

Univerzitet u Novom Sadu  
Fakultet tehničkih nauka

Gordana Milosavljević

# **UVOD U MODELOVANJE SOFTVERA**

Novi Sad, 2020.

*Edicija: „TEHNIČKE NAUKE - UDŽBENICI”*

*Naziv udžbenika: „UVOD U MODELOVANJE SOFTVERA”*

*Autor:* Gordana Milosavljević

*Recenzenti:* dr Igor Dejanović, vanredni profesor Fakulteta tehničkih nauka u Novom Sadu  
dr Danijela Boberić Krstićev, vanredni profesor Prirodno-matematičkog  
fakulteta u Novom Sadu

*Izdavač:* Fakultet tehničkih nauka, Univerzitet u Novom Sadu

*Glavni i odgovorni urednik:*

Prof. dr Rade Doroslovački, dekan Fakulteta tehničkih nauka, Univerzitet u Novom Sadu

*Priprema i štampa:* FTN - Grafički centar GRID, Trg Dositeja Obradovića 6, Novi Sad

*Štampanje odobrio:*

Savet za biblioteku i izdavačku delatnost Fakulteta tehničkih nauka, Univerzitet u Novom Sadu

*Predsednik Saveta za biblioteku i izdavačku delatnost:*

Prof. dr Milan Martinov, redovni profesor Fakulteta tehničkih nauka, Univerzitet u Novom Sadu

*Autorska prava pripadaju izdavaču*

CIP-Каталогизација у публикацији  
Библиотека Матице српске, Нови Сад

004.414.23(075.8)

**МИЛОСАВЉЕВИЋ, Гордана**

Uvod u modelovanje softvera / Gordana Milosavljević. - 1. izd. - Novi Sad :  
Fakultet tehničkih nauka, 2020 (Novi Sad : FTN, Grafički centar GRID). - 236 str. :  
graf. prikazi, tabele ; 24 cm. - (Edicija "Tehničke nauke - monografije" ; 859)

Tiraž 100. - Bibliografija.

ISBN 978-86-6022-267-3

a) Моделовање софтвера

COBISS.SR-ID 16919561

# Predgovor

Ovaj udžbenik je namenjen studentima druge godine koji slušaju predmet Specifikacija i modelovanje softvera na smeru Softversko inženjerstvo i informacione tehnologije na Fakultetu tehničkih nauka u Novom Sadu. Pored toga, može biti od koristi svima koji žele da savladaju osnove modelovanja korišćenjem UML-a (*Unified Modeling Language* – objedinjeni jezik modelovanja): najčešće korišćene vrsta dijagrama koje UML poseduje i njihovu primenu na implementaciju različitih softverskih sistema.

Tekst udžbenika je organizovan na sledeći način.

U prvom poglavlju je dat kratak prikaz UML-a i objašnjeno je šta su modeli, koje su moguće namene modela i zašto je modelovanje neizostavan deo većine metodologija razvoja softvera.

U narednim poglavljima se detaljno obrađuju najčešće korišćeni UML dijagrami.

U drugom poglavlju je prikazan dijagram slučajeva korišćenja koji se primenjuje tokom analize i specifikacije funkcionalnih zahteva razmatranog sistema.

U trećem poglavlju je prikazan dijagram aktivnosti koji se može koristiti za specifikaciju poslovnih procesa, radnih tokova, algoritama i složenih slučajeva korišćenja.

U četvrtom poglavlju je obrađen dijagram klasa koji se koristi za modelovanje strukture sistema. Uočeni koncepti i njihovi uzajamni odnosi se reprezentuju klasama, interfejsima i vezama između njih.

U petom poglavlju je obrađen dijagram objekata (instanci klasa) koji se koristi za bolje razumevanje i dokumentovanje dijagrama klasa.

Dijagram sekvence je opisan u šestom poglavlju. Koristi se za modelovanje interakcije, odnosno komunikacije između uloga sistema u vremenu. Interakcija se odvija razmenom poruka.

Dijagram komunikacije, koji takođe modeluje interakciju između uloga sistema, ali bez specifikacije vremenske dimenzije, je prikazan u sedmom poglavlju.

Dijagram prelaza stanja (konačnih automata), kojim se modeluju moguća stanja nekog sistema i događaji koji izazivaju promene stanja, je obrađen u osmom poglavlju.

U devetom poglavlju su ukratko prikazani dijagram komponenti i dijagram raspoređivanja i dati su zadaci za vežbanje.

Sva poglavlja su tako organizovana da, pored savlađivanja elemenata, notacije i pravila odgovarajuće vrste dijagrama, studenti uče i primenu date vrste dijagrama na razvoj delova realnih projektnih zadataka.

Radi lakšeg prihvatanja koncepata UML-a, dijagrami su često ilustrovani delovima programskog koda pisanog na programskom jeziku Java. Pošto je udžbenik prvenstveno namenjen studentima druge godine akademskih studija koji su tek savladali objektno-orijentisano (OO) programiranje, na ovaj način, pored modelovanja, studenti dodatno uvežbavaju i proširuju veštinu OO programiranja i dizajna.

Želim da se zahvalim generaciji studenata koji su slušali predmet Specifikacija i modelovanje softvera školske 2019/2020. godine, koji su svojim pitanjima, komentarima i detaljnom analizom ovog teksta doprineli da se otklone neke greške i nejasnoće.

Veliku zahvalnost dugujem i recenzentima, prof. dr Igoru Dejanoviću i prof. dr Danijeli Boberić Krstićev, koji su doprineli da se tekst pročisti i bolje strukturira.

Želim da se zahvalim i svom metoru, prof. dr Branku Perišiću koji me je nekada davno uveo u tajne modelovanja poslovnih sistema.

Mirjani Isakov dugujem zahvalnost za ilustraciju na koricama udžbenika.

I na kraju, želim da se zahvalim suprugu Branku i mojim devojčicama Miri i Kaći na nesebičnoj podršci tokom pisanja ove knjige.

U Novom Sadu,

Gordana Milosavljević

Maj 2020.

## Poglavlje 1

### Uvod

Cilj ovog udžbenika je da prikaže osnove modelovanja softverskih sistema korišćenjem UML-a (*Unified Modeling Language*) – objedinjenog jezika za modelovanje. Pored toga, cilj je da studenti shvate značaj modelovanja i nauče kako da ga integrišu u proces razvoja softvera, nezavisno od metodologije koju koriste.

UML je realizovan sa idejom da omogući zajedničku osnovu za analizu, projektovanje i implementaciju softverskih sistema, kao i modelovanje poslovnih procesa. Inicijalno je kreiran sa namerom da objedini i standardizuje različite notacije koje su korišćene za modelovanje objektno-orijentisanih (OO) sistema. Prvu verziju su razvili Gredi Buč (Grady Booch), Ivar Jakobson (Ivar Jacobson) i Džejms Rambau (James Rumbaugh) u okviru korporacije *Rational Software* 1995. godine. Od 1997. godine je njegov dalji razvoj i održavanje preuzela *Object Management Group* (OMG)<sup>1</sup>, sastavljena od zainteresovanih članova iz industrije (pre svega proizvođača alata za razvoj softvera) i akademske zajednice. Sve verzije UML-a se nalaze na sajtu OMG grupe<sup>2</sup>. Poslednja verzija, u trenutku pisanja ove knjige, bila je UML 2.5.1 iz decembra 2017.

UML se sastoji od više vrsta dijagrama, koji se koriste za modelovanje različitih aspekata strukture i ponašanja softverskog sistema (slika 1.1). U udžbeniku ćemo obraditi najčešće korišćene dijagrame, sa akcentom na primere modelovanja i simulaciju realnih situacija koje se dešavaju pri razvoju softvera.

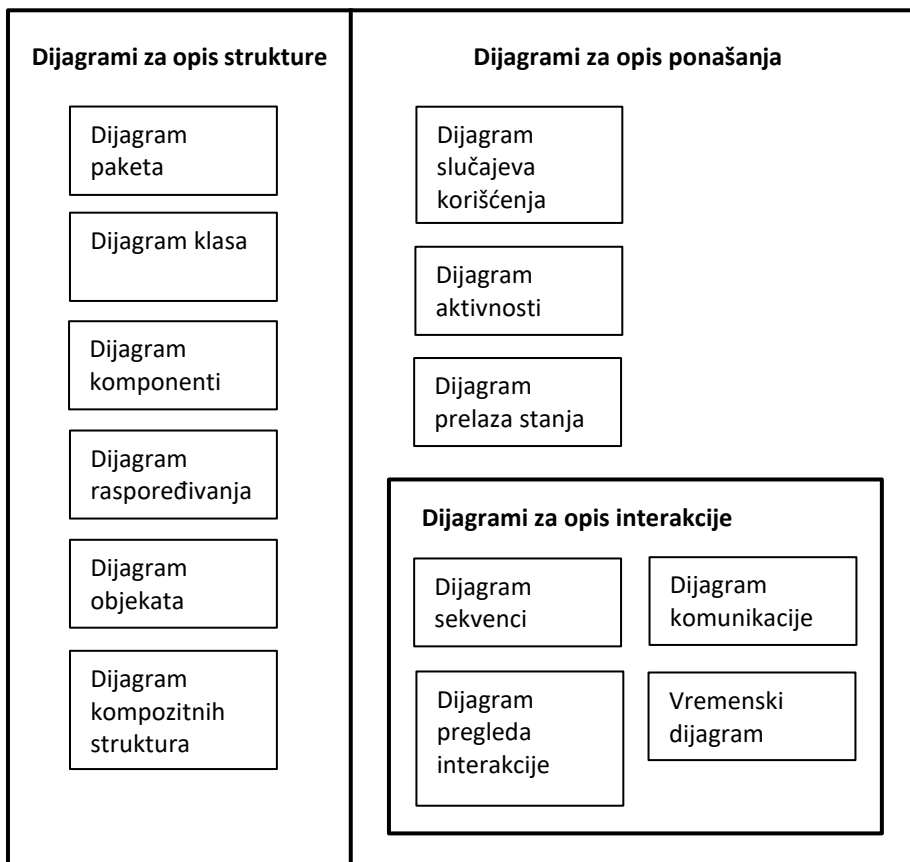
Kao što se može primetiti sa slike 1.1, UML ne poseduje dijagrame za modelovanje šema relacionih baza podataka i korisničkog interfejsa, za šta se koriste posebni jezici za modelovanje, kao što su CWM [CWM1.1] i IFML [IFML1.0]. Navedeni jezici nisu obrađeni u ovom udžbeniku, pošto se relacione baze podataka i dizajn korisničkog interfejsa proučavaju na posebnim predmetima.

---

<sup>1</sup> [www.omg.org](http://www.omg.org)

<sup>2</sup> [www.omg.org/spec/UML](http://www.omg.org/spec/UML)

Ako je čitaocu potrebno više informacija, u spisku literature su navedeni naslovi koji mogu pomoći u daljem samostalnom radu. Sažet prikaz svih vrsta UML dijagrama se može naći u [Fowler04]. Agilno modelovanje korišćenjem UML-a i drugih jezika za modelovanje je obrađeno u [Ambler04]. Detaljan opis svakog elementa UML dijagrama sa primerima upotrebe se može naći u [Rumbaugh05].



Slika 1.1 UML dijagrami [UML2.5.1]

Pre nego što se posvetimo pojedinačnim vrstama UML dijagrama, u nastavku ovog poglavlja ćemo se baviti sledećim pitanjima:

- Koji izazovi postoje u razvoju softvera?
- Zašto su nam potrebne metodologije?
- Šta su modeli i kako nam mogu pomoći u savlađivanju kompleksnosti realnih i softverskih sistema?
- Koje su moguće namene modela?

## 1.1 Izazovi u razvoju softvera

Prema rezultatima istraživanja *Standish* grupe koji se objavljuju od 1994. godine<sup>3</sup>, rezultati softverskih timova su značajno lošiji od rezultata koje postižu starije inženjerske discipline, kao što je npr. građevinsko, elektrotehničko ili mašinsko inženjerstvo. U tabeli 1.1 su prikazani rezultati izveštaja *Standish* grupe koji su u trenutku pisanja udžbenika bili javno dostupni.

Tabela 1.1 Rezultati istraživanja *Standish* grupe o uspešnosti softverskih projekata za period 1994. – 2015.

Godina	Uspešni projekti	Delimično uspešni	Neuspešni projekti
2015	36%	45%	19%
2014	36%	47%	17%
2013	41%	40%	19%
2012	37%	46%	17%
2011	39%	39%	22%
2009	32%	44%	24%
2006	35%	46%	19%
2004	29%	53%	18%
2000	28%	49%	23%
1998	26%	24%	28%
1996	27%	33%	40%
1994	16%	53%	31%

Uspešnim projektima se smatraju oni koji su ušli u primenu, u potpunosti zadovoljavaju zahteve naručioca i realizovani su u okviru predviđenog vremena i budžeta. Delimično uspešni projekti su ušli u primenu, ali nisu implementirali sve zahteve korisnika, premašili su predviđeno vreme i/ili budžet. Neuspešni projekti su bili prekinuti u nekom trenutku i nikad nisu ušli u primenu.

---

<sup>3</sup> Videti npr.

[https://www.standishgroup.com/sample\\_research\\_files/CHAOSReport2015-Final.pdf](https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf)

Razlog za navedeno je:

- tehnička složenost koju imaju softverski projekti (brz razvoj tehnologije, kontinuirana pojava novih uređaja i novih razvojnih okruženja, različiti specijalisti koji moraju da sarađuju vezano za aspekte sistema koji su u njihovoj nadležnosti),
- složenost realnih sistema za koje se realizuje softver, kao i veliki stepen neodređenosti koje u proces razvoja unose promenljivi zahtevi, teškoće u komunikaciji i različiti socio-psihološki i organizacioni problemi.

Ako kao primer uzmemo poslovne informacione sisteme kod kojih je procenat neuspeha obično najveći, možemo primetiti sve navedene poteškoće [Milosav10].

