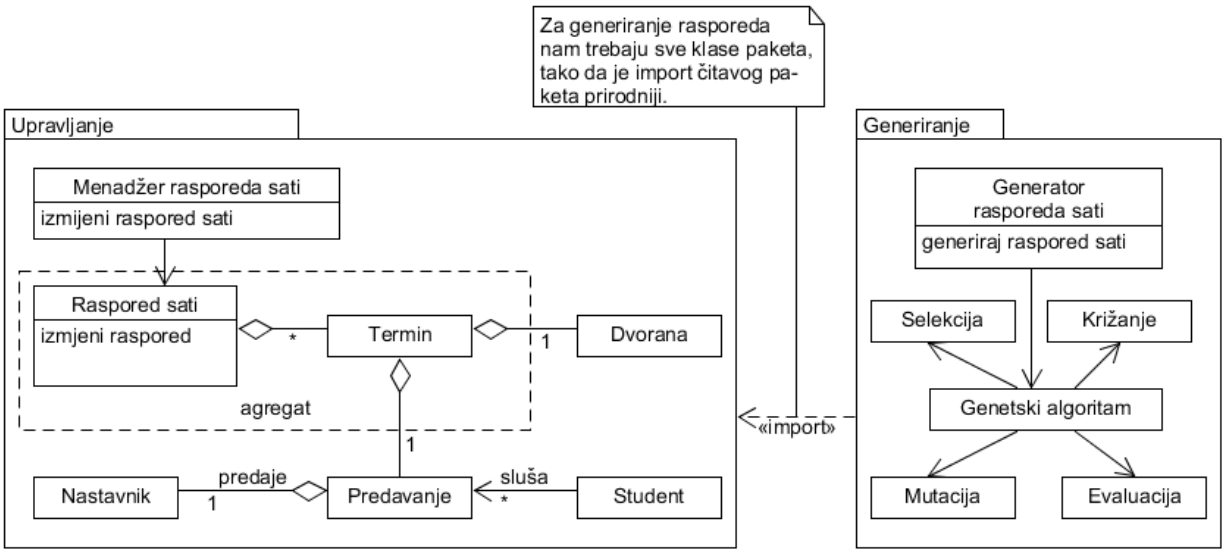
Kao što je već rečeno, agregat će u našem sustavu činiti *raspored sati* i skup *termina* koji ga grade. Ne možemo dopustiti da se se manipulacija nad terminima rasporeda vrši izvan samog rasporeda sati. Sukladno zaključcima iz pododjeljka o agregatima, korijen agregata bit će *raspored sati* te će sve promjene nad rasporedom ići iz njega.



Slika 2.7: Primjer agregata.

Moduli i servisi

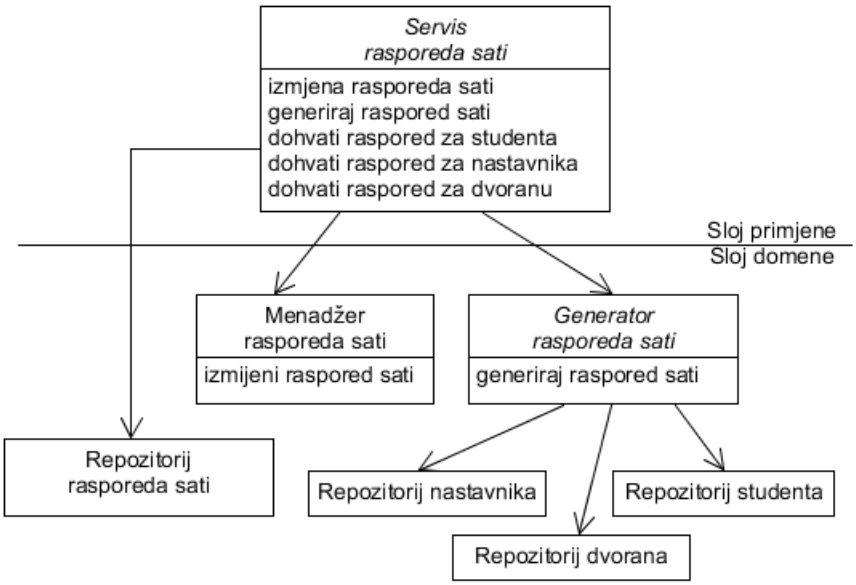
U našoj aplikaciji možemo primijetiti dvije različite funkcionalnosti: *generiranje rasporeda* i *upravljanje rasporedom*. U prvoj se bavimo odabirom parametara algoritma, njegovim pokretanjem i spremanjem novog rasporeda, dok se u drugoj bavimo upravljanjem *nakon što generiramo algoritam*. Iz tih funkcionalnosti prirodno proizlaze servisi *Menadžer rasporeda* i *Generator rasporeda* te moduli *Upravljanje* i *Generiranje*.



Slika 2.8: *Class* dijagram servisa i paketa modela.

Repozitoriji

Za sve navedene entitete izvan agregata (student, predavanje, nastavnik) i sve korijene agregata (raspored sati) moramo imati pripadne repozitorije.
Repozitorij predavanja će imati podršku i za dohvat predavanja koje sluša određeni student ili predaje određeni nastavnik. *Repozitorij rasporeda sati* će nuditi dodatnu mogućnost dohvata konkretnog rasporeda za studenta, nastavnika ili dvoranu. S obzirom da je riječ o običnom dohvatu termina po identitetu studenta / nastavnika / dvorane iz baze podataka, taj posao ćemo sakriti iza repozitorija.



Slika 2.9: *Class* dijagram servisa i repozitorija modela.

Poglavlje 3

Refaktoriranjem prema dubljem uvidu

U prethodnom poglavlju smo govorili o gradivnim blokovima kojima postižemo usku korespodenciju između modela i implementacije. Ipak, pravi izazov se krije u pronalaženju takvog modela koji u isto vrijeme dobro odgovara specifičnostima domene i iz kojeg se može izvesti praktičan dizajn. U konačnici želimo da model bude odraz dubokog razumijevanja domene kako bi više odgovarao potrebama korisnika. U ovom poglavlju pričat ćemo o refaktoriranju kao ključnoj aktivnosti pri razvijanju takvog modela.

Refaktoriranje u svijetu programiranja označava ponovno oblikovanje softvera nakon kojeg njegova funkcionalnost ostaje nepromijenjena. Većina knjiga o refaktoriranju bavi se mehaničkim izmjenama koje su motivirane poboljšanjem čitljivosti i robusnosti kôda, npr. ako vidimo priliku da primijenimo neki oblikovni obrazac. Mi ćemo se fokusirati na **refaktoriranje koje je motivirano dubljim uvidom u domenu** ili prilikom za jasnije izraženim kôdom.

Prvi modeli domene stvaraju se u razgovoru sa stručnjacima gdje subjekti i predikati sugeriraju objekte, veze i operacije koje ćemo koristiti u sustavu. Tako dobivamo inicijalni model s kojim možemo započeti implementaciju. U suštini je takav model površan i bazira se na slabom poznavanju domene. S vremenom, razvojni tim stječe dublje znanje o domeni, upoznaje se s bitnim konceptima, apstraktnim jezikom i žargonom domene te za model postepeno isplovljavaju nove ideje koje bolje odgovaraju novo-otkrivenom znanju. Bitne koncepte domene nastojimo eksplicitno prikazati kôdom. Ovakvim malim izmjenama postepeno pridonosimo jasnoći modela i otvaramo prostor za *otkriće* boljeg i dubljeg modela. Takva otkrića nalažu veće izmjene u implementaciji i naš razvoj privremeno stagnira. Ipak, implementacijom dubljeg modela nestaju poteškoće korištenja starog, plićeg modela te se nude nove mogućnosti koje možemo obavljati s novim modelom. Kao posljedica toga razvoj dobiva na brzini.

Na primjeru rasporeda sati izazov je smisliti model iz kojeg će se jednostavno moći ge-

nerirati reprezentacija rješenja algoritma i koji će se jednostavno moći prikazati na ekranu. Reprezentacija rasporeda kao skup termina u ovom slučaju predstavlja *otkriće* koje je pomoglo bržem razvitku aplikacije.

3.1 Izražavanje implicitnih koncepta eksplicitino

Duboki model je moćan jer sadrži centralne koncepte i apstrakcije koje na sažet i fleksibilan način mogu izraziti ključno znanje korisničke aktivnosti, probleme s kojima se susreću i njihova rješenja. Proces otkrivanja koncepta domene nije jednostavan, uglavnom ide postepeno kroz učenje i mnoštvo istraživanja. Te bitne koncepte možemo prepoznati na razne načine.

- *Slušanjem jezika.* Termini koje koriste stručnjaci domene, a nisu zastupljeni u modelu, mogu biti naputci gdje se krije skriveni koncept koji se još nije izrazio.
- *Istraživanjem područja nejasnoća.* Nejasni i kompleksni dijelovi dizajna dobar su pokazatelj gdje postoji prostor za istraživanje. U to istraživanje treba uključiti stručnjake domene kako bi došli do jasnijeg i elegantnijeg dizajna.
- *Uzimanjem drugog mišljenja.* Različiti stručnjaci domene imaju različite načine kojima objašnjavaju neke stvari. Ponekad će nam drugi pogled istaknuti koncepte koje nismo uočili u prijašnjim razgovorima.
- *Proučavanjem literature domene.* Čitanjem literature o specifičnoj domeni obogaćujemo naše znanje domene i pospješujemo komunikaciju sa stručnjacima domene. Ponekad, kada nam stručnjaci domene nisu od velike pomoći, ovo nam je i najbolji izvor znanja.
- *Obrasci analize (Analysis patterns).* **Obrasci analize** su grupe koncepata koji predstavljaju uobičajenu konstrukciju u modeliranju poslovanja. Oni mogu obuhvaćati jednu domenu ili više njih. Primjerice, koncept korisničkog računa je koncept koji se može susresti u mnoštvu domena. Umjesto da otkrivamo 'toplu vodu', u knjizi *Analysis patterns* od Martina Fowlera [2] možemo pronaći korisne ideje za takve probleme.

