



Funded by the
Erasmus+ Programme
of the European Union



This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



KI206 - PROCES I METODOLOGIJE RAZVOJA SOFTVERA

Projektovanje i implementacija
softvera

Lekcija 06

PRIRUČNIK ZA STUDENTE

KI206 - PROCES I METODOLOGIJE RAZVOJA SOFTVERA

Lekcija 06

PROJEKTOVANJE I IMPLEMENTACIJA SOFTVERA

- ✓ Projektovanje i implementacija softvera
- ✓ Poglavlje 1: Proces projektovanja objektno-orientisanog softvera
- ✓ Poglavlje 2: Kontekst i interakcije sistema
- ✓ Poglavlje 3: Utvrđivanje svojstava klasa
- ✓ Poglavlje 4: Specifikacija interfejsa
- ✓ Poglavlje 5: Vežba 1 - Projektovanje softvera
- ✓ Poglavlje 6: Šabloni projektovanja i njihove vrste
- ✓ Poglavlje 7: Implementacija softvera
- ✓ Poglavlje 8: Pretvaranje projektnog UML modela u kod
- ✓ Poglavlje 9: Aktivnosti pretvaranja modela u kod
- ✓ Poglavlje 10: Upravljanje implementacijom
- ✓ Poglavlje 11: Vežba – Zadatak za samostalni rad
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

✓ Uvod

UVOD

Cilj ove nastavne jedinice da vas upozna sa osnovama projektovanja objektno-orientisanog softvera primenom UML. Kao programeri, vi nećete raditi na projektovanju softvera, ali koristićete projektnu dokumentaciju, te potrebno da je razumete, jer ćete na osnovu nje razvijati vaše programe . Ova lekcija vam omogućava da:

- razumete najvažnije aktivnosti nekog uopštenog objektno-orientisanog procesa projektovanja;
- razumete različite modele koji se koriste za dokumentovanje nekog objektno-orientisanog projektnog rešenja;
- znate ideju projektnih šablonai kako se one koriste za višestruku upotrebu znanja i iskustva u projektovanju;
- razumevanje ključnih pitanja o kojima se mora voditi računa pri implementaciji (kodiranju) softvera, uključujući i pitanja višestruke primene (engl., software reuse)
- primena preslikavanja, tj. transformacije, objektnog modela softvera u UML obliku , koji je rezultat projektovanja, u programski kod, koji je rezultat implementacije.

✓ Poglavlje 1

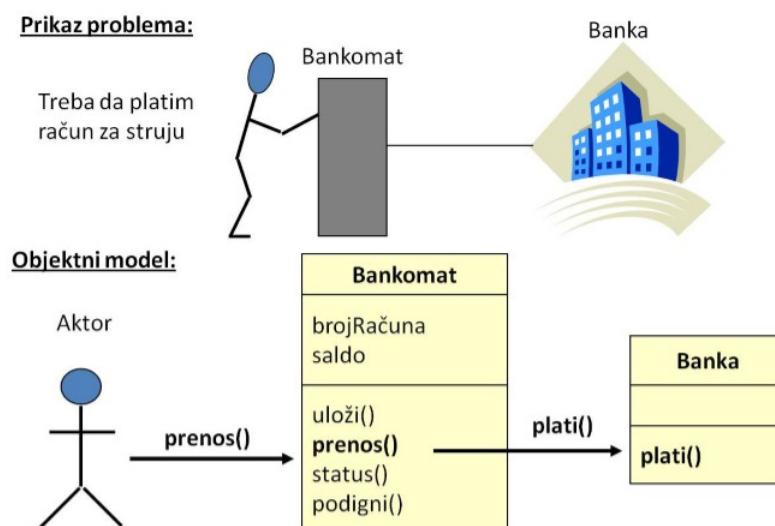
Proces projektovanja objektno-orientisanog softvera

OBJEKTNO-ORIJENTISANO PROGRAMIRANJE

Jedan objektno-orientisano softverski sistem čine interaktivni objekti koji održavaju svoje unutrašnje stanje i obezbeđuju operacije na to stanje

Jedan objektno-orientisano softverski sistem čine interaktivni objekti koje održavaju svoje unutrašnje stanje i obezbeđuju operacije na to stanje. To znači da te operacije, kada se izvrše, menjaju stanje jednog ili više objekata, zavisno koliko objekata su u interakciji kada se jedna operacija izvršava. Stanje objekta odražava trenutno vrednosti njegovih atributa.

Stanje nekog objekta je „privatno“, što znači da ne može da se menja direktnom intervencijom nekog spolja. Ako neki objekt želi da promeni stanje nekog drugog objekta, on mora da tom drugom porukom pošalje odgovarajuću poruku koja bi trebalo da aktivira odgovarajuću operaciju, tj. metod koju objekat koji prima poruku poseduje, a samo ona onda može da izvrši promenu stanja (npr. da promeni vrednost nekog atributa objekta). Slanje jedne poruke između dva objekta, koja pobuđuje odgovarajuću operaciju (metod), ili ceo lanac povezanih operacija, nazivamo interakcijom između dva objekta. Na slici 1 dat je primer plaćanja računa za struju preko bankomata. Pojednostavljen objektni model prikazuje dva objekta u interakciji i jednog aktora (čoveka koji daje nalog za isplatu računa).



Slika 1.1 Plaćanje računa za struju preko bankomata (ATM)

RAD SA OBJEKTIMA

Kada je softverski sistem razvijen i kada je u izvršnom obliku, on sam dinamički stvara potrebne objekte koristeći definicije klase iz modela klase

Ne može bilo koji objekat da šalje poruke bilo kom drugom objektu. To može da radi samo sa objektima sa kojima je u vezi, tj. relaciji. Veze (relacije) između objekata se određuju vezama među njihovim klasama, koje predstavljaju sve objekte iste strukture (tj. tzv. instance klasa), imaju među sobom. Klase jednog objektno-orientisanog sistema definišu objekte koje sistem koristi i njihove interakcije. Zato se koristi model sa klasama koji predstavlja klase i njihove veze koje postoje u nekom softverskom sistemu. Kada je softverski sistem razvijen i kada je u izvršnom obliku, on sam dinamički stvara potrebne objekte koristeći definicije klase iz modela klase.

Objektno-orientisane sisteme lakše je menjati nego sisteme koji su razvijeni primenom nekog od funkcionalnih pristupa projektovanja softverskih sistema.

Kao što je rečeno, objekti sadrže i podatke, ali i operacije koje manipulišu (tj. obradjuju i menjaju) te podatke. Bilo koja promena u klasi, koja predstavlja sve objekte kreirane kao predstavnike, tj. instance te klase, je lokalizovana samo na tu klasu. Ne remeti strukturu i rad drugih klasa koje čine sistem. Ako se želi da se promeni rad nekog objekta, vrši se promena u strukturi njegove klase (menjaju se atributi i operacije(metodi)). Tako se na primer može dodati neki servis koji neki objekat treba da obezbeđuje (drugim objektima).

Kako softverski objekti često odražavaju neke stvarne objekte (npr. komponente računarskog hardvera), to može da postoji jasno preslikavanje objekta realnog sveta u objekte softverskog sistema. To olakšava razumljivost sistema, a samim tim i njegovo održavanje (menjanje rada/ servisa objekata).

PROCES PROJEKTOVANJA SOFTVERA

Definisati kontekst i spoljne interakcije sa sistemom, projektovati arhitekturu softvera, utvrditi glavne objekte sistema, razviti projektne modele sistema i specificirati sve interfejse objekata

Pri projektovanju softverskog sistema, koje se kreće od projektovanja koncepta, pa do detaljnog projektovanja, potrebno je uraditi sledeće:

1. Razumeti i definisati kontekst (problem i okruženje) i spoljne interakcije sa sistemom.
2. Projektovati arhitekturu softvera.
3. Utvrditi glavne objekte sistema.
4. Razviti projektne modele sistema.
5. Specificirati sve interfejse objekata.

Iako su ovi uobičajeni zadaci projektanta softvera, oni ne moraju da predstavljaju i aktivnosti jednog sekvensijalnog procesa razvoja softvera. Proces razvoja softvera je obično složeniji, jer sadrži i povratne sprege, tj. putanje kretanja posla, a i odvijanje i paralelnih, a ne samo sekvensijalnih (rednih) aktivnosti. Na primer, kada počinjete projektovanje nekog sistema, vi počinjete od nekih ideja, razvijate predloge rešenja koje detaljnije razrađujete kada vam neke informacije postaju dostupne. Kasnije u toku procesa projekta, vi vidite da morate da se vratite nekoliko koraka nazad i da nešto promenite, i onda da ponovite te korake.

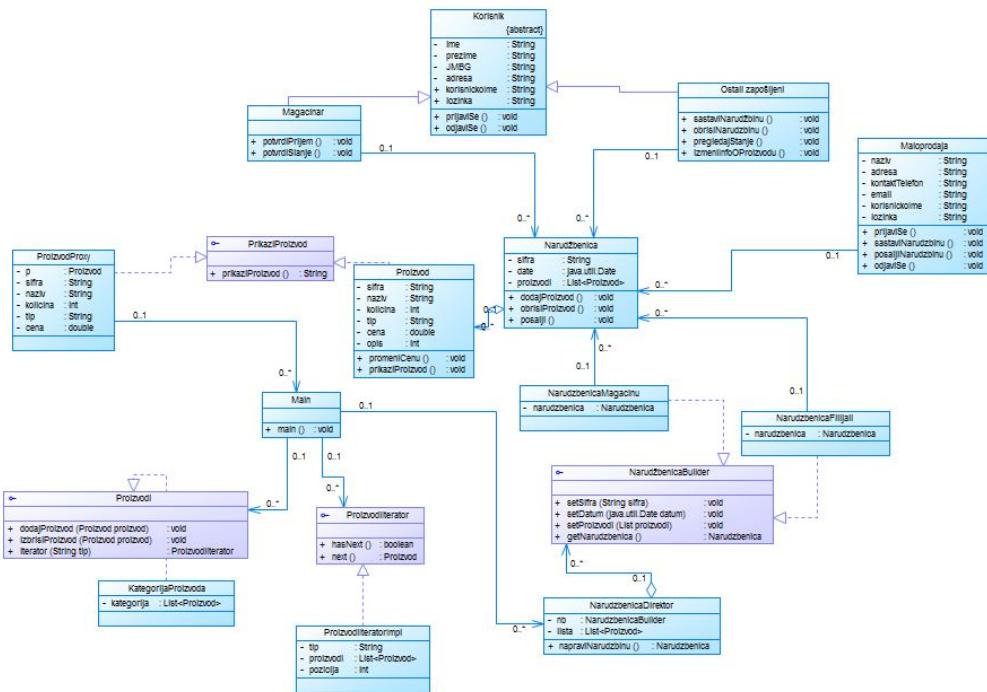
Ako bi taj proces rada opisali nekim dijagramom sa svim aktivnostima procesa, on sigurno ne bi izgledao tako jednostavno, kao pet redno povezanih aktivnosti. Pored toga što svaka glavna aktivnost se može raščlaniti na više drugih, dijagram bi pokazao i više ovih povratnih puteva rada, a i rad paralelnih ili čak, međusobno spregnutih aktivnosti (paralelne aktivnosti treba koristiti svuda gde je moguće, jer se tim skraćuje vreme projektovanja i razvoja sistema).

U prethodnim lekcijama, objasnili smo šta je potrebno raditi u prve dve navedene aktivnosti procesa projektovanja (1 i 2). U četvrtoj lekciji (**Inženjerstvo zahteva**) objašnjen je značaj razumevanje zahteva korisnika i sistema, kao i interakcije sa okruženjem. U petoj lekciji (**arhitektura sistema**) je izložena aktivnost projektovanja arhitekture sistema, te se to neće ponavljati u ovoj lekciji. U trećoj lekciji (Modelovanje sistema) dat je postupak utvrđivanja glavnih, apstraktnih objekata podeljenih u tri osnove kategorije Boundary, Control i Entity, što je osnova za dalje projektovanje klasa sistema.

U toku procesa projektovanja se ovi objekti dalje razrađuju, dele ili dodaju. Međutim, najvažniji doprinos aktivnosti projektovanja je što omogućava definisanje svojstava svih klasa objekata, tj. određivanje njihovih atributa i operacija, kao i potrebnih interfejsa objekata. To će i biti fokus ove lekcije, kao i primena šablonu projektovanja.

PRIMER RADA SA OBJEKTIMA

U primeru je dat rad sa objektima i modelovani klasni dijagram koji predstavlja glavne objekte sistema.



Slika 1.2 Klasni dijagram

Na slici 2 predstavljen je klasni dijagram sa identifikovanim glavnim objektima sistema za upravljanje narudžbenicama. Predstavljene su neophodne klase budućeg sistema. Klasni dijagram predstavlja jedan od osnovnih dijagrama u okviru projektovanja softvera i može se koristiti za kasnije generisanje programskog koda aplikacije.

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci koji obuhvataju rad sa objektima i proces projektovanja softvera.

1. Zadatak: Opis sistema: Sistem za upravljanje internet prodajom obuće. Postoje dve vrste korisnika softvera, to su administrator i običan korisnik. U zavisnosti od vrste korisnika, sistem ima različite funkcionalnosti.

Administrator bi morao prvo da se uloguje kako bi mogao da menja sadržaj proizvoda na sistemu, odnosno, da dodaje, menja postojeće i briše određene proizvode iz baze podataka. Kako bi mogao da koristi ovaj sistem, korisnik prvo mora da napravi nalog preko koga bi se kasnije prijavljivao ukoliko želi da koristi funkcije datog sistema. Kupac ima različite mogućnosti, da pregleda obuću po kategorijama, da bira odgovarajući broj, boju, itd. Takođe, proizvod može da doda u korpu i na taj način ga poruči.

Modelovati klasni dijagram i identifikovati glavne objekte sistema.

2. Zadatak: Opis sistema: Sistem "Dnevnik" ima za cilj da omogući obradu podataka vezanih za studente. Što znači omogućava da se unose svi potrebni podaci vezani za studente u bazu kao što su osnovne informacije, predmeti koje slušaju i ocene za predmete koji su položeni. Takođe potrebno je omogućiti prikaz unetih podataka o studentima u obliku tabele, histogramski prikaz i prikaz u obliku liste. Podaci će moći da se brišu iz baze, takođe i da

se sortiraju po imenu i prezimenu. Program će omogućavati pretragu svih studenata koji se nalaze u bazi po imenu, prezimenu i po oba kriterijuma zajedno.

Modelovati klasni dijagram i identifikovati glavne objekte sistema.

▼ Poglavlje 2

Kontekst i interakcije sistema

ODNOS SOFTVERA I OKRUŽENJA

Razumevanje konteksta u kome radi sistem omogućava i određivanje granica sistema, a na osnovu njih se odlučuje šta treba da ima, kao i svojstva drugih, povezanih sistema.

Prva stvar koja se treba uraditi prilikom projektovanja softverskog sistema je da se razume odnos između softvera koji se projektuje i spoljnog okruženja. Ovo je bitno za donošenje odluke o tome kako da se obezbedi zahtevana funkcionalnost sistema i kako da se napravi struktura sistema za njegovu komunikaciju sa okruženjem. Razumevanje konteksta u kome radi sistema omogućava i određivanje granica sistema (tj. šta on „pokriva“ a šta ne).

Na osnovu uspostavljenih granica sistema, odlučujete koja svojstva sistem treba da ima, kao i svojstva drugih, povezanih sistema. Analiziraćemo primer sistema za praćenje stanja vremenskih prilika koji koristi više distribuiranih stanica za merenje vremenskih parametara (temperatura vazduha, pritisak, vlažnost, brzina vetra i dr.). Pri projektovanju ovog sistema treba da odlučite i kako da raspodelite (distribuirate) funkcionalnost sistema između kontrolnog sistema i stanica za merenje vremenskih parametara, kao i unutar ugrađenog računarskog sistema u svakoj stanici.

Modeli konteksta sistema i modeli interakcija predstavljaju komplementarne poglede na veze između sistema i okruženja:

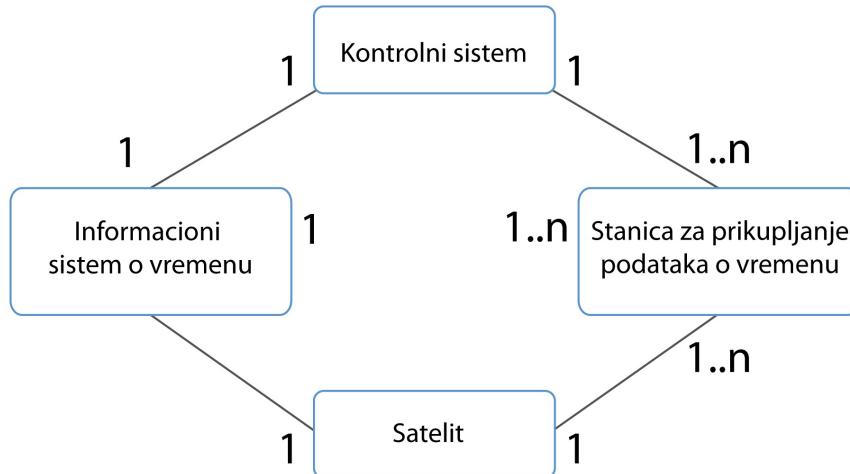
1. **Model konteksta sistema je** strukturni model koji pokazuje druge sisteme u okruženju sistema koji se razvija.
2. **Model interakcija je dinamički model** koji pokazuje kako sistem komunicira sa okruženjem pri svom radu.

MODEL KONTEKSTA SISTEMA

Model konteksta sistema sadrži asocijacije između sistema, koji se projektuje, i okruženja, tj. drugih sistema sa kojima sistem treba da bude u nekoj vezi

Model konteksta sistema sadrži asocijacije između sistema koji se projektuje, i okruženja, tj. drugih sistema sa kojima sistem treba da bude u nekoj vezi. Asocijacije pokazuju da ima neke veze između entiteta koje su u asocijaciji. To je vrlo uopšteni opis veze između dva ili više entiteta, jer ne specificira bliže tip i svojstva te veze. To je slično kada za dve osobe kažete

da su povezane. Da li su u rođačkoj vezi, prijateljskoj vezi, i dr., to još ne saopštavate. Znači, vrlo uopšteni iskaz. Na primer, na slici 1 su date asocijacije (veze) između informacionog sistema za praćenje vremenskih prilika, satelitskog sistema i kontrolnog sistema. To su sistemi u vezi koji predstavljaju okruženje svake stanice za praćenje vremenskih prilika. Brojevi na krajevima veza pokazuju kardinalnost, tj. broj očekivanih entiteta na strani veze na kojoj se nalazi broj. Vidi se da prikazano okruženje čine jedan informacioni sistem, jedan satelitski sistem, jedan kontrolni sistem, a više stanica za praćenje vremenskih prilika. Znači, model konteksta pokazuje samo strukturu okruženja sistema koga projektujemo i razvijamo.

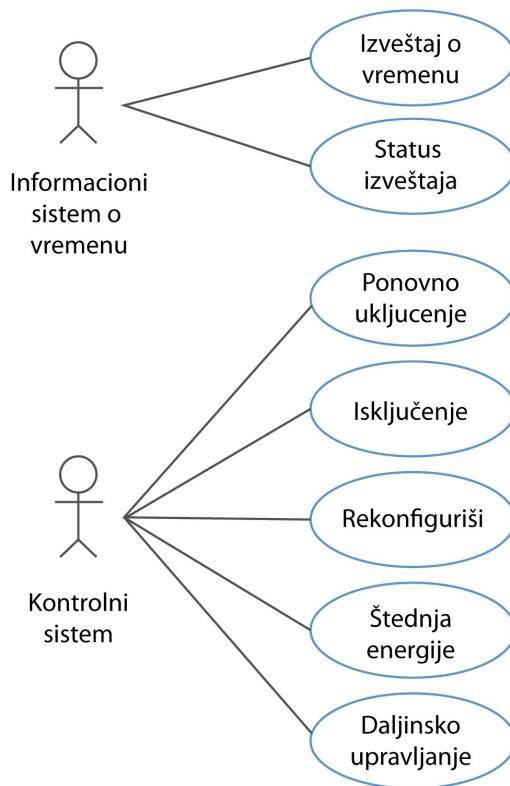


Slika 2.1 Kontekst sistema za stanice za praćenje vremenskih prilika

MODEL INTERAKCIJE

Model interakcija daje dodatne informacije, jer ukazuje kako sistem koji projektujemo komunicira sa okruženjem, tj. sa drugim sistemima u svom okruženju (a na koje ukazuje model k

Model interakcija daje dodatne informacije, jer ukazuje kako sistem koji projektujemo komunicira sa okruženjem, tj. sa drugim sistemima u svom okruženju (a na koje ukazuje model konteksta). Jedan od UML model interakcija je UML dijagrama korišćenja (engl., use case), koji opisuju po jednu interakciju sa sistemom koji projektujemo. Simbol aktera označava ili neki drugi sistem ili čoveka koji koristi naš sistem. Na slici 2 je prikazan dijagram korišćenja koji pokazuju slučajevе korišćenja stanica za praćenje vremenskih prilika, tj. sistema koji se projektuje. Svaka elipsa simboličko označava po jedan slučaj korišćenja (koji se bliže opisuje tekstualnom informacijom u vidu scenarija).



Slika 2.2 Slučajevi korišćenja stanica za praćenje vremenskih prilika

PRIMER OPISA SLUČAJA KORIŠĆENJA

Dodatne tekstualne informacije u strukturisanom prirodnom jeziku dopunjuju dijagramske prikaze slučajeva upotrebe.

Primenom modela konteksta i modela interakcija, projektant definiše glavne objekte sistema i objašnjava šta to sistem treba da radi. Dodatne tekstualne informacije u strukturisanom prirodnom jeziku (slika 3) dopunjuju dijagramske prikaz slučajeva upotrebe. Na primer, slučaj korišćenja (upotrebe) sistema „Izveštaj o vremenu“, predstavljen na dijagramu na slici 2 jednom elipsom, detaljnije je tekstualno opisan na slici 3 .

Sistem	Stanica za praćenje vremenskih prilika
Slučaj korišćenja	Izveštaj o vremenu
Akteri	Informacioni sistem o vremenu, Stanica za praćenje vremenskih prilika
Opis	Stanica za praćenje vremenskih prilika šalje sumarni izveštaj sa vremenskim podacima koji su prikupljeni od instrumenata u toku određenog vremenskog intervala koje definiše informacioni sistem. Ti podaci predstavljaju maksimalne, minimalne i srednje vrednosti temperature vazduha, vazdušnog pritiska, brzine vetra, ukupan nivo padavina i pravac vetra u intervalima od pet minuta.
Stimulans	Informacioni sistem o vremenu sadrži link za komunikaciju sa satelitom koji je u vezi sa stanicama za praćenje vremenskih prilika zahteva prikupljene podatke.
Odgovor	Sumarni prikaz podataka poslatih informacionom sistemu o vremenu.
Komentari	Od stanica za praćenje vremenskih prilika obično treba da na svaki sat pošalju podatke o vremenu, ali ta frekvencija slanja može da bude različita po stanicama, a može i da se menja u budućnosti.