Moderni informacioni sistemi se implementiraju tako da se mogu koristiti i iz lokalne mreže preduzeća (desktop i web aplikacije) kao i na terenu, korišćenjem mobilnih telefona i tableta. Ne koriste ih samo njihovi zaposleni, već i klijenti koji mogu direktno naručivati proizvode putem internet prodaje. Sarađuju sa informacionim sistemima drugih preduzeća, kao i sa sistemima platnog prometa, poreske uprave i drugim sistemima. Arhitekture poslovnih sistema su prešle put od monolitnih aplikacija za mejnfrem računare, preko dvoslojnih klijent-server arhitektura, do troslojnih i višeslojnih servisno-orijentisanih arhitektura koje su danas aktuelne. Svaki sloj i svaki uređaj zahtevaju određena specijalistička znanja. Da bi se implementirali ovakvi sistemi, potrebna je tesna saradnja različitih vrsta specijalista.

Preduzeća za koje se softver razvija su izložena različitim uticajima: pojačanoj konkurenciji, preokretima na tržištu, globalizaciji ekonomije i brzim tehnološkim promenama. Da bi opstala na tržištu, preduzeća se moraju brzo prilagođavati, što znači da se i njihov informacioni sistem mora brzo menjati, bilo da je u fazi razvoja ili održavanja. Korisnički zahtevi koji su postojali na početku razvoja projekta, već posle nekoliko meseci ne moraju više biti aktuelni. Kada korisnici počnu da upotrebljavaju razvijeni softver, po pravilu daju nove zahteve šta bi trebalo promeniti ili unaprediti.

Sa stanovišta prikupljanja i analize korisničkih zahteva, takođe postoji niz izazova. U preduzećima obično ne postoji dokumentovano i centralizovano znanje o radnim procedurama i poslovnim procesima koje zaposleni slede, kao ni osoba koja njime u potpunosti vlada. Znanje je obično rasuto u okviru lokalnih radnih procedura pojedinačnih odeljenja, smešteno u dokumentima koji se razmenjuju u okviru ograničenih grupa učesnika i/ili delimično založeno u okviru postojećeg informacionog sistema koji često nije integrisan i poseduje redundanse ili



protivrečnosti. Operativni izvršioci obično poznaju samo svoje procedure i eventualno procedure onih sa kojima neposredno sarađuju. Uprava poznaje u grubim crtama poslovne procese na nivou preduzeća, ali obično ne poseduje detaljno, operativno znanje potrebno za realizaciju softvera. Nije retka pojava da su neki poslovni procesi redundantni, u međusobnoj kontradikciji ili više nisu od značaja za preduzeće. Razvojni tim nije suočen samo sa problemom otkrivanja potreba korisnika (prikupljanje, analiza i pisanje korisničkih zahteva) već i problemom otkrivanja znanja o realnom sistemu za koji se softver razvija.

Prilikom rada na otkrivanju znanja o realnom sistemu, problem je i izabrati predstavnike korisnika koji će služiti kao izvor informacija - pošto se u razvoj ne mogu uključiti svi zaposleni jednog preduzeća. Stavljanje akcenta na bilo koju grupu ili hijerarhijski nivo korisnika pogoduje neuspehu projekta u slučaju nedovoljne podrške date grupe. Ostali korisnici, zbog nedovoljnog učestvovanja u procesu razvoja, mogu dobijeno rešenje smatrati neodgovarajućim i nametnutim i u trenutku uvođenja boriti se svim sredstvima protiv njega (sabotažama, ignorisanjem i sl).

Tehnike koje teže da uvedu demokratičnost u razvoj uključivanjem svih nivoa korisnika, takođe mogu dovesti do propusta. Korisnici nižih hijerarhijskih nivoa se često ustežu da javno iznose svoje potrebe (pogotovo u prisustvu uprave), ne žele da donose odluke jer smatraju da nisu dovoljno kompetentni ili da to nije njihov posao ili su, zbog nepoznavanja računarske terminologije, skloni da se saglase sa svim predlozima projektnog tima (što se drastično menja u trenutku uvođenja). U nekim slučajevima, „vlasnici“ ekspertskih znanja koja im daju određenu moć u njihovom okruženju (subjektivnu ili objektivnu), teže da mistifikuju ta znanja u strahu da će izgubiti na značaju ili da će ostati bez posla. Neki korisnici pružaju otpor u strahu od promena – boje se da neće umeti da koriste novi sistem, posebno ako do tada nisu koristili računar.

Postoje problemi i u komunikaciji. Korisnici obično ne poseduju tehničko, a projektni tim domensko znanje određene struke. Korisnici se uglavnom bave konkretnim pitanjima svog posla, dok projektni tim pokušava da formira generalizacije, koje su prosečnom korisniku previše apstraktne. Različiti dokumenti koji se nude korisnicima na overu, njima obično puno ne znače. Džejms Martin (James Martin) je prezentirao rezultate studije sprovedene u okviru jedne softverske firme, gde je utvrđeno da 64% grešaka potiču iz faza analize i projektovanja, iako su korisnici potpisali sve dokumente proistekle iz tih faza. Mnoge rutine i postupci se nalaze u formi „skrivenih“ znanja koja se podrazumevaju u okviru određene struke i obično ne navode u kontaktu sa razvojn timerom.

Iz svega navedenog, jasno je da se ovakvi sistemi ne mogu implementirati tako što tim programera samo počne da piše programski kod. Slično tome, ni soliter se ne može izgraditi tako što grupa zidara samo počne da slaže cigle i malter. Potrebne su metodologije, tehnike i alati koje primenjujemo sa ciljem da se u razvoj uvede sistematičnost i povećaju šanse za uspeh projekta [SWEBOK].

Modelovanje je tehnika koja ima centralno mesto u većini metodologija razvoja softvera.

## 1.2 Vrste i namena modela

Model je pojednostavljena predstava realnog sistema kreirana sa određenim ciljem. Ciljevi kreiranja modela mogu biti:

- Bolje razumevanje i dokumentovanje realnog sistema za koji se softver razvija.
- Lakša komunikacija između različitih učesnika u procesu razvoja – „jedna slika govori više od hiljadu reči“.
- Isprobavanje različitih ideja u okviru razvojnog tima na „jeftiniji“ način – manje vremena i truda zahteva kreiranje i analiza nekoliko verzija modela pre prelaska na implementaciju, nego prebrza i nepripremljena implementacija za koju se, posle puno uloženog truda i vremena, pokaže da ne zadovoljava postavljene zahteve.
- Specifikacija kako implementirati softverski sistem.
- Automatizacija implementacije, korišćenjem interpretera ili generatora koda, koji izvršavaju model ili generišu kod za ciljnu platformu (detalji u sekciji 1.4).

Da bismo ostvarili postavljene ciljeve, modeli se kreiraju tako da reprezentuju one koncepte iz stvarnog sveta koji su bitni za određenu situaciju ili namenu, a da zanemare ono što trenutno nije od važnosti. Proces zanemarivanja nebitnih informacija prilikom kreiranja modela se zove apstrakcija. Bez primene apstrakcija, morali bismo da se suočimo sa svom složenošću realnog sveta.

Modeli na visokom nivou apstrakcije se koriste za prikaz celovite slike o funkcionisanju sistema, obično u početnim fazama analize zahteva i specifikacije dizajna. Modeli na niskom nivou apstrakcije, bliski programskom kodu, se najčešće koriste tokom detaljnog dizajna sistema i tokom implementacije.

Po svojoj nameni, modeli mogu biti *deskriptivni* i *preskriptivni*. Deskriptivni modeli se koriste za bolje razumevanje sistema, dokumentovanje i komunikaciju ideja. Preskriptivni modeli se koriste radi specificiranja kako neki sistem treba da

se implementira, nebitno da li je u pitanju ručna ili automatizovana implementacija. Modeli istovremeno mogu biti i deskriptivni i preskriptivni.

### **1.3 Modeli u fazama razvoja softvera**

Cilj metodologija razvoja softvera je savlađivanje kompleksnosti i povećanje verovatnoće uspeha softverskog projekta. Nezavisno od vrste životnog ciklusa na kojoj je određena metodologija bazirana (npr. linearni, spiralni ili neka od varijacija iterativno-inkrementalnog), modelovanje i modeli imaju centralnu ulogu u većini razvojnih aktivnosti, kao što su: analiza zahteva, specifikacija dizajna, konstrukcija, testiranje i održavanje.

Tokom prikupljanja, analize i zapisivanja zahteva koje korisnici postavljaju pred budući softverski sistem, modeli imaju ulogu da pomognu u komunikaciji između svih zainteresovanih strana (različitih specijalista u razvojnom timu, naručioca, budućih korisnika, itd), kao i da omoguće kvalitetan ulaz u fazu specifikacije dizajna.

Tokom faze dizajna se kreira arhitektura softvera, što podrazumeva dekompoziciju (razlaganje) softverskog sistema na komponente i definisanje interfejsa između komponenti. Ovo je dizajn na visokom nivou apstrakcije. U zavisnosti od metodologije, u ovoj fazi se može obaviti i detaljan dizajn, odnosno specifikacija uočenih komponenti.

Cilj modela u ovoj fazi je da pomognu u komunikaciji između članova razvojnog tima, da posluže pri izboru između više mogućih rešenja, kao i da obezbede specifikacija za fazu konstrukcije (ako je obavljen detaljan dizajn). Kao što arhitekta različitim vrstama modela specificira kako građevinski objekat treba da se izgradi, tako i softverski arhitekta može da specificira programerima kako da izvrše implementaciju, kreiranjem modela koji opisuju različite aspekte strukture i ponašanja komponenti sistema. Ako je u pitanju agilni tim (videti sekciju 1.5), softverski arhitekta obično ne postoji kao izdvojena uloga, već ceo tim učestvuje u modelovanju i definisanju arhitekture.

Tokom faze konstrukcije se vrši implementacija izvršive aplikacije, što u najvećoj meri podrazumeva pisanje programskog koda i testiranje. Testiranje se sastoji od pisanja i izvršavanja jediničnih i integracionih testova koji treba da pokažu da programski kod radi ono što je razvojni tim razumeo da treba da radi. U zavisnosti od metodologije, modeli u ovoj fazi mogu da služe kao ulazna specifikacija šta i kako treba implementirati, ali i za komunikaciju ideja između članova razvojnog tima i analizu različitih rešenja. Takođe, modeli se mogu iskoristiti i za direktnu

proizvodnju programskog koda, ako se primenjuje inženjerstvo softvera vođeno modelima (videti sekciju 1.4).

Faza verifikacije podrazumeva skup aktivnosti koje treba da pokažu da softver radi baš ono što je korisniku potrebno. Tu obično najvažniju ulogu imaju modeli kreirani tokom specifikacije zahteva, radi osmišljavanja scenarija za verifikaciju na osnovu modelovanih korisničkih zahteva.

U fazi održavanja, koja podrazumeva ispravke grešaka i dodavanje novih funkcionalnosti pošto je softverski sistem uveden u primenu, modeli mogu imati sve navedene uloge, uključujući i ulogu dokumentacije.

## 1.4 Inženjerstvo softvera vođeno modelima

Inženjerstvo softvera vođeno modelima (*Model Driven Software Engineering*, MDSE) podrazumeva razvoj softvera kod kojeg se model razmatranog sistema tretira kao deo implementacije. Model se automatizovanim postupcima, kroz jednu ili više transformacija, prevodi na programski kôd koji se direktno izvršava na ciljnoj platformi. Uz oslonac na jezike specifične za domen (*Domain Specific Language*, DSL) koji su optimizovani za efikasno modelovanje u određenoj oblasti i različite alate koji implementiraju pomenute transformacije, MDSE pokušava da postigne brzinu razvoja i prevaziđe probleme koji se nedovoljno efikasno rešavaju samo korišćenjem jezika treće generacije.

MDSE ima više predstavnika, od kojih je najpoznatija MDA (*Model Driven Architecture*) inicijativa<sup>4</sup>, pokrenuta od strane OMG grupe 2001. godine. U kontekstu MDA, pod modelom se podrazumeva sistem ili deo sistema zapisan korišćenjem dobro definisanog jezika koji podržava automatsku interpretaciju od strane računara.

U okviru MDA, početni model kojim se vrši apstrakcija razmatranog problema se realizuje tako da *ne zavisi* od konkretne implementacione platforme (*Platform Independent Model*, PIM). Cilj ovog koraka je da znanje o funkcionisanju realnog sistema koje se uloži u model ostane iskoristivo i u slučaju promene razvojne tehnologije.

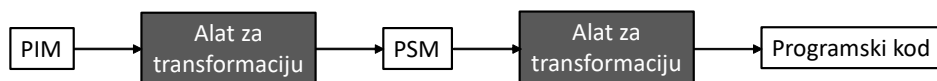
---

<sup>4</sup>Ostali predstavnici su: Razvoj softvera vođen modelima (*Model-Driven Software Development*, MDSD), Računarstvo integrisano modelima (*Model Integrated Computing*, MIC), Programiranje orijentisano ka jezicima (*Language Oriented Programming*, LOP), Modelovanje specifično za domen (*Domain-Specific Modeling*, DSM), Generativno programiranje (*Generative Programming*), Modelovanje specifično za okvir za razvoj (*Framework-Specific Modeling*), Izvršivi UML (*Executable UML*, xUML) itd.

Zadatak projektnog tima je da opiše svoj sistem u okviru PIM-a, posle čega se koriste alati koji, na osnovu definisanih pravila za transformaciju, generišu jedan ili više modela *zavisnih* od platforme (*Platform Specific Model*, PSM) i veze između njih (slika 1.1). U kontekstu MDA, transformacija predstavlja automatsko generisanje ciljnog modela na osnovu početnog modela, u skladu sa definicijom transformacije. Definicija transformacije predstavlja skup pravila od kojih svako opisuje kako se jedna ili više konstrukcija iz početnog jezika može preslikati na jednu ili više konstrukcija ciljnog jezika.