Ograničenja

Ograničenja čine jednu bitnu skupinu koncepta modela koja često nisu eksplicitno izražena u kôdu. U primjeru rasporeda sati imamo mnoštvo ograničenja koja moramo poštivati prilikom generiranja rasporeda. Primjerice, moramo paziti da se predavanje održava u dvorani odgovarajućeg kapaciteta. Možemo eventualno dopustiti da se u jednoj dvorani

nalazi 10% više osoba nego što je za dotičnu dvoranu standardno predviđeno. Sve više od toga vodi do prenapučenosti dvorane.

Ilustrirat ćemo provođenje tog ograničenja idućim kôdom. Pretpostavit ćemo da klasa `Classrooom` ima varijablu `capacity` tipa `int` koja predstavlja kapacitet neke dvorane te klasa `Lecture` ima varijablu `attendingNum` tipa `int` koja predstavlja broj slušatelja nekog predavanja.

```
public boolean changeClassroomForLecture(Lecture lecture, Classrooom classroom) {
  if (classroom.getCapacity() * 1.1 < lecture.getAttendingNum()) {
    return false;
  }
  // Kod za izmjenu ucionice
  return true;
}
```

Kôd 3.1: Primjer koncepta koji je sadržan implicitno u kôdu.

Ovakav kôd će raditi, ali je malena vjerojatnost da će drugi programer moći povezati izraz u `if` uvjetu s ograničenjem koje stoji iza njega. To ograničenje i druga ograničenja koja predavanja nalažu spadaju u skup *pravila raspoređivanja predavanja*. Treba definirati klasu čija će odgovornost biti provođenje tih pravila, koja će za svako ograničenje imati metodu kojom provjerava je li ograničenje zadovoljeno. Tako ograničenje postaje imenovana stvar koju možemo diskutirati te jednostavno proširiti i izmijeniti.

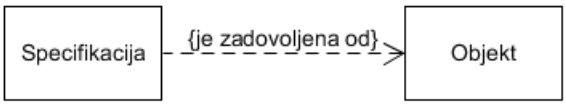
```
public boolean changeClassroomForLecture(Lecture lecture, Classrooom classroom) {
  if (ClassroomPolicies.classroomIsTooSmall(classroom, lecture)) {
    return false;
  }
  // Kod za izmjenu ucionice
  return true;
}

public class ClassroomPolicies {
  public static boolean classroomIsTooSmall(Classroom classroom, Lecture lecture) {
    return classroom.getCapacity() * 1.1 < lecture.getAttendingNum();
  }
}
```

Kôd 3.2: Primjer koncepta koji je sadržan eksplicitno u kôdu.

Specifikacije

U aplikacijama su česte tzv. *metode istinitosti* koje su dio nekih manjih pravila. Sve dok su ta pravila jednostavna, *metoda istinitosti* će zadovoljiti naše potrebe. Čim se pojave složenija pravila, koja mogu uključivati logičke operatore, ta složenost počinje preopterećivati osnovno značenje tog domenskog objekta. Rješenje je kreirati vrijednosne objekte (tzv. **specifikacije**), koji se ponašaju kao predikati i ispituju ispunjava li objekt neki određeni kriterij. Zapravo je riječ o inverziji kontrole, umjesto da objekt ispituje zadovoljava li on određeni kriterij, mi definiramo kriterij i pitamo se je li zadovoljen od određenog objekta.



Slika 3.1: Odnos specifikacije i objekta koji ju treba zadovoljiti.

Kada koristimo specifikacije često nailazimo na situacije u kojima bismo ih htjeli kombinirati pomoću logičkih operatora kao što i predikate možemo kombinirati logičkim operatorima kunjunkcije, disjunkcije i negacije. Uz prikladna nasljeđivanja možemo upravo to postići i sa *specifikacijom*.

```
public interface Specification {
  boolean isSatisfiedBy(Object object);
  Specification and(Specification other);
  Specification or(Specification other);
  Specification not();
}

public abstract class AbstractSpecification implements Specification {
  public abstract boolean isSatisfiedBy(Object object);

  private Specification and(Specification other) {
    return new AndSpecification(this, other);
  }
  private Specification or(Specification other) {
    return new OrSpecification(this, other);
  }
  private Specification not(Specification other) {
    return new NotSpecification(this);
  }
}

public class AndSpecification extends AbstractSpecification {
  private Specification first;
  private Specification second;
```

```
  public AndSpecification(Specification first, Specification second) {
    return this.first = first;
    return this.second = second;
  }

  @Override
  public boolean isSatisfiedBy(Object object) {
    return first.isSatisfiedBy(object) && second.isSatisfiedBy(object);
  }
}
// Ostala prosirenja specifikacije idu analogno
```

Kôd 3.3: Specifikacija koja podržava logičke operatore.

Specifikacije imaju nekoliko svrha u oblikovanju.

1. Za validaciju objekta, kako bi vidjeli zadovoljava li određena svojstva.

2. Kao kriterij pri selekciji iz neke kolekcije objekata.

3. Kako bi specificirali stvaranje nekog objekta.

Validacija. Najjednostavnija specifikacija je ona koja služi u svrhu *validacije*. Primjerice, trebamo definirati zaustavni kriterij optimizacijskog algoritma. Često želimo postaviti više kriterija, od kojih je jedan kada se postigne određena dobrota, a drugi kada prođe određeni period. Umjesto da kriterije zapišemo direktno u algoritam, definirat ćemo specifikaciju koja će provjeravati zadovoljava li algoritam uvjete zaustavljanja. (Detaljnije u odjeljku 3.3).

Selekcija. U validaciji ispitujemo objekt da utvrdimo zadovoljava li određen kriterij. Kod selekcije, specifikacija nam služi kao **filter koji primjenjujemo nad nekom kolekcijom objekata** kako bismo dobili samo objekte koji zadovoljavaju određen kriterij. Ovakav tip specifikacija može imati bitnu ulogu u aplikacijama u kojima korisnik primjenjuje nekakav filter nad proizvodima koji ga zanimaju. Na primjer, korisnik želi vidjeti ponude svih televizora definirane veličine, određenog proizvođača, čija je cijena u određenom rangu.

Stvaranje Posljednja svrha specifikacije je ona u kojoj *specifikacija* sadrži informaciju kakav objekt (ili više njih) želimo stvoriti. U ovom slučaju nemamo jedan objekt koji želimo ispitati, niti imamo kolekciju objekata koje treba profiltrirati, već želimo nekom delegirati posao **stvaranja objekta koji zadovoljava neku specifikaciju**. Na primjeru zaustavnog kriterija algoritma - specifikacija zaustavnog kriterija je u neku ruku i specifikacija stvaranja. Naime, mi smo algoritmu dali specifikaciju da želimo da nam *stvori* raspored sati koji zadovoljava određen kriterij (određenu dobrotu).

3.2 Gipki dizajn

U procesu konstantnog refaktoriranja sam dizajn mora biti oblikovan tako da podržava izmjene. Izmjene su pogotovo teške ako postoje nejasne ovisnosti među objektima i metode čije posljedice nisu predvidive. Ako u našem kôdu ima puno ovakvih slučajeva naš će se razvoj softvera u bliskoj budućnosti svesti na borbu s vlastitom baštinom. Htjeli bismo da dizajn bude takav da je jednostavno činiti izmjene u njemu, bez nepredvidivih posljedica.

Imenovanje koje otkriva namjenu

U *oblikovanju vođenom domenom* želimo razmišljati o procesima domene. Ako član razvojnog tima mora razmišljati o implementaciji komponente da bi ju koristio, vrijednost enkapsulacije je izgubljena. Krivo shvaćena namjena te komponente lako je moguća. Moguć je scenarij u kojem dva člana razvojnog tima različito tumače namjenu jedne komponente. Kada ju jedan od njih odluči izmijeniti posljedice ne mogu biti predvidive i mogu narušiti funkcionalnosti dijela sustava koji je napisao drugi programer.

Kako bi se ovakve zabune izbjegle treba imenovati module, klase, metode, argumente i varijable na razumljiv način, koji otkriva svoju namjenu. Ta imena bi trebala što više moguće korespondirati sa *sveprisutnim jezikom* kako bi članovi tima odmah mogli dokučiti njihovo značenje.

Slično vrijedi i za složene mehanizme sustava. Za jednu klasu, koja u sebi krije mnoštvo složenih operacija, treba uložiti dodatan kognitivni napor kako bi se razaznala njegova namjena. Takve složene mehanizme trebalo bi sakriti preko sučelja koja nam govore što klasa radi, a ne kako. Takvo sučelje nazivamo **sučelje koje otkriva namjenu**.

Razvoj vođen testiranjem zahtijeva da prvo moramo napisati test prije nego što implementiramo neku komponentu. Takav pristup nas tjera da razmišljamo kao klijent te komponente i da provodimo imenovanje koje otkriva namjenu prije nego što napišemo komponentu.

Funkcije bez popratnih posljedica

Mijenjanje dizajna postaje poprilično teško kada u svom dizajnu imamo mnoštvo operacija koje mijenjaju objekt ili koje pozivaju druge operacije koje mijenjaju objekt. Kako bi razumjeli što se u konačnici dogodilo s tim objektom, trebamo razumjeti implementaciju dotične operacije, te implementaciju svih operacija koje ona koristi. Ako moramo razumjeti čitavu hijerarhiju operacija kako bi razumjeli što objekt radi, sve uvedene enkapsulacije gube smisao.

Rješenje je koristiti operacije koje samo vraćaju rezultat, a pri tome nemaju popratnih posljedica na predani objekt. Takve operacije još zovemo i **funkcijama**. Ako pak trebamo činiti izmjene nad objektima, preporučuje se prvo iz objekta dohvatiti varijable potrebne

za izračun, izvršiti izračun *funkcijom* te rezultat izračuna *set* metodom postaviti na pripadajuću vrijednost. Na ovakav način možemo imati funkcije koje pozivaju druge funkcije bez brige o nekontroliranim promjenama unutarnjih stanja objekata. Funkcija može pozivati druge funkcije za svoje operacije bez da se zamaramo o posljedicama gniježđenja. Ako se složene operacije događaju unutar same klase čija stanja treba izmijeniti, dobra praksa je vidjeti može li se takve operacije izdvojiti u vrijednosni objekt (kao u slučaju specifikacija).