Slika 2.3

ZADACI ZA INDIVIDUALNI RAD

Zadaci koji se odnose na kontekst i interakcije sistema.

- 1. Zadatak:** Za sistem za upravljanje insulin pumpom (opisan u prethodnim lekcijama) potrebno je razviti model konteksta sistema. Potrebno je predstaviti strukturu okruženja navedenog sistema, prikazati veze između identifikovanih entiteta kroz uopšteni iskaz.
- 2. Zadatak:** Korišćenjem modela interakcije prikazati kako sistem za upravljanje insulin pumpom komunicira sa okruženjem (drugim sistemima u svom okruženju). Za prikazivanje modela interakcije koristiti UML dijagram korišćenja.
- 3. Zadatak:** Za svaki identifikovani slučaj korišćenja (kroz model interakcije) izvršiti detaljan opis slučajeva korišćenja sistema za upravljanje insulin pumpom (za prikaz opisa koristiti tabelu na slici 3). Navesti aktere, opis slučaja korišćenja, stimulans, odgovor i komentar vezan za slučaj korišćenja.

▼ Poglavlje 3

Utvrđivanje svojstava klasa

MODELI SISTEMA

Modeli projektovanja, ili modeli sistema, prikazuju objekte ili klase objekata sistema. Oni prikazuju i asocijacije (veze) između ovih entiteta (objekata ili klasa).

Modeli projektovanja, ili modeli sistema, prikazuju objekte ili klase objekata sistema. Oni prikazuju i asocijacije (veze) između ovih entiteta (objekata ili klasa). Ovi modeli su svojevrsni „most“ između aktivnosti utvrđivanja zahteva i aktivnosti implementacije sistema. Oni bi trebalo da budu dovoljno apstraktni kako bi sakrili nepotrebne detalje, ali bi trebalo da uključe dovoljno detalja potrebnih programerima za njihov rad.

Da bi se ovi oprečni zahtevi uspešno zadovoljili, najčešće se koriste više modela sistema sa različitim nivoom apstrakcije (uopštavanja). Zato, kritična odluka u projektovanju sistema je odluka o nivou detalja koje model sistema, ili dela sistema treba da sadrži.

Modeli zavise i od vrste sistema koji se projektuje. Na primer, ako je u pitanju sistem za sekvencijalnu (rednu) obradu podataka, model se razlikuje od modela za slučaj nekog ugrađenog računarskog sistema. UML nudi 13 različitih tipova modela, ali u praksi se najčešće koristi samo nekoliko. Korišćenje manjih broja modela smanjuje troškove projektovanja, kao i potrebno vreme.

Pri primeni UML u projektovanju sistema, obično se razvija dve vrste modela projektovanja (sistema):

Strukturni modeli, koji opisuju statičku strukturu sistema upotreboom klase objekata i njihove veze (relacije). Važne veze su i one koje se naseleđuju (od superklasa) ili koje se uopštavaju i prebacuju na superklase, kao i veze tipa „upotrebljava/upotrebljen od“ i kompozitne (složene) veze.

Dinamički modeli, koji opisuju dinamičku strukturu sistema i pokazuju interakcije između objekata sistema. Te interakcije najčešće obuhvataju: niz zahteva za servisima koje generišu objekti i promene stanja objekata (npr. vrednosti njihovih atributa) do kojih dolazi zbog dejstva interakcija, tj. poruka koje neki objekt dobija od drugog objekta.

Od ovih modela, najviše se koriste:

- **dijagram klasa** koji prikazuje klase nekog od podsistema softverskog sistema koji se projektuje
- **sekvencijalni model**, koji pokazuje redosled interakcija između objekata sistema

- **model stanja**, koji prikazuje stanja i njihovu promenu usled dejstva pojedinih događaja kod objekata

Prvi je strukturni model, a druga dva su dinamički modeli.

DETALJNO PROJEKTOVANJE OBJEKATA SISTEMA

Primena UML slučajeva korišćenja sistema pomaže da se utvrde ne samo objekti već i operacije u sistemu koje su neophodne za njegov rad

Kada ste odredili glavne objekte koji čine strukturi softverskog sistema (tj. njegovu arhitekturu), treba da dalje detaljnije projektujete ove objekte. Primena UML slučajeva korišćenja sistema pomaže da se utvrde ne samo objekti već i operacije u sistemu koje su neophodne za njegov rad. Na primer, iz opisa slučaja upotrebe stanice za praćenje vremenskih prilika na nekoj lokaciji, jasno je koji su instrumenti potrebni toj stanici da bi mogla da obezbedi tražene podatke o vremenu.

Na osnovu utvrđenih neophodnih glavnih objekata koje sistem treba da ima, sada je mogućno početi sa utvrđivanjem klasa objekata koje softverski sistem treba da obezbedi. Za to utvrđivanje klase, mogu se koristiti nekoliko tehnika:

1. *Upotreba analiza gramatike prirodnog jezika* u kome je dat tekstualni opis sistema. Objekti i njihovi atributi su imenice, a operacije ili servisi su glagoli u tom tekstu.
2. *Upotreba opštijih pojmoveva za pojedine reči* u tekstualnom opisu sistema:

- a) opipljivi entiteti (stvari) u aplikacionom domenu, kao što je avion
- b) uloge (engl. roles), kao što su menadžer ili doktor,
- c) događaji (engl. events), kao što su zahtevi,
- d. instrukcije, kao što su sastanci
- e.) lokacije, kao što su kancelarije
- f) organizacione jedinice, kao što su kompanije, itd.

Upotreba analize scenarija datih u okviru slučajeva upotrebe. Svaki scenario se analizira da bi se utvrdili zahtevani objekti, atributi, i operacije.

Upotrebom ovih tehnika, kao i izvora kao što su dokumenti koji opisuju zahteve za sistem, informacije prikupljene diskusijama sa korisnicima, i analizom postojećih sistema dobijaju se potrebne informacije.

U lekci br. 5 (Modelovanje i analiza sistema) dali smo način kako identificirati apstraktne objekte i kako ih razvrstati po kategorijama: Boundary, Control i Entity. Tada smo napomenuli da apstraktne klase nemaju ni atribut ni operacije, u fazi analize. Međutim, u fazi projektovanja sistema, moraju se naći veze (asocijacije među klasama, moraju se odrediti atribute i operacije klase). U ovom poglavljtu ćemo videti kako se to radi.

UTVRĐIVANJE ASOCIJACIJA (VEZA) IZMEĐU OBJEKATA

Utvrđuju se uglavno analizom glagola u tekstu scenarija

Asocijacije između objekata se mogu utvrditi analizom tekstova scenarija datih u okviru slučajeva korišćenja. Od posebnog značaja su fraze kao što su: "ima", "deo je", "upravlja", izveštava ", podtsaknute sa..." "je sadršan u ..." "govori nekome", uključuje")

Svaka asocijacija treba da ima svoj naziv, a na svakom kraju bi trebalo da budu upisane uloge.

Ovo su preporuke za utvrđivanja asocijacija objekata (klasa):

- analiziraj rečenice sa glagolima.
- Precizno imenuj asocijaciju i uloge na krajevima
- Upotrebi kvalifikatore što ćešće, da bi identifikovao prostor imena i ključne atribute.
- Ukloni bilo koju asocijaciju koja je dobijena iz druge asocijacije.
- ne obraćajte pažnu na kardinalnost sve dok se ne stabilišu asocijacije
- Isuviše mnogo asocijacija čini model nečitkovim.

Asocojacije sa agregacijom označavaju pripadnost objekta nekom drugom objektu. One s elako nalaze u tekstu koji opisuje neku strukturu i šta pripada toj strukturi. Ako niste sigurni u agregaciju, možete, bar na početku, koristiti običnu asocijaciju tipa "one to many".

UTVRĐIVANJE ATRIBUTA

Atributi objekta se utvrđuju analizom teksta scenarija, podvlačenjem imenica.

Atributi su svojsva objekata (slika 1) i treba ih specificirati samo ako su od značaja za sistem. Ostale atribute možete ignorisati. Na primer, informacioni sistem univerziteta ne koristi infomraciju o broju pasoša studenta, te taj podatak ne bi trebalo da nude autribut objekta Student. Međutim, za neki drugi sistem, na primer, informacioni sistem turističke agencije, taj podatak može biti potreban, te ga tada i treba koristiti kao atribut.

EmergencyReport
emergencyType:{fire,traffic,other}
location:String
description:String

Slika 3.1 Atributi obajekta

Ne smatraju se atributima koji zavrednost imaju neke druge objekte, Zato se prvo definišu asocijacije, pa tek onda atributi.

Atributi objekta se utvrđuju analizom teksta scenarija, podvlačenjem **imenica**. U slučaju Entity objekata, svaka informacija ili podatak kojaj mora da se trajno čuva, tj. stavi u trajnu memoriju, treba da bude atribut Entity objekta.

Atributi su najnestabilniji deo objekta. Nije problem da neke uključimo i pri kraju porektovanja.

Preporuke za utvrđivanje atributa:

1. Ispitaj posesivne rečenice u scenarijima
2. Predtsavit memorisan stanje kao atribut Entity objekta
3. Opiši svaki atribut.
4. Ne predstavi objekat kao atribut. Koristi za to asocijaciju ka tom objektu.
5. Ne gubi vreme na detaljima sve dok struktura objekta ne postane stabilna.

Entity objekti najčešće imaju karakteristike koje ih identificuju od strane aktera sa kojima komuniciraju. Jedan od atributa objekta bi trebalo da bude identifikator po kome mu se akteri obračaju. Na primer, za studenta, to je broj indeksa.

UTVRĐIVANJE OPERACIJA PRIMENOM SEKVENCIJALNIH DIJAGRAMA

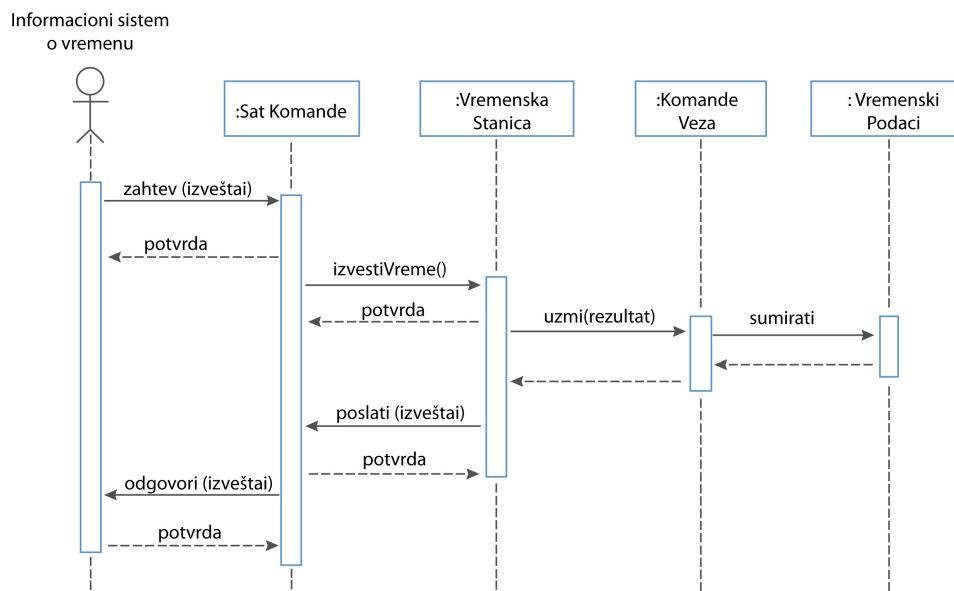
Sekvencijalni dijagrami se koriste za modelovanje kombinovanog ponašanja grupe objekata.

U fazi projektovanja najčešće se definišu **glavne klase podsistema**, te njihovi dijagrami klasa ne sadrže sve klase koje će sistem imati na kraju razvoja. Drugi deo klasa se definišu u fazi implementacije, tj. kada programeri pišu kod i određuju vrlo detaljno projektno rešenje svakog pod sistema.

Sekvencijalni dijagrami se pravi za svaku značajniju interakciju. Kada napravite modul slučaj akorišćenje, trebalo bi da napravite za njega i sekvencijalni dijagram (model). Na slici 2 je prikazan sekvencijalni dijagram koji pokazuje interakcije između objekata stanice za prikupljanje podataka o vremenskim prilikama kada stanica primi zahtev za slanje prikupljenih podataka. Sekvencijalni dijagram se čita odozdo ka dole (duž vremenske ose). Sekvencijalni dijagrami se koriste za modelovanje kombinovanog ponašanja grupe objekata.

Svaka poruka određuje operaciju primajućeg objekta, a parametri poruke, mogu biti atributi primajućeg objekta.

Objekat *SatKomunikacija* osluškuje i prima poruke spolnjih sistema, dekodira te poruke i pokreće rad stanice, koju predstavlja objekat *VremenskiPodaci*. Ova dva objekta mogu da rade paralelno (istovremeno).



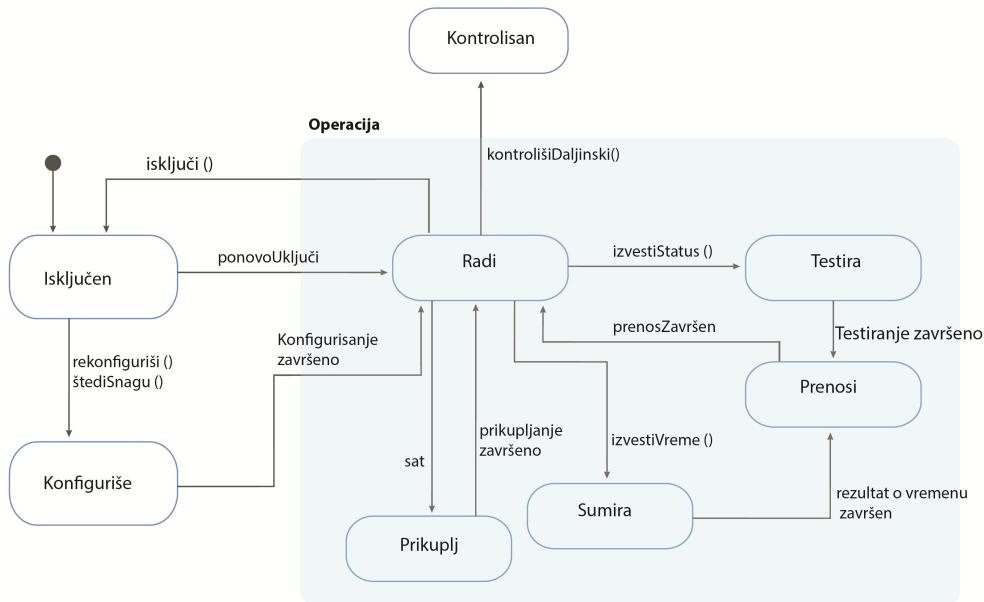
Slika 3.2 Sekvencijalni dijagram koji opisuje interakcije neophodne za prikupljanje podataka

DIJAGRAM STANJA

Dijagram stanja ne pokazuje, kao sekvencijalni dijagram, ko šalje poruke, već samo poruke koje objekt prima, i stanja u koja ulazi kao njegovu reakciju na primljene poruke

Međutim, potrebno je i videti kako koji objekt menja stanje (vrednost svojih atributa) kada dobije neku poruku, tj. kada se javi neki događaj koji pobuđuje neku njegovu operaciju. Za tu svrhu se koristi **dijagram stanja objekta**. Na slici 3 prikazan je primer dijagrama stanja za slučaj stanice za prikupljanje podataka o vremenskim prilikama (ili kraće, stanica) koji pokazuje kako stanica, kao objekt sistema, reaguje na različite servise, tj. događaje (poruke) u svom okruženju. Na dijagramu se vide stanja u koji dolazi objekt koji predstavlja stanicu, kada dobije navedene poruke. Dijagram stanja ne pokazuje, kao sekvencijalni dijagram, ko šalje poruke, već samo poruke koje objekt prima, i stanja u koja ulazi kao njegovu reakciju na primljene poruke. Do promene stanja dolazi zato što ove poruke aktiviraju bar jednu operaciju (metod) objekta koji vrši promenu vrednosti bar jednog atributa objekta.

Dijagrami stanja su korisni za uopštenje (apstraktne) modele sistema ili za opisivanje rada nekog objekta. Obično se ne koriste za sve objekte sistema. Objekti koji su jednostavni, ne moraju se opisivati dijagramima stanja.



Slika 3.3 Dijagram stanja objekata koji predstavlja stancu za prikupljanje podataka o vremenskim prilikama

UTVRĐIVANJE PONAŠANJE OBEKATA USLOVLJENO NJIHOVIM STANJEM

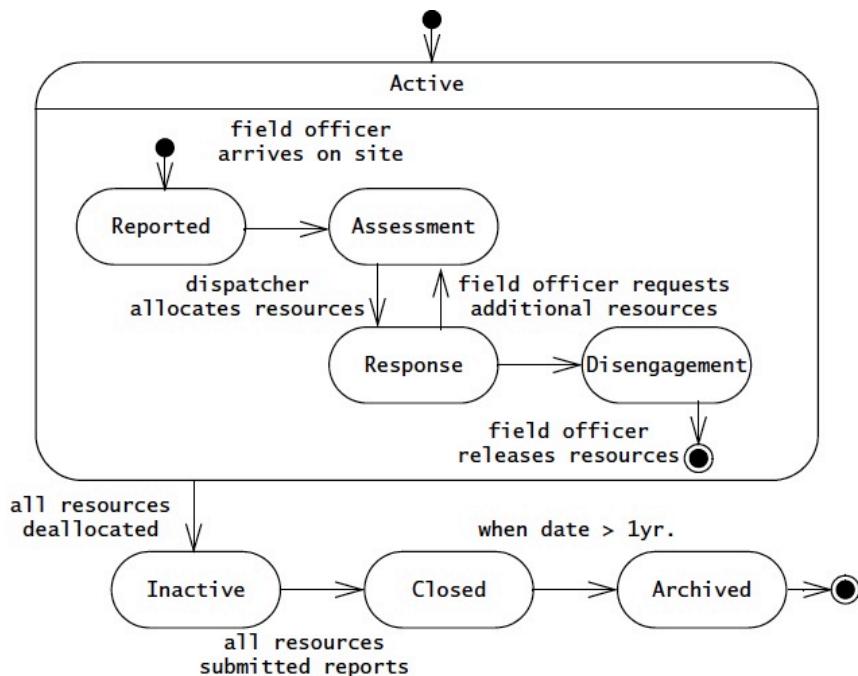
Dijagram stanja pomaže projektanu da otkrije nove slučajevе korišćenja i da doradi postojeće

Sekvencijalni dijagrami distribuiraju ponašanje sistema po objektima i utvrđuju operacije koje objekti treba da realizuju. Sekvencijalni dijagrami predstavljaju ponašanje sistema iz perspektive jednog slučaja korišćenja.

Dijagram stanja jednog objekta predstavlja ponašanje iz perspektive samo jednog objekta. Ovo omogućava da projektant sistema pripremi više formalnog opisa ponašanja objekta, i da utvrdi i nedostatak nekih slučajeva korišćenja. Fokusiranjem na pojedinačna stanja objekta, projektanti mogu da identifikuju i novo ponašanje.

Nije potrebno razviti dijagram stanja za svaki objekat sistema. Samo objekti koji imaju duži životni ciklus i čije ponašanje se menja stanjem u kome se nalaze se uzimaju u obzir za razvoj dijagrama stanja. To su najčešće Control objekti, ređe Entity objekti, a skoro nikada Boundary objekti.

Na slici 4 prikazan je primer jednog dijagrama stanja. Njegovom analizom, projektant može da primeti da li ima definisane sve potrebne slučajevе korišćenja koji prate objekat u svim njegovim stanjima, od otvaranja, pa do zatvaranja. Projektant može dodati neke detalje na neke akcije korisnika, koje menjaju stanje objekta.



Slika 3.4 Dijagram stanja objekta Incident.

PRIMER: KLASE STANICA ZA PRAĆENJE VREMENSKIH PRILIKA

Kada se utvrde osnovni objekti i njihova glavna svojstva (atributi i operacije), vrši se dalje, detaljnije projektovanje ovih objekata

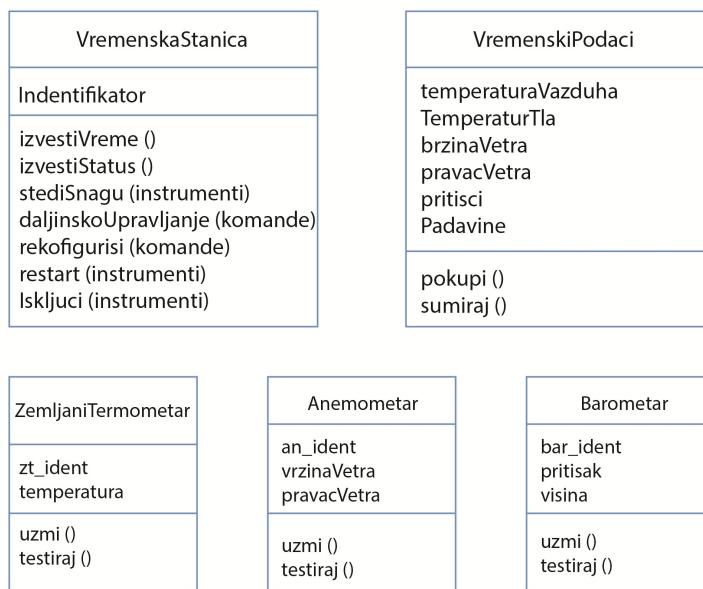
Na slici 5 prikazano je pet klasa utvrđenih za slučaj stanice za praćenje vremenskih prilika, a koje su utvrđene primenom navedenih tehnika. To su Termometar na zemlji, Anemometar (za merenje brzine vetra), Barometar (za merenje pritiska), VremenskaStanica i VremenskiPodaci.

Objekat VremenskaStanica obezbeđuje interfejs stanice sa okruženjem. Ovaj objekt sadrži sve operacije koje pominje scenario koji opisuje njegov rad.

Objekat VremenskiPodaci je odgovoran za obradu vremenskog izveštaja. On šalje informacionom sistemu sumirane podatke dobijene od instrumenata koje sadrži stanica.

Objekti Termometar, Anemometar i Barometar predstavljaju klase koje su direktno povezane sa instrumentima koje predstavljaju. Sadrže neophodne operacije za kontrolu ovih instrumenata. Ovi objekti autonomno prikupljaju u određenoj frekvenciji i lokalno ih smeštaju u memoriju objekata. Na poseban zahtev, ovi podaci se šalju o

Kada se utvrde osnovni objekti i njihova glavna svojstva (atributi i operacije), vrši se dalje, detaljnije projektovanje ovih objekata. Analiziraju se zajednička svojstva klasa, da bi se definisala eventualna hijerarhija klasa. Na primer, u slučaju stanice za



Slika 3.5 :Klase objekata stanice za praćenje vremenskih prilika

prikupljanje vremenskih prilika, može se definisati superklasa za sve instrumente, npr. nazvana Instrument koja sadrži zajednička svojstva, kao što su identifikator, i operacije za čitanje i testiranje podataka. Ovde se mogu dodati i novi atributi u super klasi (Instrument), kao što je frekvencija prikupljanja podataka.