Posle automatskog generisanja primenom transformacija, predviđeno je da se PSM-ovi obogate dodatnim informacijama specifičnim za datu implementacionu platformu, kroz ručna podešavanja od strane razvojnog tima. Posle završenih podešavanja, PSM-ovi se koriste za automatsko generisanje programskog koda ili za direktno izvršavanje od strane odgovarajuće virtuelne mašine.

U slučaju otkrivanja greške ili promene u domenu primene, razvojni tim se vraća na rad na PIM-u, a opisani koraci se ponavljaju.



Slika 1.1 Osnovni koraci u MDA razvojnom procesu [Kleppe03]

## 1.5 Modelovanje u klasičnim i agilnim metodologijama

Klasične metodologije se oslanjaju na različite vrste modela i dokumentacije, za koju je predviđeno da se sve vreme usaglašava sa programskim kodom i međusobno. Rad počinje detaljnim prikupljanjem skupa zahteva, posle čega sledi projektovanje, konstrukcija, verifikacija i uvođenje, sa obimnom dokumentacijom koja prati svaku od faza, što može da troši puno vremena i da dovode do iscrpljivanja projektnog tima i odvlačenja pažnje od realizacije finalnog proizvoda. Cilj obimnog modelovanja i dokumentovanja je prenos informacija između različitih faza razvoja, pokušaj izbegavanja zavisnosti od bilo kog učesnika u razvojnom procesu i postizanje kontrole projekta [Cockburn02]. Modeli u klasičnim metodologijama obično imaju preskriptivnu ulogu, kao i ulogu dokumentacije.

Nažalost, zbog velike složenosti problemskog domena, teškoća u komunikaciji između projektnog tima i korisnika, nekad neadekvatnoj komunikaciji između različitih specijalista u okviru tima baziranoj na dokumentaciji i nedostatku povratnih informacija od strane korisnika na „nešto što radi“ često do kasnih faza razvoja projekta, dešava se da realizovano rešenje znatno odstupa od

inicijalne vizije i da kao takvo postaje neprihvatljivo za korišćenje (videti tabelu 1.1).

Agilne metodologije su nastale sredinom 90-tih godina prošlog veka kao reakcija na probleme klasičnih metodologija razvoja softvera da brzo prezentuju konkretne rezultate i odgovore na promene. Zbog brzih promena iz okruženja koje utiču na razmatrani sistem, autori agilnih metodologija smatraju da na početku projekta nije moguće definisati kompletan, nepromenljiv skup zahteva koji bi služio kao podloga za ostale faze razvoja. Oni se zalažu za minimiziranje dokumentacije tokom razvojnog procesa koja se svodi samo na formiranje „markera“. To mogu biti modeli na visokom nivou apstrakcije i kratki pregledni dokumenti koji pomažu da se tražene informacije pronađu direktno u programskom kodu, koji predstavlja najtačniju specifikaciju i odraz stanja projekta. Umesto razmene i pohranjivanja informacija putem dokumenata i modela, teže postizanju deljenog znanja (*shared knowledge*) u okviru tima: boravkom u istoj prostoriji, komunikaciji sa korisnikom u kojoj svi učestvuju, izbegavanjem specijalizacija na analitičare, projektante i programere (svi su „specijalisti opšte prakse“), radom u parovima (*pair programming*), samoorganizaciji i podsticanju kreativnosti svih učesnika.

Vrednosti za koje se zalažu pripadnici agilnog pokreta, definisane u okviru Agilnog manifesta<sup>5</sup> su:

- Pojedinci i njihove interakcije vrede više od metodoloških procesa i razvojnih alata.
- „Nešto što radi“ vredi više od obimne dokumentacije.
- Stvaranje bliske saradnje sa korisnikom vredi više od pregovaranja oko ugovora.
- Reagovanje na promene vredi više od striktnog praćenja planova.

Agilni pokret ne negira potrebu postojanja metodoloških procesa, planova, ugovora i dokumentacije, već samo izražava stanovište da oni nisu od presudne važnosti za uspeh projekta. Razvojni tim sastavljen od motivisanih pojedinaca, koji kroz uzajamne interakcije dolaze do kreativnih rešenja, ima veće mogućnosti da postigne željeni cilj od tima koji samo formalno sledi pravila propisana određenim metodološkim procesom ili nametnuta od strane korišćenog alata. Mada je posedovanje modela i dokumentacije korisno, oni bez izvršivog koda ne vrede puno, jer se kvalitetne povratne informacije mogu dobiti samo na bazi

---

<sup>5</sup> [www.agilemanifesto.org](http://www.agilemanifesto.org)

„nečega što radi“<sup>6</sup>. Rad na dobroj saradnji sa naručiocima softvera omogućava stvaranje atmosfere poverenja i prijateljstva gde su svi na istoj strani, što može učiniti nepotrebnim postojanje striktnih ugovora. Planovi, pogotovo dugoročni, često odstupaju od realne situacije zbog brzih promena koje se dešavaju tokom razvoja – tada ih nema smisla striktno slediti, već se, u cilju uspeha projekta, treba prilagođavati promenjenim okolnostima.

Nažalost, ovakav način rada najviše pogoduje malim razvojnim timovima i može dovesti do gubitka informacija kada razvojni tim, posle završetka projekta, prestane da postoji (dolazi do gubitka deljenog znanja).

Iz navedenih razloga, Bari Bem (Barry Boehm) se zalaže da se za svaki projekat, na osnovu njegovih karakteristika, odredi kombinacija agilnih i klasičnih tehnika (sa težištem na jednoj ili drugoj strani) i da se na taj način prevaziđu nedostaci oba pristupa [Boehm03]. Karakteristike koje se razmatraju su: dinamičnost projekta (sa stanovišta učestalosti izmene zahteva), kritičnost projekta (sa stanovišta mogućih gubitaka usled greške u softveru – od gubitka novca, do gubitka velikog broja ljudskih života), veličina tima, kvalitet članova tima i korporativna kultura (bazirana na haosu ili na disciplini).

### 1.5.1 Agilno modelovanje

Na početku agilnog pokreta, modelovanje je smatrano pre kao balast, nego kao pomoć u razvoju, mada je zvanični stav bio „koristite ga ako vam može biti od pomoći“. Pojavom metodologije pod nazivom *Agilno modelovanje* (AM), čiji je cilj bio razvoj tehnika za realizaciju modela i dokumentacije u agilnom maniru, modelovanje je šire prihvaćeno u agilnim timovima. Autor AM-a je Skot Ambler (Scott Ambler) [Ambler02].

Preporuka AM-a je da sesije za razvoj modela traju što je moguće kraće. Tokom početne faze projekta dozvoljeno je potrošiti od nekoliko sati do nekoliko dana (gornja granica je dve nedelje) za prikupljanje osnovnih korisničkih zahteva i kreiranje kostura arhitekture. Smatra se da sve preko ovoga dovodi projekat u opasnost zbog nedostatka povratnih informacija na nešto što radi.

U okviru implementacione faze, preporuka je da sesije za modelovanje traju od 10 do 20 minuta, koliko je potrebno da se skicira model nekog segmenta

---

<sup>6</sup>Čuvena je izjava: „...bubbles and arrows, as opposed to programs, ...never crash“ - B. Meyer „UML: The Positive Spin“, American Programmer, 1997.

problema, posle čega se odmah prelazi na njegovu implementaciju. Implementacija obično traje od nekoliko sati do nekoliko dana.

U okviru sesija za modelovanje učestvuju članovi tima i korisnici koji obezbeđuju informacije. Preporuka je da ne postoje ekskluzivni specijalisti (softverske arhitekture) čiji je posao isključivo izrada modela, već da svi učestvuju i u modelovanju i u realizaciji programskog koda.

Osnovni principi AM-a su:

- Modelovati sa svrhom, tj. ne kreirati modele radi njih samih. Softver koji zadovoljava potrebe korisnika je primarni cilj projekta, a ne „lepi“ modeli i dokumentacija. Kreirati onoliko modela i dokumentacije koliko je minimalno potrebno za efikasan i pouzdan rad.
- Razvijati sistem inkrementalno, po jedan manji deo u jedinici vremena, umesto pokušati da se ceo sistem implementira odjednom.
- Koristiti različite vrste modela - softver je suviše kompleksan da bi samo jedna vrsta modela u potpunosti mogla da ga opiše.
- Sadržaj modela je mnogo bitniji od njegove reprezentacije - nisu uvek neophodni alati za modelovanje, često su dovoljni papir i olovka ili tabla i flomaster, radi skiciranja ideje. Koristiti najjednostavnije moguće alate za modelovanje koji odgovaraju datoj primeni.
- Dobro poznavati dobre i loše strane različitih vrsta modela i koristiti odgovarajuće vrste modela za određenu namenu (nisu sve vrste modela jednako dobre za sve primene). Preći na drugu vrstu modela kada se primeti da sa tekućom vrstom postoji problem da se opiše neki aspekt sistema.
- Paralelno realizovati više vrsta modela koji ilustruju različite aspekte sistema (strukturne, dinamičke i sl).
- Kreirati jednostavna rešenja, odnosno uzdržavati se od modelovanja dodatnih aspekata sistema do trenutka kada su zaista potrebni.
- Kreirati svedene modela, odnosno koristiti podskup raspoloživih notacija radi realizacije jednostavnih dijagrama koji ilustruju samo ključne aspekte sistema koje je potrebno razumeti. Nisu svi upoznati sa svakim detaljem UML-a.
- Modelovati u malim inkrementima - malo modelovati, pa zatim to programirati, testirati i prikazati korisniku radi dobijanja povratnih informacija. Ispravnost modela se dokazuje programskim kodom.
- Modelovati sa drugima, radi postizanja boljeg rešenja – više očiju bolje vide. Opasno je modelovati sam!



- Usvajati i primenjivati konvencije u modelovanju na nivou celog tima.
- Primenjivati gotove šablone u modelovanju (*design patterns*)<sup>7</sup> samo kada je to neohodno, pošto preterana primena može da uvede kompleksnost u modele i programski kod, koja možda nije neophodna za datu namenu.

## 1.6 Šta smo naučili

Razvoj softvera je kompleksna aktivnost sa velikim procentom neuspeha, posebno kod velikih projekata. Za povećanje verovatnoće pozitivnog ishoda, potrebno je koristiti metodologije, tehnike i alate koji pogoduju datoj vrsti projekta i razvojnom timu.

Modelovanje je centralna aktivnost u većini metodologija. Model je pojednostavljena predstava realnog sistema, kreirana primenom apstrakcija (zanemarivanjem detalja koji nisu bitni u određenom kontekstu).

Prema nameni, modeli mogu biti deskriptivni i preskriptivni. Deskriptivni modeli se koriste radi boljeg razumevanja sistema, dokumentovanja i komunikacije ideja. Preskriptivni modeli se koriste radi specificiranja kako neki sistem treba da se implementira, nebitno da li je u pitanju ručna ili automatizovana implementacija. Modeli istovremeno mogu biti i deskriptivni i preskriptivni.

Modelovanje nije samo sebi cilj - modeli treba da budu u funkciji implementacije softverskog rešenja koje odgovara potrebama korisnika. Ispravnost modela se proverava praćenjem reakcija korisnika na izvršivu aplikaciju koja je implementirana na bazi datog modela.

---

<sup>7</sup> Projektni šabloni ili obrasci (*design patterns*) opisuju kako projektovati rešenja za određene situacije u razvoju softvera koje se često ponavljaju. Nastali su kao rezultat dugogodišnje prakse u razvoju softvera i prvi put opisani u okviru [Gamma04]. Mogu pomoći da i manje iskusni projektanti kreiraju kvalitetne softverske arhitekture, ali, ako se koriste kada to nije neophodno, mogu dovesti do prekomplikovanih rešenja.



## Poglavlje 2

### Dijagram slučajeva korišćenja

Razvoj softverskog projekta u većini metodologija počinje fazom analize zahteva. Cilj ove faze je da se stekne uzajamno razumevanje razvojnog tima i naručioca kako funkcionise realni sistem i šta se od softverskog sistema, koji treba da ga podrži, očekuje. Zahtevi koji se postavljaju pred sistem mogu biti funkcionalni i nefunkcionalni.

Funkcionalni zahtevi definišu šta sistem treba da radi – koje funkcije (usluge, servise) treba da obezbedi svojim korisnicima i drugim sistemima koji su sa njim u interakciji. Funkcionalni zahtevi se obično prevode na skup opcija aplikacije koje korisnik ili neki drugi sistem može da pokrene i kao rezultat dobije potrebnu uslugu.

Zajedno sa funkcionalnim zahtevima se otkrivaju i beleže i poslovna pravila. Poslovna pravila su ograničenja na funkcionisanje sistema koja nameće poslovni domen za koji se sistem razvija. Prilikom implementacije funkcionalnih zahteva, uvek se moraju uzimati u obzir i poslovna pravila, tako da je njihovo otkrivanje izuzetno važno. Primeri poslovnih pravila za informacioni sistem fakulteta su: ocena studenta može biti od 5 do 10; student mora imati propisani broj bodova da bi mogao da upiše sledeću školsku godinu; u jednom danu se ne sme održavati više od jednog ispita za određenu godinu studija, itd.

Nefunkcionalni zahtevi definišu dodatna ograničenja koja se postavljaju pred sistem. Obično se odnose na bezbednost, otpornost na ispade, performanse, dostupnost, lakoću korišćenja i sl. Aplikacija za podršku rada bioskopa i aplikacija za podršku kontrole leta ili upravljanje nuklearnom elektranom imaju potpuno različite zahteve po pitanju bezbednosti, dostupnosti i otpornosti na ispade, što bitno utiče na arhitekturu sistema i način implementacije.