Korištenjem ovih principa možemo i testove pisati na jednostavan način. Kada bi se operacije vršile nad objektima, trebali bismo kreirati čitav objekt sa željenim vrijednostima prije provjere tvrdnji. Taj objekt može imati i po nekoliko desetaka varijabli stanja u sebi. Ponekad postoje i razna *'not null'* ograničenja koja treba ispoštovati prije nego što napišemo same tvrdnje testa. *Razvoj vođen testiranjem* nije moguć te u samom testu moramo razmišljati i o dodatnim stvarima uz namjenu operacije koju koristimo.

Provođenjem gornjih pravila možemo testirati jedinicu po jedinicu, osiguravajući integritet svake pojedinačno. Moguće je prvo napisati i test, pa tek onda funkciju koju koristimo, čime pospješujemo princip imenovanja koje otkriva namjenu.

Utvrđivanja (*assertions*)

Čak iako koristimo funkcije bez popratnih posljedica i dalje nam ostaje dio operacija nad entitetima koji mogu imati popratne posljedice. Na primjer, ako atribut nekog entiteta nepažnjom poprimi vrijednost `null`, grešku u programu ćemo uočiti tek kada jedna od operacija koja koristi taj atribut baci `NullPointerException`. Kako bismo otkrili zašto je došlo do te greške, morat ćemo tragati unatrag (kroz kôdove koje možda niti nismo pisali), sve dok ne dođemo do trenutka kada je postavljena ta `null` vrijednost.

Takvih primjera grešaka ima mnogo i jedini način da se osiguramo od popratnih posljedica je da ispitamo stanja entiteta na kraju operacija koje provodimo. Neki programski jezici nude tzv. `assert` naredbe kojima pišemo *tvrdnje* koje moraju vrijediti u tom trenutku. Za konkretni slučaj propagirane `null` reference Java npr. nudi metodu <T> `Object.requireNonNull (T obj)` koja vraća predani objekt, osim ako je predan `null` - onda baca iznimku!

Ako nam programski jezik ne nudi mogućnosti provjeravanja tvrdnji tijekom izvođenja kôda onda trebamo testovima utvrditi stanja objekta nakon operacije koju izvodimo.

Konceptualne konture

Ponekad se određena kompleksna funkcionalnost razdvaja na nekoliko jednostavnih cjelina. Ponekad se nekoliko funkcionalnosti spaja u jednu enkapsuliranu cjelinu. Bitno je razlikovati kada ćemo se prikloniti kojem pristupu. Kada nekoliko različitih elemenata do-

mene ugradimo u jednu strukturu povećavamo vjerojatnost dupliciranja kôda. Sučelje kojim sakrivamo tu strukturu ne govori korisniku jasno njegovu namjenu jer se iza tog sučelja krije mnoštvo međusobno pomiješanih koncepta domene. S druge strane, rastavljanje klasa i metoda na male cjeline može nepotrebno zakomplicirati čitavu strukturu, otežavajući klijentu da njih zajedno poima kao dio cjeline. Postavlja se pitanje kada cjepkati, a kada sastavljati cjeline.

Generalnog pravila kako pristupiti ovom pitanju - nema! U različitim situacijama postoje različiti odgovori. Ono što zasigurno postoji su logičke cjeline neke domene. To ne znači da su cjeline domene savršeno konzistentne, već da iza njih postoje dobri naputci kako definirati cjeline unutar modela. **Kada se model podudara sa nekim dijelom domene, veća je vjerojatnost da će model ostati konzistentan s ostalim dijelovima domene koje otkrijemo kasnije**. Zato treba paziti da cjeline koje definiramo u modelu imaju unutar sebe jaku koheziju i da odgovaraju logičkim cjelinama domene. Otkrivanje tih logičkih cjelina nije odmah jasno i često se tek nakon nekoliko uspješnih refaktoriranja može vidjeti pravilnost koja ih spaja tj. razdvaja.

Dobar primjer kako se iz logičkih cjelina domene mogu uočiti konceptualne konture su algoritmi koje učimo na nastavi. Algoritam se nikada ne predstavlja studentu kao njegova implementacija, već je student upoznat sa 'pseudokodom' koji otkriva ponašanje algoritma, a detalje ostavlja u obliku jednostavnog teksta. Primjerice, klasičan genetski algoritam (generacijski) će govoriti o selekciji, križanju, mutaciji i pridruživanju novoj populaciji, a neće govoriti kako se ti procesi vrše ili kako će se stara populacija zamijeniti novom. To je dobra naznaka da naš algoritam upravo treba biti sastavljen što vjernije tom pseudokodu, definirajući manje cjeline kao što su selekcija, križanje i mutacija. Proces zamjena populacije može biti izdvojen u zasebnu metodu. Upravo tako možemo biti sigurniji da naš algoritam radi baš ono što treba (ono što piše u pseudokodu).

Samostojeće klase

Međuovisnosti čine model i dizajn manje razumljivim. Svaka asocijacija je ujedno i ovisnost. Razumjeti klasu podrazumijeva razumjeti i klase s kojima je povezana. Povezanost može biti u obliku nasljeđivanja, varijabla klase, argumenata metode ili njene povratne vrijednosti. Što više povezanosti imamo među klasama, to veći napor moramo uložiti kako bismo razumjeli njihovo funkcioniranje te općenito poimali njih kao cjelinu. Vjerojatnost da ćemo nešto previdjeti u izmjeni je velika. Iz tog razloga koristimo *module* i *agregate* kako bismo smanjili mrežu međuovisnosti.

Ipak, čak i unutar modula ponekad zna biti teško razumjeti što se događa, osobito kada se tijekom razvoja u modelu pojave dodatne međuovisnosti. Jedan od principa kojim nastojimo smanjiti broj ovisnosti je tzv. *samostojeća klasa*.

Samostojeća klasa je klasa koja se može razumjeti i testirati **bez referenciranja dru-**

gih klasa osim primitivnih varijabli i klasa osnovnih biblioteka (liste, skupovi itd.). Slabe veze su osnova objektnog oblikovanja. Iz klase bi trebalo ukloniti sve koncepte nebitne za tu klasu, a ostaviti samo bitne. Ako je moguće, unutarnja računanja koje referenciraju druge klase bilo bi dobro prepustiti vrijednosnim objektima koje koriste druge klase. Tada će klasa biti samosadržavajuća i može se promatrati i razumjeti sama po sebi. Svaka takva samostojeća klasa značajno olakšava razumijevanje nekog modula.

Zatvorenost operacije

Ipak, ne možemo ukloniti sve međuovisnosti modela svođenjem svega na primitive. Neke međuovisnosti se nalaze unutar modela kao posljedica fundamentalnih koncepta domene i kao takve trebaju i ostati u modelu. U tom slučaju dobra praksa je, ako je moguće, definirati metode tako da vraćaju isti tip kakav je i tip argumenata koje prima. U duhu matematike ovu praksu nazivamo **zatvorenost operacije**. Ovakva praksa je česta kod vrijednosnih objekata koji primaju vrijednosne objekte istog tipa te kao rezultat vraćaju vrijednosni objekt istog tipa. Na ovaj način ne unosimo dodatne međuovisnosti u model.

Ponekada smo zadovoljni i kada osiguramo približnu zatvorenost operacije, primjerice, kada se argumenti operacije poklapaju s implementatorom operacije, ali ne s povratnom vrijednosti, ili obratno. Ako je dodatan tip podatka primitivni podataka ili jedna od baznih klasa programskog jezika, osjećamo se sigurno kao i kad imamo zatvorenost operacije.

Deklarativni stil oblikovanja

Deklarativni stil oblikovanje je stil oblikovanja u kojem naš kôd nastoji objasniti *što* naš program mora raditi u terminima problema domene, umjesto *kako* to radi. Primjenom gore navedenih principa za *gipki dizajn* dobivamo kôd koji poprima izgled izvršive specifikacije. Kada naš kôd poprimi takav oblik, možemo biti znatno sigurniji u ishod njegovog izvršavanja.

Dobar primjer deklarativnog oblikovanja je kôd za specifikacije, u kojem je omogućeno ulančavanje specifikacija logičkim operatorima. Možemo primijetiti da su varijable i metode oblikovane tako da otkrivaju namjenu, a metode (osim metode provjere) su *funkcije* i zatvorene su na operande.

Postoje i tzv. **jezici specifični domeni** (*domain-specific language*) koji su posebna vrsta programskih jezika čija je sintaksa prilagođena određenoj domeni. Primjerice, postoji porodica programskih jezika koju si prilagođeni financijskom sektoru. Program se potom prevede u jedan od konvencionalnih objektno-orijentiranih jezika. Prednost korištenja takvih jezika je visoka povezanost programa sa sveprisutnim jezikom, dok je nedostatak poteškoća pri održavanju takvog programa (većina ljudi se ne zna koristiti takvim jezikom).

3.3 Primjena gipkog dizajna na studijskom primjeru

Najzahtjevniji dio pisanja aplikacije za raspored sati je pisanje algoritma za generiranje rasporeda. Iza svakog zanimljivog optimizacijskog algoritma krije se mnoštvo složenih operacija koje se izvode kako bi dobili što bolji rezultat. Nepravilnim funkcioniranjem jedne od tih komponenti algoritam bi proizvodio loša rješenja pri čemu ne bi dobili bilo kakvu informaciju gdje je problem. Zapravo, teško je razlučiti dobivamo li loša rješenja kao posljedicu neispravnosti neke komponente ili kao posljedicu lošeg algoritma. S obzirom na to da optmizacijski algoritmi uglavnom koriste mehanizme slučajnog odabira, nemamo očekivane rezultate s kojima možemo provjeravati ispravnost algoritma. Jedina način da osiguramo ispravnost algoritma je dobar dizajn i temeljito testiranje komponenti.