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci koji se odnose na projektovanje objekata sistema, utvrđivanje asocijacija između objekata, atributa i operacija.

1. Zadatak: Izvršiti detaljno projektovanje sistema za upravljanje insulin pumpom. Definisati objekte sistema, modelovati klasni dijagram, razraditi veze između objekata i njihove attribute. Navesti operacije identifikovanih objekata.

2. Zadatak: Modelovati klasni dijagram i dijagram stanja za sistem za prodaju PC komponenti. Sistem je namenjen firmama koje žele da naveliko kupuju PC komponente za svoje radnje. Napravljen je tako da olakša proces naručivanja i isporuke svih porudžbina. Podaci i informacije o svim porudživanjima čuvaće se u bazi podataka i time smanjiti broj potrebne dokumentacije za sve narudžbine.

Glavne komponente od kojih će sistem da se sastoji su komponenta za naručivanje i isporuku PC delova, komponenta za upravljanje proizvodima, komponenta za informisanje klijenata i komponenta za upravljanje nalogom.

▼ Poglavlje 4

Specifikacija interfejsa

ŠTA JE INTERFEJS?

Interfejs nekog objekta definiše poruke na koje objekat reaguje. Svaka poruka se definiše tzv. potpisom tj. nazivom operacije koju objekat sadrži i njenih parametara.

Interfejs nekog objekta (tj. klase objekata) definiše poruke na koje objekat reaguje. Svaka poruka se definiše tzv. *potpisom* (**signature**), tj. nazivom operacije (metoda) koju objekat sadrži i argumenata (podataka) koji su neophodni da bi ta operacija mogla uspešno da se izvrši.

Interfejs ne određuje kako će objekt, tj. klasa koja predstavlja taj objekt, primeniti i izvršiti tu operaciju. Zato se projektovanje interfejsa i projektovanje klase mogu raditi paralelno (istovremeno). Vi možete kasnije menjati način kako se operacija izvršava, a da pri tom ništa ne menjate u interfejsu (ako se signature poruka ne menjaju). To je jedan od prednosti objektno-orientisanih sistema, jer promene koje se rade unutar jedne klase, ne moraju da izazivaju nikakve promene u komunikaciji sa drugim klasama, ako se signature poruka koje razmenjuju (interakcije), ne menjaju.

Projektovanje interfejsa se svodi na definisanje signatura i semantike servisa koje obezbeđuje neki objekt ili grupa objekata (klasa). U UML-u, interfejs se predstavlja isto kao i klasa, samo što nema deo koji je namenjen atributima. Semantika interfejsa se može definisati primenom OCL jezika (Object Constraint Language).

Interfejs sadrži listu servisa, ili operacija, koje objekt (ili klasa) objekt koji je povezan sa interfejsom nudi spoljnjem okruženju. Te operacije, menjaju vrednosti podataka u objektu (set metodi) ili čitaju i prosleđuju njihove vrednosti (get metodi).

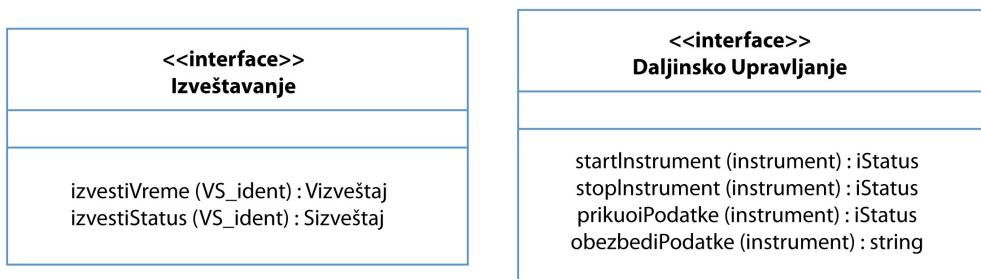
Jedan objekt može imati i više interfejsa. To se primenjuje najčešće kada se želi da pruže različiti pogledi na objekt, ili da se ograniči pristup određenoj grupi korisnika, samo određenom skupu servisa. U Javi se interfejs posebno definiše, tj. nezavisno od objekta koji realizuje operacije (metode) navedene u interfejsu.

S druge strane, jedan interfejs može da obezbedi pristup i većem broju objekata.

PRIMER: INTERFEJSI VREMENSKE STANICE

Dva interfejsa koji su definisani za primer stanice za prikupljanje podataka o vremenskim prilikama. Oni definišu četiri operacije koje realizuje jedan metod klase VremenskaStanica.

Slika 1 prikazuje dva interfejsa koji su definisani za primer stanice za prikupljanje podataka o vremenskim prilikama. Na levoj strani je interfejs koji je namenjen izveštavanju, jer sadrži spisak operacija koje se nalaze u objektu VremenskaStanica i koje pripremaju podatke za izveštaje koji se traže. Drugi interfejs, s desne strane, je namenjen za podršku upravljanja iz daljine, jer obezbeđuje četiri operacije opravljivanja. Međutim, iako interfejs navodi četiri operacije, one se sve realizuju u objektu od strane samo jednog metoda objekta VremenskaStanica. Njen metod daljinskaKontrola izvršava sve četiri operacije



Slika 4.1 Interfejsi objekta VremenskaStanica

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci koji obuhvataju specifikaciju interfejsa u procesu projektovanja.

1. Zadatak: Aplikacija za prepoznavanje kuhinjskog posuđa: Aplikacija bi radila na mobilnim uređajima sa Android operativnim sistemom i davala bi rezultate na osnovu uslikane fotografije. Analizom fotografije obrađivali bi se rezultati i korisniku slala informacija o prepoznatom kuhinjskom posudu. Za svakog korisnika bi se u bazi čuvale informacije o količini posuđa koje poseduje, tj. o njegovoj virtualnoj kuhinji. Nakon uslikavanja novog posudja korisnik ima opciju da sačuva nov element u svoju virtualnu kuhinju ili da odustane.

Za opisanu aplikaciju potrebno je izvršiti detaljnu specifikaciju interfejsa i definisanje operacija koje se izvršavaju.

2. Zadatak: Chatspace sistem simulira osnovne funkcionalnosti jedne chat aplikacije. Sistem omogućava korisniku da pristupi sistemu sa svojim nalogom na osnovu koga dobija sve privilegije korisnika sistema. Korisnik dobija mogućnost da razmenjuje poruke sa drugim korisnicima sistema. Poruke mogu biti u okviru teksta, priče, pesme, i ostalih tekstualnih objava. Korisnik ima mogućnost da poruke šalje drugim korisnicima. Takođe, korisnik ima mogućnost da čita poruke drugih korisnika kao i uvid u listu ostalih korisnika koji koriste sistem.

Za opisanu aplikaciju potrebno je izvršiti detaljnu specifikaciju interfejsa i definisanje operacija koje se izvršavaju.

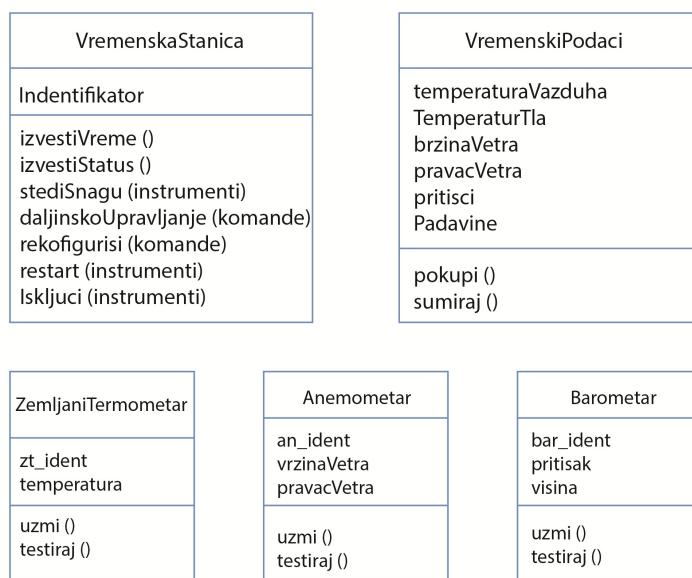
✓ Poglavlje 5

Vežba 1 - Projektovanje softvera

ZADACI 1 I 2 ZA INDIVIDUALNI RAD

Zadaci koji se odnose na deo lekcije koja se bavi metodama projektovanja softvera, bez primene šablon projektovanja.

1. Zadatak: Za studiju slučaja - Sistem za prikupljanje sa stanica za merenje metereoloških podataka, definisane se mnoge klase. Na slici 1 prikayano je samo jedan deo tih klasa. Utvrdite druge oobjekte tog sistema, analizom slučajeva korišćenja i njihovih scenaria. Napravite hiojerarhijsku strukturu klasa tih objekata, uz korišćenje i svojstva nasleživanja.



Slika 5.1 Deo utvrđenih klasa saistema za merenje meteroloških podataka

2. Zadatak: Razvite projektno rešenje stanice za prikupljanje metereoloških podataka, koje će pokazati interakciju između podsistema i instrumenata koji prikupljaju metereološke podatke. Upotrebite sekvensijalni dijagram da prikažete tu interakciju.

ZADACI 3-5 ZA INDIVIDUALNI RAD

Ovi zadaci očekuju da student uradi projektovanje traženog softverskog sistema.

3. Zadatak: Utvrdite moguće objekte sledećih sistema, i za njih razvite objektno-orijetisano projektno rešenje. Možete sami usvojite predpostavke potrebne za ovo projektovanje:

- a) Sistem za upravljanje vremenom i dnevnicima zaposlenih u jednoj firmi. Sistem ima za cilj da omoguća zakazivanje sastanaka grupe zaposlenih. Kada je potrebno organizovati sastanak određene grupe zaposlenih, sistem analizira njihove elektronske rokovnike, i nalazi vremenski slot u kome su svi oni slobodni i zakazuje sastanak, obaveštavajuće sve članove grupe. Ukoliko ne može da nađe slobodan vremenski slot, sistem pregovara sa korisnicima tako da promenama u svojim rokovnicima, omoguće zajednički vremenski slot za sastanak.
- b) Stanica za prodaju goriva za automobile projektna je da automatski rad, bez radnika da sipe goriva. Vozač treba da ubaci svoju kreditnu karticu u čitač kartica, koji je povezan s apumpom. Kartica se verifikuje komunikacijom sa kompjuterm iydavaoca kartice, određuje se gorna granica goriva koja se može uzeti. Vozač onda može da sipa gorivo. Kada završi sipanje i vrati crevo na svoje mesto, sistem zadužuje kreditnu karticu za vrednost sisanog goriva. Kreditna kartica se vraća vozaču, od strane čitača. Ako kreditna kartica nije ispravna ili važeća, pumpa je vraća vozaču pre nego što bi počelo punjenje (nema punjenja goriva).

4. Zadatak: Uradite sekvencijalni dijagram za zadatak 3.a.

5. Zadatak: Uradite dijagram stanja sa zadatak 3.a

▼ Poglavlje 6

Šabloni projektovanja i njihove vrste

ŠTA JE ŠABLON PROJEKTOVANJA?

Šabloni i jezici za šablone su načini opisivanja najbolje prakse, dobrih projektnih rešenja, i prikupljanje iskustva na način koji omogućava drugima da ponovo upotrebe to iskustvo.

Šablon (**pattern**) je opis problema i suština njegovog rešenja koje se može više puta koristiti u različitim slučajevima. Šabloni su način da se ponovo upotrebni znanje i iskustvo drugih projektanta. One obezbeđuju ponovnu upotrebljivost projektnih rešenja koja su se pokazala dobrim u praksi.

Šablon ne daje detaljnu specifikaciju, već više opis akumuliranog iskustva i znanja, u primeni oprobanog rešenja nekog zajedničkog problema. Hillside Group daje sledeću definiciju: „*Šabloni i jezici za šablone su načini opisivanja najbolje prakse, dobrih projektnih rešenja, i prikupljanje iskustva na način koji omogućava drugima da ponovo upotrebe to iskustvo.*“

Obično se neko projektno rešenje softverskog sistema objašnjava navođenjem šabloni koja je korišćenja za njen razvoj. Šabloni za projektovanje se najčešće koriste pri projektovanju objektno-orientisanih sistema. Zato mnoge šabloni primenjuju objektne karakteristike, kao što je nasleđivanje svojstava i poliformizam. Međutim praksa primenjivanja učaurivanja (engl., encapsulation) se može primeniti i kod drugih sistema (koji nisu objektno orijentisani). Jedan šablon najčešće treba da sadrži bar sledeća četiri elementa:

1. **Naziv šabloni** koji odražava upotrebljivost Šabloni.
2. **Opis problemskog područja** u kome se šablon može primeniti.
3. **Opis rešenja** za delove sistema, njihovih veza i odgovornosti. To nije opis rešenja, već uzorak za njegovu izradu, koji se kreira u svakoj korektnoj situaciji. To je obično u formi dijagrama klase.
4. **Iskaz u posledicama**, tj. o rezultatima i učinjenim kompromisima pri primeni šablonia Na taj način korisnici šablonu (projektanti sistema) mogu da se lakše odluče da li da koriste ponuđenu mustru.

Korisno je istaći zašto je uzorak koristan i opisati situacije u kojima se šablon može uspešno primenjivati. Opis rešenja obuhvata strukturu šablonu, učesnike, kolaboracije i implementacijuš

OSNOVNE VRSTE ŠABLONA PROJEKTOVANJA

Šabloni se mogu klasifikovati na sledeće osnovne vrste: šabloni kreiranja, šabloni strukture, šabloni ponašanja i J2EE šabloni

Pri projektovanju objektno-orientisanih softverskih sistema treba koristiti poznate (objavljene) šabloni za projektovanje softverskih sistema, jer oni su rezultat uspešne prakse u projektovanju ovakvih sistema. Pri tome se koriste dva principa u vidu preporuka:

1. Programirajte objekat (primerak klase) a ne implementaciju
2. Dajte prednost kompoziciji umesto nasleđivanju

Šabloni projektovanja su parcijalna rešenja opštih problema. Šablon projektovanja čini mali broj klasa koji pute delegiranja ili nasleđivanja, obezbeđuju robusno i prilagodljivo rešenje. Te klase se mogu prilagoditi i dopuniti pri projektovanju specifičnog sistema koji se razvija. Šabloni projektovanja daju standardnu terminologiju i specifični su za određeni scenario.

Na primer: Singleton šablon. On pojednostavljuje upotrebi jednog objekta. Svi inženjeri koji prave jedan objekat saopštavaju jedni drugima da koriste Singleton šablon

Šabloni projektovanja (engl. **design patterns**) obezbeđuju najbolja rešenja određenih problema iz prakse razvoja softvera. Njihovom primenom, neiskusni softver inženjer uči kako da projektuje softver na lak i brzi način.

Šabloni se mogu klasifikovati na sledeće osnovne vrste:

1. Šabloni kreiranja
2. Šabloni strukture
3. Šabloni ponašanja
4. J2EE šabloni

Šabloni kreiranja daju način kreiranja objekata sa skrivenom logikom kreiranja, tj. bez korišćenja operatora **new**. Na taj način program ima veću fleksibilnost u kreiranju objekata u određenom slučaju korišćenja.

Šabloni strukture se bavi kompozicijom klasa i objekata. Koncept nasleđivanja se koristi da bi se definisali interfejsi i da bi se definisao način postavljanja objekata radi dobijanja nove funkcionalnosti

Šabloni ponašanja se prvenstveno bave komunikacijom između objekata

J2EEE šabloni se prvenstveno bave prezentacionim slojem sistema (npr. GUI). Ovi šabloni su utvrđeni od strane Sun Java Center.

PRIMENA ŠABLONA PROJEKTOVANJA

Za uspešnu primeni projektnih šablon potrebno je odgovarajuće iskustvo u njihovom korišćenju. Potrebno je da prepozname situacije u kojima se može primeniti neki šablon.

Pri projektovanju softverskog sistema, uzmite uvek u obzir mogućnost korišćenja odgovarajućeg šablonu, jer to ubrzava projektovanje, a obezbeđuje i pouzdanost sistema, jer se koristi već provereno projektno rešenje. Postoje knjige sa prikazima različitih šablonu (mustri) za projektovanje softverskih sistema, i njih treba izabrati i kreativno primeniti.

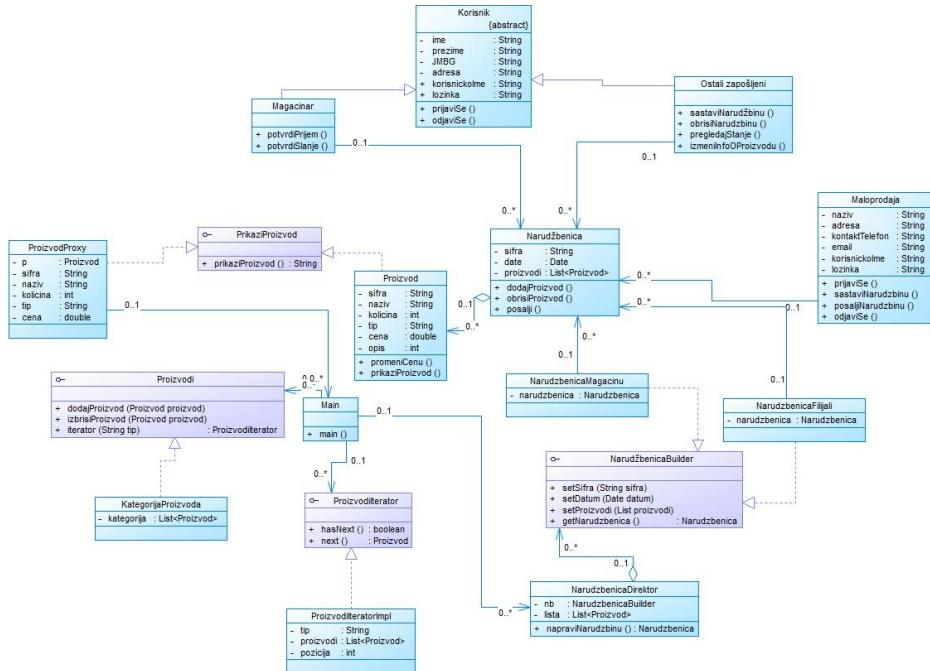
Primenom šablonu vrši se ponovna upotreba već razrađenih koncepcijskih rešenja. U slučaju ponovne upotrebe izvršnih komponenti, javlja se problem ograničenja koja se javljaju usled odluka vezanih za detaljno projektovanje, a koje su doneli implementatori (programeri) tih izvršnih komponenata. Ta ograničena mogu biti vezana za izabrane algoritme u tim komponentama, ali i za tipove interfejsa komponenata. U slučaju kada su ova vrlo konkretna rešenja u sukobu sa zahtevima postavljenim pri projektovanju novog sistema, ponovna upotreba takvih komponenti postaje nemoguća ili unosi veliku neefikasnost rada sistema. Zbog toga, upotreba šablonu (mustri) ne znači upotrebu gotovih, izvršnih komponenata, već samo ponovnu upotrebu ideja, koncepata, iskustva rešavanja sličnog problema, što otvara mogućnost reorganizacije sistema tako da se može upotrebiti odgovarajuća šablon.

Ako se radi na tom, u opštem nivou, mogu se naći odgovarajuće šabloni u knjizi projektnih šablonu i uspešno primeniti. Međutim, ako je vaš problem vrlo specifičan, onda je verovatno da ne možete naći odgovarajuću šablon koju bi mogli da primenite u vašem projektovanju rešenja.

Za uspešnu primeni projektnih šablon potrebno je odgovarajuće iskustvo u njihovom korišćenju. Potrebno je da prepozname situacije u kojima se može primeniti neka šablon. Neiskusni projektanti imaće teškoća da izaberu odgovarajući šablonu iz knjige raspoloživih šablonu

PRIMER PRIMENE ŠABLONA PROJEKTOVANJA

U primeru je dat klasni dijagram sistema u kome je izvršena primena šablonu projektovanja.



Slika 6.1 Klasni dijagram sistema

Builder šablon je iskorišćen za kreiranje narudžbenice. U sistemu bi postojale dve vrste narudžbenica – ona koja se šalje filijali i ona koja se šalje magacinu, odnosno one koju šalje maloprodaja i one koju zaposleni šalje magacioneru. Razlika između ove dve implementacije interfejsa narudžbina je jedino u šifri narudžbenice pri čemu se dodaju ili dve nule ili dve jedinice u zavisnosti od toga o kojoj se narudžbenici radi.

PRIMER PROGRAMSKOG KODA U KOME JE PRIMENJEN ŠABLON PROJEKTOVANJA

Primer prikazuje programski kod interfejsa i klasa u kojima je primenjen šablon projektovanja.

Problem je rešen primenom builder šablona koji gradi narudžbenicu, koji je jedini kome klijent pristupa. Klijent ne zna koji konkretni builderi postoje. On pristupa samo direktoru i builderu i time gradi odgovarajuću narudžbenicu.

```
public interface NarudzbenicaBuilder {  
    void setSifra(String sifra);  
    void setDatum(Date datum);  
    void setProizvodi(List proizvodi);  
    Narudzbenica getNaruzbenica();  
}
```

Slika 6.2 Narudžbenica builder interfejs

```
public class NarudzbenicaDirektor {
    private NarudzbenicaBuilder nb;
    public List<Proizvod> lista;
    public NarudzbenicaDirektor(NarudzbenicaBuilder nb) {
        this.nb = nb;
    }

    public Narudzbenica napraviNarudzbinu() {
        lista = new ArrayList<Proizvod>();
        lista.add(new Proizvod("333", "jabuka", 3));
        lista.add(new Proizvod("555", "kruska", 5));
        lista.add(new Proizvod("666", "mleko", 6));
        nb.setDatum(new Date(2016, 10, 27));
        nb.setProizvodi(lista);
        nb.setSifra("3355");
        return nb.getNaruzbenica();
    }
}
```

Slika 6.3 Narudžbenica direktor klasa

```
public class Narudzbenica {
    private String sifra;
    private Date date;
    private List<Proizvod> proizvodi;

    public Narudzbenica() {
    }

    public Narudzbenica(String sifra, Date date, List<Proizvod> proizvod) {
        this.sifra = sifra;
        this.date = date;
        this.proizvodi = proizvod;
    }

    public String getSifra() {
        return sifra;
    }

    public void setSifra(String sifra) {
        this.sifra = sifra;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
```

Slika 6.4 Klasa narudžbenica

ZADACI ZA SAMOSTALNI RAD

Zadaci se odnose na primenu šablonata projektovanja u procesu razvoja softvera.