Za brzo i uspešno otkrivanje potrebnih informacija neophodan je čest kontakt sa korisnicima. Kontakt se može ostvarivati tako da korisnik često boravi na lokaciji razvojnog tima (*user on site*), za šta se zalažu agilne metodologije, ali i obrnuto - da razvojni tim često posećuje radno mesto korisnika. Boravkom na mestu događaja i posmatranjem, stiće se osećaj za atmosferu i navike koje vladaju u datom okruženju. Razvojni tim na taj način može da istinski razume aktivnosti korisnika, njihovu organizaciju i širi kontekst u kome se njihov rad odvija, umesto

da se samo osloni na ono što čuje od njihovih izabranih predstavnika u nekom kontrolisanom ambijentu. Od pomoći je i analiza dokumenata koje različiti nivoi korisnika kreiraju pri obavljanju posla, kao i analiza funkcija postojećeg softverskog rešenja, ako je u pitanju reinženjering.

Posebnu pomoć u komunikaciji predstavlja prezentacija (a ako je moguće i zajednički razvoj) prototipova i praćenje reakcija korisnika na njih. Prototip može biti sve što pomaže u komunikaciji, od skice ekrana na tabli ili papiru, do izvršivog prototipa. Prototip može da se kreira samo radi komunikacije ideja i odbaci, kada ispuni svoju funkciju. Ako se prototip pažljivo implementira i nadograđuje dok ne preraste u finalno rešenje, zove se evolutivni prototip.

Postoji više tehnika za prikupljanje, analizu i beleženje zahteva. Tehniku baziranu na slučajevima korišćenja (*use cases*) je osmislio Ivar Jakobson 1994. godine. Ona se primenjuje isključivo na funkcionalne zahteve. Slučaj korišćenja je funkcija ili servis kojim razmatrani sistem obezbeđuje merljivu vrednost nekom od učesnika. Specificira se kao niz akcija koje sistem i učesnik razmenjuju radi ostvarivanja postavljenog zahteva.

Učesnik (akter, *actor*) predstavlja ulogu u funkcionisanju razmatranog sistema. Određenu ulogu mogu imati realni korisnici – osobe, kao i drugi sistemi sa kojima je dati sistem u interakciji. Jedan realni korisnik može imati više uloga.

Skup slučajeva korišćenja predstavlja pogled na sistem sa stanovišta korisnika – koje servise treba da pruži, ne na koji način razvojni tim treba da ih implementira. Prilikom prikupljanja skupa slučajeva korišćenja, obično se preporučuju sledeći koraci:

- odrediti učesnike,
- za svakog učesnika izolovati slučajeve korišćenja koji mu pripadaju i
- detaljno analizirati i opisati svaki slučaj korišćenja.

Razvojni tim kreće od minimalnog skupa slučajeva korišćenja koji modeluju osnovne funkcije sistema i zatim postepeno, kroz dalju analizu i komunikaciju sa naručiocima, proširuje taj skup. Prilikom opisivanja slučajeva korišćenja, obično se otkrivaju nedorečenosti, a nekad i protivrečnosti u inicijalnim zahtevima, pa su potrebne dodatne informacije od naručioca. Tada se otkrivaju novi slučajevi korišćenja i novi učesnici i menjaju postojeći. Obično je potrebno nekoliko iteracija dok se stigne do skupa slučajeva korišćenja na osnovu kojeg može da se krene sa narednim fazama projekta (specifikacija dizajna, implementacija). Agilne metodologije zastupaju stav da analiza zahteva ne mora biti savršena, niti

kompletna, da bi se krenulo sa implementacijom, ali ne treba krenuti ni sa očigledno nepotpunim i nedovoljno dobro shvaćenim zahtevima.

**Napomena:** Ako niste sigurni šta neki zahtev predstavlja ili kako bi sistem trebalo da ga izvrši, obavezno se obratite naručiocu za razjašnjenje. Nikad nemojte raditi na osnovu pretpostavki, ako postoji mogućnost da dobijete tražene informacije! Najskuplje za ispravljanje su greške koje se u kasnijim fazama pojavljuju kao posledica pogrešno shvaćenih ili propuštenih zahteva.

Razmotrimo kao primer sledeću specifikaciju dobijenu od naručioca.

---

**Inicijalna specifikacija aplikacije za testiranje studenata.** *Potrebno je realizovati softver za elektronsko testiranje studenata. Nastavnici kreiraju testove za predmete koje predaju. Testovi se sastoje od pitanja, gde svako pitanje treba da ima više ponuđenih odgovora. Jedno pitanje mora imati minimalno jedan tačan odgovor.*

*Testiranje počinje tako što nastavnik aktivira test. U tom trenutku, svi studenti koji slušaju taj predmet mogu da mu pristupe i počnu rešavanje. Kada istekne vreme za rešavanje testa, sistem treba da blokira dalje rešavanje, izvrši ocenjivanje testova i pošalje rezultate studentima i nastavniku na e-mail. Student može da pogleda svoj rešeni test (kao i sve prethodne rešene testove) i vidi gde je pogrešio. Tačni odgovori treba da budu označeni drugom bojom.*

---

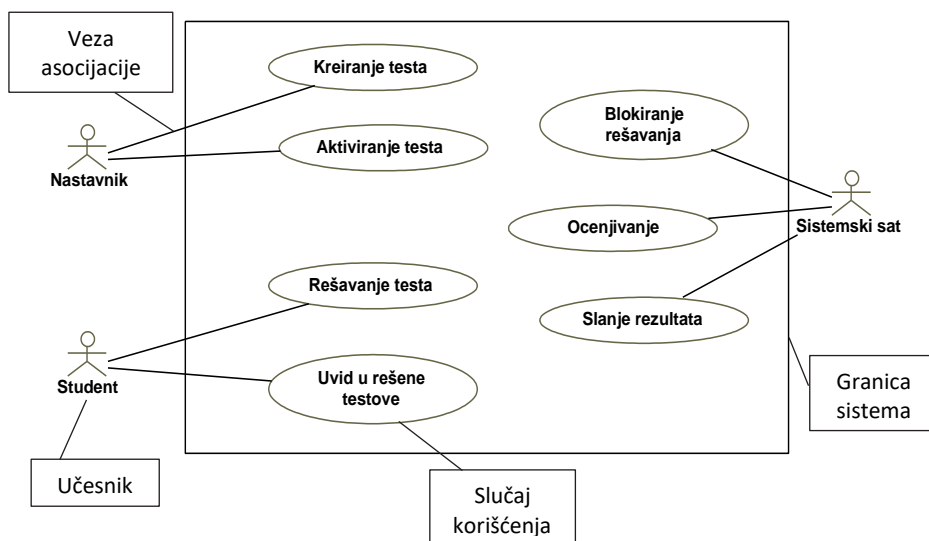
Iz navedene specifikacije možemo odmah da primetimo da su učesnici Nastavnik i Student. Kandidati za slučajeve korišćenja koje aktivira Nastavnik su Kreiranje testa i Aktiviranje testa, a koje aktivira Student su Rešavanje testa i Uvid u rešeni test (kandidati su zato što će detaljnija analiza možda pokazati da su neki od njih prekompleksni i da ih treba zameniti grupom jednostavnijih slučajeva korišćenja).

Pored navedenih kandidata, postoje i sledeći: Blokiranje rešavanja, Ocenjivanje testova i Slanje rezultata. U tekstu specifikacije piše da njih pokreće sistem: „Kada istekne vreme za rešavanje testa, sistem treba da blokira dalje rešavanje, izvrši ocenjivanje testova i pošalje rezultate studentima i nastavniku na e-mail“.

Pošto su učesnici uloge korisnika sistema (osobe ili drugi sistemi), u većini literature se preporučuje da se ne uvodi učesnik koji se zove Sistem, već da se umesto njega koristi učesnik sa nazivom Sistemski sat (*Timer*). Taj naziv je i korektniji, pošto se slučajevi korišćenja koje sistem „samostalno“ pokreće po pravilu aktiviraju od strane sistemskog sata. Primeri su: pokretanje različitih

složenih obrada u noćnim satima, kreiranje sigurnosnih kopija u tačno određenom danu i satu u sedmici i sl.

Do sada izolovani učesnici i kandidati za slučajeve korišćenja su prikazani na UML dijagramu na slici 2.1. Notacija je projektovana sa namerom da bude jednostavna i intuitivna, tako da modelovanje može da se vrši sa predstavnicima korisnika. Na slici 2.1 vidimo simbole za učesnika (stilizovana silueta osobe), slučaj korišćenja (elipsa), vezu asocijacije koja povezuje učesnika sa slučajem korišćenja i granicu sistema (pravougaonik koji obuhvata sve slučajeve korišćenja koje sistem treba da obezbedi).



Slika 2.1 UML dijagram slučajeva korišćenja za sistem za elektronsko ocenjivanje studenta. Prikazuje učesnike i slučajeve korišćenja izolovane na osnovu inicijalne specifikacije.

Granica sistema ne mora da se crta, osim ako želimo da navedemo i neke slučajeve korišćenja koje sistem neće podržavati. Na primer, mogli smo da modelujemo Uvid u rezultate testa izvan granica sistema, pošto korisnici rezultate dobijaju na e-mail i pristupaju im putem svoje e-mail aplikacije. Takođe, izvan granica se mogu crtati i slučajevi korišćenja koji su ostavljeni za neku buduću implementaciju, ali trenutno nisu planirani da budu deo sistema.

UML dijagram služi za brz uvid u korisničke uloge i funkcije (usluge, servise) koje sistem treba da im obezbedi, ali ne daje potrebne detalje za naredne faze razvoja. Kompletan specifikacija mora da se obezbedi u okviru tekstualnih dokumenata. Pre pojave UML-a, slučajevi korišćenja su imali samo pomenutu

tekstualnu specifikaciju. Opis ovih dokumenata će biti prezentovan u narednim sekcijama.

## 2.1 Učesnici

Učesnik modeluje korisničku ulogu, nekoga ko traži uslugu od sistema. U određenoj ulozi može biti osoba ili drugi sistem sa kojima je razmatrani sistem u interakciji. Učesnik može biti i sistemski sat (*timer*) ili hardverski uređaj koji, slanjem signala, aktivira izvršavanje neke funkcije. Bitno je da je učesnik neko ili nešto *izvan* datog sistema i da je on taj koji pokreće izvršavanje željene funkcije.

Zbog različitih problema koji mogu nastati prilikom analize zahteva, naručioci nam prilikom inicijalne specifikacije zahteva obično neće artikulirati sve učesnike, pa je potrebna detaljnija analiza da bismo došli do njih. Pri određivanju učesnika, mogu pomoći sledeća pitanja i njihove varijacije [Ambler04]:

1. Ko su glavni korisnici razmatranog sistema?
2. Kome su potrebne informacije koje se mogu dobiti od sistema?
3. Ko obezbeđuje informacije potrebne za rad sistema?
4. Ko ažurira (unos, menja i briše) podatke?
5. Ko pokreće sistem?
6. Ko gasi sistem?
7. Ko podržava funkcionisanje sistema?
8. Da li postoje sistemi sa kojima je dati u interakciji?
9. Da li se nešto dešava u tačno određeno vreme?

U razgovoru sa naručiocima se ova pitanja mogu prilagoditi njihovom konkretnom poslovnom domenu i sistemu koji se projektuje. Pri komunikaciji je bitno ne koristiti računarsku terminologiju, osim ako znamo da osoba koja je predstavnik korisnika ima tehničko znanje. Ova pitanja se ne moraju postavljati redom, već se može preći na otkrivanje slučajeva korišćenja za svakog novog učesnika, čim ga primetimo.

Pitanja koja mogu pomoći da se otkriju slučajevi korišćenja za nekog učesnika su: [Ambler04]

1. Koji su osnovni zadaci ovog učesnika?
2. Šta mu je potrebno da bi mogao da radi svoj posao?
3. Koje podatke treba da vidi, unosi, menja i briše?
4. Da li sistem treba da šalje neke informacije ili obaveštenja ovom učesniku?

Takođe, mogu se uključiti i dodatna pitanja, u zavisnosti od situacije.

Razmotrimo sledeći razgovor sa naručiocima, pošto smo razumeli njihovu inicijalnu specifikaciju i otkrili osnovne učesnike i slučajeve korišćenja na slici 2.1.

- Da li još neko treba da koristi sistem za elektronsko ocenjivanje, osim studenata i nastavnika?

*Da, asistenti.*

- Da li asistenti mogu da rade sve što i nastavnici?

*Ne, oni ne smeju da menjaju testove, ali mogu da aktiviraju test da bi studenti počeli sa rešavanjem. Oni treba da dobiju rezultate, kao i nastavnici.*

- Da li su još nekom, osim studentima, asistentima i nastavnicima, potrebne informacije koje se mogu dobiti od ovog sistema?

*Ne.*

- Kako dobijamo podatke o studentima, nastavnicima i predmetima koje nastavnici predaju?

*Ove podatke dobijamo od studentske službe.*

- U kom obliku dobijate podatke? Na papiru, u elektronskom formatu, na neki drugi način?

*Dobijamo ih u elektronskom formatu. Treba obezbediti učitavanje tih podataka u sistem za elektronsko ocenjivanje.*

- Da li studentska služba automatski šalje podatke u neko određeno vreme ili ih dobijate na zahtev?

*Na zahtev.*

- Ko treba da aktivira učitavanje?

*Neko ko se bavi održavanjem sistema.*

- Da li je to neko od nastavnika?

*Ne. To je zaposleni iz Službe za tehničku podršku.*

- Da li ta osoba ili neko drugi sme da menja učitane podatke?

*Ne.*

- Ko održava (unos, menja i briše) ostale podatke?