Komponente genetskog algoritma

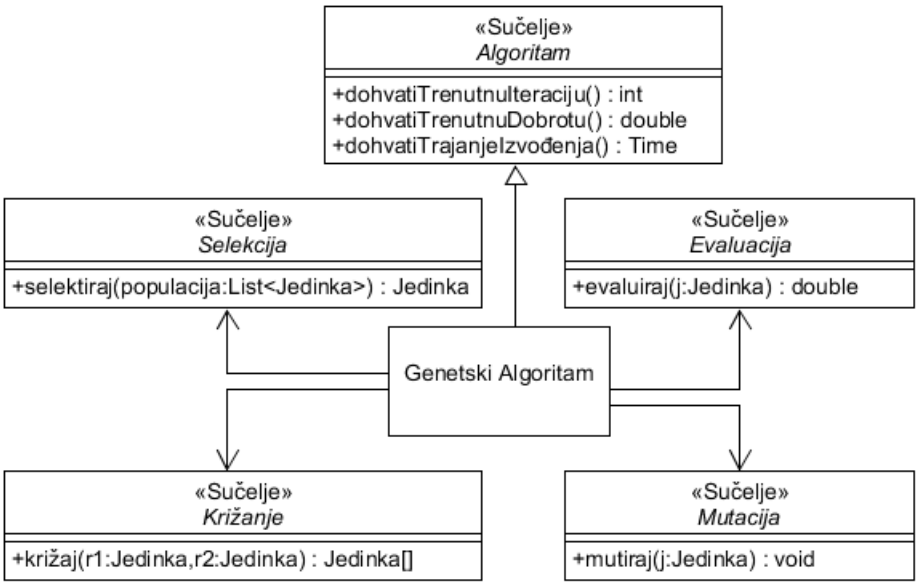
Genetski algoritam se sastoji od operacija selekcije, križanja, mutacije i evaluacije. Svaka od njih ima mnoštvo implementacija, tako da nam je želja metodama enkapsulacije i polimorfizma sakriti implementaciju operacija od centralnog dijela algoritma. Čak iako sa sigurnošću znamo koje ćemo točno implementacije selekcija, križanja i mutacije koristiti, mehanizmi iza njih su vrlo složeni i ne želimo s njima opterećivati korisnika. Svaka od tih operacija bit će predstavljena *sučeljem koje otkriva namjenu*.

- *Selekcija.* Iz populacije (liste *jedinki*) trebamo izabrati jedinku koja će postati roditelj. Metoda *selektiraj* prima listu jedinki i vraća selektiranu jedinku. Zatvorenost operacije je zadovoljena.

- *Križanje.* Križanjem dva roditelja dobivamo dva djeteta. Operacija prima dvije jedinke i vraća dvije jedinke pri čemu roditelji ostaju nepromijenjeni. Riječ je o *funkciji* zatvorenoj na operacije.

- *Mutacija.* Prima jedinku i vrši mutaciju nad njom. Za razliku od prethodnog križanja, predanu jedinku ćemo mijenjati u metodi koja neće imati povratnu vrijednost. Naime, mutiranje nije funkcija jer po svojoj definiciji mijenja predanu jedinku. Transformacijom u funkciju zatvorenu na operande samo bismo zbunili korisnika.

- *Evaluacija.* Evaluacija prima jedinku te vraća broj koji predstavlja njenu dobrotu. S obzirom na to da je `double` primitivna varijabla, zadovoljni smo kao da imamo zatvorenost operacije.



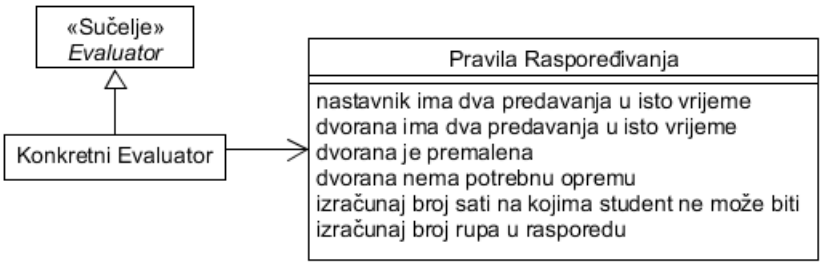
Slika 3.2: Genetski algoritam i njegove operacije.

Evaluacija

U evaluaciji provjeravamo koliko je naše rješenje dobro, zadovoljava li ili krši stroga ograničenja nad rasporedom.

- Jedan nastavnik ne može predavati na dva predavanja u isto vrijeme. *Jako* ograničenje.
- U jednoj dvorani ne mogu se održavati dva predavanja u isto vrijeme. *Jako* ograničenje.
- Predavanje se ne može održavati u dvorani koja nema potrebnu opremu. *Jako* ograničenje.
- Predavanje se ne može održavati u dvorani u koju ne stanu svi studenti. *Jako* ograničenje.
- Student ne može prisustvovati svim predavanjima koje sluša. *Slabo* ograničenje.
- Student nema 'rupa' u rasporedu. *Slabo* ograničenje.

Kao što smo ranije spomenuli, sva pravila dobrog raspoređivanja ćemo staviti u posebnu klasu `SchedulingPolicies`. Za jaka ograničenja ćemo definirati metode koje ispituju krši li dani raspored neko od tih ograničenja, a za slaba ograničenja ćemo vraćati broj sati na kojima student ne može biti, odnosno, broj 'rupa' u rasporedu.



Slika 3.3: Detaljniji prikaz komponente *evaluacije*.

Kriterij zaustavljanja

Još nam ostaje definirati kako će algoritam znati kada treba prestati s radom. Primjerice, algoritam može stati s radom nakon određenog broja iteracija (generacija), nakon što nam rješenje postane 'dovoljno dobro' ili nakon nekog određenog perioda. Realan zahtjev bio bi omogućiti zaustavljanje algoritma ako se dosegne određena dobrota ili ako prođe određen period vremena.

Treba nam mehanizam kojim možemo validirati je li ostvaren uvjet zaustavljanja algoritma. Kao logičan izbor nam se nameću *specifikacije*. Proširit ćemo specifikaciju iz kôda 3.3 s navedenim kriterijima zaustavljanja.

```
public abstract StopCriteria extends AbstractSpecification {

  public abstract boolean isSatisfiedBy(Algorithm alg);

  @Override
  public boolean isSatisfiedBy(Object object) {
    if (object instanceof Algorithm) {
      return isSatisfiedBy((Algorithm) object);
    } else {
      return false;
    }
  }
}

public class MaxIterationReachedCriteria extends StopCriteria {
  private int maxIterNum;

  public MaxIterationReachedCriteria(int maxIterNum) {
    this.maxIterNum = maxIterNum;
  }

  @Override
  public boolean isSatisfiedBy(Algorithm alg) {
    return alg.getCurrentIter() >= maxIterNum;
  }
```

```
}

// Analogno definiramo i druga dva kriterija
public class FitnessReachedCriteria extends StopCriteria { ... }
public class TimeElapsedCriteria extends StopCriteria { ... }

public class StopCriterias {
  public static StopCriteria isMaxIterReached(int maxIter) {
    return new MaxIterationReachedCriteria(maxIter);
  }

  public static StopCriteria isFitnessReached(double fitnessGoal) {
    return new FitnessReachedCriteria(fitnessGoal);
  }

  public static StopCriteria isMaxTimeReached(long timeInMillis) {
    return new TimeElapsedCriteria(timeInMillis);
  }
}
```

Kôd 3.4: Kriterij zaustavljanja oblikovan kao *specifikacija*.

Ako želimo da algoritam stane kad se dosegne određena dobrota ili protekne određeno vrijeme, kôd bismo napisali na sljedeći način.

```
StopCriteria sc = StopCriterias.isFitnessReached(fitnessGoal)
        .or(StopCriterias.isMaxTimeReached(maxTime));
```

Kôd 3.5: Primjer definiranja zaustavnog kriterija.

Spajanje svih komponenti u cjelinu

Nakon što smo definirali sve komponente algoritma, vrijeme je da prikažemo kako genetski algoritam radi kao cjelina. U sljedećem kôdu ćemo fokus staviti na proces samog algoritma, a izostavit ćemo detalje (atribute, naslijeđene metode itd.) vezane uz samu klasu.

```
public class GeneticAlgorithm implements Algortihm {
    // ... svi potrebni atributi ...
    private Specimen bestSpecimen;
    private StopCriteria stopCriteria;
    // ... sve naslijedene metode ...

    @Override
    public Specimen start() {
        List<Specimen> population = initializePopulation();
        evaluatePopulation(population);

        while (!stopCriteria.isSatisfied(this)) {
            List<Specimen> newPoulation = new ArrayList<>();
            while (newPoulation.size() != population.size()) {
                Specimen parent1 = selection.select(population);
                Specimen parent2 = selection.select(population);
                Specimen[] children = crossover.cross(parent1, parent2);
```

```
                mutation.mutate(children[0]);
                mutation.mutate(children[1]);
                newPoulation.addAll(children);
            }
        }
        return bestSpecimen;
    }

    private List<Specimen> initializePopulation {...}

    private void evaluatePopulation(List<Specimen> population) {
        for (Specimen specimen : population) {
            specimen.setFitness(evaluator.evalute(specimen));
            if (specimen.getFitness() > bestSpecimen.getFitness()) {
                bestSpecimen = specimen;
            }
        }
    }
}
```

Kôd 3.6: Centralni dio genetskog algoritma.

U ovom *centralnom dijelu* algoritma možemo vidjeti da nigdje nije zapisana informacija o tome kako se vrši selekcija, križanje, mutacija i evaluacija, ne znamo koji je točno zaustavni uvjet algoritma. Jedino znamo *što* algoritam radi pa je i stil pisanja takav da kôd nalikuje na pseudokod genetskog algoritma. Tako možemo biti donekle sigurni u ispravnost rada centralnog dijela algoritma. Ono što nam preostaje je temeljito testirati sve komponente algoritma. Zahvaljujući ovakvom oblikovanju, gdje su međuovisnosti minimalizirane, taj posao će biti jednostavan!

Jedini segment studijskog primjera koji je ostao neobjašnjen je povezanost modela rasporeda sati iz poglavlja o gradivnim objektima i ovog algoritma. Kako iz modela rasporeda sati, koji se sastoji od termina, dobijemo jedinku genetskog algoritma? Taj segment ćemo pokriti u posljednjem poglavlju gdje ćemo govoriti o vezanim kontekstima.

Poglavlje 4

Strateško oblikovanje

Neki sustavi su toliko veliki i složeni da ih ne možemo razumjeti na razini objekata. Potrebno je uvesti nove tehnike modeliranja i oblikovanja koje će tim objektima dati kontekst unutar čitavog sustava, kojima ćemo smanjiti međuovisnosti dijelova te povećati jasnoću i sinergiju čitavog sustava. Takve odluke modeliranja i oblikovanja na razini čitavog sustava nazivamo **strateškim oblikovanjem**.