1. Zadatak: Prvi deo sistema: Poslovni sistem namenjen ugostiteljskim objektima ima za svrhu da obezbedi lako vođenje evidencija o porudžbinama napravljenim u objektu, kao i automatizacija procesa izdavanja fiskalnih računa. Međutim, sistem nije namenjen samo radnicima u objektu već i gostima, koji dolaze u dodir sa sistemom preko veb sajta. Na veb sajtu ugostiteljskog objekta gosti mogu videti sve informacije o lokaluu i napraviti rezervaciju.

Za ovaj deo sistema definisati i obrazložiti primenu šablona projektovanja. Definisati deo opisanog sistema u kome je moguće izvršiti primenu.

2. Zadatak: Drugi deo sistema: Konobari imaju mogućnost pregleda svih stolova u njihovom sektoru i mogu da prave porudžbine za svaki od tih stolova. Takođe, konobari imaju mogućnost da pregledaju sve artikle koji su dostupni u objektu. Na osnovu napravljenih porudžbina konobar može da generiše fiskalni račun koji će biti odštampan na kasi. Pored toga, konobar je zadužen i za unošenje inventara u toku nedeljnih popisa i ima mogućnost pregleda napravljenih rezervacija.

Da li je moguće primeniti šablon projektovanja na ovaj deo sistema? Modelovati klasni dijagram dela sistema i prikazati primenu šablona projektovanja ukoliko je to moguće.

▼ Poglavlje 7

Implementacija softvera

ŠTA JE IMPLEMENTACIJA SOFTVERA?

Implementacija obuhvata razvoj programa primenom programskih jezika visokog ili niskog nivoa, ili pak, korišćenje opštih, gotovih sistema koji se onda prilagođavaju zahtevima korisnika.

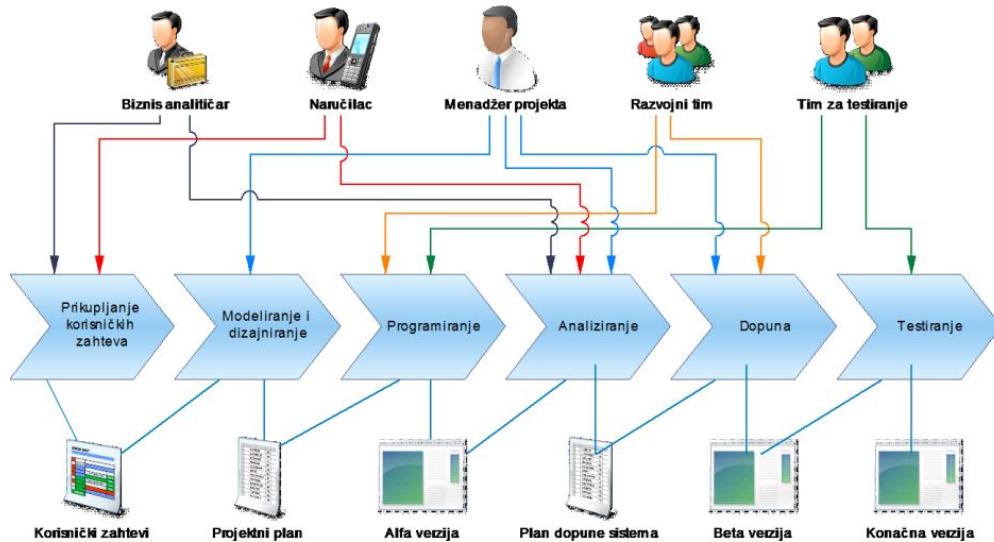
Softversko inženjerstvo obuhvata sve aktivnosti procesa razvoja softvera, počev od specifikacije početnih zahteva sistema, pa do održavanja i upravljanja razvijenim i primjenjenim sistemima. Kritična faza ovog procesa je implementacija sistema, kada se kreira izvršna verzija softvera. Implementacija obuhvata razvoj programa primenom programskih jezika visokog ili niskog nivoa, ili pak, korišćenje opštih, gotovih sistema koji se onda prilagođavaju i uklapaju u sistem koji zadovoljava specifične zahteve organizacije.

Ovde se nećemo baviti dobrom programerskom praksom i primenom konkretnih programskih jezika, jer se to izučava u posebnim predmetima. Umesto toga, ovde ćemo razmatrati posebno interesantne i važne aspekte implementacije softvera, tj. razvoja izvršnog softvera, a koje se obično ne izučavaju u predmetima programiranja. To su sledeći aspekti:

1. **Ponovno korišćenje:** Savremeni softverski sistemi koriste ponovno upotrebljive komponente ili sisteme. Pri razvoju svog softvera, trebalo bi u što većoj meri da koristite već razvijen i provere kod, a to obezbeđuje primena ranije razvijenih i primjenjenih softverskih komponenti.
2. **Upravljanje konfiguracijom:** Prilikom procesa razvoja, kreiraju se mnoge verzije svake od softverskih komponenata. Ako ne vodite računa o njima, primenom posebnog sistema za upravljanje konfiguracijom softvera, može se desiti da koristite pogrešne verzije komponenti u vašem sistemu. Konfiguraciju softvera čini skup softverskih komponenta sa ispravnim verzijama.
3. **Razvoj na razvojnem serveru a za rad na ciljnem serveru:** Obično se softver razvija na serveru koji ne mora da bude istog tipa kao i server na kome će softver raditi. To otvara puno izazova u razvoju o kojima se mora voditi računa.

PRIMER PROCESA IMPLEMENTACIJE SOFTVERA

Dat je primer procesa implementacije softvera.



Slika 7.1.1 Tok procesa implementacije softvera

Na slici 1 dat je tok procesa implementacije softvera gde je prikazana verzija sistema shodno različitim fazama. Kao primer uzećemo sistem za upravljanje insulin pumpom. Nakon prikupljanja korisničkih zahteva, izrade projektnog plana dolazi se do alfa verzije aplikacije u fazi programiranja. U toj verziji postoje osnovne funkcionalnosti sistema kao što su merenje nivoa šećera u krvi. Vrši se analiziranje alfa verzije (od strane menadžera projekta) i predlaže se mogućnost dopune testiranih funkcionalnosti. U fazi dopune razvojni tim radi na unapređenju alfa verzije shodno navedenim dopunama i tada se dolazi do beta verzije. U ovom slučaju vrši se unapređenje merenja nivoa šećera u krvi. Beta verzija sadrži sve funkcionalnosti budućeg sistema i takođe prolazi kroz proces testiranja od strane tima za testiranje i ukoliko su rezultati testiranja zadovoljili potrebe korisnika spremna je konačna verzija. Nakon konačne verzije moguće je izvršiti konfigurisanje shodno potrebama i zahtevima budućih korisnika.

ZADACI ZA SAMOSTALNI RAD

Zadaci se odnose na proces implementacije softvera.

- 1. Zadatak:** Za proizvoljnu aplikaciju definisati proces implementacije softvera. Podeliti proces na faze, definisati učesnike u fazama i odrediti zadatke shodno svakoj identifikovanoj fazi.
- 2. Zadatak:** Opisati razvoj na razvojnem serveru proizvoljne aplikacije. Identifikujte i opišite nedostatke razvoja aplikacije na razvojnem serveru a koja će raditi na drugom ciljnom serveru.

✓ 7.1 Ponovna upotreba softvera

NIVOI PONOVNE UPOTREBE SOFTVERA

Razvoj softvera koji se višestruko koristi se vrši na više nivoa: apstraktni nivo (projektni šabloni), nivo objekta (objekti iz biblioteke), nivo komponenti i nivo sistema

Do 90-tih godina, softver je najčešće razvijan „od početka“, za svakog naručioca posebno, korišćenjem programskih jezika višeg nivoa za pisanje koda (programa). Ponovno korišćenje softvera je bilo prisutno samo u slučaju korišćenja programa za pojedine uobičajene funkcije iz programske biblioteke. Međutim, pritisak da se brže i što jeftinije razvije softver, pogotovu poslovnih sistema, doveo je do korišćenja ranije razvijenog softvera, tj. do *razvoja softverskih proizvoda koji se višestruko koriste*.

Ponovna upotreba softvera je moguća na više različitih nivoa:

1. **Apstraktni nivo:** Na ovom nivou nema direktnе ponovne upotrebe softvera, već se koristi znanje primjeno prilikom projektovanja softvera. Koriste se projektni šabloni i strukturni šabloni kao apstraktna znanja za ponovno korišćenje.
2. **Nivo objekta:** Na ovom nivou ponovno se koriste objekti iz biblioteke. Pri razvoju softvera, treba naći pogodnu biblioteku objekata i koristiti objekte čije metode obezbeđuju potrebnu funkcionalnost (na primer, JavaMail biblioteka).
3. **Nivo komponente:** Komponente su kolekcije objekata i klase objekata koje zajedno rade na obezbeđivanju odgovarajućih funkcija i servisa. Pri ponovnoj upotrebi neke komponente, integracijom nekoliko ranije razvijenih softverskih sistema najčešće je potrebna određena promena softvera ili njegova dopuna, da bi se komponenta prilagodila potrebama novog korisnika, tj. softvera u koji se ugrađuje (na primer, grafički korisnički interfejs - GUI).
4. **Nivo sistema:** Na ovom nivou se upotrebljava ponovo ceo softverski sistem. U tom slučaju se njegova konfiguracija (komponente i dr.) prilagođava specifičnim potrebama korisnika (npr. SAP). Na ovaj način se može novi sistem kreirati konfiguriranjem novog sistema

STRUKTURA TROŠKOVA PONOVNOG KORIŠĆENJA SOFTVERA

Korišćenjem ranije razvijenog softvera, ubrzava se razvoj novog softvera, smanjuje se rizik neuspeha i smanjuju se troškovi razvoja

Korišćenjem ranije razvijenog softvera, ubrzava se razvoj novog softvera, smanjuje se rizik neuspeha i smanjuju se troškovi razvoja. Kako je softver koji se ponovo koristi, već ranije u primeni proveren, to njegovom upotrebom povećava se pouzdanost sistema koji se razvija. Međutim, postoje i određeni troškovi karakteristični kod ponovnog korišćenja softvera:

1. *Trošak vremena utrošenog za nalaženje i analizu softvera koji se može ponovno upotrebiti, a sa stanovišta zadovoljenja potreba korisnika.*

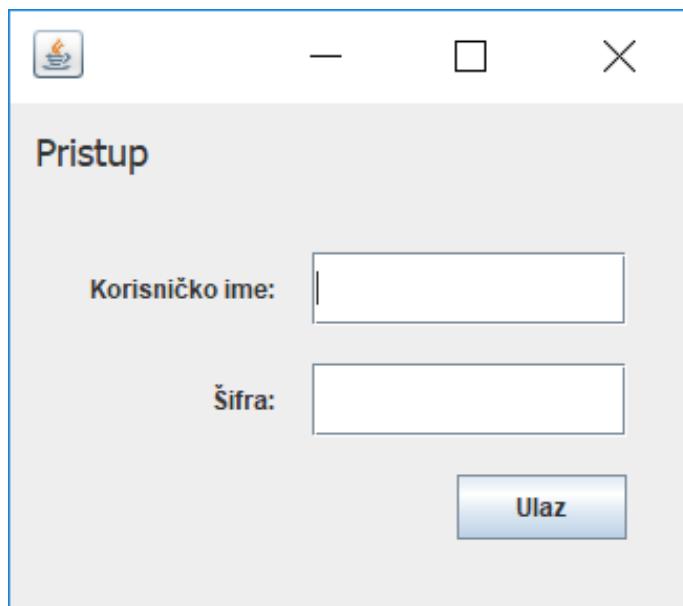
2. *Trošak nabavke softvera* . u cilju zadovoljenja za ponovno korišćenje. Veliki sistemi, koji se kupuju radi prilagođavanja, mogu da budu i vrlo skupi.
3. *Troškovi prilagođavanja kupljenog softvera*, tj. softverskih komponenti u skladu sa zahtevima.
4. *Troškovi integracije softverskih komponenti* namenjenih za ponovnu upotrebu.

Pri razvoju novog softvera, prvo što projektant treba da uradi je da odluči *kako da ponovno upotrebi postojeće SW*. Ta se odluka donosi na početku, tj. pre projektovanja novog sistema. U slučaju razvoja objektno-orientisanih sistema, kada se iscrpe mogućnosti prilagođavanja postojećih elemenata, onda se menjaju zahtevi i arhitektura softvera kako bi se u što većoj meri koristili ranije razvijeni softverski elementi

PRIMER DELA PROGRAMSKOG KODA KOJI JE MOGUĆE PONOVO UPOTREBITI

Primer se odnosi na deo sistema koji služi za registraciju korisnika i koji može biti korišćen u različitim sistemima nakon minimalnog prilagođavanja.

Primer predstavlja klasu GUI iz sistema za naručivanje u restoranu. Klasa omogućava prikaz korisničkog interfejsa za pristupanje i proveru unetih korisničkih kredencijala. Programski kod se može iskoristiti u drugim sistemima uz minimalne modifikacije u samom programskom kodu i kroz dopunu ostalih klasa.



Slika 7.2.1 Izgled interfejsa za pristup

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
```

```
/*
package gui;

import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPasswordField;
import javax.swing.JTextField;

/**
 * Klasa predstavlja login panel
 * @author nebojsa
 */
public class Login extends JFrame{

    private JButton ulaz;
    private JLabel naslov;
    private JLabel usernameLabela;
    private JLabel sifraLabela;
    private JPasswordField pass;
    private JTextField user;

    /**
     * Konstruktor koji setuje sve na panelu
     */
    public Login() {
        naslov = new JLabel();
        user = new JTextField();
        usernameLabela = new JLabel();
        sifraLabela = new JLabel();
        pass = new JPasswordField();
        ulaz = new JButton();

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

        naslov.setFont(new Font("Tahoma", 0, 18));
        naslov.setText("Pristup");

        usernameLabela.setText("Korisnicko ime:");
        sifraLabela.setText("Šifra:");

        ulaz.setText("Ulaz");
        ulaz.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                ulazAkcija(evt);
            }
        });
        javax.swing.GroupLayout layout = new
```



```
    pack();

    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);
}
```

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci u okviru ponovne upotrebe softvera.

1. Zadatak: Osmisliti i prikazati deo programskog koda koji je moguće ponovo koristiti u različitom softveru. Odrediti nivo ponovne upotrebe softvera.

2. Zadatak: Na osnovu realizacije prvog zadatka izvršiti prikaz strukture troškova ponovog korišćenja softvera kroz sledeće stavke:

- trošak utrošenog vremena
- trošak nabavke softvera
- trošak prilagođavanja softvera
- trošak integracije softverskih komponenti

✓ 7.2 Upravljanje konfiguracijom softvera

ŠTA JE KONFIGURACIJA SOFTVERA?

Konfiguracija softvera je skup određenih verzija softverskih komponenata. Upravljanje konfiguracijom je opšti proces upravljanja promenama softverskog sistema

Proces razvoja softvera je vrlo dinamičan, jer dolazi do čestih promena planova, zahteva i dr. Zbog toga je vrlo važno uspostaviti dogovarajući sistem upravljanja ovim promenama. Kako sinhronizovati rad članova razvojnog tima tako da se ne mešaju jedan drugom u posao (bez kontrole)? Ako jedan od njih unosi neku promenu, da li se ta promena tiče i rada ostalih? Da li neće ta promena „uništiti“ rad nekog drugog člana tima? Kako obezbediti da svi članovi tima u svakom momentu rade sa najnovijim verzijama softverskih komponenata? Ako mi sa novom verzijom komponente nešto loše ide, kako da se vratim na prethodnu verziju? Sve su ovo pitanja na koje očekujemo odgovor primenom odgovarajućeg sistema za upravljanje konfiguracijom softvera.

Konfiguracija softvera je skup određenih verzija softverskih komponenata. Moguće je da se za različite korisnike koriste različite verzije softverskih komponenata. Te verzije ne moraju

da budu samo vremenski uslovljene, već mogu biti rezultat obezbeđivanja nešto različitih funkcija, ili primene različitih tehnologija.

Upravljanje konfiguracijom (engl., Configuration Management) je opšti proces upravljanja promenama softverskog sistema. Cilj primene sistema za upravljanje konfiguracijom je da se podrži proces integracije sistema tako da svi članovi razvojnog tima mogu da na kontrolisan način pristupaju kodu softvera i njegovoj dokumentaciji, da pronađu promene do kojih je došlo, i da izvrše prevođenja u izvršni oblik (tj. da izvrše kompilacije) i da povežu sve komponente koje čine sistem.

OSNOVNE AKTIVNOSTI UPRAVLJANJA KONFIGURACIJOM

Upravljanje konfiguracijom je upravljanje verzijama, integracija sistema i praćenje problema radi prijava grešaka i problema u softveru.

Postoј tri osnovne aktivnosti upravljanja konfiguracijom:

1. **Upravljanje verzijama**, kojim se obezbeđuje praćenje različitih verzija softverskih komponenata, kao i koordinacija rad više programera. Na taj način se sprečava da jedan programer promeni kod koji je drugi programer prethodno pripremio i ubacio u sistem.
2. **Integracija sistema**, kojom se obezbeđuje pomoć softverskim inženjerima da definišu verzije komponenata koje čine svaku verziju sistema. To omogućava automatsku izradu izvršnog koda sistema (tj. njegovu kompilaciju i povezivanje) sastavljanog od različitih komponenata.
3. **Praćenje problema**, koji omogućava korisnicima korisnicima da prijave greške (bagove) i druge probleme u radu sistema, a razvojnom timu omogućava da koordiniše rad na otklanjanju prijavljenih grešaka i rešavanju problema.

Svaki alat koji se bavi upravljanju konfiguracijama softvera (npr. ClearCase) mora da podrži sve tri navedene aktivnosti. Ovi sistemi integrišu sve navedene aktivnosti obezbeđujući korišćenje istog korisničkog interfejsa i iste baze podataka.

Pored integrisanih sistema za upravljanje konfiguracijama, koriste se i posebni alati, koji obezbeđuju samo jednu od navedenih aktivnosti. Na primer, alat Subversion vrši upravljanje verzijama komponenata i sistema. Integraciju sistema može da obavlja alat koji samo to radi, kao na primer, Unix make. Sistemi za praćenje grešaka, kao što je Bugzilla, se koriste za prijavljivanje grešaka i za druge stvari, i vode evidenciju o tome koja je greška otklonjena, a koja još nije.

PRIMER UPRAVLJANJA VERZIJAMA SOFTVERA

U primeru je prikazano upravljanje različitim verzijama softvera korišćenjem GitHub repozitorijuma.

Na slici 1 je dat primer unosa nove verzije softvera na GitHub repozitorijum. Potrebno je napraviti repozitorijum u okviru koga je potrebno smestiti prvu verziju programskog koda.

Nakon unosa prve verzije programskog koda svaka naredna verzija vrši ažuriranje prethodno unetog programskog koda i prikaz izmena. Korisnik može videti prethodnu verziju a i izmene u novoj verziji koja je postavljena. U svakom trenutku može izvršiti vraćanje na prethodnu verziju.

Repozitorijumu može pristupiti više korisnika, koji prvo preuzimaju poslednju verziju koja se nalazi u repozitorijumu, vrše usklađivanje sa svojom verzijom na lokalnom računaru a nakon toga ažuriraju verziju unutar repozitorijuma.

Kroz ovakve alate moguće je izvršiti kontrolu verzija softvera i načinjenih izmena u razvojnom procesu.

```
Showing 14 changed files with 154 additions and 19 deletions.

17 pom.xml
@@ -18,6 +18,23 @@
 18     </properties>
 19
 20     <dependencies>
 21     +
 22     +         <dependency>
 23     +             <groupId>org.springframework</groupId>
 24     +             <artifactId>spring-aop</artifactId>
 25     +             <version>${spring.version}</version>
 26     +         </dependency>
 27     +         <dependency>
 28     +             <groupId>org.aspectj</groupId>
 29     +             <artifactId>aspectjrt</artifactId>
 30     +             <version>1.6.11</version>
 31     +         </dependency>
 32     +         <dependency>
 33     +             <groupId>org.aspectj</groupId>
 34     +             <artifactId>aspectjweaver</artifactId>
 35     +             <version>1.6.11</version>
 36     +         </dependency>
 37     +
 38     <dependency>
 39     +             <groupId>javax</groupId>
 40     +             <artifactId>javaee-web-api</artifactId>
 41