*Nastavnici dodaju, menjaju i brišu podatke o testovima, kao i pitanja i odgovore. Rešeni testovi ne smeju se menjati. Ocene niko ne sme da menjati, pošto sistem treba da vrši automatsko ocenjivanje.*

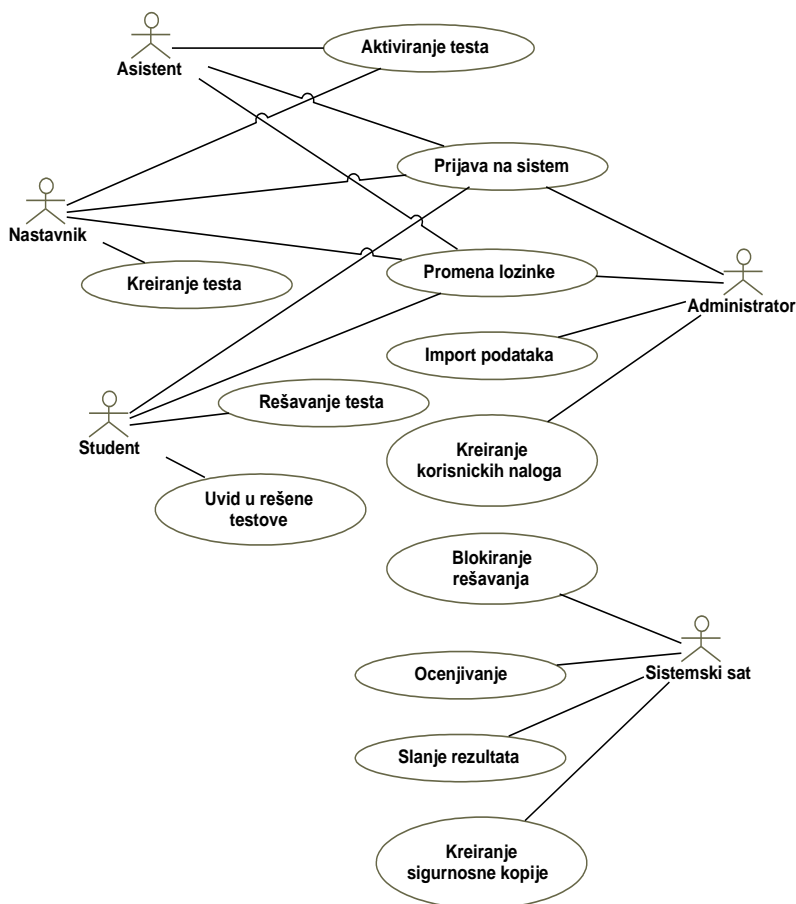


- Da li postoji ograničenje ko može da vidi određene podatke – o nastavnicima, predmetima, studentima, testovima, ocenama?  
*Tehnička podrška može da vidi sve podatke. Nastavnici i asistenti mogu da vide svoje podatke i predmete, kao i testove za te predmete. Studenti mogu da vide svoje podatke, svoje rešene testove, kao i test koji im je dodeljen za rešavanje.*
- Ko pokreće sistem?  
*Studenti i nastavnici treba da pokrenu aplikaciju i prijave se da bi mogli da rade.*
- Da li je vreme kada mogu da se prijave ograničeno?  
*Ne, treba im omogućiti da mogu da se prijave u bilo koje vreme.*
- Da li je lokacija za prijavu ograničena?  
*Za nastavnike nije. Studenti mogu raditi testove samo u predviđenoj učionici.*
- Na koji način bi trebalo da se korisnici prijave?  
*Treba da unesu korisničko ime i lozinku.*
- Ko će im obezbediti korisničko ime i lozinku?  
*Osoba iz tehničke podrške. Svaki korisnik treba da ima mogućnost da promeni svoju lozinku.*
- Ko gasi sistem?  
*Korisnici treba da ugase svoju aplikaciju kada su završili rad.*
- Da li postoje sistemi sa kojima naš sistem treba da sarađuje?  
*Naš sistem treba da dobija podatke od sistema studentske službe.*
- Da li studentskoj službi naš sistem treba da šalje neke podatke (npr. zaključene ocene)?  
*Ne.*
- Da li nešto treba da se dešava u tačno određeno vreme?  
*Da. Po završetku trajanja testa, sistem blokira studentima rad, vrši ocenjivanje i šalje rezultate. Takođe, svakog dana u ponoć treba napraviti sigurnosnu kopiju (backup) podataka.*

Možemo primetiti da smo kombinovali pitanja za otkrivanje učesnika sa pitanjima za otkrivanje slučajeva korišćenja, zahvaljujući čemu su se iskristalisali još neki potencijalni učesnici i slučajevi korišćenja. Postoji Administrator (osoba zaposlena u tehničkoj podršci) koja kreira korisničke naloge i obavlja import

podataka dobijenih od studentske službe. Asistent može da izvršava neke od funkcija nastavnika.

Sve vrste korisnika (student, asistent, nastavnik, administrator) se prijavljuju na sistem korišćenjem svog korisničkog imena i lozinke. Svaki korisnik može da menja svoju lozinku.



Slika 2.2 Dijagram slučajeva korišćenja koji je dobijen posle još jednog razgovora sa naručiocima. Izolovani su dodatni učesnici i slučajevi korišćenja. Čitljivost dijagrama se može popraviti uvođenjem veza generalizacije, kao na slici 2.3.

Sistemska sat pokreće još jednu funkciju – kreiranje sigurnosne kopije.

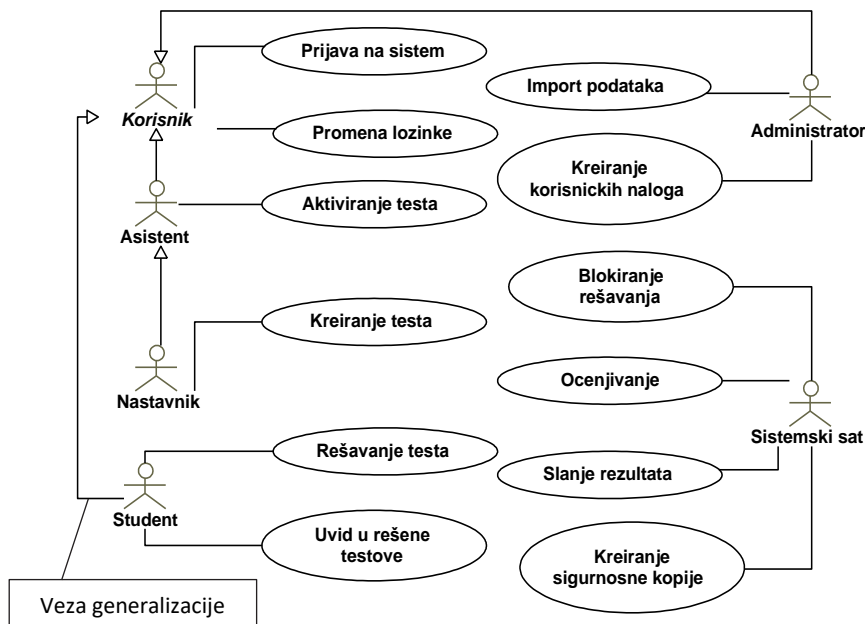
Sistem studentske službe nije učesnik, iako se od njega dobijaju podaci, zato što ne aktivira nijednu funkciju sistema za elektronsko ocenjivanje.

Sve navedeno bi se moglo modelovati kao na slici 2.2. Vidimo da su slučajevi korišćenja povezani sa svim učesnicima koji imaju pravo da ih pokrenu, što je dovelo do puno uskrštenih veza asocijacije i teško čitljivog modela.

Iz dodatne analize su se iskristalisala i dva nefunkcionalna zahteva. Iz zahteva da se korisnici mogu prijaviti u bilo koje doba, proizilazi da serverska komponenta ovog rešenja treba da je stalno dostupna. Prilikom prijave studenata, treba proveriti da li se prijavljuju sa računara predviđenih za testiranje.

### 2.1.1 Nasleđivanje učesnika

Iako je način modelovanja kao na slici 2.2 korektan, model postaje pregledniji i jasniji ako se koriste veze generalizacije (nasleđivanja) između učesnika, kao na slici 2.3. Pravac nasleđivanja određuje smer strelice: Asistent nasleđuje Korisnika, Administrator nasleđuje Korisnika. Nastavnik nasleđuje Asistenta, itd.



Slika 2.3 Dijagram slučajeva korišćenja ekvivalentan dijagramu na slici 2.2. Veze generalizacije (nasleđivanja) su pomogle da se popravi čitljivost modela i da se uvede dodatna informacija: student, asistent, nastavnik i administrator su vrste korisnika, koji imaju mogućnost da se prijave na sistem i promene svoju lozinku.

Veza generalizacije znači da naslednik ima sve slučajeve korišćenja svog pretka,

bez potrebe da sa njima bude direktno povezan. Ukoliko je potrebno, može se koristiti i višestruko nasleđivanje.

Uveden je i apstraktni učesnik koji se zove Korisnik, koji ima pravo da se prijavi na sistem i da promeni svoju lozinku. U UML-u se svi apstraktni elementi modela označavaju tako što im se naziv piše kurzivnim (*italic*) slovima, nezavisno od dijagrama u kojem se nalaze.

**Napomena:** Alati za modelovanje, kao što su Papyrus<sup>8</sup> ili MagicDraw<sup>9</sup> u potpunosti implementiraju UML sintaksu i nazive apstraktnih elemenata pišu kurzivnim slovima. Neki drugi alati, kao što je PowerDesigner<sup>10</sup>, označavaju da je klasa apstraktna dodavanjem ograničenja {abstract} ispod imena elementa modela.

Svi naslednici korisnika su direktno povezani sa slučajevima korišćenja koji su karakteristični samo za njih. Možemo primetiti da za sve navedene naslednike važi pravilo da predstavljaju specijalizaciju korisnika (vrstu korisnika). Ne treba povlačiti vezu nasleđivanja između učesnika koji nisu iste vrste, samo na osnovu činjenice da dele neke slučajeve korišćenja. Na primer, ako bi zahtev bio da sve slučajeve korišćenja koje pokreće Sistemski sat može pokrenuti i Administrator, ne bi trebalo da Administrator nasledi Sistemski sat. Administrator je osoba, vrsta korisnika, ne vrsta sistemskog sata, tako da ovakvo nasleđivanje ne bi bilo korektno. Ovo je česta greška pri modelovanju slučajeva korišćenja.

## 2.2 Slučajevi korišćenja

Slučaj korišćenja opisuje uslugu ili servis sistema koji aktivira neki od učesnika i koji tom učesniku obezbeđuje neku merljivu vrednost. Slučajeve korišćenja treba imenovati tako da se može pretpostaviti kakvu funkciju obavljaju (npr. Slanje narudžbenice, Aktiviranje zahteva, umesto Narudžbenica ili Zahtev). Takođe, treba izbegavati nejasne formulacije (npr. Rad sa narudžbenicom). Ako se ne može naći precizan naziv, moguće je da nije u pitanju pojedinačni slučaj korišćenja već grupa povezanih slučajeva, pa treba tražiti dodatne informacije od naručioca.

---

<sup>8</sup> [www.eclipse.org/papyrus](http://www.eclipse.org/papyrus)

<sup>9</sup> [www.nomagic.com](http://www.nomagic.com)

<sup>10</sup> <https://powerdesigner.biz>

Na slikama 2.1, 2.2 i 2.3 su prikazani različiti kandidati za slučajeve korišćenja, izolovani na osnovu inicijalne specifikacije zahteva naručioca. Da li su zaista u pitanju slučajevi korišćenja zaključićemo na osnovu dalje analize svakog kandidata.

Analiza se sprovodi dodatnom komunikacijom sa naručiocima radi dogovora oko detalja funkcionisanja svakog uočenog slučaja korišćenja i pisanjem dokumenta čije su sekcije date u tabeli 2.1.

Tabela 2.1 Sekcije dokumenta za specifikaciju slučaja korišćenja

Naziv sekcije	Objašnjenje
Identifikator	Jedinstveni identifikator koji pomaže da se slučaj korišćenja referencira od strane drugog slučaja korišćenja (detalji u sekciji 2.3)
Naziv	Naziv slučaja korišćenja
Učesnici	Učesnici koji mogu da aktiviraju dati slučaj korišćenja ili koji učestvuju u njegovom izvršavanju
Opis	Opis slučaja korišćenja koji u par rečenica objašnjava njegovu namenu. Ne mora se uneti ako je namena jasna na osnovu naziva.
Preduslovi ( <i>preconditions</i> )	Uslovi koji moraju biti zadovoljeni da bi se dati slučaj korišćenja mogao pokrenuti. Ova sekcija se popunjava samo ako postoje uslovi za pokretanje koji su dovoljno značajni. Očigledni uslovi se ne navode.
Posledice ( <i>postconditions</i> )	Posledice koje ostavlja izvršavanje slučaja korišćenja. Slično kao kod preduslova, unose se samo netrivialne posledice, ako postoje.
Osnovni tok izvršavanja	Koraci koje izvršavaju učesnik (učesnici) i sistem, da bi se opisalo osnovno ponašanje slučaja korišćenja. Koraci se obično pišu kao niz u kojem se vidi šta je akcija korisnika i koji je odgovor sistema, sve dok se ne stigne do kraja izvršavanja.

Alternativni tok A	Alternativni tokovi se navode u slučaju kada postoji grananje izvršavanja u zavisnosti od nekog uslova.
Alternativni tok B	
...	

Prilikom pisanja dokumenta, treba da odlučimo da li ćemo specificirati esencijalni ili sistemski slučaj korišćenja. Esencijalni slučajevi korišćenja su mnogo jednostavniji i pišu se tako da se specificira ponašanje nezavisno od implementacione platforme, a ako je moguće i od računara - da nije bitno da li korake izvršava softver ili ih ručno sprovodi osoba. Cilj esencijalnih slučajeva korišćenja je da se izoluje srž funkcionisanja realnog sistema koja ostaje nepromenjena u slučaju promene implementacione platforme, kao i tokom automatizacije procesa koji su se do tada odvijali ručno.