U velikim sustavima, u kojima postoje različiti modeli za isto područje domene, potrebno je paziti da **očuvamo integritet modela** na razini čitavog sustava. Područje domene zna postati toliko veliko i kompleksno da moramo **izdvojiti jezgru domene**, kako bismo razlučili srž sustava od njegovih potpornih elemenata. Ponekad je korisno nad sustavom **velikih razmjera** definirati određenu **strukturu** kako bismo jednostavnije mogli razumjeti ulogu dijelova u cjelini.

4.1 Očuvanje integriteta modela

U idealnom svijetu htjeli bismo imati jedan model koji je primjenjiv nad čitavim sustavom. Nažalost, u praksi je takva ideja vrlo nepraktična za velike sustave na kojim radi više timova ljudi (a većina zanimljivih sustava spada u tu kategoriju). Željom da zadovoljimo sve timove i sve kontekste istim modelom moramo uvoditi kompleksna rješenja čime model činimo težim za upotrebu. Takav je model teže mijenjati jer ne znamo kako će te izmjene utjecati na timove koji koriste model za druge kontekste.

U odjeljku 2.4 definirali smo jedan model rasporeda sati koji nam je pogodan za prikaz rasporeda na ekran i za ručnu izmjenu. Mogli bismo zahtijevati da isti model koristimo i u genetskom algoritmu. Za to bismo trebali izmisliti posebne vrste selekcija i mutacija koje bi činile izmjene nad takvom implementacijom rasporeda. Jednostavnije je imati **reprezentaciju rasporeda** nad kojim ćemo izvoditi jednostavnije, već poznate operacije algoritma.

Za model nam je jako bitna njegova unutarnja konzistentnost. Bitno nam je da njegovi termini uvijek imaju isto značenje i da ne sadrži kontradiktorna pravila. Kako bi to održali, ponekad je potrebno imati više modela od kojih svaki daje drukčiji pogled na domenu. U tom slučaju potrebno je označiti granice i odnose među modelima kako bi očuvali integritet modela. U ovom ćemo odjeljku govoriti o tehnikama za raspoznavanje, komunikaciju i biranje granica modela i njihovih međusobnih odnosa.

Vezani kontekst

Ako se odlučimo za više modela unutar jednog projekta, onda je bitno da uz svaki model identificiramo kontekst uz koji je vezan. Kombiniranjem kôdova iz dva različita modela dobivamo softver koji je podložniji greškama, nepouzdan i težak za razumijevanje. Komunikacija u timu postaje otežana i često se ne zna u kojem kontekstu se model *ne* koristi. Za vezane kontekste je jako bitno da se razmotre iz više perspektiva.

1. Iz perspektive organizacije timova.

2. Iz perspektive specifičnih dijelova aplikacije u kojima ih koristiti.

3. Iz perspektive konačnog produkta - kôda, sheme baze podataka itd.

Kada definiramo kontekst koji će zadovoljavati sve tri perspektive, možemo definirati model koji će biti vezan za taj kontekst te granice njegove primjene. S jasno definiranim granicama, timovi mogu nesmetano raditi, bez međusobne interferencije. Bitno je pripaziti da je model na razini jednog konteksta potpuno konzistentan i bez kontradiktornih pravila.

Konzistentnost modela se postiže na dva načina: *dobrom komunikacijom* u timu koja se temelji na *sveprisutnom jeziku* i *alatima za gradnju sustava*. **Dobra komunikacija** nam je prva linija obrane od fragmentacije modela. S njom možemo preduhitriti eventualne greške koje se mogu dogoditi zbog nerazumijevanja dijelova modela. Kao njen temelj nalazi se *sveprisutni jezik* kojim osiguravamo da svi ljudi na projektu pod jednim pojmom misle na isti koncept.

Ako se i provuku greške prilikom pisanja kôda, moramo imati mehanizam kojim ih već u ranoj fazi možemo otkriti. Za te potrebe koristimo **alate za gradnju sustava** koji nam pružaju automatiziranu podršku za povezivanje izvornih kôdova, izgradnju sustava i pokretanje niza testova. Takav ciklus još zovemo i *merge/build/test* ciklus. Prolaženjem jednog *merge/build/test* ciklusa možemo uočiti ako je došlo do simultanog mijenjanja istog dijela kôda, ako se sustav ne može niti izgraditi ili ako izgradnjom novog sustava neki od testova ne prolaze.

Kada smo osigurali integritet modela unutar konteksta, trebamo urediti odnose između modela koji su u međusobnoj komunikaciji. Scenarija međusobne komunikacije ima puno.

Primjerice, postoji preklapanje među modelima, jedan model koristi metodu drugog modela drugog konteksta, koristimo baštinjen sustav i sl.. Kada se nađemo u toj situaciji potrebno je vršiti **preslikavanje konteksta**. S preslikavanjem konteksta se stalno susrećemo kada pristupamo relacijskoj bazi podataka. S jedne strane imamo podatke u relacijskoj paradigmi, a s druge strane podatke u objektnoj paradigmi. Ono što će nas zanimati su preslikavanja na razini objekata domene. Koristimo objekte prevoditelje koji objekte jednog konteksta prevode u objekte drugog konteksta.

Ipak, prije nego što krenemo s prevođenjem, bitno je prvo na razini projekta identificirati sve vezane kontekste, odrediti njihove granice, komunikaciju i eventualna preklapanja te ih dokumentirati. Tako nam granice rada razvojnih timova postaju jasnije i lakše uočavamo kada možemo primijeniti jedan od oblikovnih obrazaca. U nastavku ćemo nabrojati nekoliko obrazaca kojima možemo oblikovati preslikavanje konteksta.

Odnosi vezanih konteksta

Zajednička jezgra. Ponekad nekoliko vezanih konteksta imaju jedan manji, ali bitan podskup koji svi koriste (najčešće jezgra domene). U ovakvom slučaju ne želimo da svaki tim implementira isti komad modela, osobito ako je riječ o jezgri domene. Naime, troškovi dupliciranja kôda i preslikavanja konteksta u ovom slučaju mogu biti veliki. Ono što želimo je takav bitan dio konteksta prozvati **zajedničkom jezgrom** koju ne smijemo mijenjati bez dogovora s ostalim timovima koji ju koriste.

Kupac i dostavljač. Često imamo slučaj da jedan tim dobavlja podatke drugom timu, čija suradnja podsjeća na onu između kupca i dostavljača. Preduvjet ovakvoj suradnji je dobra komunikacija timova. Bez nje se može dogoditi da tim kupca ostane bespomoćan jer tim dostavljača ne dostavlja ono što tim kupca želi. Druga stvar koja se može dogoditi jest da tim dostavljača izmjenom svojeg kôda sruši sustav tima kupca. Kako bi to izbjegli, moramo definirati sučelja koja su maksimalno prilagođena potrebama kupca i testovima kojima testiramo željeno ponašanje. Ako nije moguće osigurati dobru komunikaciju među timovima, tada pribjegavamo drugoj strategiji - *konformisti*.

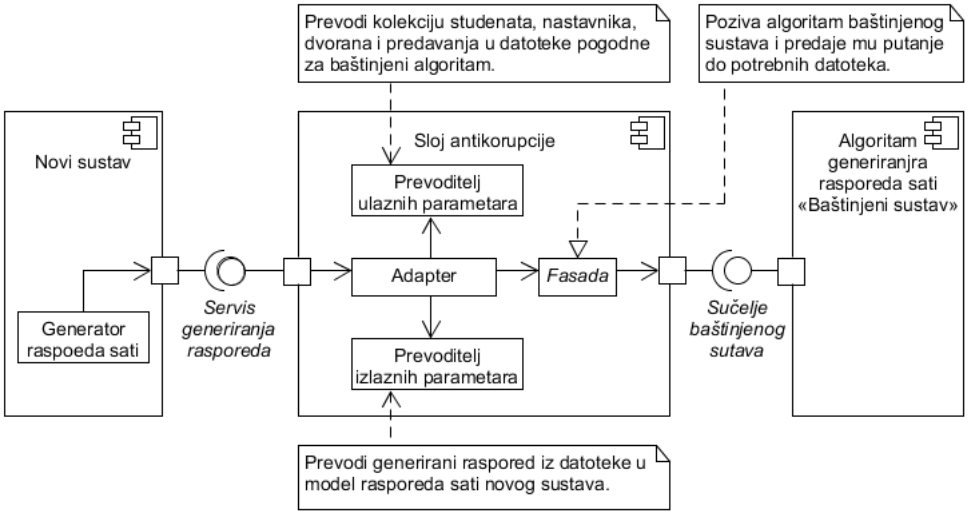
Konformist. Konformist je u suštini vrlo sličan obrascu kupca i dostavljača, samo što dobre komunikacije između timova nema i tim kupca se ne može pouzdati u 'dostavu dostavljača'. S obzirom da tim kupca ne može utjecati na tim dostavljača, ponekad je efikasnije i jeftinije rješenje u potpunosti se prikloniti modelu tima dostavljača. Iako time dobivamo lošiji dizajn i model, iznimno smo pojednostavili integraciju izbjegavajući potrebu za kompleksnim prevođenjem iz modela dostavljača u model kupca. Između ostalog, tim kupca može se posvetiti svom poslu, a da ne ovisi o isporuci tima dobavljača. U slučaju

kada imamo 'sustav dostavljača', koji može biti baštinjen, koristimo se drugim obrascem - obrascem *sloja antikorupcije*.

Sloj antikorupcije. Ovaj obrazac koristimo kada gradimo sustav koji intenzivno koristi neki stariji sustav, u kojem su modeli poprilično slabi. Ono što se može dogoditi je da model novog sustava postane *konformist* starom sustavu te da model novog sustava postane preslika (neprikladnog) modela starog sustava. Ako ne radimo *arhitekturnu transformaciju* starog sustava, želimo imati model koji je prikladan za novi sustav. To činimo s tzv. *slojem antikorupcije*.

Stari sustavi često mogu imati kompleksna sučelja. Zato želimo svu logiku poziva funkcionalnosti starog sustava sakriti od ostatka antikorupcijskog sloja iza jednostavnijeg sučelja tzv. **fasade**. S obzirom na to da fasada koristi drukčiji model od modela starog sustava, trebat će nam **adapter** koji će preslikavati objekte novog sustava u objekt starog sustava i obratno. *Adapter* će interno koristi nekoliko **prevoditelja** za posao preslikavanja. Funkcionalnost koju obavljamo pomoću starog sustava klijentu ćemo ponuditi, naravno, kroz **servis**, tako da klijent ne bude niti svjestan da se iza servisa krije stari sustav.