16 src/main/java/com/mycompany/aspect/InterceptorLog.java
```

Slika 7.3.1 Primer unosa nove verzije softvera na GitHub repozitorijum

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci za upravljanje konfiguracijom softvera.

1. Zadatak: Napraviti GitHub nalog, kreirati repozitorijum KI206 i u njega uneti programski kod proizvoljne aplikacije. Prvi uneti programski kod je prva verzija tog softvera. Izvršiti izmenu u programskom kodu i ažurirati repozitorijum sa novom verzijom. Pronaći i ispratiti izveštaj o načinjenim izmenama.

2. Zadatak: Napraviti Bitbucket nalog, kreirati repozitorijum KI206 i u njega uneti programski kod proizvoljne aplikacije. Prvi uneti programski kod je prva verzija tog softvera. Izvršiti izmenu u programskom kodu i ažurirati repozitorijum sa novom verzijom. Pronaći i ispratiti izveštaj o načinjenim izmenama.

Uporediti unos i ažuriranje programskog koda kroz GitHub i Bitbucket. Prikazati i opisati uočene razlike.

✓ 7.3 Razvojna i izvršna platforme

RAZVOJNA PLATFORMA

Razvojne i izvršne platforme obuhvataju i hardversku i softversku opremu, koja se koristi pri razvoju i pri korišćenju razvijenog sistema.

Najčešće se softver razvija na jednom kompjuteru, a koristi se na nekom drugom. Zato se govorи o razvojnim i izvršnim platformama, koje obuhvataju i hardversku i softversku opremu koja se koristi pri razvoju i pri korišćenju razvijenog sistema. Tu ulazi i operativni sistem, i softver za podršku, kao što su sistemi baza podataka (DBMS), ili sistemi za interaktivni razvoj softvera (u slučaju razvojnih platformi).

U slučaju ugrađenih računarskih sistema, često se koriste **simulatori** koji vrše simulaciju rada određenih hardverskih uređaja, kao što su senzori, događaju u okruženju u kojima će sistem raditi. Simulatori ubrzavaju proces razvoja ugrađenih sistema, jer omogućavaju da svaki inženjer razvoja ima svoju izvršnu platformu, te ne mora da preuzima softver sa zajedničkog servera. Međutim, skup je razvoj simulatora, te se oni koriste samo u slučaju najpopularnijih hardverskih arhitektura (uređaja).

Ukoliko izvršno okruženje koristi određeni **posrednički softver** (middleware), tj. serverski softver za upravljanje radom aplikacija, kao i za druge softvere, onda se i oni moraju koristiti prilikom testiranja sistema. Nije praktično da sav taj softver instalirate na vašem računaru, pre svega zbog problema sa licencama. Zato se najčešće razvijen softver prebacuju na testiranje na izvršnu platformu.

Razvojna platforma treba da obezbedi sledeće softverske alate:

- *Integriran kompajler i sistem za promenu sintakse, radi kreiranja, promene i prevodenja (kompilacije) izvornog koda.*
- *Sistem za otklanjanje grešaka za softver pisan u određenom programskom jeziku.*
- *Grafički alati za rad sa softverskim modelima, kao što su UMPL modeli.*
- *Alati za testiranje, kao što su JUnit, koji mogu da automatski izvršavaju skup testova nove verzije programa.*
- *Alati za podršku projektima razvoja softvera, koji organizuju kod za različite razvojne projekte.*

Radi podrške razvojnom timu, koristi se i zajednički server, na kome se nalaze sistemi za upravljanje promenama i konfiguracijama, a i za upravljanje zahtevima. Alati za razvoj softvera se najčešće grupišu tako da čine integrisane razvojne platforme (engl., **Integrated Development Environment**- IDE). u okviru zajedničkog okvira i korisničkog interfejsa. IDE je skup softverskih alata koji podržavaju različite aspekte razvoja softvera, Najčešće IDE podržava primenu jednog programskog jezika, ka na primer, Javu, ali mogu da podržavaju i više programske jezike.

NAČIN RAZVOJA SOFTVERA ZA CILJNU PLATFORMU

Odluka o izvršnoj konfiguraciji softverskog sistema se najčešće dokumentuje primenom UML dijagrama instalacije

IDE opšte namene je okvir za softverske alate koji obezbeđuje upravljanje podataka za potrebe softvera koji se razvija, kao i mehanizme za integraciju rada alata. Jedna od najviše korišćenih IDE opšte namene je Eclipse okruženje. Njegovim konfiguriranjem, može se podesiti za razvoj softvera u različitim programskim jezicima i u različitim aplikacionim domenima.

Jedna od odluka koja se mora doneti se tiče načina razvoja softvera za ciljnu platformu, tj. izvršnu platformu na kojoj će se softver koristiti. To je jasno u slučaju ugrađenih sistema, jer se tačno zna sistem u koji se ugrađuje. Međutim, kod distribuiranih sistema, mora se doneti odluka na koje platforme će biti instalirana svaka od softverskih komponenti sistema. Prilikom odlučivanja, potrebno je voditi računa o sledećem:

1. *Hardverski i softverski zahtevi komponente:* Ako je komponenta razvijena za specifičnu hardversku arhitekturu, ili zavisi od drugih softverskih sistema, ona mora da se razvija na platformi koja obezbeđuje zahtevani hardver i softver.
2. *Raspoloživost sistemskih zahteva:* Mnogi sistemi zahtevaju primenu na različitim platformama. Ovo znači da u slučaju otkaza rada komponente na jednoj platformi, može se primeniti komponenta razvijena za neku drugu platformu.
3. *Komponenta komunikacija:* U slučaju da se koristi vrlo intenzivna komunikacija između komponenti, najbolje da se one grupišu i da rade na istoj platformi, ili na fizički bliskim platformama. To smanjuje kašnjenja u kretanju signala između komponenata.

Odluka o izvršnoj konfiguraciji softverskog sistema se najčešće dokumentuje primenom UML dijagrama instalacije, koji pokazuje kako su softverske komponente raspoređene po hardverskim platformama.

U slučaju razvoja ugrađenih sistema, uzimaju se u obzir karakteristike ciljnog uređaja (u koji se ugrađuje softver) kao što su fizička veličina, raspoloživa snaga, potrebna brzina odgovora na signale sa senzora u realnom vremenu, fizičke karakteristike aktuatora i karakteristike upotrebljenog operativnog sistema za rad u realnom vremenu.

PRIMER RAZVOJNE PLATFORME

Dat je primer razvojne platforme za softver.

```

10  * @author nebojsa
11  */
12
13
14  public class Vezba {
15      static void proveraNivo(int nivoSecera){
16          if(nivoSecera>6) {
17              throw new ArithmeticException("Potrebno je aktivirati pumpu i ubrzati insulin.");
18          }
19          else {
20              System.out.println("Secer je u normalnim granicama!");
21          }
22      }
23
24      /**
25       * @param args the command line arguments
26      */
27      public static void main(String[] args) {
28          System.out.println("Proveravanje nivosa seceru u krvi je u toku...");
29          proveraNivo(3);
30      }
31
32  }
33

```

Slika 7.4.1 Primer razvojne platforme

Primer razvojnog alata (NetBeans) gde se sa leve strane nalaze projekti sa paketima koji sadrže klase na kojima se može raditi. Sa desne strane se nalazi sadržaj otvorene klase odnosno programski kod koji izvršava neku funkcionalnost. Na slici 1 dat je primer izutetka koji je prethodno korišćen za simulaciju izuzetaka. Deo ispod programskog koda predstavlja konzolu u kojoj se ispisuju rezultati pokretanja same aplikacije (i eventualne uočene greške). Kompajler kroz razvojni alat pokreće aplikaciju simulirajući tako rad u realnom okruženju što korisniku omogućava da proveri rad programskog koda.

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci koji se odnose na razvojne i izvršne platforme.

- 1. Zadatak:** Simulirati rad razvojne platforme, prikazati pokretanje aplikacije kroz razvojnu platformu i način proveravanja grešaka u programskom kodu.
- 2. Zadatak:** Testirati različite razvojne alate dostupne za Java programski kod. Pokrenuti aplikaciju u različitim razvojnim okruženjima i testirati korisnički interfejs razvojnog alata.

▼ Poglavlje 8

Pretvaranje projektnog UML modela u kod

PROBLEMI U IMPLEMENTACIJI

Zbog ad-hoc izmena u softveru, koje nisu u skladu sa projektnom dokumentacijom, nastaju problemi pri integraciji većih softverskih sistema.

Ako je dosledno primjenjen o UML modeliranje, na kraju faze projektovanja sistema, trebalo bi da imamo kompletну dokumentaciju sistema. Na osnovu nje, u fazi implementacije, trebalo bi da se izvrši kodiranje, tj. Implementacija projektnog rešenja sistema koja kao rezultat ima izvršni kod koji primjenjuje projektovano rešenje. Na žalost, život je malo komplikovaniji, te se ovaj "normalan" scenario ne realizuje uvek ovako, kako bi normalno trebalo da se realizuje. Koji su to problemi i zašto se oni javljaju?

Obično veći sistemi, koji i primjenjuju planski razvoj softvera i koji primjenjuju "tradicionalan" način razvoja u kome se prvo razvija kompletna projektna i druga dokumentacije, pre faze implementacije, imaju više podsistema, a svaki ima više komponenata. Ti podsistemi mogu biti razvijeni od strane različitih timova. Neke komponente su razvijene specijalno za projekat, a neke su nabavljene ili kupljene, gotove, ili se koriste iz prethodnih projekata. Kada se krenulo u integraciju svih ovih podsistema i komponenata, posle njihovog kodiranja, često nastaju sledeći problemi:

- Pojedini podsistemi i komponenti na različite način primjenjuju iste interfejse, jer su ih razvijali različiti programeri. U programskim interfejsima (API) javljaju se i parametri

Kojih nema u dokumentaciji projektovanih interfejsa, jer su naknadno dodati zbog novih zahteva koji su prihvaćeni u toku razvoja ovih podsistema i komponentama.

- Dodati su novi atributi u objektni model međutim, njih ne podržava primjenjen sistem baze podataka, verovatno zbog greške u komunikaciji, jer tim koji je radio vezu sa bazama podataka, nije doneo informaciju o ovim dodatim atributima.

Zbog ovih i sličnih problema, dolazi do razvoja improvizovanog koda (koji nema podlogu u projektnoj dokumentaciji), jer se na improvizovani način dodaje kod koji rešava uočene probleme prilikom integracije sistema. Taj softver onda malo podseća na projektno rešenje dato u dokumentaciji, te se teško razume, jer ga ne prati projektna dokumentacija.

U ovom delu predavanja, analiziraćemo moguća rešenja navedenih i sličnih problema, koja bi sprečila javljanje ovih problema. Primenom disciplinovanog pristupa, izvršiće se preslikavanja (prevođenje) projektnog rešenja u programske kod ,

- optimizacijom modela klasa,
- preslikavanjem asocijacija u kolekcije
- preslikavanjem ugovorenih operacija u programske izuzetke,
- preslikavanjem modela klasa u šemu baze podataka.

PRIMER PROBLEMA U IMPLEMENTACIJI

U primeru je dat primer nastalog problema u toku implementacije.

Result List			
Category	Check	Object	Location
Class	Operation implementation	Operation 'Proizvodi.dodajProizvod' by class 'Proizvod...' <Model>	
Class	Operation implementation	Operation 'Proizvodi.izbrisniProizvod' by class 'Proizvod...' <Model>	
Class	Operation implementation	Operation 'ProizvodIterator' by class 'ProizvodIterator...' <Model>	
Class	Operation implementation	Operation 'NarudzbenicaBuilder.setSifra' by class 'Pr...' <Model>	
Class	Operation implementation	Operation 'NarudzbenicaBuilder.setDatum' by class '...' <Model>	
Class	Operation implementation	Operation 'NarudzbenicaBuilder.setProizvodi' by clas... <Model>	
Class	Operation implementation	Operation 'NarudzbenicaBuilder.getNarudzbenica' by... <Model>	
Class	Operation implementation	Operation 'NarudzbenicaBuilder.setSifra' by class 'Pr...' <Model>	
Class	Operation implementation	Operation 'NarudzbenicaBuilder.setDatum' by class '...' <Model>	
Class	Operation implementation	Operation 'NarudzbenicaBuilder.setProizvod' by clas... <Model>	
Class	Operation implementation	Operation 'NarudzbenicaBuilder.getNarudzbenica' by... <Model>	
Class	Role name assignment	Association 'Association_7' <Model>	
Class	Role name assignment	Association 'Association_11' <Model>	
Class	Role name assignment	Association 'Association_1' <Model>	
Class	Role name assignment	Association 'Association_2' <Model>	
Class	Role name uniqueness	Class 'NarudzbenicaFiliali' <Model>	
Class	Role name uniqueness	Class 'NarudzbenicaMagacin' <Model>	
Class Attribute	Initial value for final attribute	Attribute 'Korisnik.korisnickoIme' <Model>::Korisnik	
Class Attribute	Initial value for final attribute	Attribute 'Korisnik.lozinka' <Model>::Korisnik	

Slika 8.1.1 Primer problema u toku implementacije

Na slici 1 prikazan je problem u toku implementacije. Kreiran je klasni dijagram aplikacije i sledeći korak je generisanje programskog koda. Razvojno okruženje PowerDesigner ne dozvoljava generisanje programskog koda zbog grešaka u atributima klasa (prikazani na slici 1). Shodno tome, razvojni tim mora detaljno da proveri operacije definisane u programskom kodu i ponovo pokuša generisanje programskog koda. Problem je nastao u toku modelovanja i mora biti ispravljen pre implementacione faze. Navedenim izmenama ispunjen je deo optimizacije modela klasa navedene aplikacije.

ZADACI ZA SAMOSTALNI RAD

Zadaci koji se odnose na probleme u implementaciji.

1. Zadatak: Modelovati klasni dijagram za ATM uređaj. Definisati sve potrebne klase sistema. Nakon modelovanja izvršiti analizu dijagrama i navesti gde je moguće izvršiti optimizaciju navedenog klasnog dijagrama.

2. Zadatak: Na modelovanom dijagramu simulirati preslikavanje asocijacije u kolekcije.

❖ 8.1 Preslikavanje modela

NAJAVAŽNIJE AKTIVNOSTI TRANSFORMACIJE

Transformacija ima za cilj da poboljša jedan aspekt modela zadržavajući sva druga svojstva modela.

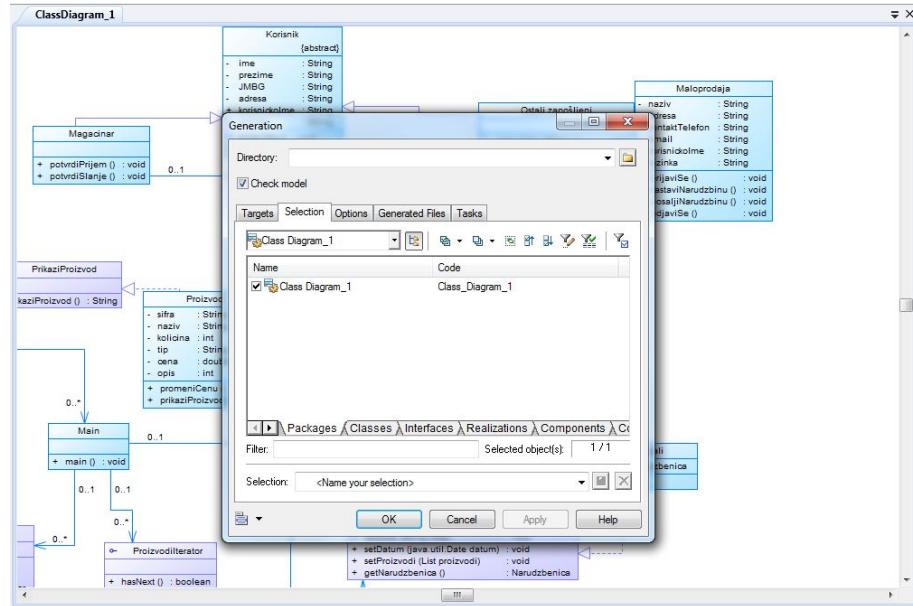
Preslikavanje projektnog rešenja izraženog u vidu UML modela u programski kod (npr. u Javi), je proces transformacije UML komponenata u Java naredbe (instrukcije).

Transformacija ima za cilj da poboljša jedan aspekt modela (npr. modularnost) zadržavajući sva druga svojstva modela. Njeno dejstvo je *lokalizovano*, obuhvata samo nekoliko klasa, atributa i operacija i izvršava se u seriji malih koraka . Ovakve transformacije se pojavljuju za vreme mnogobrojnih aktivnosti projektovanja i implementacije. Najvažnije aktivnosti ovih transformacija su sledeće aktivnosti:

1. **Oprimizacija:** Ova aktivnost ima za cilj da zadovolji zahteve za performansama sistema. To se realizuje smanjivanjem kardinalnosti (broj veza) u asocijacijama UML modela da bi se ubrzale operacije (upiti) nad bazama podataka, dodavanjem redundantnih (ponovljenih) asocijacija povećava se efikasnost izvršnog koda, i dodavanjem izvedenih (obračunatih) atributa da bi se skratilo vreme pristupa objektima.
2. **Realizacija asocijacija:** Za vreme ove aktivnosti, preslikavaju se asocijacije u izvorni kod, kao što su reference i kolekcije referenci.
3. **Preslikavanje ugovora (interfejsa) u izuzetke:** Za vreme ove aktivnosti, opisujemo ponašanje operacija kada dolazi do narušavanja ugovora. To obuhvata generisanje izuzetaka kada se utvrde povrede ugovora, kao i rad sa izuzecima na nivoima sistema sa većom apstrakcijom
4. **Preslikavanje modela klasa u šemu baze podataka:** U fazi projektovanja, odredili smo: strategiju trajnog memorisanja (smeštaja objekata u trajnu memoriju, npr. u bazu podataka), skup datoteka ili kombinaciju baze i datoteka. Za vreme ove aktivnosti, vrši se transformacija modela klasa i šemu memorisanja, npr. u šemu relacione baze podataka.

PRIMER PRESLIKAVANJA MODELA U PROGRAMSKI KOD - REALIZACIJA

Dat je primer u okviru koga je izvršeno preslikavanje modela u Java programski kod.



Slika 8.2.1 Klasni dijagram u Power Designer okruženju

Na slici 1 prikazan je klasni dijagram u Power Designer okruženju. Potrebno je odabratи jezik Java i klase dijagrama koje želimo da pretvorimo u programski kod. Moguće je selektovati samo neke od klasa koje se nalaze na dijagramu i iz njih generisati programski kod. Za ovaj primer odabrana je klasа "Maloprodaja" i generisan je Java programski kod.

```

private String kontaktTelefon;
/** @pdOid 6280e0a4-962e-4234-9ddc-b862f27c7310 */
private String email;
/** @pdOid 4a56e929-9db9-48a4-a8dc-fad3a5204da7 */
private String korisnickoIme;
/** @pdOid 3a676c2d-f9e4-4863-b82e-31a8780b87ce */
private String lozinka;

/** @pdOid 14e9f3c1-89bd-4404-bbc7-ab347426df1f */
public void prijavaSe() {
    // TODO: implement
}

/** @pdOid ec21e5dc-e5e1-4918-af03-eca801ba051f */
public void sastaviNarudzbinu() {
    // TODO: implement
}

/** @pdOid 961d38b8-8981-4404-9cc8-5be506093065 */
public void posaljiNarudzbinu() {
    // TODO: implement
}