Ako zavisnost od računara nije moguće izbeći, esencijalni slučajevi korišćenja se pišu tako da beležimo šta sistem radi kao odgovor na akcije korisnika, bez korišćenja terminologije implementacione platforme - elemenata korisničkog interfejsa vezanih za konkretnu implementacionu platformu i sl.

Sistemski slučajevi korišćenja su mnogo detaljniji i po pravilu imaju više koraka od esencijalnih. Oni se pišu kada je dogovorena implementaciona platforma, a njihove sekcije se tada mogu pisati korišćenjem terminologije specifične za datu platformu. Pri njihovom pisanju se obično sa naručiocem obavlja i osnovni dizajn ekrana (izgled i ponašanje) preko kojih će korisnik aktivirati određene akcije.

U okviru slike 2.4 je prikazan esencijalni slučaj korišćenja za prijavu na sistem.

<b>Identifikator:</b> ESC1
<b>Naziv:</b> Prijava na sistem
<b>Učesnik:</b> Korisnik
<b>Opis:</b> Prijava korisnika na sistem unošenjem korisničkog imena i lozinke.
<b>Preduslovi:</b> Sistem je pokrenut. Prethodni korisnik je odjavljen (prisutna je forma za prijavu na sistem).
<b>Posledice:</b> Korisnik može da koristi funkcije sistema
<b>Osnovni tok izvršavanja:</b> <ol style="list-style-type: none"> <li>1. Korisnik unosi korisničko ime i lozinku</li> <li>2. Sistem proverava da li su korisničko ime i lozinka ispravni</li> </ol>

3. Sistem prikazuje raspoložive opcije za prijavljenog korisnika [Alternativni tok A] 4. Slučaj korišćenja se završava
<b>Alternativni tok A: Neispravno korisničko ime ili lozinka</b>
1. Sistem obaveštava korisnika da su korisničko ime ili lozinka neispravni 2. Korisnik potvrđuje da je video obaveštenje 3. Sistem prikazuje formu za prijavu na sistem 4. Slučaj korišćenja se završava

Slika 2.4 Specifikacija esencijalnog slučaja korišćenja za prijavu na sistem

Primetimo da ne postoji alternativni tok za odustajanje od prijave. Pri pisanju koraka, podrazumeva se da korisnik u svakom trenutku može da odustane, tako da se koraci osnovnog toka time ne opterećuju.

Koraci osnovnog i alternativnih tokova se mogu pisati i u dve kolone, da bi se jasnije razdvojile akcije korisnika i odgovor sistema, ako takav stil više odgovara razvojnog timu (slika 2.5).

<b>Osnovni tok izvršavanja:</b>	
1. Korisnik unosi korisničko ime i lozinku	2. Sistem proverava da li su korisničko ime i lozinka ispravni 3. Sistem prikazuje raspoložive opcije za prijavljenog korisnika [Alternativni tok A] 4. Slučaj korišćenja se završava
<b>Alternativni tok A: Neispravno korisničko ime ili lozinka</b>	
2. Korisnik potvrđuje da je video obaveštenje	1. Sistem obaveštava korisnika da su korisničko ime ili lozinka neispravni 3. Sistem prikazuje formu za prijavu na sistem 4. Slučaj korišćenja se završava

Slika 2.5 Pisanje akcija u dve kolone, radi razdvajanja akcija korisnika i odgovora sistema

Treći način pisanja je da se uopšte ne koriste koraci, već da se u više rečenica objasni šta slučaj korišćenja radi (kao opis, samo sa više detalja).

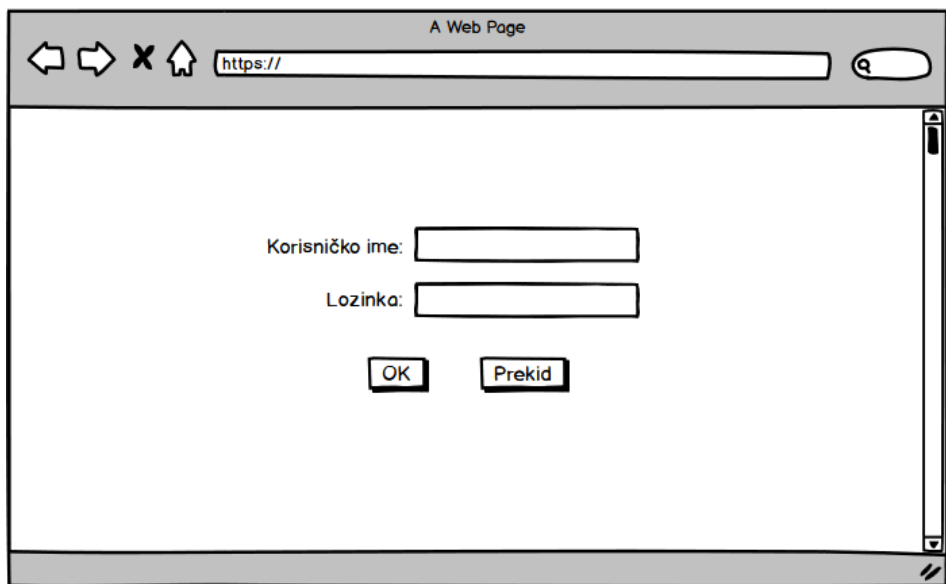
Razmotrimo sada sistemski slučaj korišćenja za prijavu korisnika, pod pretpostavkom da se razvojni tim dogovorio sa naručiocima da je potrebno razviti web aplikaciju. Primetimo da se u sistemskom slučaju koriste termini kao što su stranice i linkovi, kao i pridružena skica ekrana (slika 2.6).

<b>Identifikator:</b> SSC1
<b>Naziv:</b> Prijava na sistem
<b>Učesnik:</b> Korisnik
<b>Opis:</b> Prijava korisnika na sistem unošenjem korisničkog imena i lozinke.
<b>Preduslovi:</b> Sistem je pokrenut. Prethodni korisnik je odjavljen (na ekranu stoji stranica za prijavu na sistem).
<b>Posledice:</b> Korisnik može da koristi funkcije sistema.
<b>Osnovni tok izvršavanja:</b> <ol style="list-style-type: none"> <li>1. Korisnik unosi korisničko ime i lozinku u stranici za prijavu S1</li> <li>2. Korisnik potvrđuje da je obavio unos klikom na „OK“ dugme</li> <li>3. Sistem proverava da li su korisničko ime i lozinka ispravni</li> <li>4. Sistem prikazuje osnovnu stranicu sa raspoloživim linkovima [Alternativni tok A]</li> <li>5. Slučaj korišćenja se završava</li> </ol>
<b>Alternativni tok A: Neispravno korisničko ime ili lozinka</b> <ol style="list-style-type: none"> <li>1. Sistem prikazuje stranicu sa obaveštenjem da su korisničko ime ili lozinka neispravni</li> <li>2. Korisnik potvrđuje da je video obaveštenje klikom na dugme „OK“</li> <li>3. Sistem prikazuje stranicu za prijavu na sistem</li> <li>4. Slučaj korišćenja se završava</li> </ol>

Slika 2.6 Specifikacija sistemskog slučaja korišćenja za prijavu na sistem

Razvojni tim bira da li će prvo pisati esencijalne slučajeve korišćenja, pa onda na bazi njih i sistemske, ili će odmah preći na sistemske. Postojanje esencijalnih slučajeve korišćenja obezbeđuje da se razvojni tim fokusira na suštinu, ali je nekad pravljenje dve verzije specifikacije za jedan slučaj korišćenja previše vremenski zahtevno.





Slika 2.7 Skica S1 – stranica za prijavu na sistem

Prijava na sistem je relativno jednostavna i svima poznata, tako da je specifikacija slučajeva korišćenja mogla da se odradi bez puno interakcije sa naručiocem.

Razmotrimo sada slučaj korišćenja za rešavanje testa. Uslov da bi student mogao da krene sa rešavanjem je da je test aktiviran i da student sluša predmet za koji je test kreiran. Da bismo mogli da specificiramo korake, moramo da obavimo detaljnu analizu u saradnji sa naručiocem. Neka od pitanja na koje treba dobiti odgovor su:

- Kako će student pristupiti testu? Da li će se stranica za rešavanje testa automatski prikazati svim prijavljenim studentima, čim asistent ili nastavnik aktiviraju test? Ako je tako, šta se dešava ako student zakasni na test?
- Da li postoji mogućnost da istovremeno postoji više aktiviranih testova koje student ima pravo da rešava? Ako je odgovor da, šta raditi u takvoj situaciji?
- Kako stranica za rešavanje testa treba da izgleda? Da li sva pitanja sa svojim odgovorima treba da budu na jednoj stranici, ili je potrebno tu stranicu organizovati na neki drugi način?
- Da li student može da označi da je završio rešavanje testa ili mora da čeka da sistem okonča rešavanje? Ako je označio da je završio test, da li može da nastavi rešavanje, ako vreme još nije isteklo?

Primetimo da nam treba da specificiramo slučaj korišćenja pomaže da postavljamo konkretnija pitanja i dobijemo detaljnije informacije koje naručioci obično samostalno ne artikuliraju. Ovo se posebno odnosi na granične slučajeve, koji često ostanu neprepoznati.

Kao rezultat analize ćemo dobiti i skice (prototipe) ekrana kreirane sa naručiocima, pošto od njihovog dizajna zavise i koraci koje treba da unesemo u opis sistemskih slučajeva korišćenja.

Pretpostavimo da je dogovor sa naručiocima sledeći.

---

*Istovremeno može postojati više aktivnih testova – na primer, ako student polaže testove iz više različitih godina studija. Linkovi na sve aktivne testove treba da stoje na stranici koja se pojavljuje posle prijave studenta. Osvežavanjem stranice student može dobiti uvid u sve testove koji su u međuvremenu aktivirani. Student bira koji test želi da rešava. Student može i pre isteka roka da označi da je završio test. Kada je test završen, nije dozvoljeno menjanje odgovora, nebitno na koji način je test završen (od strane studenta ili sistema).*

*Da bi test bio pregledniji i da studenti ne bi prevideli neka pitanja, test treba da bude organizovan kao na slici 2.8. Sva pitanja treba da imaju indikator da li su odgovorena ili ne, da bi se student lakše snašao sa testovima sa velikim brojem pitanja. Izborom linka za davanje odgovora, otvara se forma u kojoj student vidi ponuđene odgovore i može da označi koji su tačni (slika 2.9). Potvrdom da je rešio pitanje, status pitanja na osnovnoj stranici za rešavanje testa se menja u „rešen“, a odgovori se trajno čuvaju. Čuvanje odgovora treba aktivirati čim student potvrdi da je rešio pitanje - ne treba čekati kraj rešavanja celog testa da ne bi došlo do gubitka svih odgovora u slučaju neke nepredviđene situacije (nestanak struje, ispad aplikacije u slučaju greške u sistemu i sl).*

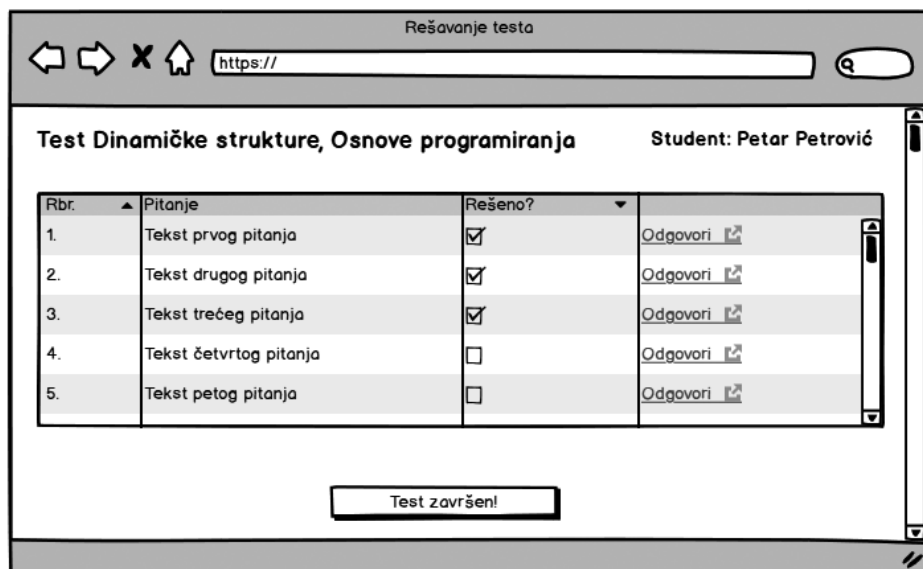
*Student može aktivirati link za davanje odgovora i ako je pitanje rešeno i promeniti date odgovore, ako test nije zaključen. Ako je neko pitanje bilo rešeno i student poništi sve odgovore (na način da nijedan ne ostane izabran), pritiskom na dugme „OK“ se pitanje označava da nije rešeno.*

---

Pri crtanju ekrana, nije neophodno koristiti softverske alate – dovoljno je i skiciranje na tabli ili papiru, da bi proces bio brži. Alati se mogu koristiti da se kasnije nacrtaju skice koje se smeštaju u zvaničnu dokumentaciju, ako je to

potrebno ili ako postoji takav zahtev, od strane naručioca ili onih koji upravljaju projektom<sup>11</sup>.

Sistemska slučaj korišćenja za rešavanje testa je specificiran na slici 2.10. Možemo primetiti da je davanje odgovora na pitanje modelovano u okviru alternativnog toka A. Uslov za prelazak na njega je da je korisnik izabrao link za davanje odgovora na neko pitanje. Ako student odluči da ne želi da odgovara na pitanja, može odmah završiti test, što se vidi iz koraka osnovnog toka na slici 2.10.