U našem je primjeru moguć scenarij u kojem naša aplikacija za raspored sati, napisana u programskom jeziku visoke razine kao što je Java, koristi stari algoritam napisan u jeziku niže razine kao što je C. Taj algoritam se poziva preko komandne linije, a podaci o studentima, nastavnicima, dvoranama i predavanjima se nalaze u datotekama čije putanje predajemo algoritmu preko komandne linije.



Slika 4.1: Struktura antikorupcijskog sloja.

Odvojeni putovi. Integracija je uvijek skupa. Ponekada je toliko skupa da su prednosti manje nego troškovi. U tom slučaju treba definirati vezane kontekste tako da nemaju međusobnih veza i da omogućavaju razvojnim timovima nesmetan i jednostavan razvoj svojih rješenja.

Servis otvorenog domaćina i publicirani jezik. U slučaju kad se nekim podsustavom koristi mnoštvo korisnika, prilagođavanje svakom korisniku pisanjem prevoditelja može predstavljati ogroman teret timu koji radi na tom podsustavu. Više toga treba održavati i izmjene su skuplje. U ovom slučaju je jednostavnije definirati pristup podsustavu kroz niz servisa koji su dostupni klijentima. Pristup tim servisima treba omogućiti kroz protokol koji je javno dostupan svim klijentima. Od toga dolazi i naziv **servis otvorenog domaćina**. Pri tome model koji se koristiti za razmjenu podataka ne mora biti model koji domaćin interno koristi.

Kvaliteta 'usluge' koju **otvoren domaćin** pruža klijentima usko je vezana uz taj model koji se koristi za razmjenu podataka i njegovu dokumentiranost. Prekompleksne, loše faktorirane i slabo dokumentirane modele teže je održavati te postoji tendencija da se s vremenom njihov razvoj zamrzne i da se prestanu koristiti. Rješenje je jezik tog modela 'publicirati' ili koristiti već publicirani. Za **jezik** vezan uz model kažemo da je **publiciran** ako je dobro dokumentiran i jasan za upotrebu.

Primjerice, razna sveučilišta i instituti objavljuju skupove podataka i najbolja rješenja za određene optimizacijske probleme, među kojima ima i za problem rasporeda sati. Ponekad su ti skupovi podataka vezani uz određeno natjecanje. U interesu institucije je da privuče što više korisnika koji bi iskušali svoj algoritam u nadi da će naći bolje rješenje. Kako bi to omogućili, uz skupove podataka trebaju pružiti i dobru dokumentaciju kako ih koristiti.

4.2 Destilacija (izdvajanje)

Kako sustav raste, povećava se i broj komponenti koje se u njemu nalaze. Svaka od njih je kompleksna, isprepliće se s ostalima, a u isto vrijeme je neophodna za funkcioniranje softvera. U konačnici, ne možemo više razlučiti srž našeg modela od njegovih komponenti.

Želimo iz čitavog modela izdvojiti komponente modela, posebno pazeći na srž domene. Taj proces nazivamo **izdvajanjem** ili **destilacijom**. Izraz destilacija potječe iz kemije, gdje označava proces kojima se iz neke mješavine izdvajaju komponente. U konačnici su produkti destilacije tj. destilati, vredniji od same mješavine.

U softverskom inženjerstvu, izdvajanje komponenti ima nekoliko prednosti.

- Članovi razvojnog tima imaju jasniju sliku o čitavoj domeni.
- Pospješuje bolju komunikaciju pružajući bolji *sveprisutni jezik*.
- Može nam ukazati na priliku za refaktoriranje.
- Može nam ukazati na priliku za korištenje gotovih produkata.
- Stavlja fokus na bitnija područja u modelu (srž domene).

Srž domene

Najvrednija komponenta modela je ona koja obuhvaća **srž domene**. To je dio modela koji se bavi centralnim zahtjevima korisnika, razlogom našeg razvijanja, onaj karakterističan dio koji aplikaciju čini jedinstvenom. Često se u projektima iskusnije programere postavlja na tehnički zahtjevnije zadatke, dok se funkcionalnosti srži domene ostavljaju manje iskusnim programerima, koji ne pokušavaju razviti dubok model. Kao produkt toga dobiva se model koji ne sadrži duboko znanje domene i dizajn cjelokupnog sustava koji lošije ističe bitne dijelove domene što u konačnici rezultira lošijim softverskim produktom kojeg je teže održavati.

Bitno je u početku razvoja definirati koji dio našeg modela čini *srž domene*, a koji dijelovi pripadaju njezinim potpornim elementima. Kad prepoznamo srž domene, trebamo joj dodijeliti najiskusnije razvojne inženjere koji će znati razviti *dubok model* i *gipki dizajn*.

Takav pristup ima nekoliko prednosti.

- Srž domene se koristi u ostalim dijelovima aplikacije. Ako je ona dobro modelirana, razvoj svih povezanih potpornih sustava je jednostavniji.
- Srž domene je specifična za aplikaciju koju razvijamo, a potporni elementi nisu. Ako razdvojimo srž domene od potpornih elemenata, lakše je uočiti potporne komponente za koje već postoje rješenja ili koje su pogodne za *outsourcing*.
- Jasnim izdvajanjem domene smanjujemo mogućnost da nebitni dijelovi sustava postanu fokus projekta.

Generičke pod-domene

Često postoji dio domene koji je neophodan i izuzetno bitan faktor u modelu, ali sadrži koncepte koje možemo identificirati u raznim poslovnim domenama. Takva pod-domena ne sadrži specifično znanje o domeni, već sadrži generička znanja koja postoje i u raznim

drugim domenama. Takve **generičke pod-domene** treba identificirati, izdvojiti u posebne module i staviti kao dio sustava nižeg prioriteta.

Nakon što identificiramo i izdvojimo jednu generičku pod-domenu, na raspolaganju stoji nekoliko rješenja.

1. *Korištenje COTS (Commercial off-the-shelf) rješenja*. Ponekada je jednostavnije kupiti rješenje ili koristiti rješenje kojem je dostupan izvorni kôd. Umjesto pisanja algoritma za generiranje rasporeda, možemo provjeriti postoji li već dostupan algoritam koji zadovoljava naše potrebe.

2. *Publicirani dizajn ili model.* Neki problemi su toliko česti da već postoje javno dostupni modeli za njihovo rješavanje. Problem sveučilišnog rasporeda sati je jedan od njih. Postoji mnoštvo akademskih članaka na tu temu koji nam mogu dati ideju kako reprezentirati rješenje i koji algoritam koristiti.

3. *Outsourcing.* Posao implementiranja generičke pod-domene možemo prepustiti nekom drugom timu ljudi.

4. *Implementacija unutar kuće.* Sami razvijamo pod-domenu.

Vezani mehanizmi

Jedna od najbitnijih komponenti našeg studijskog primjera je algoritam kojim generiramo raspored sati. Jedan optimizacijski algoritam, kao što je genetski, koristi mnoštvo operacija kao što su: selekcija, križanje, mutacija i evaluacija. Jedna selekcija može biti proporcionalna, a može biti i turnirska. I turnirska i proporcionalna selekcija imaju svoje razne implementacije, a svaka od njih svoje varijacije. Ovakvi detalji, *kako* smo implementirali ovaj algoritam, mogu prevladati *što* radi ovaj algoritam.

Zbog ovakvih složenih računa u sustavima, teško je razlučiti srž domene od njenih komponenti. Preporuka je takve **vezane mehanizme** izdvojiti u aplikacijski okvir (*framework*) koji će svoje funkcionalnosti pružati srži domene preko sučelja. Bitno je ne pretjerati u apstrahiranju i težiti prema 'univerzalnom aplikacijskom okviru', samo ga treba izdvojiti do te mjere da jezgra domene ostaje jednostavnija i jasnija.

Slika 3.2, na kojoj smo ilustrirali povezanost genetskog algoritma i njegovih operacija, upravo prikazuje primjer jednog *aplikacijskog okvira* za generiranje rasporeda sati. Ako se odlučimo koristiti *mravlji algoritam* umjesto genetskog, sve što trebamo je napisati novi algoritam koji implementira sučelje `Algorithm`.

Odvojena jezgra

Slično kao i u prethodnim primjerima, jezgra nam je suviše kompleksna, sadrži mnoštvo potpornih komponenti (generičke pod-domene, vezani mehanizmi). Sve zajedno, imamo nakupinu ovisnosti u kojoj jezgra domene nije posve jasna. U prethodnim smo primjerima identificirali generičke pod-domene i vezane mehanizme te ih izdvajali kako bi dobili jasniju jezgru domene. Ponekad je jednostavnije prepoznati pod-domenu jezgre i izdvojiti nju iz nakupine, refaktorirati ju i ne obazirati se previše na ono što je ostalo. Time je jednostavnije iz ostatka nakupine uočiti neku generičku pod-domenu ili vezni mehanizam.

Apstraktna jezgra

Ponekad je i jezgra domene sama po sebi previše kompleksna. Radi ilustracije ćemo se prebaciti u perspektivu u kojoj implementiramo samo algoritam za generiranje rasporeda sati i ništa više. Više ne možemo govoriti o vezanom kontekstu jer je algoritam jezgra domene, a ne potporni element. I dalje smo suočeni s kompleksnim operacijama kao što su (za genetski algoritam) selekcija, križanje, mutacija i evaluacija, koje u ovom slučaju predstavljaju operacije domene. Umjesto da vežemo naš algoritam za implementaciju operacija, naš algoritam se može sastojati od apstraktnih komponente ili sučelja koje samo definiraju tu operaciju. S obzirom na to da je naša jezgra sastavljena od apstraktnih komponenti, ovako definiranu jezgru nazivamo **apstraktna jezgra**.

4.3 Strukture velikih razmjera

Kada model postane velik, rastavljamo ga na manje dijelove kako bismo mogli na model gledati kao na suradnju njegovih dijelova. Kada model postane *golem*, broj njegovih dijelova je toliko velik da postane teško povezati sve te dijelove u jednu cjelinu. Nedostaje nam drugačiji pogled na model kojim možemo vidjeti ulogu pojedinog dijela modela bez razmišljanja o ostalim dijelovima modela.