/** @pdOid 014b0c12-b7cf-4a75-ae88-fabf9bcf7121 */
public void odjavise() {
}

```

Slika 8.2.2 Generisani Java programski kod otvoren u NetBeans razvojnem okruženju

Kada je izvršeno generisanje Java programskog koda tako dobijenu klasu moguće je otvoriti kroz NetBeans razvojno okruženje kao što je prikazano na slici 2. Ukoliko je potrebno korisnik može izvršiti izmene u programskom kodu i modifikovati automatski generisan kod.

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci koji se odnose na preslikavanje modela.

1. Zadatak: Napraviti klasni dijagram za sistem za podršku rada hotela. Sistem će recepcioneru omogućiti da se prijavi sa svojim korisničkim imenom i lozinkom kako bi pristupio bazi podataka hotela. U bazi su smešteni podaci o gostima.

Moći će da unosi u sistem nove goste i da uklanja goste kojima se boravak u hotelu završio. Dodavanje gosta će biti jednostavno. Repcioner izabere datum dolaska i datum odlaska gosta, zatim upisuje njegove osnovne podatke i u dogovoru sa gostom rezerviše sobu. Sobe mogu biti sa klimom i bez klime. Takođe sobe se mogu razlikovati po broju kreveta i tipu kreveta. Cene za jednu noć provedenu u hotelu se razlikuju u zavisnosti od izabrane sobe. Za VIP goste moći će da se naruči prevoz, tj. personalni vozač za određeni vremenski period.

Izvršiti preslikavanje modela u Java programski kod. Prikazati dobijene klase u razvojnem okruženju.

2. Zadatak: Napraviti klasni dijagram za sistem za podršku autobuskog prevoza putnika. Administrator sistema predstavlja i samog direktora firme koji ima mogućnost upravljanja svim podacima i izmena istih. Radnici imaju uvid u svoj raspored vožnji, raspored vozila koja koriste, imaju mogućnost štikliranja obavljenih poslova i imaju mogućnost nekih manjih izmena. Dok korisnici mogu da se informišu o vremenu polaska, rutama, vremenu trajanja, ceni i ostalo što je bitno jednom korisniku da zna za svoje putovanje.

Izvršiti preslikavanje modela u Java programski kod. Prikazati dobijene klase u razvojnem okruženju.

▼ 8.2 Koncepti preslikavanja

VRSTE TRANSFORMACIJA

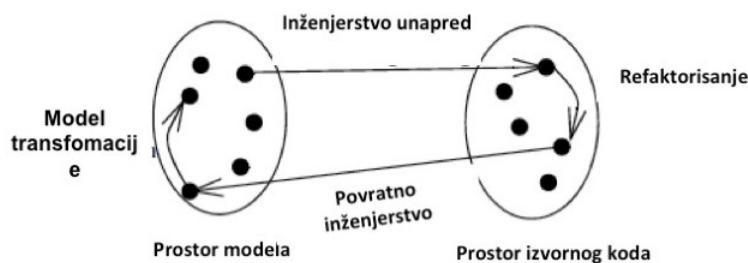
Postoji četiri vrste transformacija između i unutar prostora UML modela i prostora izvornog koda.

Koriste se četiri vrste transformacija:

1. **Transformacije modela** (Model transformations): Transformacije modela se vrše nad modelima objekata. Na primer: konverzija prostog atributa (npr. Adrese predstavljenje kao String) u neku klasu (npr. Klasa sa adresom, Zip kodom, gradom, državom i dr.)
2. **Refaktorisanje** (Refactoring): Refaktorisanja su transformacije koje se vrše nad izvornim kodom. Slične su transformacijama modela objekata, jer i one poboljšavaju jedan aspekt sistema, bez promene njegove funkcionalnosti. Razlika je u tome što se refaktorisanjem menja izvorni kod.

3. **Inženjerstvo unapred** (Forward engineering): Inženjerstvo "unapred" proizvodi uzorak izvornog koda koji odgovara objektnom modelu. Mnogi elementi modela (npr. Asocijacije, atributi) se mehanički preslikavaju u izvorni kod određenog programskog jezika. Telo metoda ostaje da bude uneseno od strane programera
4. **Inženjerstvo unazad** (Reverse engineering): Inženjerstvo unazad proizvodi model koji odgovara izvornom kodu. Ova transformacija se koristi kada je projekat softvera (dizajn) izgubljen, te se mora rekonstruisati na osnovu izvornog koda. Mnogi alati za razvoj softvera podržavaju ovaj postupak, ali se ipak traži i dodatna intervencija programera da bi se dobio tačan objektni model, jer kod ne sadrži sve potrebne informacije

Na slici 1 prikazane se ove četiri vrste transformacije između UML modela. I modela izvršnog koda, tj. prostora u kome je definisan UML model i prostora u kome je definisan izvorni kod (npr. u Javi).



Slika 8.3.1 Četiri vrste transformacije modela

TRANSFORMACIJA MODELA

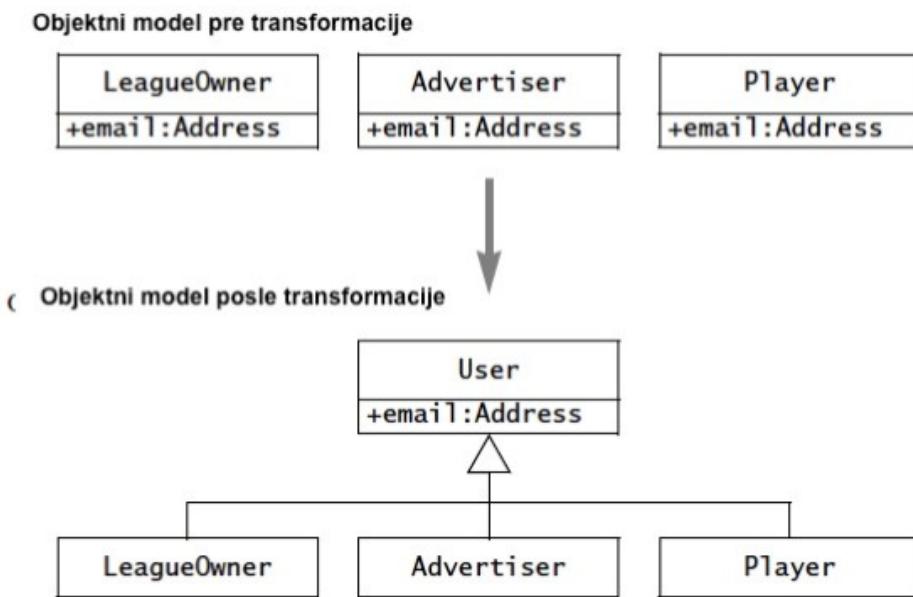
Svrha ove transformacije je pojednostavljenje ili optimizacija originalnog modela, a u skladu sa specifikacijom zahteva

Transformacijom modela, jedan objektni model se pretvara u drugi. Svrha ove transformacije je pojednostavljenje ili optimizacija originalnog modela, a u skladu sa specifikacijom zahteva.

Transformacija može da doda ili ukloni klasu, da joj promeni ime, operacije, asocijacije ili attribute. Može da doda informaciju modelu ili da je ukloni iz modela

Na slici 2 prikazan je slučaj modela sa više klasa koje imaju isti atribut. Transformacijom se uklanja ta redundantnost jer klasa User postaje super klasa ostalim klasama koji sadrže taj atribut

U stvari, razvojni proces čini niz transformacija modela, kojim se model analize postepeno pretvara u model projektovanih objekata.



Slika 8.3.2 Transformacija modela sa više klasa koje imaju isti atribut

REFAKTORISANJE

Refaktorisanje je transformacija izvornog koda radi poboljšanja njegove jasnoće (čitljivosti) ili promenljivosti, a bez promena ponašanja sistema

Refaktorisanje je transformacija izvornog koda radi poboljšanja njegove jasnoće (čitljivosti) ili promenljivosti, a bez promena ponašanja sistema. Refaktorisanje ima za cilj da poboljša projektno rešenje sistema koji je u upotrebi fokusirajući se na specifično polje ili metod neke klase. Primjenjuje se inkrementalnim koracima, zajedno sa testovima. Teži se da se ne menja interfejsa klase.

Na primer, objektni model na slici se dobija posle tri refaktoranja. Prva (Pull Up Field) pomera atribut email sa podklasa u superklasu. Druga (Pull Up Constructor) pomera kod inicijalizacije sa podklasa na superklasu. Treća (Pull Up Method) pomera metode koje manipuliše atributom email u superklasu.

Pre refaktoranja	Posle refaktoranja
<pre> public class Player { private String email; ... } public class LeagueOwner { private String eMail; ... } public class Advertiser { private String email_address; ... } </pre>	<pre> public class User { protected String email; } public class Player extends User { ... } public class LeagueOwner extends User { ... } public class Advertiser extends User { ... } </pre>

Slika 8.3.3 Primena Pull Up Field refaktorisanje

Before refactoring	After refactoring
<pre>public class User { private String email; } public class Player extends User { public Player(String email) { this.email = email; //... } } public class LeagueOwner extends User { { public LeagueOwner(String email) { this.email = email; //... } } public class Advertiser extends User { public Advertiser(String email) { this.email = email; //... } }</pre>	<pre>public class User { public User(String email) { this.email = email; } } public class Player extends User { public Player(String email) { super(email); //... } } public class LeagueOwner extends User { { public LeagueOwner(String email) { super(email); //... } } public class Advertiser extends User { public Advertiser(String email) { super(email); //... } }</pre>

Slika 8.3.4 Primena Pull Up Field refaktorisanje

INŽENJERSTVO UNAPRED

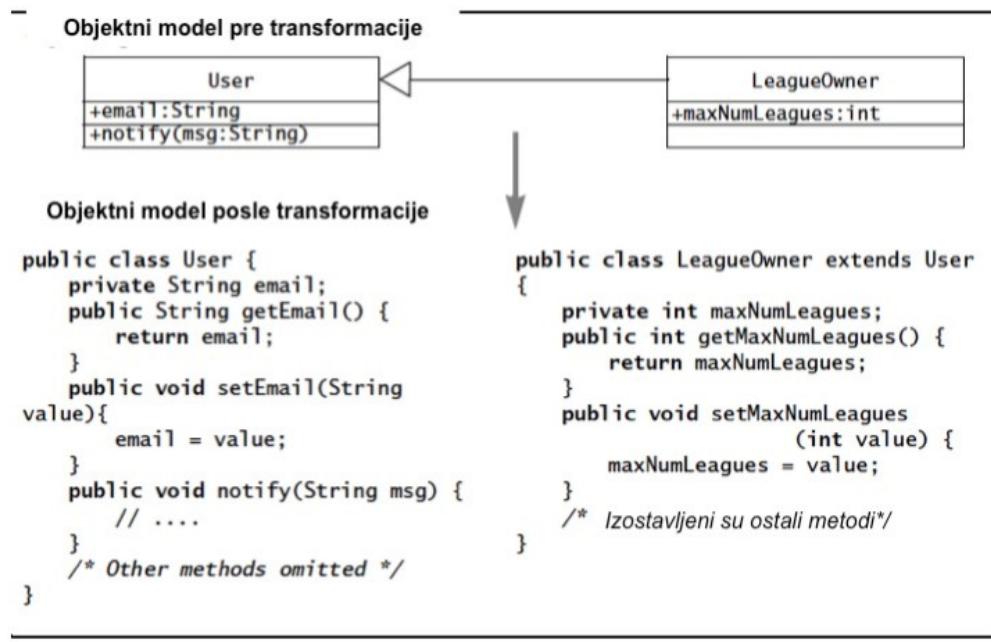
Inženjerstvo unapred se primenjuje na skup elemenata modela i dovodi do odgovarajućih komandi izvršnog koda, kao što su deklaracije klase, Java izrazi ili šema baze podataka.

Inženjerstvo unapred se primenjuje na skup elemenata modela i dovodi do odgovarajućih komandi izvršnog koda, kao što su deklaracije klase, Java izrazi ili šema baze podataka.

Svha primene inženjerstva unapred je održavanje stroge povezanosti dizajna objektnog modela i programskog koda, i smanjivanje broja grešaka koje su se javile za vreme implementacije, a radi smanjivanja posla implementacije.

Na slici 5 je prikazana primenena inženjerstva unapred na klasama **User** i **LeagueOwner**. Svaka UML klasa se transformiše u Java klasu. UML generalizacija klase se pretvara u Javini **extends** iskaz. Svaki atrbut UML klase se transformiše u privatno polje podatka Java klase i u dva metoda (get i set) za čitanje i pisanje vrednosti polja podataka u Java klasi.

Posle automatske transformacije (koju rade alati za razvoj softvera), programer može da doda dodatna polja podataka, kao na primer **maxNumLeagues**. Sem imena atributa i metoda, ostali deo automatski generisanog koda je identičan, te programeri brzo postanu familijarni sa njim, a i zbog toga, on kasnije ima manje grešaka. Programer metode mora ručno da programira.



Slika 8.3.5 Primer transformacije unapred UML klase User i LeagueOwner u odgovarajuće Java klase

INŽENJERSTVO UNAZAD

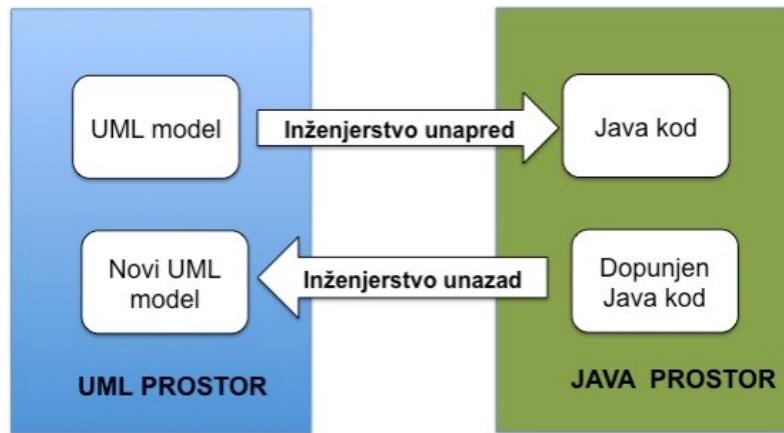
Inženjerstvo unazad se primenjuje nad skupom elemenata izvornog koda da bi se dobio skup elemenata modela.

Inženjerstvo unazad se primenjuje na skupom elemenata izvornog koda da bi se dobio skup elemenata modela.

Svrha je rekonstrukcija modela postojećeg sistema. To je transformacija inženjerstvu unapred. Inženjerstvo unazad kreira UML klasu za svaku deklarisanu klasu u izvornom kodu, dodaje atribute za svako polje, i dodaje operacije za svaki metod.

Na slici 6 je prikazan smer transformacija u slučaju inženjerstva unapred i inženjerstva unazad.

Treba imati u vidu da transformaciju Java koda u UML model obično daje model koji se razlikuje i po izgledu od originalnog UML modela, a da se mogu javiti i neke greške kod transformacija pojedinih Java iskaza. Takođe, ne vrši se transformacija koda Java metoda. Vrši se samo transformacija potpisa metoda u Javi u nazine UML operacija, sa odgovarajućim modifikatorima pristupa i drugim modifikatorima.



Slika 8.3.6 Smerovi transformacija pri primeni inženjerstva unapred i inženjerstva unazad

PRINCIPI TRANSFORMACIJA

Primenom principa transformacija objektnog modela umanjujete mogućnost javljanja grešaka.

Pomenute čitiri vrste transformaciju imaju svoje specifične ciljeve:

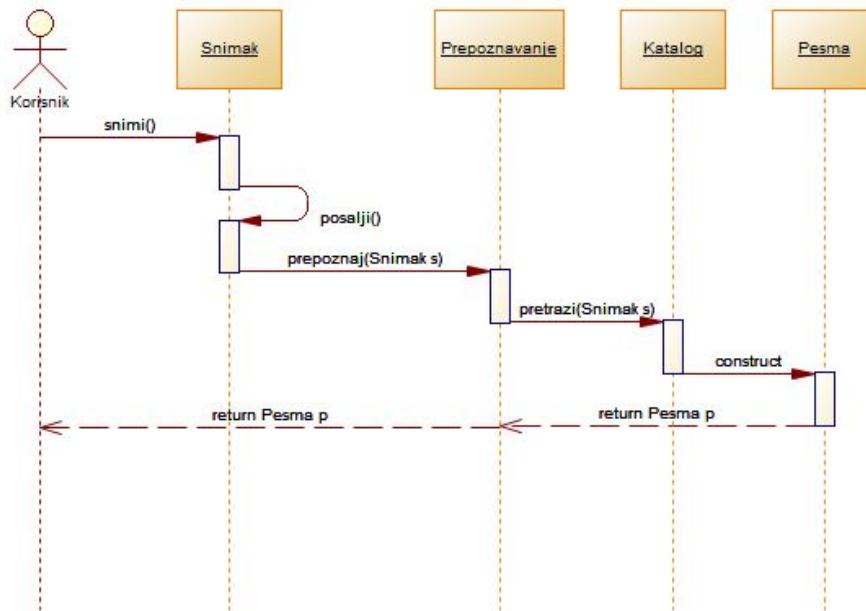
1. **Transformacija modela:** Poboljšava usaglašavanje objektnog modela sa ciljevima projektovanja sistema.
2. **Refaktorisanje:** Poboljšava čitljivost i olakšava menjanje koda.
3. **Inženjerstvo unapred:** Poboljšava konsistentnos izvornog koda sa objektnom modelom.
4. **Inženjerstvo unazad:** Otkriva projektno rešenje na osnovu izvornog koda.

Kako svaka izmena modela ili koda nosi rizik uvođenja novih grešaka, ovde se daju principi čije poštovanja umanjuje mogućnost njihovog javljanja:

1. *Svaka transformacija treba da se odnosi samo na jedan kriterijum:* Transformacija treba da dovede do zadovoljenja samo jednog cilja projektovanja, a ne da prestavlja multi-kriterijumsku optimizaciju, jer bi kod posato isuviše složen, te bi doveo i do grešaka.
2. *Svaka transformacija treba da ima samo lokalni efekat:* To znači da transformacija treba da menja samo nekoliko metoda ili klase. Ako se menja i interfejs, onda i klijent mora o tome da bude obavešten. Previše izmena bi onda napravilo zbrku kod korisnika.
3. *Svaka transformacija treba da bude realizovana nezavisno od drugih:* U jednom trenutku treba vršiti samo jednu transformaciju. Na primer, ako u nekom metodu poboljšavate performansu, onda ne radite istovremeno i dodavanje nove funkcionalnosti, jer onda ne možete utvrditi da li ste optimizirali kod ili niste.
4. *Iza svake transformacije mora da bude provera efekta:* Ako ste primenili transformaciju modela, osvežite odgovarajuće UML dijagrame, kako bi bili ažurni sa promenom. Ako ste vršili refaktorisanje, uradite testove klasa ili metoda koje ste menjali. Nalaženje eventualne greške je uvek lakše ako se utvrdi odmah posle promene koda, nego kasnije.

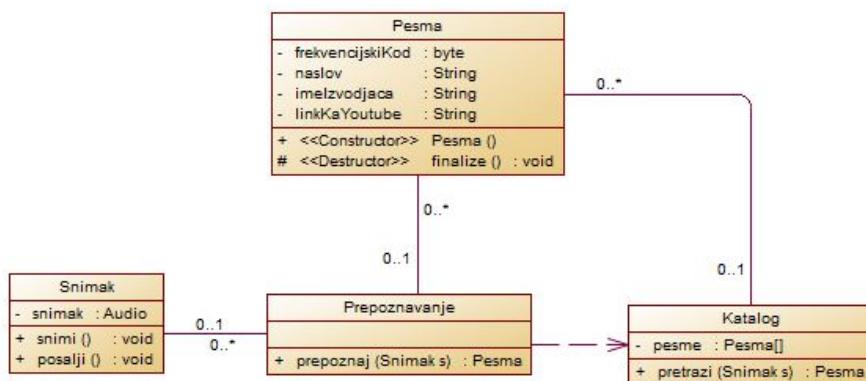
PRIMER TRANSFORMACIJE MODELA

Dat je primer transformacije modela.



Slika 8.3.7 Sekvencijalni dijagram aplikacije za prepoznavanje naziva pesama

Na slici 7 prikazan je sekvencijalni dijagram aplikacije za prepoznavanje naziva pesama. Korisnik vrši snimanje pesme, zatim šalje snimak na prepoznavanje, vrši se pretraga kroz katalog pesama i ukoliko pesma postoji vraća se povratna poruka korisniku sa nazivom pesme. Nakon kreiranja sekvencijalnog dijagrama identifikovani objekti postaju klase a poruke postaju metode u klasnom dijagramu. (slika 8)

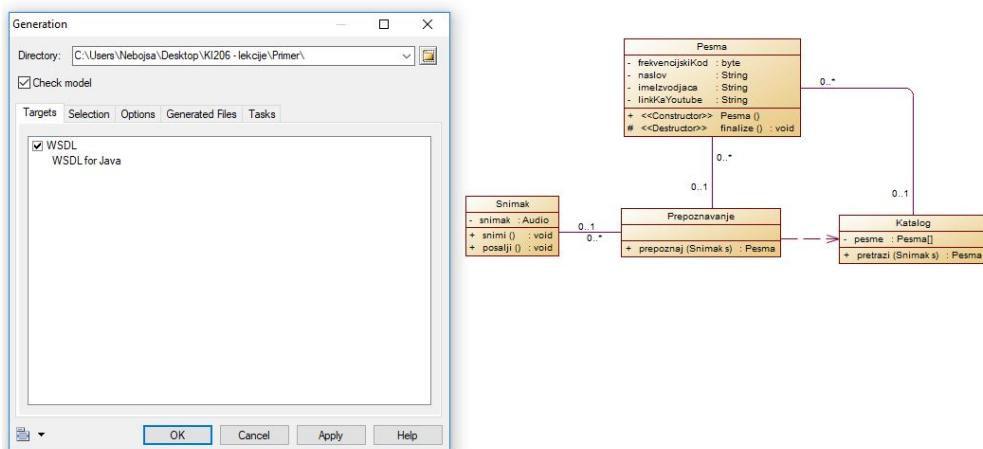


Slika 8.3.8 Klasni dijagram aplikacije za prepoznavanje naziva pesama

Početni dijagram koji je modelovan je sekvencijalni dijagram, na osnovu koga je modelovan klasni dijagram. Slika 8 prikazuje klasni dijagram koji je modelovan na osnovu sekvencijalnog dijagrama na slici 7. Klasni dijagram sadrži klase pesma, snimak, prepoznavanje i katalog što su prethodno bili objekti u okviru sekvencijalnog dijagrama. Metode koje se nalaze u klasama prikazuju poruke iz sekvencijalnog dijagrama. Sledeći korak podrazumeva generisanje Java programskega koda (Java klasa) na osnovu klasnog dijagrama koji je prethodno nastao od sekvencijalnog dijagrama.

PRIMER INŽENJERSTVA UNAPRED

Dat je primer generisanja Java programskog koda iz klasnog dijagrama aplikacije za prepoznavanje naziva pesama.



Slika 8.3.9 Primer generisanja programskog koda aplikacije za prepoznavanje naziva pesama iz klasnog dijagrama

Na osnovu klasnog dijagrama moguće je izvršiti generisanje Java programskog koda kroz PowerDesigner razvojno okruženje. Potrebno je odabrati "Language" i nakon toga "Generate Java Code" i definisati lokaciju za smeštanje generisanih Java fajlova. Kada je generisanje završeno, dobijene su Java klase koje je moguće pokrenuti kroz razvojno okruženje i izvršiti izmene u samom programskom kodu ukoliko je to potrebno.

REZULTAT INŽENJERSTVA UNAPRED - KLASE KATALOG.JAVA I PESMA.JAVA

Na osnovu klasnog dijagrama izvršeno je generisanje Java klase Katalog.java i Pesma.java

```
*****
* Module: Katalog.java
* Author: nebojsa
* Purpose: Defines the Class Katalog
*****
```

```
import java.util.*;  
  
/** @pd0id cae57b05-8243-4a20-ac79-181cb6285306 */  
public class Katalog {  
    /** @pd0id 01bf2441-60af-4ddf-b781-3394a9cef0a */  
    private Pesma[] pesme;  
  
    /** @pdRoleInfo migr=no name=Pesma assc=association1 coll=java.util.Collection  
    impl=java.util.HashSet mult=0..* type=Composition */  
    public java.util.Collection<Pesma> pesma;  
  
    /** @param s  
     * @pd0id fd3db707-396f-4a1c-a973-0c5e77176f2c */  
    public Pesma pretrazi(Snimak s) {  
        // TODO: implement  
        return null;  
    }  
  
    /** @pdGenerated default getter */  
    public java.util.Collection<Pesma> getPesma() {  
        if (pesma == null)  
            pesma = new java.util.HashSet<Pesma>();  
        return pesma;  
    }  
  
    /** @pdGenerated default iterator getter */  
    public java.util.Iterator getIteratorPesma() {  
        if (pesma == null)  
            pesma = new java.util.HashSet<Pesma>();  
        return pesma.iterator();  
    }  
  
    /** @pdGenerated default setter  
     * @param newPesma */  
    public void setPesma(java.util.Collection<Pesma> newPesma) {  
        removeAllPesma();  
        for (java.util.Iterator iter = newPesma.iterator(); iter.hasNext();)  
            addPesma((Pesma)iter.next());  
    }  
  
    /** @pdGenerated default add  
     * @param newPesma */  
    public void addPesma(Pesma newPesma) {  
        if (newPesma == null)  
            return;  
        if (this.pesma == null)  
            this.pesma = new java.util.HashSet<Pesma>();  
        if (!this.pesma.contains(newPesma))  
            this.pesma.add(newPesma);  
    }  
}
```

```
/** @pdGenerated default remove
 * @param oldPesma */
public void removePesma(Pesma oldPesma) {
    if (oldPesma == null)
        return;
    if (this.pesma != null)
        if (this.pesma.contains(oldPesma))
            this.pesma.remove(oldPesma);
}

/** @pdGenerated default removeAll */
public void removeAllPesma() {
    if (pesma != null)
        pesma.clear();
}

}
```

```
*****
* Module: Pesma.java
* Author: nebojsa
* Purpose: Defines the Class Pesma
*****/

import java.util.*;

/** @pd0id b048cd59-81d8-4737-8740-888fabc4c62d */
public class Pesma {
    /** @pd0id 4b813e1a-f459-4367-9cc4-c5be448d1c67 */
    private byte frekvencijskiKod;
    /** @pd0id 0f6a7ac4-a349-4648-a8cc-5156d3222e98 */
    private String naslov;
    /** @pd0id 969f5a3c-29fc-4d26-abe8-d8261ac083c5 */
    private String imeIzvodjaca;
    /** @pd0id a9fe182d-491d-49c6-aa88-4ebc26bf2078 */
    private String linkKaYoutube;

    /** @pd0id fd4ffd41-853d-471a-alfb-801c6da75f0f */
    protected void finalize() {
        // TODO: implement
    }

    /** @pd0id 5fbaba6c-1d3a-477e-9408-6cd49ba7f8d3 */
    public Pesma() {
        // TODO: implement
    }

}
```

REZULTAT INŽENJERSTVA UNAPRED - KLASE PREPOZNAVANJE.JAVA I SNIMAK.JAVA

Dat je automatski generisan Java programski kod iz klasnog dijagrama aplikacije.