Slika 2.8 Skica S2 - stranica za rešavanje testa

Odgovori na pitanje

**3. Ovde se piše redni broj i tekst pitanja**  
☐ prvi odgovor  
☒ drugi odgovor  
☐ treći odgovor  
☒ četvrti odgovor

OK

Prekid

Slika 2.9 Skica S3 - forma za davanje odgovora na pitanje

<sup>11</sup> Za detalje videti <http://agilemodeling.com/artifacts/uiPrototype.htm>

<b>Identifikator:</b> SSC2
<b>Naziv:</b> Rešavanje testa
<b>Učesnik:</b> Student, Sistemski sat
<b>Opis:</b> Student rešava test koji bira iz spiska aktivnih testova. Student aktivira formu za davanje odgovora za svako pitanje na koje želi da odgovori i označava tačne odgovore. Ako je završio ranije, aktivira zaključenje testa, u suprotnom, zaključenje se aktivira od strane sistemskog sata.
<b>Preduslovi:</b> Student je prijavljen. Student sluša predmet za koji se test organizuje. Test je aktiviran.
<b>Posledice:</b> Svi odgovori studenta su trajno sačuvani. Rešavanje testa (davanje i menjanje odgovora) je blokirano.
<b>Osnovni tok izvršavanja:</b> <ol style="list-style-type: none"> <li>1. Student aktivira osvežavanje osnovne stranice</li> <li>2. Sistem prikazuje sve aktivirane testove</li> <li>3. Student bira test koji treba da rešava</li> <li>4. Sistem prikazuje test studentu (skica S2).</li> <li>5. [Alternativni tok A]</li> <li>6. Student označava da je završio test izborom dugmeta „Test završen!“ [Alternativni tok C]</li> <li>7. Sistem pita studenta da li zaista želi da završi test</li> <li>8. Student potvrđuje</li> <li>9. Slučaj korišćenja se završava</li> </ol>
<b>Alternativni tok A: Student želi da aktivira formu za davanje odgovora</b> <ol style="list-style-type: none"> <li>1. Student aktivira link za davanje odgovora na pitanje</li> <li>2. Sistem prikazuje formu za davanje odgovora (skica S3)</li> <li>3. Student označava jedan ili više odgovora za koje misli da su tačni [Alternativni tok B]</li> <li>4. Student potvrđuje da je odgovorio na pitanje, klikom na dugme „OK“ na formi za davanje odgovora</li> <li>5. Sistem zatvara formu</li> <li>6. Sistem postavlja indikator na stranici S2 da je pitanje rešeno</li> <li>7. Davanje odgovora se završava</li> </ol>

**Alternativni tok B: Student želi da poništi date odgovore**

1. Student poništava sve ranije date odgovore
2. Student potvrđuje svoju akciju klikom na dugme „OK“ na formi za davanje odgovora
3. Sistem zatvara formu
4. Sistem postavlja indikator na stranici S2 da pitanje nije rešeno
5. Davanje odgovora se završava

**Alternativni tok C: Isteklo vreme**

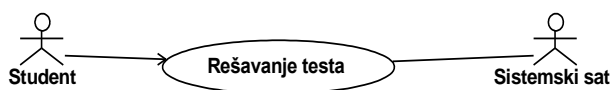
1. Sistemski sat aktivira zaključenje testa
2. Slučaj korišćenja se završava

Slika 2.10 Sistemski slučaj korišćenja za rešavanje testa

### 2.2.1 Primarni i sekundarni učesnici

U okviru alternativnog toka A na slici 2.10. vidimo da korak zaključenja testa izvršava učesnik Sistemski sat. Ako želimo da prikažemo da jedan slučaj korišćenja ima više učesnika od kojih svaki izvršava određene korake, možemo to prikazati kao na slici 2.11. Strelica na vezi asocijacije označava primarnog učesnika (onog koji aktivira slučaj korišćenja). Veza asocijacije bez strelice označava sekundarnog (pomoćnog) učesnika.

Ako je jedan slučaj korišćenja vezan za više učesnika, kao na slici 2.2, a veze nemaju strelice, ili sve imaju strelice, podrazumeva se da je svaki od njih primarni učesnik.



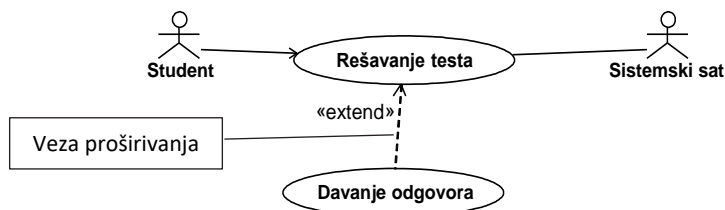
Slika 2.11 Primarni i sekundarni učesnik. Primarni učesnik ima strelicu na vezi asocijacije sa slučajem korišćenja.

### 2.3 Veze između slučajeva korišćenja

Prilikom detaljne analize slučajeva korišćenja, možemo otkriti i veze između njih. Moguće veze su: proširivanje (*extend*), generalizacija i uključivanje (*include*). One su obrađene u nastavku sekcije.

### 2.3.1 Veza proširivanja (extend)

Pogledajmo ponovo slučaj korišćenja za rešavanje testa sa slike 2.10. Možemo primetiti da osnovni tok i alternativni tok A opisuju dve zaokružene funkcionalnosti, za manipulaciju testom i davanje odgovora na pitanje, tim redosledom. Svaka funkcionalnost se odvija u okviru posebne ekranske forme i ima svoj alternativni tok. Davanje odgovora se može izvršiti nijednom, jednom ili više puta, u zavisnosti od toga koliko puta student izabere link koji aktivira ovu funkciju.



Slika 2.12 Funkcionalnost za davanje odgovora je izdvojena u poseban slučaj korišćenja koji proširuje osnovnu funkcionalnost za rešavanje testa

Iz navedenih razloga, rešavanje testa može da se modeluje kao na slici 2.12, gde su osnovni i alternativni tok razdvojeni u dva slučaja korišćenja, povezana vezom proširivanja. Slučaj korišćenja Davanje odgovora proširuje slučaj korišćenja Rešavanje testa (obratiti pažnju na smer veze!). Veza proširivanja se koristi kada se, prilikom izvršavanja osnovnog slučaja korišćenja, može izvršiti dodatni slučaj korišćenja, ako je zadovoljen određeni uslov. Ako je uslov zadovoljen, izvršavaju se i koraci osnovnog i koraci slučaja korišćenja koji ga proširuje. Ako uslov nije zadovoljen, izvršavaju se samo koraci osnovnog slučaja korišćenja. Uslov se navodi u tački proširenja (*extension point*) ili u preduslovima slučaja korišćenja koji treba da proširi osnovni.

Specifikacija navedenih slučajeva korišćenja je data na slikama 2.13 i 2.14. Tačka proširivanja je navedena u koraku 5 na slici 2.13. Uslov za proširivanje je naveden u preduslovima slučaja korišćenja na slici 2.14.

<b>Identifikator:</b> SSC2-2
<b>Naziv:</b> Rešavanje testa
<b>Učesnik:</b> Student, Sistemski sat
<b>Opis:</b> Student rešava test koji bira iz spiska aktivnih testova. Student aktivira formu za davanje odgovora za svako pitanje na koje želi da odgovori (SSC3

Davanje odgovora). Ako je završio ranije, označava da želi da završi test. U suprotnom, zaključenje se aktivira od strane sistemskog sata.
<b>Preduslovi:</b> Student je prijavljen. Student sluša predmet za koji se test organizuje. Test je aktiviran.
<b>Posledice:</b> Rešavanje testa (davanje i menjanje odgovora) je blokirano.
<b>Osnovni tok izvršavanja:</b> <ol style="list-style-type: none"> <li>1. Student aktivira osvežavanje osnovne stranice</li> <li>2. Sistem prikazuje sve aktivirane testove</li> <li>3. Student bira test koji želi da rešava</li> <li>4. Sistem prikazuje test studentu (skica S2).</li> <li>5. [Tačka proširenja: SSC3 Davanje odgovora]</li> <li>6. Student označava da je završio test izborom dugmeta „Test završen!“ [Alternativni tok A]</li> <li>7. Sistem pita studenta da li zaista želi da završi test</li> <li>8. Student potvrđuje</li> <li>9. Slučaj korišćenja se završava</li> </ol>
<b>Alternativni tok A: Isteklo vreme</b> <ol style="list-style-type: none"> <li>3. Sistemski sat aktivira zaključenje testa</li> <li>4. Slučaj korišćenja se završava</li> </ol>

Slika 2.13 Druga verzija sistemskog slučaja korišćenja Rešavanje testa je napisana tako da ga proširuje sistemski slučaj korišćenja Davanje odgovora (videti korak 5)

<b>Identifikator:</b> SSC3
<b>Naziv:</b> Davanje odgovora
<b>Učesnik:</b> Student
<b>Opis:</b> Davanje odgovora na jedno pitanje u okviru testa (stranica S2).
<b>Preduslovi:</b> Stranica za rešavanje testa je aktivna. Student je aktivirao link za davanje odgovora.
<b>Posledice:</b> Odgovori su trajno sačuvani.

**Osnovni tok:**

1. Student aktivira link za davanje odgovora na pitanje
2. Sistem prikazuje formu za davanje odgovora (skica S3)
3. Student označava jedan ili više odgovora za koje misli da su tačni [Alternativni tok A]
4. Student potvrđuje da je odgovorio na pitanje, klikom na dugme „OK“ na formi za davanje odgovora
5. Sistem čuva sve odgovore
6. Sistem zatvara formu
7. Sistem postavlja indikator na stranici S2 da je pitanje rešeno
8. Slučaj korišćenja se završava

**Alternativni tok A: Student želi da poništi date odgovore**

1. Student poništava sve ranije date odgovore
2. Student potvrđuje svoju akciju klikom na dugme „OK“ na formi za davanje odgovora
3. Sistem briše prethodno sačuvane odgovore
4. Sistem zatvara formu
5. Sistem postavlja indikator na stranici S2 da pitanje nije rešeno
6. Slučaj korišćenja se završava

Slika 2.14 Sistemski slučaj korišćenja za davanje odgovora koji proširuje slučaj korišćenja Rešavanje testa sa slike 2.13.

**Napomena:** Obratiti pažnju da je veza proširivanja sa slike 2.12 usmerena prema osnovnom slučaju korišćenja, koji je proširen dodatnom funkcionalnošću. Česta je greška da se obrne smer veze!

Treba napomenuti da su oba načina specifikacije korektna (na slici 2.10 kao i na slikama 2.12, 2.13 i 2.14). U konkretnom slučaju rešavanja testa, koji način specifikacije ćemo koristiti, zavisi od toga da li želimo da na dijagramu prikazemo više ili manje detalja.

Međutim, ako dokument specifikacije nekog slučaja korišćenja postane jako kompleksan, sa puno alternativnih tokova, tada se preporučuje da se pojedini tokovi izdvoje u posebne slučajeve korišćenja i povežu vezama proširivanja sa osnovnim slučajem.

Slučajevi korišćenja koji proširuju osnovno ponašanje se obično ne povezuju sa učesnikom (videti sliku 2.12). Autori poput [Rumbaugh05] tvrde da ne bi ni smeli da se povezuju, pošto su oni često samo fragmenti neke funkcionalnosti, ne i



samostalni slučajevi korišćenja. Osnovni slučaj korišćenja bi morao da bude celovit (da opisuje neku zaokruženu funkcionalnost) i samostalan, nezavisno od toga da li se aktiviraju ili ne slučajevi koji ga proširuju.

### 2.3.2 Veza generalizacije (nasleđivanja)

Ramotrimo slučaj korišćenja Slanje rezultata. U dogovoru sa naručiocem, studenti bi na e-mail trebalo da dobiju samo obeveštenje o svojim rezultatima, dok bi nastavnik i asistent trebalo da dobiju obaveštenje o rezultatima za sve studente koji slušaju predmet za koji je test organizovan. Ako student nije polagao test, ne bi trebalo da dobije obaveštenje. Nastavnik i asistent bi trebalo da dobiju rezultate za sve studente koji slušaju predmet, a ako student nije polagao test, umesto broja poena u poruci bi trebalo da stoji crtica „-“. Tokom analize, dogovoren je i sadržaj poruka koje student i nastavnik treba da dobiju (slike 2.15 i 2.16).

---

Poštovani <ime i prezime>,

Šaljem Vam rezultate testa <ime testa> iz predmeta <ime predmeta>.

Osvojili ste <broj poena> od ukupno <ukupan broj poena> poena.

Srdačan pozdrav!

*Napomena: Ova poruka je automatski generisana od strane Sistema za elektronsko ocenjivanje. Molim Vas, nemojte odgovarati na ovu poruku.*

---

Slika 2.15 Dokument D1 - sadržaj poruke za studenta o rezultatu koji je ostvario na testu

---

Poštovani <ime i prezime>,

Šaljem Vam rezultate testa <ime testa> iz predmeta <ime predmeta>.

Rbr.	Broj indeksa	Ime i prezime studenta	Broj poena
<rbr>	<broj indeksa>	<ime i prezime studenta>	<broj poena> ili „-“

Srdačan pozdrav!

*Napomena: Ova poruka je automatski generisana od strane Sistema za elektronsko ocenjivanje. Molim Vas, nemojte odgovarati na ovu poruku.*

---

Slika 2.16 Dokument D2 - sadržaj poruke za nastavnika i asistenta o rezultatima testa

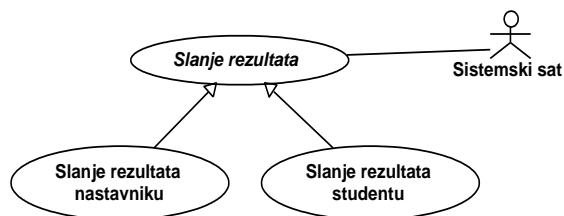
Slučaj korišćenja Slanje rezultata bi mogao da se opiše kao na slici 2.17.