Slojevi odgovornosti

Jedan od pristupa pojednostavljivanja velikih modela je definiranje strukture nad tim modelom, najčešće slojevite. Ako u mnoštvu objekata možemo prepoznati određen skup odgovornosti, možemo grupirati te objekte u slojeve po njihovim odgovornostima. Time ćemo jednostavnije naći objekt koji nam treba te na čitav sustav možemo gledati kao na priču koja započinje od vršnog sloja, a završava sa slojem na dnu.

Odgovornosti koje često nalazimo u sustavima su odgovornosti *potencijala*, *operacija*, *politike* i *odluke*.

- U **sloju potencijala** nalazimo resurse (često pohranjeni u bazi podataka) koji sudjeluju u nekoj operaciji. Primjerice, nastavnik, dvorana i kolegij su informacije potrebne za generiranje rasporeda sati.
- U **sloju operacije** nalazimo objekte koji nastaju kao rezultat potencijala, a često referenciraju objekte potencijala (ali ne obratno!). U našem slučaju to je raspored sati.
- U **sloju politike** nalazimo sve objekte koji sadrže neko pravilo ili cilj. Pravilo je da jedan nastavnik ne može predavati na nekoliko kolegija u isto vrijeme, cilj je da generirani raspored mora zadovoljavati određen stupanj dobrote. Ovdje nalazimo objekte koji nalažu ograničenja i strategije koje koristimo u nekom procesu.
- U **sloju odluke** nalazimo objekte koji imaju odgovornost odabira akcije, postavljanja politike i koriste objekte potencijala da generiraju objekte operacije.

Moramo paziti da prilikom primjene slojevite strukture (ili bilo koje strukture) na neki model ne otežamo nepotrebno posao razvojnom timu. Nepromišljena primjena strukture nad nekim modelom može dovesti do toga da razvojni tim mora provesti dodatno vrijeme prilagođavajući svoj kôd nametnutoj strukturi. Struktura je nešto što bi trebalo nastati kao produkt evolucije softvera, kada model postane takav da zahtijeva postavljanje strukture radi lakše čitljivosti. Takav je slučaj i sa studijskim primjerom. U njemu ne nalazimo razuman broj objekata koje bismo rasporedili po slojevima. Model je moguće shvatiti i bez posebne strukture.

Metafora sistema

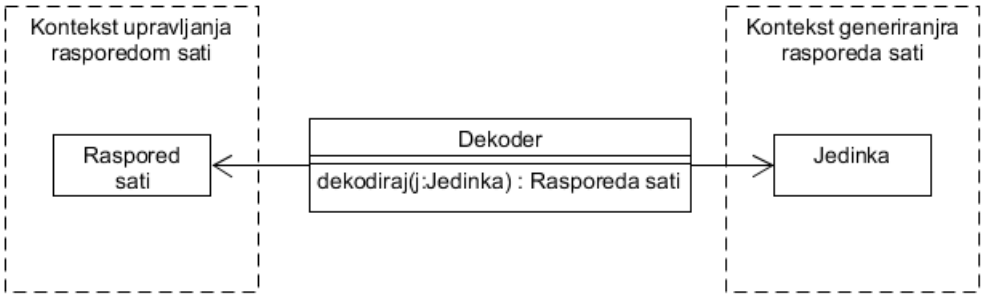
Ponekad je dizajn toliko apstraktan da niti uvođenjem slojeva ne možemo dobiti jasnu sliku sustava. Umjesto da pokušavamo naći prigodnu strukturu, ponekad je jednostavnije naći pogled na sustav koji ima analogiju s opipljivom i zamislivom pojavom u svijetu. Kada pronađemo jednu takvu **metaforu sustava**, čitav dizajn sustava treba preoblikovati prema toj metafori i termine ubaciti u *sveprisutni jezik*. Kad jednom oblikujemo sustav prema metafori potrebno je paziti da nove dijelove sustava prilagodimo toj metafori svojim dizajnom i jezikom.

Dobar primjer metafora sustava su prirodom inspirirani optimizacijski algoritmi, kao što je genetski. Riječ je o algoritmima koji su dobili inspiraciju iz prirode, a uz to su i dalje zadržali termine iz prirode. Kada pričamo o operacijama genetskog algoritma ne govorimo o 'kombinaciji rješenja' i 'manjoj izmjeni rješenja', već koristimo termine *križanje* i *mutacija jedinki* iz genetike. Neki genetski algoritmi definiraju nekoliko populacija koji mogu s vremena na vrijeme izmjenjivati jedinke među sobom. Ljudi koji su oblikovali algoritme su, u duhu metafore, te populacije nazvali selima.

4.4 Primjena strateškog oblikovanja na studijskom primjeru

Očuvanje integriteta domene. U studijskom primjeru možemo uočiti dva vezana konteksta - onaj za upravljanje rasporedom i onaj za generiranje rasporeda. U prvom vezanom kontekstu pogodan je model rasporeda definiran u odjeljku 2.4, dok isti nije pogodan za algoritam koji generira raspored sati. U kontekstu generiranja rasporeda nam je pogodnija *reprezentacija rješenja* tj. *jedinka* genetskog algoritma.

Očuvanje integriteta domene ćemo osigurati pomoću tzv. *dekodera* koji će preslikavati *jedinke* u objekte rasporeda sati definirane u odjeljku 2.4. Taj *dekoder* će koristiti *evaluator* prilikom evaluacije jedinke i genetski algoritam nakon što završi s radom, kada treba vratiti rješenje klijentu. Izostavit ćemo smjer kodiranja rasporeda sati u jedinku jer taj smjer nije nužan za rad genetskog algoritma.



Slika 4.2: Preslikavanje konteksta dekoderom.

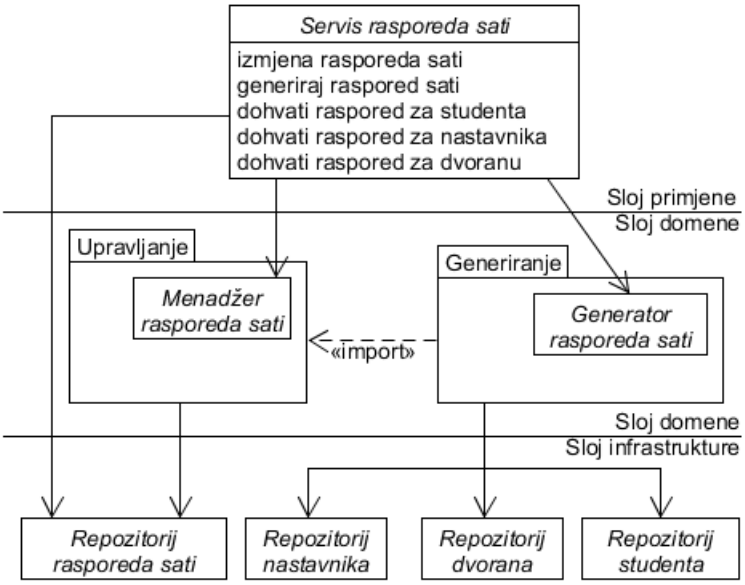
Srž domene. Model rasporeda sati i operacije upravljanja rasporedom su te koje su centralne za aplikaciju i razlikuju ju od drugih aplikacija. Modul za generiranje rasporeda sati je zamjenjivi, može se upotrijebiti već gotov produkt ili se može *outsorcati*. Zato je modul za upravljanje rasporedom sati *srž domene*, a modul za generiranje rasporeda sati *generička poddomena*.

Strateško oblikovanje. Kao što smo ustanovili ranije, studijski primjer nije velik projekt, model je relativno jednostavan i nije potrebno nad njime definirati posebnu strukturu. Na algoritam možemo gledati kao na metaforu i time pospješujemo jednostavnije i prirodnije shvaćanje rada algoritma.

Poglavlje 5

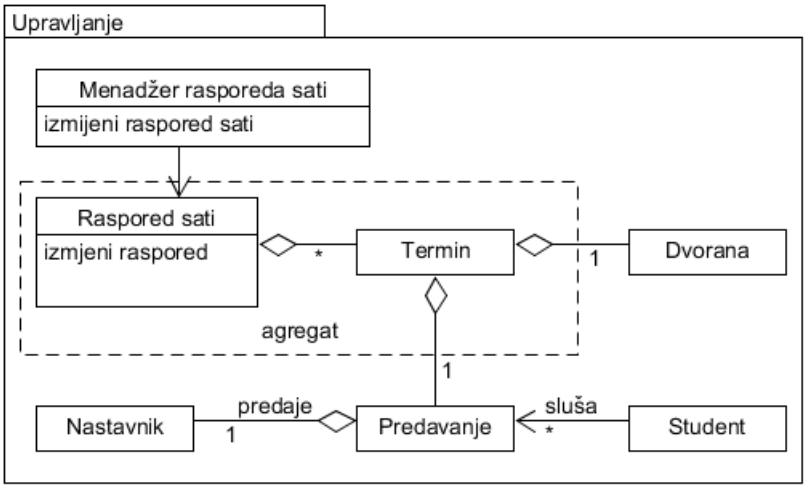
Sažetak studijskog primjera

Kao što smo napomenuli, postoje dva dijela aplikacije za raspored sati - onaj koji obuhvaća proces generiranja rasporeda i onaj koji obuhvaća upravljanje rasporedom sati. U dizajnu ćemo to realizirati kao dva modula - *modul generiranja* i *modul upravljanja*. Objekti modula za generiranje rasporeda, kao što su *evaluator* ili *dekoder* koriste objekte iz modula za upravljanje rasporedom, tako da *modul generiranja* koristi *modul upravljanja*. Funkcionalnosti sloja domene su pruženi sloju primjene preko servisa *Generator rasporeda sati* i *Menadžer rasporeda sati*.



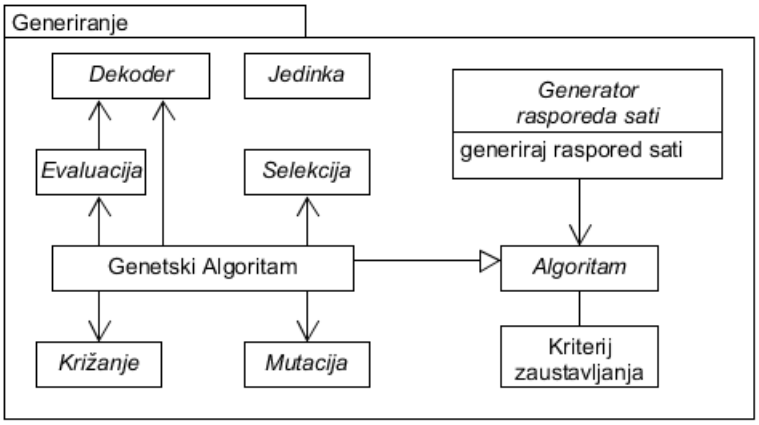
Slika 5.1: Sloj domene i njegov odnos s ostalim slojevima.