```
/*****************************************************************************  
 * Module: Prepoznavanje.java  
 * Author: nebojsa  
 * Purpose: Defines the Class Prepoznavanje  
******/  
  
import java.util.*;  
  
/** @pdOid 7bb3e803-496e-474c-84b6-ba63883e4e7f */  
public class Prepoznavanje {  
    /** @pdRoleInfo migr=no name=Pesma assc=association3 coll=java.util.Collection  
     * impl=java.util.HashSet mult=0..* */  
    public java.util.Collection<Pesma> pesma;  
  
    /** @param s  
     * @pdOid 17a10ed4-a607-487b-b286-c14b01358eb6 */  
    public Pesma prepoznaj(Snimak s) {  
        // TODO: implement  
        return null;  
    }  
  
    /** @pdGenerated default getter */  
    public java.util.Collection<Pesma> getPesma() {  
        if (pesma == null)  
            pesma = new java.util.HashSet<Pesma>();  
        return pesma;  
    }  
  
    /** @pdGenerated default iterator getter */  
    public java.util.Iterator getIteratorPesma() {  
        if (pesma == null)  
            pesma = new java.util.HashSet<Pesma>();  
        return pesma.iterator();  
    }  
  
    /** @pdGenerated default setter  
     * @param newPesma */  
    public void setPesma(java.util.Collection<Pesma> newPesma) {  
        removeAllPesma();  
        for (java.util.Iterator iter = newPesma.iterator(); iter.hasNext();)  
            addPesma((Pesma)iter.next());  
    }  
  
    /** @pdGenerated default add
```

```
* @param newPesma */
public void addPesma(Pesma newPesma) {
    if (newPesma == null)
        return;
    if (this.pesma == null)
        this.pesma = new java.util.HashSet<Pesma>();
    if (!this.pesma.contains(newPesma))
        this.pesma.add(newPesma);
}

/** @pdGenerated default remove
 * @param oldPesma */
public void removePesma(Pesma oldPesma) {
    if (oldPesma == null)
        return;
    if (this.pesma != null)
        if (this.pesma.contains(oldPesma))
            this.pesma.remove(oldPesma);
}

/** @pdGenerated default removeAll */
public void removeAllPesma() {
    if (pesma != null)
        pesma.clear();
}

}
```

```
*****
* Module: Snimak.java
* Author: nebojsa
* Purpose: Defines the Class Snimak
*****/

import java.util.*;

/** @pd0id f94d23a2-7e5c-4613-a27b-79c7487b8997 */
public class Snimak {
    /** @pd0id 351b3250-ed33-4501-b469-3112f35b5308 */
    private Audio snimak;

    /** @pdRoleInfo migr=no name=Prepoznavanje assc=association2
    coll=java.util.Collection impl=java.util.HashSet mult=0..* */
    public java.util.Collection prepoznavanje;

    /** @pd0id 1852233e-b601-4411-9c1d-8dba176c1941 */
    public void snimi() {
        // TODO: implement
    }

    /** @pd0id f9b9d8a3-8c18-4274-b81e-88a526728c66 */
    public void posalji() {
```

```
// TODO: implement
}

/** @pdGenerated default getter */
public java.util.Collection getPrepoznavanje() {
    if (prepoznavanje == null)
        prepoznavanje = new java.util.HashSet();
    return prepoznavanje;
}

/** @pdGenerated default iterator getter */
public java.util.Iterator getIteratorPrepoznavanje() {
    if (prepoznavanje == null)
        prepoznavanje = new java.util.HashSet();
    return prepoznavanje.iterator();
}

/** @pdGenerated default setter
 * @param newPrepoznavanje */
public void setPrepoznavanje(java.util.Collection newPrepoznavanje) {
    removeAllPrepoznavanje();
    for (java.util.Iterator iter = newPrepoznavanje.iterator(); iter.hasNext();)
        addPrepoznavanje((Prepoznavanje)iter.next());
}

/** @pdGenerated default add
 * @param newPrepoznavanje */
public void addPrepoznavanje(Prepoznavanje newPrepoznavanje) {
    if (newPrepoznavanje == null)
        return;
    if (this.prepoznavanje == null)
        this.prepoznavanje = new java.util.HashSet();
    if (!this.prepoznavanje.contains(newPrepoznavanje))
        this.prepoznavanje.add(newPrepoznavanje);
}

/** @pdGenerated default remove
 * @param oldPrepoznavanje */
public void removePrepoznavanje(Prepoznavanje oldPrepoznavanje) {
    if (oldPrepoznavanje == null)
        return;
    if (this.prepoznavanje != null)
        if (this.prepoznavanje.contains(oldPrepoznavanje))
            this.prepoznavanje.remove(oldPrepoznavanje);
}

/** @pdGenerated default removeAll */
public void removeAllPrepoznavanje() {
    if (prepoznavanje != null)
        prepoznavanje.clear();
}
```

}

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci koji se odnose na primenu inženjerstva unapred.

1. Zadatak: Osmisliti i napraviti sekvenčni dijagram za sistem za podršku rada hotela. Sistem će recepcioneru omogućiti da se prijavi sa svojim korisničkim imenom i lozinkom kako bi pristupio bazi podataka hotela. U bazi su smešteni podaci o gostima.

Moći će da unosi u sistem nove goste i da uklanja goste kojima se boravak u hotelu završio. Dodavanje gosta će biti jednostavno. Recepcioner izabere datum dolaska i datum odlaska gosta, zatim upisuje njegove osnovne podatke i u dogovoru sa gostom rezerviše sobu. Sobe mogu biti sa klimom i bez klime. Takođe sobe se mogu razlikovati po broju kreveta i tipu kreveta. Cene za jednu noć provedenu u hotelu se razlikuju u zavisnosti od izabrane sobe. Za VIP goste moći će da se naruči prevoz, tj. personalni vozač za određeni vremenski period.

Primeniti inženjerstvo unapred modelovanjem sekvenčnog dijagrama i klasnog dijagrama koji nastaje na osnovu sekvenčnog dijagrama (obratiti pažnju da poruke i objekti iz sekvenčnog dijagrama prelaze u metode i klase unutar klasnog dijagrama). Generisati Java programski kod klasa.

2. Zadatak: Osmisliti i napraviti sekvenčni dijagram za sistem za podršku autobuskog prevoza putnika. Administrator sistema predstavlja i samog direktora firme koji ima mogućnost upravljanja svim podacima i izmena istih. Radnici imaju uvid u svoj raspored vožnji, raspored vozila koja koriste, imaju mogućnost štikliranja obavljenih poslova i imaju mogućnost nekih manjih izmena. Dok korisnici mogu da se informišu o vremenu polaska, rutama, vremenu trajanja, ceni i ostalo što je bitno jednom korisniku da zna za svoje putovanje.

Primeniti inženjerstvo unapred modelovanjem sekvenčnog dijagrama i klasnog dijagrama koji nastaje na osnovu sekvenčnog dijagrama (obratiti pažnju da poruke i objekti iz sekvenčnog dijagrama prelaze u metode i klase unutar klasnog dijagrama). Generisati Java programski kod klasa.

✓ Poglavlje 9

Aktivnosti pretvaranja modela u kod

UVOD U AKTIVNOSTI TRANSFORMACIJA

Implementacija je preslikavanje, tj. transformacija objektnog modela u izvorni kod, kao i optimizacija samog objektnog modela, kao i njegovo preslikavanje u šemu baze podataka.

U ovom delu predavanja, detaljnije ćemo analizirati navedene transformacije koje se najčešće primenjuju pri implementaciji softvera, tj:

1. Optimizacija objektnog modela
2. Preslikavanje asocijacija u kolekcije
3. Preslikavanje ugovora (interfejsa) u izuzetke,

✓ 9.1 Optimizacija objektnog modela

OPTIMIZACIJA PRISTUPA

Čest izvor neefikasnosti je ponavljanje prolaza preko višestrukih asocijacija i pogrešno postavljanje atributa.

Direktno prevođenje modela analize u izvorni kod je često neefikasno. Model analize je usmeren na funkcionalnost i ne uzima u obzir sistem donošenja projektantskih odluka. Za vreme projektovanja modela objekata, transformiše se model objekata da bi zadovoljio ciljeve projektovanja, definisanje tokom projektovanja sistema.

Ovde ćemo opisati četiri često korišćene optimizacije:

- Dodavanje asocijacija radi optimizacije pristupnih puteva
- Pretvaranje obekata u attribute
- Odlaganje skupih proračuna
- Hvatanje rezultata skupih proračuna

Kod primene optimizacija, traži se ravnoteža između efikasnosti i jasnoće. Optimizacije povećavaju efikasnost, ali i složenost modela, što otežava razumevanje sistema

Ponovljen prelazak asocijacija: Utvrđite operacije koje se često pozivaju (sa sekvencijalnim dijagramima). Nađite onaj deo operacija koje zahtevaju višestruku prekaženje višestruke asocijacije. Frekventne operacije ne treba da imaju puno prelazaka, već direktnu komunikaciju objekta koji šalje upite i objeka koji ih prima i odgovara.

Asocijacije sa kardinalnošću "mnogo": Pokušajte da smanjite kardinalnost da bi smanjili vreme pretraživanja. Koristite asocijacije sa kvalifikovanim socijacijama. Možete koristiti indeksiranje objekata sa strane sa kardinalnošću "mnogo" da bi smanjili vreme pristupa.

Pogrešno postavljeni atributi: I prekomerno modelovanje može negativno da utiče na efikasnost. Ima klasa koje nemaju značajnija ponašanja. Ako se njihovi atributi javljaju samo u **set()** i **get()** metodima, onda verovatno treba da sklonite te atributе u klasu koja ih poziva. Ako klasa ima više takvih atributa, na kraju se može pokazati da takva klasa nije ni potrebna.

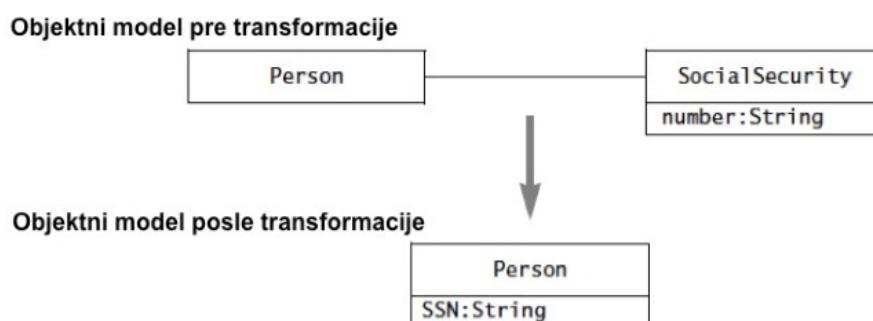
Primenom ovih optimizacija Sistematsko ispitivanje modela objekata, u skladu sa navedenim pitanjima, vodi ka modelu sa: odabranim ponovljenim asocijacijama, manjim brojem neefikasnih "mnogo-sa-mnogo" asocijacija i sa manjim brojem klasa.

PRIMER PRETVARANJA OBJEKATA U ATRIBUTE

Klasa koja ostane sa samo nekoliko atributa ili metoda (posle više optimizacija) i ako ima asocijaciju samo sa jednom klasom, može se zameniti sa jednim atributom

Efikasnije i pouzdanije radi softverski sistem koji je jednostavniji, tj. koji ima manje klasa. Klasa koja ostane sa samo nekoliko atributa ili metoda (posle više optimizacija) i ako ima asocijaciju samo sa jednom klasom, može se zameniti sa jednim atributom. To smanjuje složenost modela.

Pogledajmo primer na slici 1 . klasa SocialSecurity ima samo jedan atribut i povezana je samo jednom klasom: Person. U ovom slučaju, možemo klasu SocialSecurity zameniti sa atributom SSN u klasi Person.



Slika 9.1.1 Primer zamene klase sa atributom radi pojednostavljenja modela.

Ekvivalentan rezultat se može postići sledećim refaktorisanjem:

1. Deklariši javnih polja i metoda izvorne klase (SocialSecurity) u klasi koja je absorbuje (Person)
2. Promeni reference na izvornu klasu u referencu na klasu koja je absorbuje.

3. Promeni ime izvorne klase tako da kompjajler uhvati svaku absolutnu referencu.
4. Izvrši kompilaciju u testiranje.
5. Obriši izvornu klasu

PRIMER ODLAGANJA SKUPIH PRORAČUNA

Operacije koje troše mnogo resursa, treba odložiti u izvršenju što kasnije.

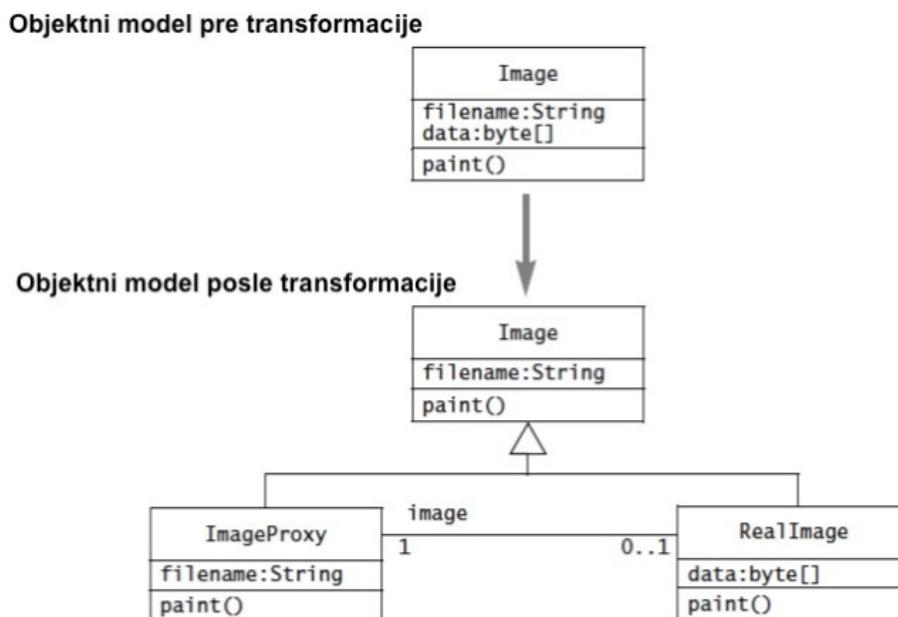
Često, neki objekti su skupi za kreiranje. Kreiranje se može odložiti sve dok njihov sadržaj postane potreban. Na primer (slika 2), vrši se prikazivanje slike (transfer mnogo piksela). Koristi se **Proxy** šablon projektovanja **ImageProxy** objekat koji preuzima mesto **Image** objekta i obezbeđuje se isti interfejs kao **Image** objekat.

Jednostavne operacije, kao **width()** i **height()** ostaju u **ProxyImage** klasi. Metod **paint()** se ubacuje u klasu tek kada se slika treba prikaže.

KEŠIRANJE REZULTATA SKUPIH OBRAČUNA

Neki metodi se često pozivaju, ali njihovi rezultati se ne menjaju ili retko menjaju. Da bi se smanjilo vreme odziva, može se rezultat takvih proračuna smestiti u vidu privatnog atributa.

Na primer, :metod **League.Boundary.getStatistics()** daje statistiku svih **Players** i **Tournaments** u **League**. Stistika se menja tek kada je završen **Match**. Nije potrebno proračunavati statistiku svaki put. Ona se smešta (kešira) kao privremeni atribut (struktura podataka) sve do završetka **Match**.



Slika 9.1.2 Odlaganje skupog obračuna pri transformaciji objektnog modela primenom Proxy šablonu

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci koji obuhvataju optimizaciju objektnog modela.

1. Zadatak: Opis aplikacije: Aplikacija ima funkciju da automatizuje zakazivanje termina korisnicima kod svog omiljenog frizera. Korisnik ima pregled termina kod svog omiljenog frizera gde vidi pregled slobodnih i zakazanih termina kod frizera. Korisnik dalje odabira jedan od slobodnih termina koji mu odgovara i šalje zahtev za zakazivanje termina. Frizer dobija obaveštenje o zahtevu korisnika i može da odgovori na njega - da prihvati ili da odbije pružanje usluge. U slučaju prihvatanja pružanja usluge termin prelazi iz stanja slobodan u zakazan, a korisnik dobija poruku o uspešno zakazanom terminu. U slučaju odbijanja zahteva za pružanje usluge, korisnik dobija poruku o nemogućnosti zakazivanja sa opcionim dodatnim tekstom o obrazloženju.

Modelovati klasni dijagram aplikacije. Izvršiti optimizaciju pristupa. Prikazati primer pretvaranja objekata u attribute.

2. Zadatak: Opis aplikacije: Ova aplikacija namenjena je za zaposlene u privatnoj klinici (medicinska sestra, doktor, direktor klinike). Svaki korisnik se prijavljuje i na osnovu prijave ima različite opcije korišćenja sistema. Opcije su sledeće:

1. Prijavljivanje na sistem klinike
2. Otvaranje kartona pacijenta
3. Pristup kartonu pacijenta
4. Slanje pacijentovih podataka iz kartona lekaru
5. Prepisivanje terapije
6. Izdavanje recepta za leka
7. Davanje uputa za dalje lečenje
8. Pretraživanje zaposlenih
9. Potpisivanje naloga, zapisnika i pravilnika klinike
10. Prikaz pravilnika klinike

Modelovati klasni dijagram aplikacije. Izvršiti optimizaciju pristupa. Prikazati primer pretvaranja objekata u attribute.

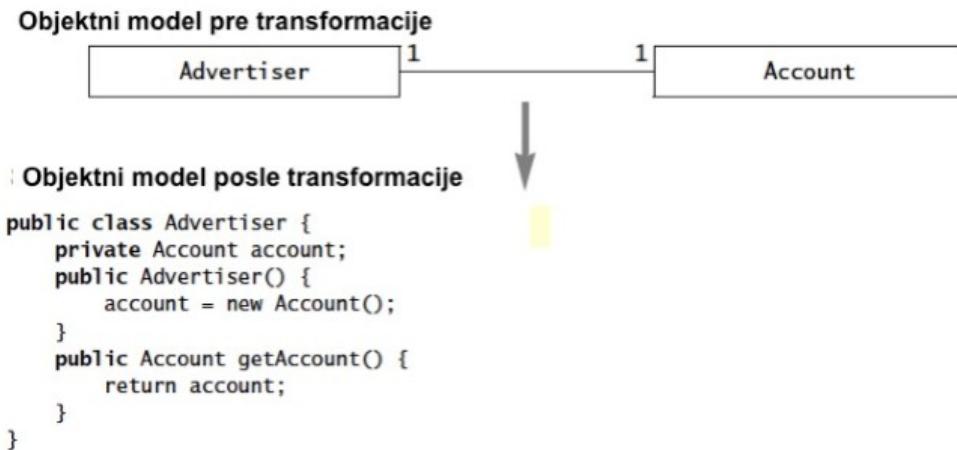
▼ 9.2 Preslikavanje asocijacija u kolekcije

PRESLIKAVANJE JEDNOSMERNE “JEDAN-NA-JEDAN” ASOCIJACIJE

Asocijacije su UML koncepti koje označavaju kolekcije dvosmernih veza između dva ili više objekata. Programski jezici podržavaju koncept reference.

Asocijacije su UML koncepti koje označavaju kolekcije dvosmernih veza između dva ili više objekata. Međutim, objektno-orientisano jezici nemaju koncept asocijacije. Podržavaju **koncept reference**: jedan objekat sadrži operacije sa drugim objektom i kolekcijama. One sadrže reference ka nekoliko objekata. Reference su jednosmerne i koriste se kao veza dva objekta. U takvim slučajevima, asocijacije treba da se preslikaju u reference.

Kako se referenca na **Account** objekatne menja vremenom, pravimo privatno **account** polje i dodjemo metod:**public Advertiser.getAccount()**. To sprečava pozivara da slučajno promeni polje **account**-



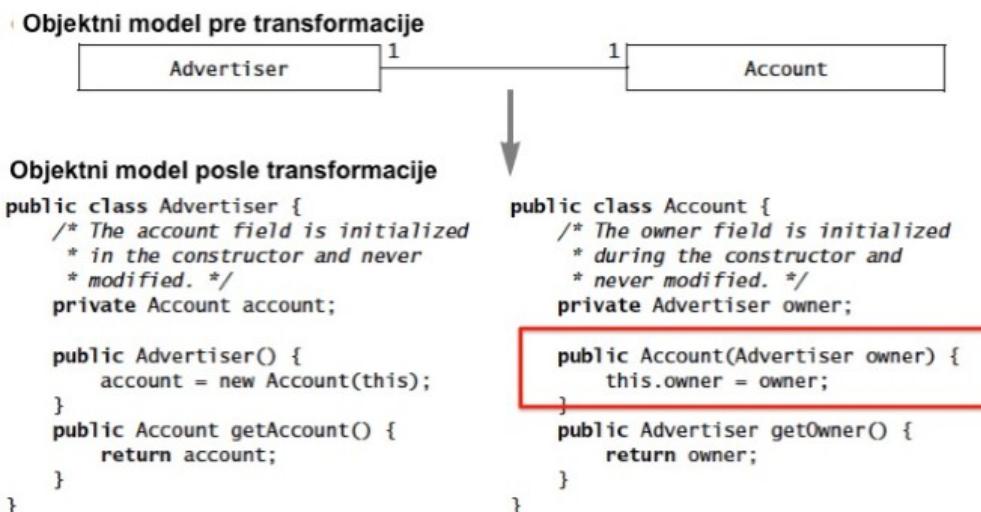
Slika 9.2.1 Preslikavanje jednosmerne "jedan na jedan" asocijacije

PRESLIKAVANJE DVOSMERNE “JEDAN-NA-JEDAN” ASOCIJACIJE

Dvosmerne asocijacije su složenije, jer uvode uzajamne zavisnosti među klasama.

Smer asocijacije se često menja za vreme razvoja sistema. Dvosmerne asocijacije su složenije, jer uvode uzajamne zavisnosti među klasama. Na primer (slika 2), menjamo **Account** klasu tako da prikazuje ime **Account** objekta koji je dobijen iz imena **Advertiser** objekta. Sada, **Account** mora da pristupi odgovarajućem **Advertiser** objektu – stvara se dvosmerna asocijacija. Dodali smo **owner atribut u Account** objektu, ali to nije dovoljn. Uneli smo redundantnost u model. Moramo da obezbedimo da dati **Account** objekat ima referencu određenom **Advertiser** objektu, a da **Advertiser** objekat ima referencu ka istom **Account** objektu. To se rešava stavljanjem parametra u **Account** konstruktoru koji inicijalizuje owner polje sa tačnom vrednošću.

Dvosmerne asocijacije su složenije, jer uvode uzajamne zavisnosti među klasama. One čine model složeniji. Preporuka: je da po pravilu, u startu, načinimo sve attribute privatnim i obezbedimo im odgovarajuće `getAttribute()` i `setAttribute()` metode. To minimizira promene API kada je nužno da se jednosmerna asocijacija mora da zemeni dvosmernom



Slika 9.2.2 Preslikavanje dvosmerne asocijacije "jedan-na-jedan"

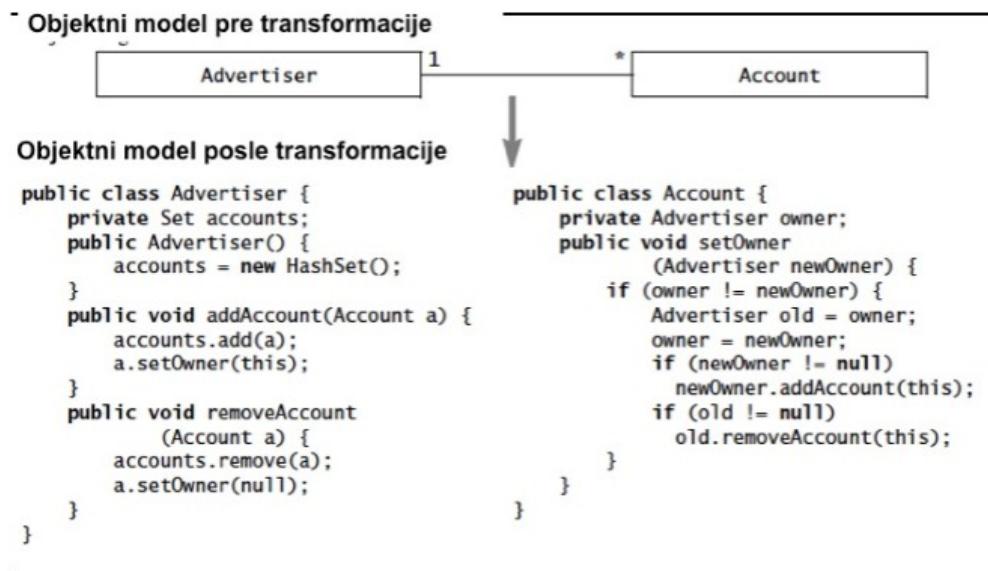
PRESLIKAVANJE ASOCIJACIJE “JEDAN-KA-PUNO”.

Stranu sa “puno” možemo realizovati samo sa kolekcijom referenci.

Asocijacije tipa “jedan-ka-puno” se ne mogu da realizuju upotrebom jedne reference ili par referenci. Stranu sa “puno” možemo realizovati samo sa kolekcijom referenci. Na primer (slika 3), zamislimo da **Advertiser** ima nekoliko **Account** objekata da bi pratio troškove vezane za **AdvertiserBanner** za različite proizvode. U ovom slučaju, **Advertiser** klasa ima asocijaciju tipa “jedan-ka-mnogo” prema klasi **Account**. Kako redosled **Account** objekata nije bitan, to se stranea “puno” može predstaviti sa nizom referenci, koji je nazvan **accounts**, Odlučili smo da ovu asocijaciju realizujemo kao dvosmernu asocijaciju, te smo dodali metode **addAccount()**, **removeAccount** i **setOwner()** metode u **Account** i **Advertiser** klase da bi mogli da menjamo vrednosti polja podataka **accounts** i **owner**.

Kako **Advertiser** objekat može imati promenljiv broj **Account** objekata, **Advertiser** objekt ne poziva konstruktor **Account** objekta. Umesto toga, koristi se jedan kontrolni objekat za kreiranje i arhiviranje objekata **Account**, i on je odgovoran za pozivanje konstruktora.

Ako niz objekata sa **Account** objektima mora da ima redosled, onda se umesto Set kolekcije, koristi List kolekcija. Da bi minimizirali promene u interfeksu kada se ovo ograničenje menja, to se za povratni tip **getAccount()** koristi Collection, koja je superklasa i klase List i klase Set.



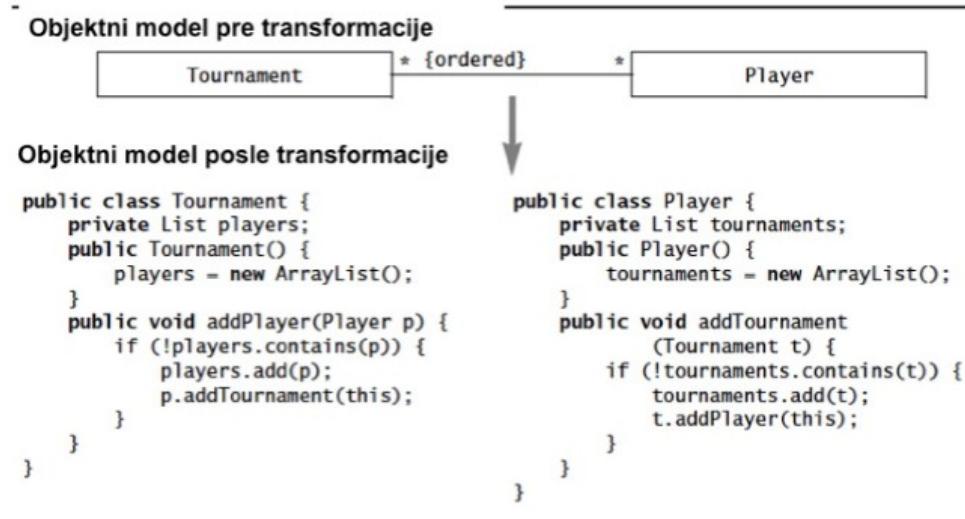
Slika 9.2.3 Preslikavanje dvosmerne asocijacije "jedan-ka-puno"

PRIMER PRESLIKAVANJA ASOCIJACIJE “PUNO-KA-PUNO”

Obe strane imaju klase koje imaju polja podataka tipa kolekcije, a koje sadrže reference na objekte, kao i operacije koje su neophodne za njihovo ažuriranje

U ovom slučaju, obe strane imaju klase koje imaju polja podataka tipa kolekcije, a koje sadrže reference na objekte, kao i operacije koje su neophodne za njihovo ažuriranje, tj. ubacivanje i izbacivanje objekata u kolekcije i dr. Na primer (Slika 4), klasa **Tournament** ima asocijaciju tip “puno-ka-puno” sa redosledom, a sa klasom **Player**. Ova asocijacije se realizuje upotrebom atributa tipa **List** u svakoj od ove dve klase. Oni se mogu menjati sa metodama: **addPayer()**, **removePlayer()**, **addRournament()** i **removeTournament()**. Ove operacije omogućavaju održavanje konzistentnosti obe liste.

Ako treba da se razmatrana dvosmerna asocijacija pretvori u jednosmernu, trebalo bi da se ukloni atribut **tournaments** i povezane metode.



Slika 9.2.4 Realizacija dvosmerne asocijacije "puno-ka-puno"

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci koji obuhvataju preslikavanje asocijacija.

1. Zadatak: Aplikacija se bavi online prodajom (kupovinom) mobilnih uređaja, zavisno da li korisnik pristupa sistemu kao prodavac (ima mogućnost dodavanja uređaja za prodaju i odgovaranju poruka koje je dobio od kupaca) ili korisnik pristupa sistemu kao kupac (ima mogućnost kupovine i kontaktiranja prodavca). Korisnik ne može koristiti ovaj sistem bez naloga.

Modelovati dijagram za opisanu aplikaciju i predstaviti jednosmernu i dvosmernu "jedan-na-jedan" asocijaciju.

2. Zadatak: Aplikacija treba da omogući međusobnu interakciju korisnika putem društvene mreže na kojoj korisnici mogu da dele video sadržaje, četuju međusobno i prate sadržaje koji drugi korisnici dele na društvenoj mreži. Korisnik da bi postao član društvene mreže prvo mora da se registruje, proces registracije jeste jako jednostavan i ne zahteva previše podataka od korisnika već samo one osnovne. Kada se registruje korisnik će moći da deli svoj video sadržaj sa drugim korisnicima, prati sadržaje koji su drugi korisnici podelili, lajkuje i komentariše te iste sadržaje kao i mogućnost četovanja sa drugim registrovanim korisnicima. Takođe, registrovani korisnik će imati opciju da prati druge korisnike koji izbacuju video sadržaj na društvenoj mreži.

Modelovati dijagram za opisanu aplikaciju i predstaviti asocijaciju "jedan-ka-puno" i "puno-ka-puno".

✓ 9.3 Preslikavanje intefejsa (ugovora) u izuzetke

PRIMENA IZUZETAKA KOD PROVERE INTERFEJSA (UGOVORA)

Ukoliko dođe do kršenje ugovora (operacije na interfejsu), izbacuje se izuzetak te se izvršenje programa zaustavlja.

Java nema ugrađenu podršku za automatsku generaciju izuzetaka kada dođe do kršenja ugovora, što bi pomoglo programerima da na vreme neki nedostatak otklone. Može se ipak koristiti Javin mehanizam izuzetaka

throw <objekat izuzetka>

Objekat izuzetka je mesto za stavljanje informacije o izuzetku, poruka greške Izuzetak zaustavlja izvršenje programa dok se ne izvrši **catch** iskaz. Iskaz **catch** sadrži parametar koji povezuje objekat izuzetka, i blok za rad sa izuzetkom. Ako je objekat izuzetka istog tipa kao i tip parametra, **catch** iskaz se poklapa i blok naredbi izuzetka se izvršava.

Na primer (slika 1), operacija **acceptPlayer()** klase **TournamentControl** se poziva sa igračem koji je učesnik takmičenja.

Tournament.addPlayer() izbacuje izuzetak tipa **KnownPlayer**, koji hvata (**catch**) pozivar **TournamentForm.addPlayer()**, koji prosleđuje izuzetak klasi **ErrorConsole**.Prosto preslikavanje bi posmatralo svaku operaciju u ugovoru nezavisno i dodalo bi kod u telo metoda radi provere preduslova, postuslova, i nepromenljivih operacija.