Modul upravljanja rasporedom sati sastoji se od elemenata koji su opisani u odjeljku 2.4 i nije se od tog odjeljka značajnije promijenio.



Slika 5.2: Modul upravljanja rasporedom sati.

Modul upravljanja rasporedom sati sastoji se od elemenata koji su opisani u odjeljku 3.3 i odjeljku 4.4. Budući da **jedinku** koriste gotovo svi objekti modula generiranja, njene veze s ostalim objektima modula neće biti istaknute.



Slika 5.3: Modul generiranja rasporeda sati.

Sve komponente algoritma oblikovne su kao sučelja, sukladno onome na slici 3.2. **Evaluator** je oblikovan kao i na slici 3.3. Dodatno koristi i *dekoder* kojim prevodi *jedinku* u *raspored sati* prije provođenja *pravila dobrog raspoređivanja*. **Zaustavni kriterij** je oblikovan kao specifikacija, sukladno kôdovima u odjeljku 3.3. **Genetski algoritam** koristi *dekoder* nakon završetka rada, kada treba prevesti *jedinku* u *raspored sati* koji je prigodan za prikaz na ekranu.

Poglavlje 6

Zaključak

U ovom radu navedeno je mnoštvo dobro poznatih tehnika iz svijeta objektno-orijentiranog programiranja koje nam pomažu razviti bolji softver. Bitno je da čitatelj ovog rada ne zaključi da je oblikovanje vođeno domenom skup oblikovnih obrazaca i pravila iz objektno-orijentiranog programiranja. Oblikovni obrasci su samo naputci za oblikovanje, a objektno-orijentirana paradigma je programska paradigma koja je *trenutno* popularna. Prava vrijednost oblikovanja vođenog domenom leži u načinu razmišljanja u kojem se dobar i održiv sustav razvija tako da njegov dizajn usko korespondira s dotičnom poslovnom domenom.

Središnja aktivnost oblikovanja vođenog domenom je **modeliranje domene**. Model je intermedijator između domene i sustava - on sadrži znanje o domeni koje je bitno za oblikovanje sustava. Svaka nova spoznaja o domeni ulazi u model, a potom i u dizajn sustava. Svaka izmjena u dizajnu sustava treba se odraziti i na model. Time razvijamo model koji sadrži dublje znanje o domeni te sustav koji bolje opisuje poslovna pravila i procese domene. Ako sustav koji razvijamo bolje opisuje domenu, bolje ćemo ga razumjeti, bit će nam ga lakše održavati i možemo biti sigurniji u korektnost njegovog rada.

Provođenje tehnika oblikovanja vođenog domenom ima svoju cijenu. Naime, održavanje modela i provedba izolacija i enkapsulacija predstavlja dodatni posao koji razvojni tim mora obavljati uz razvoj samog sustava. Taj dodatni posao može biti i suvišan ukoliko je riječ o manjem projektu koji ima relativno jednostavnu domenu.

U suvremenom programiranju postoji mnoštvo popularnih aplikacijskih okvira koji nalažu korisnicima da procese domene implementiraju u zasebnom sloju koristeći repozitorije i servise za izoliranje sloja domene od ostalih slojeva. Danas su sve popularnije tehnike *aspektno-orijentiranog programiranja* koje nam omogućavaju da tehničke detalje, kao što su npr. sigurnost ili *logiranje*, odvojimo od objekata domene. Možemo reći da je oblikovanje vođeno domenom pristup koji je našao svoje mjesto u modernom programiranju.

Bibliografija

[1] Evans, *Domain-Driven Design: Tacking Complexity In the Heart of Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003, ISBN 0321125215.

[2] M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley series in object-oriented software engineering, Addison Wesley, 1997, ISBN 9780201895421, `https://books.google.hr/books?id=4V8pZmpwmBYC`.

[3] Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995, ISBN 0-201-63361-2.

[4] Robert Manger, *Softversko inženjerstvo - skripta*, nadopunjeno 2. izadnje., PMF-Matematički odjel, Sveučilište u Zagrebu, rujan 2013.

[5] Marko Čupić, *Prirodom inspirirani optimiziacijski algoritmi. Metaheuristike*, prosinac 2013.

Sažetak

U ovom radu obrađuje se tema oblikovanja vođenog domenom koju je, u istoimenoj knjizi, osmislio i opisao Eric Evans. Principi oblikovanja vođenog domenom ilustrirani su na primjeru sustava za oblikovanje i praćenje rasporeda sati na fakultetu.

Kako bi mogao izgraditi softver koji obavlja korisne aktivnosti, razvojni tim mora biti upoznat s poslovnom domenom tih aktivnosti. Ta domena je često kompleksna i strana razvojnim inženjerima. Potrebno je u razgovoru sa stručnjacima domene procesirati to znanje domene i stvoriti jedan model domene koji će sadržavati znanje o domeni i predstavljati naputak za oblikovanje sustava. Razvijanjem modela razvija se i sveprisutni jezik koji služi kao zajednički jezik između stručnjaka domene i razvojnog tima te kao standardni jezik unutar samog razvojnog tima.

Nakon što se sakupi dovoljno znanja o domeni, potrebno je razvrstati objekte domene na entitete i vrijednosne objekte, a procese definirati kao servise. Objekti koji se pojavljuju u grupi, unutar koje postoje invarijante koje uvijek moraju biti zadovoljene, grupiraju se u agregate. Stvaranje agregata i kompleksnih objekata delegira se tvornicama. Pronalazak i dohvat ranije stvorenih objekata vrši se preko repozitorija.

Tijekom razvoja često se dođe do novih spoznaja o domeni ili do otkrića drukčijeg dizajna koji omogućava da kôd bolje opisuje domenu. U oba slučaja treba refaktorirati dijelove sustava sukladno novim spoznajama i otkrićima. Koncepte koji su izraženi implicitno unutar kôda treba izraziti eksplicitno, dijelove dizajna koji su nezgrapni treba preoblikovati tako da budu *gipkiji*, podložniji izmjenama, s manje međuovisnosti. To se postiže korištenjem tehnika kao što su sučelja koja otkrivaju namjenu, funkcije bez popratnih posljedica, utvrđivanja (*assertions*), konceptualne konture, samostojeće klase, zatvorenost operacija i deklarativno oblikovanje.

U velikim sustavima, u kojima postoje različiti modeli za isto područje domene, potrebno je paziti da očuvamo integritet modela na razini čitavog sustava. Područje domene zna postati toliko veliko i kompleksno da moramo izdvojiti jezgru domene, kako bismo razlikovali srž sustava od njegovih potpornih elemenata. Ponekad je korisno nad sustavom velikih razmjera definirati određenu strukturu kako bismo jednostavnije mogli razumjeti ulogu dijelova u cjelini.

Summary

This thesis is about domain driven design, a design approach that Eric Evans describes in his book of the same title. The principles of domain driven design have been illustrated on an example of a system for designing and monitoring academic timetables.

In order to make a software that does some useful activities, a developmnent team must have knowledge about the underlying business domain. That domain is often complex and unfamiliar to the developers. Developers must engage in knowledge crunching activities with domain experts in order to create a model that contains rich domain knowledge and dictates the design of the software. As a result, a ubiquitous language is developed that is used as a common language between developers and domain experts and as a standard language between the developers.

After certain knowledge is obtained, domain objects are specified as entities or value objects and operations are defined as services. Some objects come in groups and have invariant that always need to be satisfied. They are formed in aggregates. Creating aggregates and complex objects are delegated to factories. Finding and retrieving objects that have previously been created and stored is done through repositories.

During development, refactoring is needed to reflect new insights about the domain or to obtain a design that allows better expression of the domain through code. Implicit concepts must be made explicit, cumbersome parts of the design should be redesigned in a more supple way in order to make it easier to change, with less interdependencies. That is accomplished by using techniques such as intention-revealing interfaces, side-effect-free functions, assertions, conceptual contours, standalone classes, closure of operations and declarative design.

In large systems that have several models for the same domain segment we need to focus on maintaining model integrity on system scale. A domain can be so complex and huge that it is hard to distinguish the core domain from its supporting elements. Sometimes, it is useful to impose a certain structure to a large-scale system in order to better understand the roles of the parts of the system in the whole system.

Životopis

Rođen sam 23. travnja 1991. u Zagrebu, gdje sam završio osnovnu školu Petar Zrinski, a zatim VII. gimnaziju. Položio sam državnu maturu 2010. te sam iste godine upisao *Preddiplomski sveučilišni studiji - Matematika* na matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu.

Tijekom preddiplomskog studija upisao sam i položio na FER-u vještinu *Osnove programskog jezika Java* kod tadašnjeg asistenta Marka Čupića, gdje sam se prvi put susreo s oblikovnim obrascima i izazovima oblikovanja. Od tada sam se trudio postati što bolji u oblikovanju softvera, pa sam odlučio naučiti nešto o oblikovanju vođenom domenom.

Preddiplomski studij sam završio 2013. godine i upisao *Diplomski sveučilišni studiji - Računarstvo i matematika*. Tijekom diplomskog studija proveo sam jedan semestar studirajući u Beču, a zatim sam upisao još jednu vještinu kod asistenta Čupića na FER-u - *Rješavanje optimizacijskih problema algoritmima evolucijskog računanja u Javi*. Na toj vještini sam napisao brojne optimizacijske algoritme te poradio na vještinama oblikovanja. U sklopu kolegija *Softversko inženjerstvo* dizajnirao sam aplikaciju za upravljanje rasporedom sati, a u sklopu kolegija *Računarski praktikum 3* sam ju i implementirao.

Godine 2015. zaposlio sam se u firmi SV Group d.o.o. te počeo pisati ovaj diplomski rad u kojem nastojim znanje oblikovanja vođenog domenom primijeniti na aplikaciji generiranja rasporeda sati.