```
public class TournamentControl {
    private Tournament tournament;
    public void addPlayer(Player p) throws KnownPlayerException {
        if (tournament.isPlayerAccepted(p)) {
            throw new KnownPlayerException(p);
        }
        //... Normalno ponašanje sa addPlayer()
    }
}
public class TournamentForm {
    private TournamentControl control;
    private List players;
    public void processPlayerApplications() {
        // Go through all the players who applied for this tournament
        for (Iterator i = players.iterator(); i.hasNext();) {
            try {
                // Delegate to the control object.
                control.acceptPlayer((Player)i.next());
            } catch (KnownPlayerException e) {
                // If an exception was caught, log it to the console, and
                // proceed to the next player.
                ErrorConsole.log(e.getMessage());
            }
        }
    }
}
```

Slika 9.3.1 Primer rada sa izuzetcima u Javi

REALIZACIJA KODA ZA REALIZACIJU OPERACIJA

Svaki preduslov odgovara jednom izuzetku. Svaki postuslov odgovara jednom logičkom iskazu u kome if iskaz izbacuje izuzetak ako je došlo do krešenja uslova.

Kod u telima metoda koji definišu preduslove, postulsove i druge nepromenljive operacije, realizuju se na sledeći način:

- **Provera preduslova:** Preduslove bi trebalo proveriti pre nego što počne bilo koja oprada. Svaki preduslov odgovara jednom izuzetku, tako da klasa klijenta može ne samo da uoči da je došlo do kršenja nekog preduslova, već da tačno utvrdi kod kog parametra je greška.
- **Provera postuslova:** Svaki postuslov odgovara jednom logičkom iskazu u kome **if** iskaz izbacuje izuzetak ako je došlo do krešenja uslova. Ako ima više postuslova koji nisu zadovoljeni, samo se kod prvog prijavljuje kršenje uslova.
- **Provera nepromenljivih (*invariants*):** Kod razmatranja svakog ugovora pojedinačno, nepromenljive se istovremeno proveravaju kada i postuslovi.
- **Rad sa nasleđivanjem:** Kod za proveru preduslova i postuslova trebalo bi da se ugnezdi u posebnim metodama koji mogu da se pozivaju iz podklasa.

Na slici 2 prikazan je primer primene ovih pravila za slučaj **Tournament.addPlayer()** ugovor, koji onda dovodi do koda prikazanim na slici 2 .



REALISTIČNOST PRIMENE POSTUPKA PRESLIKAVANJA

U mnogim slučajevima, kod za proveru preduslova i postuslova je obimniji i složeniji nego kod koji obavlja zahtevanu funkciju, te je izvor novih grešaka kao i pada performansi softvera.

Prethodno izložen postupak preslikavanja, u kome se proverava svaki poziv metoda u skladu sa navedenim pravilima, nije realističan iz sledećih razloga:

- **Rad na kodiranju:** U mnogim slučajevima, kod za proveru preduslova i postuslova je obimniji i složeniji nego kod koji obavlja zahtevanu funkciju. Taj rad bo mogao bolje da se upotrebi za testiranje i čišćenje koda.
- **Veća mogućnost za unos grešaka:** Kod za proveru može takođe da unosi greške i da poveća rad na testiranju. Ako isti programer piše i kod za proveru kao i fukcionalnog koda, ond aće ovaj da zamaskira greške u fukcionalnom kodu.
- **Zamršen kod:** Kod za proveru je obično složeniji od fukcionalni i teško s emenja kada dođe do promene kod fukcionalnog. To dovodi do uvođenja novih grešaka u sistem, što onda dovodi do gubitka svrhe ovog softvera.

- **Problem sa performansama:** Sistematska provera svih ugovora može da primetno pogorša performanse softvera. Povežanje tačnosti semože postići, ali se onda ciljne performanse ne ostvaruju.

Sve dok se ne javi sistem koji može automatski da generiše kod za proveru preslikavanja, moramo da donosimo pragmatične odluke.

Kod za proveru bi trebalo da ima dokumentaciju o svakom ograničenju, i to na engleskom i OCL. To bi olakšalo promenu ovog koda kada dođe do promene funkcionalnog koda.

PRIMER PRIMENE IZUZETAKA ZA PROVERU NIVOA ŠEĆERA U KRVI

Dat je primer primene izuzetaka.

Primer je realizovan u NetBeans razvojnem okruženju i prikazuje proveru nivoa šećera u krvi. Postavljen je uslov da ukoliko je nivo veći od 6 aplikacija izbaci izuzetak: "Potrebno je aktivirati pumpu i ubrizgati insulin.". Ukoliko je nivo manji od 6 aplikacija treba da izbaci izuzetak: "Secer je u normalnim granicama!".

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package pokaznavezba;

/**
 *
 * @author nebojsa
 */

public class PokaznaVezba {
    static void proveraNivoa(int nivoSecera){
        if(nivoSecera>6) {
            throw new ArithmeticException("Potrebno je aktivirati pumpu i ubrizgati insulin.");
        }
        else {
            System.out.println("Secer je u normalnim granicama!");
        }
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Proveravanje nivoa secera u krvi je u toku...");
        proveraNivoa(7);
    }
}
```

}

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci za primenu izuzetaka u toku realizacije koda.

- 1. Zadatak:** Prikazati realizaciju programskog koda za proizvoljnu aplikaciju. Izvršiti definisanje preduslova, postuslova i prikazati nasleđivanje. Prikazati primenu izuzetaka u proizvoljnem delu programskog koda.
- 2. Zadatak:** Da li primena izuzetaka olakšava realizaciju programskog koda? Prikazati proveru preslikavanja na programskom kodu proizvoljne aplikacije.

▼ Poglavlje 10

Upravljanje implementacijom

DOKUMENTOVANJE TRANSFORMACIJA

Neophodnost je ažurno unositi sve urađene transformacije i izmene

Neophodnost je ažurno unositi sve urađene transformacije i izmene. Kako preslikavanje objektnog modela u programski kod nije "jedan-na-jedan", već se vrše promene modela pre preslikavanja, mora da postoji dokumentacija o tim promenama. Bez toga, pri primeni obrnutog inženjerstva došlo bi do gubitka informacija.

Daju se sledeće preporuke za održavanje konzistentnosti objektnog modela i programskog koda:

1. Za određenu transformaciju, upotrebljavajte isti alat.
2. Stavite ugovore u izvršni kod, a ne u objektni model.
3. Upotrebljavajte ista imena za iste objekte.
4. Koristite eksplisitne transformacije.

Napomena: Pod "ugovorom" se u ovom predmetu podrazumeva opis ponašanja metoda i ograničenja koja se određuju sa njegove parametre i attribute. Programeri menjaju ugovor promenom tela metoda u izvornom kodu.

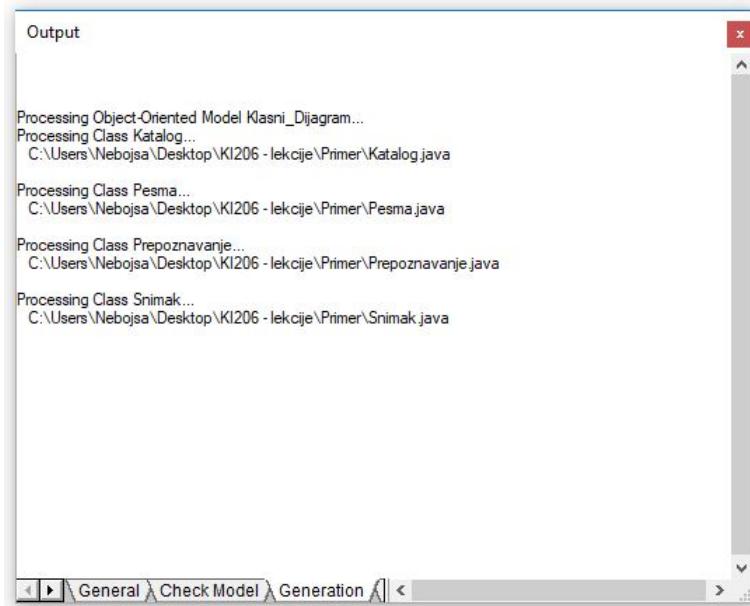
Za dokumentovanje transformacija odgovornost ima nekoliko članova projektnog tima:

1. **Glavni arhitekta:** Bira transformacije koje će se sistematski sprovoditi.
2. **Pomoćni arhitekte:** Odgovaran za dokumentovanje ugovora koji odgovaraju interfejsima podsistema. Ako se ugovori menjaju, moraju se obavestiti svi korisnici klasa.
3. **Programer:** odgovoran je da primenjuje konvencije definisane od strane glavnog arhitekta za transformacije. Odgovorni su da održavaju ažurnost komentara izvornog koda sa modelom.

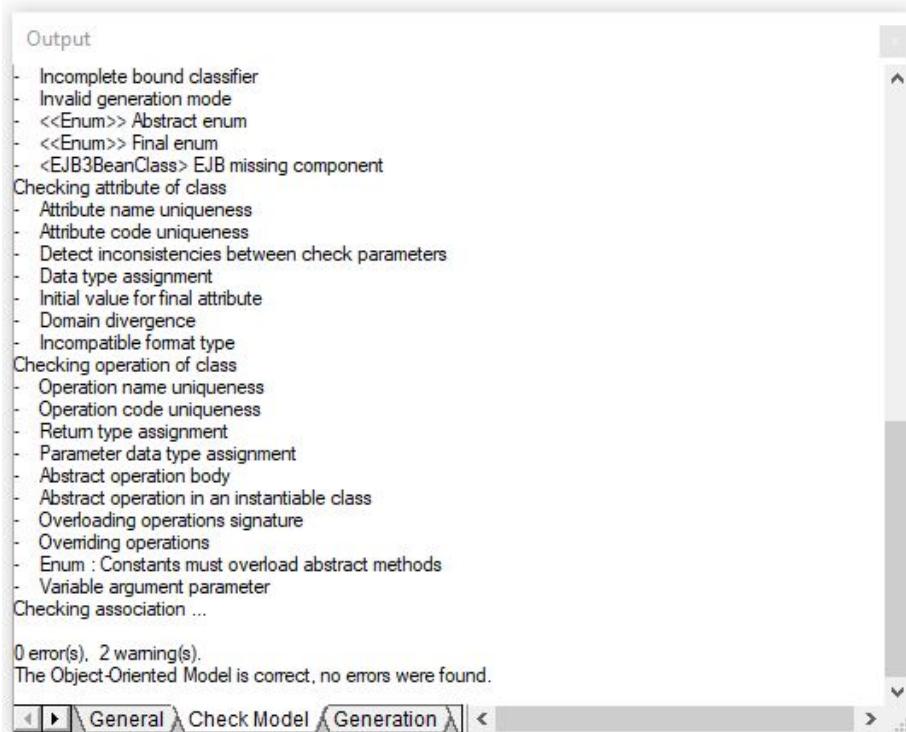
PRIMER DOKUMENTOVANJA TRANSFORMACIJE KLASNOG DIJAGRAMA U JAVA PROGRAMSKI KOD

Na primeru je prikazano dokumentovanje generisanja Java programskog koda iz klasnog dijagrama.

Na slici 1 je prikazano dokumentovanje transformacije klasnog dijagrama aplikacije u Java programski kod. Predstavljene su sve akcije koje su izvršene u tom procesu tako da razvojni tim ima uvid u svaki korak u toku transformacije i procesa generisanja.



Slika 10.1 Prikaz dokumentovanja generisanja Java klasa



Slika 10.2 Proveravanje modela u procesu generisanja

ZADACI ZA SAMOSTALNI RAD

Zadaci koji se odnose na dokumentovanje transformacija.

- 1. Zadatak:** Modelovati klasni dijagram za aplikaciju za naručivanje autodelova. **Opis aplikacije:** Aplikacija omogućava svakom korisniku laku kupovinu bilo kog autodela koji mu je potreban i brzu isporuku na kućnu adresu. Prikazati transformaciju klasnog dijagrama u Java programski kod ili opisati korake u dokumentovanju izvršene transformacije.
- 2. Zadatak:** Za proizvoljnu aplikaciju izvršiti transformaciju klasnog dijagrama u Java programski kod. Nakon transformacije na osnovu dobijene dokumentacije koraka izvršiti izmenu u Java klasi i sačuvati je kroz razvojno okruženje.

▼ Poglavlje 11

Vežba – Zadatak za samostalni rad

ZADATAK ZA SAMOSTALAN RAD

Zadaci transformacije modela u programski kod

Za sledeći scenario potrebno je uraditi objektno modelovanje kompletognog sistema i transformisati ga u Java programski kod.

Potrebno je izraditi IS za apoteku. Apoteka poseduje dva magacina koji u sebi sadrže lekove. Svaki lek u sebi sadrži informaciju o imenu i datumu isteka. Apoteka vodi evidenciju svojih klijenata. Više klijenata može kupovati više lekova. Od podataka klijenta čuvaju se informacije (ime, prezime, adresa). Napraviti odgovarajući klasni dijagram i transformisati taj dijagram u kod. Nakon transformacije dijagrama kreirati main metodu koja će demonstrirati prvo kreiranje magacina, zatim 3 leka u magacinu, kreiranje jednog klijenta i njegovou kupovunu 2 leka.

Napomena: U ovom zadatku se mogu primeniti više različitih veza (jedan-na-više, više-na-više).

✓ Poglavlje 12

Zaključak

ZAKLJUČAK

1. Projektovanje softvera i njegova implementacija su dve spregnute aktivnosti. Nivo detalja koje projektno rešenje softvera treba da sadrži zavisi od tipa softvera koji se razvija i od toga da li se koristi razvoj vođen planom ili se koristi agilni pristup u projektovanju.
2. Proces projektovanje objektno-orientisanog softverskog sistema uključuje aktivnosti projektovanja arhitekture sistema, utvrđivanja osnovnih objekata sistema, opisivanje projektnog rešenja upotrebom različitih UML objektnih modela. I dokumentovanjem interfejsa komponenata.
3. Broj i vrste UML modela koji se radi u toku projektovanja može da bude različit. Koriste se statički modeli (modeli klasa, modeli generalizacije, modeli asocijacije) i dinamički modeli (seqvencijalni modeli, modeli stanja).
4. Interfejsi komponenata moraju se precizno definisati tako da ih drugi objekti mogu da koriste. U tu svrhu se koristi UML stereotip za interfejse.
5. Pri razvoju softvera, uvek gledajte da u što većoj meri koristite postojeći softver, bilo u vidu komponenti, u vidu servisa, ili u vidu celih sistema.
6. Upravljanje konfiguracijom je proces upravljanja promenama softverskog sistema koji je u razvoju. Vrlo je bitno da razvojni tim međusobno sarađuje u toku razvoja softvera.
7. Najčešće se softver razvija na razvojnoj platformi, a realizuje na izvršnoj platformi kod kupca. Na razvojnoj platformi se koristi sistem za razvoj softvera (IDE), a razvijeni izvršni kod softvera se onda prebacuje na ciljnu mašinu, tj. računar na kome će raditi u toku njegove eksploatacije (tzv. izvršna platforma).
8. Razvoj softvera sa otvorenim (izvornim) kodom je razvoj softvera čiji je izvorni kod javno dostupan. To znači da mnogi mogu da predlože promene i poboljšanja softvera.