<b>Identifikator:</b> SSC4
<b>Naziv:</b> Slanje rezultata
<b>Učesnik:</b> Sistemski sat
<b>Opis:</b> Slanje rezultata testa nastavniku, asistentu ili studentu na e-mail. Rezultate treba formatirati kako je navedeno u dokumentima D1 i D2.
<b>Preduslovi:</b> Vreme izvršavanja testa je isteklo i test je zaključen i ocenjen. Nastavnik ili asistent izvodi predmet za koji je test organizovan. Student sluša predmet za koji je test organizovan i polagao je test.
<b>Posledice:</b> Rezultati su poslani na e-mail.
<b>Osnovni tok izvršavanja:</b> <ol style="list-style-type: none"> <li>1. Sistem preuzima e-mail adresu osobe kojoj želi da pošalje poruku</li> <li>2. [Aternativni tok A] [Aternativni tok B]</li> <li>3. Sistem šalje poruku na e-mail adresu</li> <li>4. Slučaj korišćenja se završava</li> </ol>
<b>Aternativni tok A: Rezultate treba poslati studentu</b> <ol style="list-style-type: none"> <li>1. Sistem sastavlja poruku kako je opisano u dokumentu D1.</li> </ol>
<b>Aternativni tok B: Rezultate treba poslati asistentu ili nastavniku</b> <ol style="list-style-type: none"> <li>1. Sistem sastavlja poruku kako je opisano u dokumentu D2.</li> </ol>

Slika 2.17 Slučaj korišćenja za slanje rezultata – prva verzija

Ako pogledamo opis, preduslove i korake slučaja korišćenja za Slanje rezultata na slici 2.17, vidimo da imamo varijacije u zavisnosti od uloge osobe kojoj se šalju rezultati. Osnovni tok ima dva alternativna toka koja se tiču formiranja poruke, za studenta i nastavnika.

Drugi način za specifikaciju navedenog slučaja korišćenja je upotrebom generalizacije, odnosno nasleđivanja slučajeva korišćenja. Na slici 2.18 možemo videti da Slanje rezultata nastavniku i Slanje rezultata studentu nasleđuju (specijalizuju) apstraktni slučaj korišćenja Slanje rezultata, čiji je naziv zapisan zakošenim slovima.



Slika 2.18 Slanje rezultata je modelovano kao apstraktni slučaj korišćenja kojeg nasleđuju dva specijalizovana slučaja korišćenja, za slanje rezultata nastavniku i slanje rezultata studentu.

Naslednik može redefinisati bilo koji element pretka: preduslove, posledice, opis, cele tokove kao i pojedinačne korake u okviru tokova. Ako neki deo specifikacije nije neophodno redefinisati kod naslednika, taj deo treba da ostane prazan ili da se čitalac uputi da pogleda odgovarajuću sekciju pretka.

Na slici 2.19 je navedena specifikacija apstraktnog slučaja korišćenja koji opisuje osnovnu funkcionalnost slanja rezultata.

<b>Identifikator:</b> SSC4-2
<b>Naziv:</b> Slanje rezultata, apstraktni slučaj korišćenja
<b>Učesnik:</b> Sistemska sat
<b>Opis:</b> Slanje rezultata testa nastavniku, asistentu ili studentu na e-mail. Pogledati dokumente D1 ili D2.
<b>Preduslovi:</b> Vreme izvršavanja testa je isteklo i test je zaključen i ocenjen.
<b>Posledice:</b> Rezultati su poslani na e-mail.
<b>Osnovni tok izvršavanja:</b> <ol style="list-style-type: none"> <li>1. Sistem preuzima e-mail adresu osobe kojoj želi da pošalje poruku</li> <li>2. Sistem kreira poruku</li> <li>3. Sistem šalje poruku na e-mail adresu</li> <li>4. Slučaj korišćenja se završava</li> </ol>

Slika 2.19 Apstraktni slučaj korišćenja za slanje rezultata – druga verzija

Na slici 2.20 je navedena specijalizacija apstraktnog slučaja korišćenja sa slike 2.19, koja opisuje slanje rezultata nastavniku. Na slici 2.21 je naveden slučaj korišćenja za slanje rezultata studentu.

U oba naslednika je redefinisano samo korak 2 u okviru osnovnog toka. Opis je potpuno redefinisano, a preduslovi su prošireni u odnosu na preduslove pretka.

<b>Identifikator:</b> SSC5
<b>Naziv:</b> Slanje rezultata nastavniku
<b>Učesnik:</b> Kao kod SSC4-2.
<b>Opis:</b> Slanje rezultata testa nastavniku ili asistentu. Pogledati dokument D2.
<b>Preduslovi:</b> Kao kod SSC4-2. Nastavnik ili asistent kome treba poslati poruku predaje predmet za koji je test organizovan.
<b>Posledice:</b> Kao kod SSC4-2.
<b>Osnovni tok izvršavanja:</b> 2. Sistem sastavlja poruku kako je opisano u dokumentu D2.

Slika 2.20 Slučaj korišćenja za slanje rezultata nastavniku – naslednik apstraktnog slučaja korišćenja sa slike 2.19

<b>Identifikator:</b> SSC6
<b>Naziv:</b> Slanje rezultata studentu
<b>Učesnik:</b> Kao kod SSC4-2.
<b>Opis:</b> Slanje rezultata testa studentu na e-mail. Pogledati dokument D1.
<b>Preduslovi:</b> Kao kod kod SSC4-2. Student kome treba poslati poruku sluša predmet za koji je test organizovan. Student je polagao test.
<b>Posledice:</b> Kao kod SSC4-2.
<b>Osnovni tok izvršavanja:</b> 2. Sistem sastavlja poruku kako je opisano u dokumentu D1.

Slika 2.21 Slučaj korišćenja za slanje rezultata studentu – naslednik apstraktnog slučaja korišćenja sa slike 2.19

**Napomena:** Nemojte povezivati vezom generalizacije slučajeve korišćenja koji opisuju raznorodne funkcionalnosti! Ako funkcionalnosti nisu „iste vrste“, iskoristite veze proširivanja, uključivanja ili alternativne tokove u okviru osnovne aktivnosti.

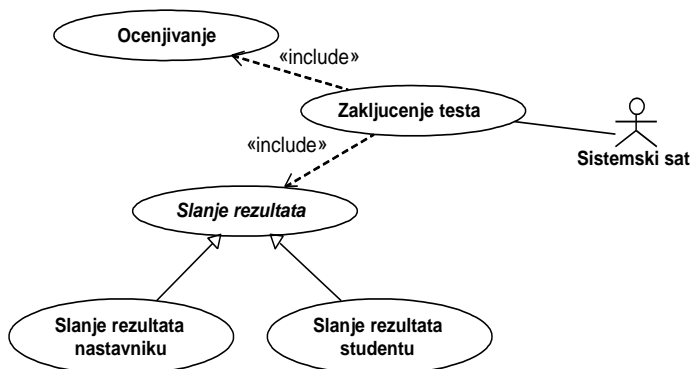
### 2.3.3 Veza uključivanja (*include*)

Na osnovu zahteva naručioca, kada istekne vreme za rešavanje, potrebno je da sistem obavi blokiranje daljeg rešavanja, zatim ocenjivanje testova i na kraju slanje rezultata, tim redom. Ako pogledamo slike 2.2 ili 2.3, možemo primetiti da su navedeni slučajevi korišćenja modelovani nezavisno, tako da se, sa stanovišta sintakse i semantike UML modela, mogu izvršiti bilo kojim redom, proizvoljan broj puta.

Redosled bismo mogli modelovati pažljivo sročnim preduslovima, ali, ako želimo da bude očigledno i sa dijagrama, potrebno je uvesti novi slučaj korišćenja Zaključenje testa, i sa njim, vezama uključivanja, povezati sve slučajeve korišćenja koje treba aktivirati kada se test završi (slika 2.22).

Ukoliko jedan slučaj korišćenja uključuje drugi, mora uvek izvršiti i svoje korake i korake uključenog. Ako uključuje više drugih, sve ih mora izvršiti svaki put kada se aktivira. Mesto uključivanja se navodi u koracima osnovnog slučaja korišćenja, tako da se redosled može precizno definisati.

Prilikom analize sa naručiocima, primećeno je da se Blokiranje testa svodi samo na jedan korak, pa je on direktno unet u osnovni tok za Zaključenje testa (slika 2.23). Ovo je normalna situacija prilikom detaljne analize – neke kandidate za slučajeve korišćenja ćemo brisati, neke razlagati na više drugih slučajeva korišćenja, a neke ćemo otkrivati i dodavati.



Slika 2.22 Zaključenje testa uključuje Ocenjivanje i Slanje rezultata. Blokiranje testa je ukinuto i modelovano kao korak Zaključenja testa. Uključeni slučajevi korišćenja se pokreću tačno jednom, na mestu naznačenom u koracima Zaključenja testa.

Ocenjivanje testa je specificirano na slici 2.24. Slanje rezultata je specificirano u sekciji 2.3.2.

<b>Identifikator:</b> SSC7
<b>Naziv:</b> Zaključenje testa
<b>Učesnik:</b> Sistemski sat
<b>Opis:</b> Kada istekne vreme za rešavanje testa, treba blokirati rešavanje testova koje studenti još nisu završili, oceniti testove i poslati rezultate studentima, nastavnicima i asistentima koji su uključeni u predmet za koji se test organizuje.
<b>Preduslovi:</b> Isteklo je vreme za rešavanje testa
<b>Posledice:</b> Menjanje testova je blokirano, testovi su ocenjeni i ocene poslate.
<b>Osnovni tok izvršavanja:</b> <ol style="list-style-type: none"> <li>1. Sistem označava da su završeni svi testovi koje studenti još rešavaju</li> <li>2. [Uključi: SSC8 Ocenjivanje]</li> <li>3. [Uključi: SSC4-2 Slanje rezultata]</li> <li>4. Slučaj korišćenja se završava</li> </ol>

Slika 2.23 Slučaj korišćenja za zaključenje testa

<b>Identifikator:</b> SSC8
<b>Naziv:</b> Ocenjivanje
<b>Učesnik:</b> Sistemski sat
<b>Opis:</b> Računanje osvojenog broja bodova za sve studente koji su radili test.
<b>Preduslovi:</b> Testovi su završeni.
<b>Posledice:</b> Izračunat je broj bodova za svakog studenta koji je radio test.
<b>Osnovni tok izvršavanja:</b> <p>Za svako pitanje u okviru rešenog testa, sistem sabira poene koje nose odgovori koje je student izabrao. Odgovor može nositi pozitivne poene, ako je tačan, i negativne poene, ako je netačan. Poene za svaki odgovor je uneo nastavnik, prilikom kreiranja testa (videti sliku 2.28). Kada sabere poene svih</p>

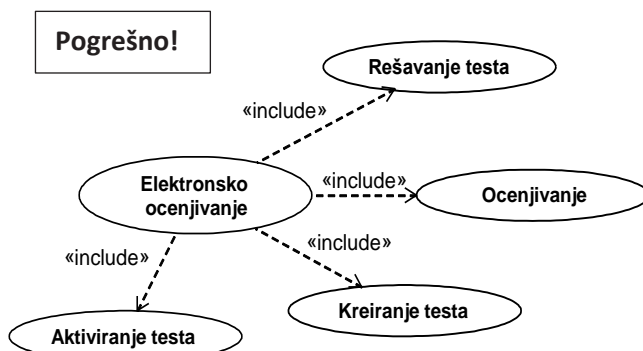
označenih odgovora za sva pitanja, sistem trajno čuva broj poena koje je dati student osvojio na datom testu.

Slika 2.24 Slučaj korišćenja za ocenjivanje testa. Korišćen je treći način specifikacije – opis ponašanja, umesto koraka.

Uključivanje se koristi:

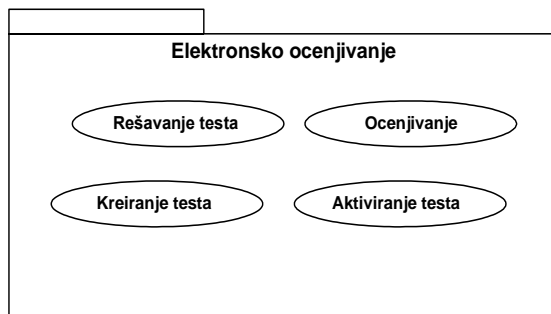
1. kada se prilikom detaljne analize prepozna da se neka funkcionalnost sreće više puta u okviru koraka različitih slučajeva korišćenja, pa se izdvoji i uključi u sve slučajeve korišćenja gde je potrebno (kao kod poziva bibliotečke funkcije u višim programskim jezicima),
2. kada jedan slučaj korišćenja direktno zavisi od rezultata izvršavanja drugog slučaja korišćenja (videli smo primer za zaključenje testa).

Greška je da se uključivanje koristi za grupisanje nezavisnih slučajeva korišćenja ili za dekompoziciju sistema, kao na slici 2.25. Primer je rešenje zadatka sa ispita iz ovog predmeta. Autor je verovatno želeo da prikaže koje funkcije nudi aplikacija za elektronsko ocenjivanje, Međutim, ono što se iz modela može pročitati je da, svaki put kada se pokreće Elektronsko ocenjivanje, ono tačno jednom pozove Rešavanje testa, Ocenjivanje, Kreiranje testa i Ažuriranje testa, što ne odgovara zahtevima. Takođe, Elektronsko ocenjivanje nije slučaj korišćenja, već aplikacija u celini.



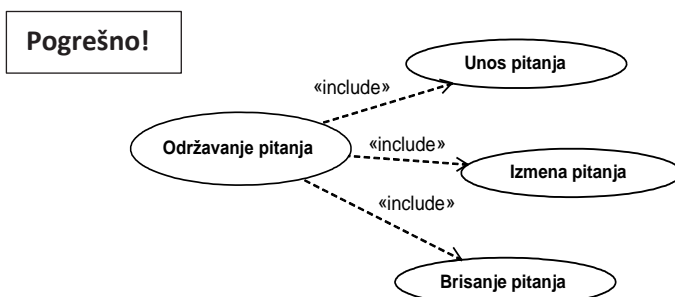
Slika 2.25 Primer **pogrešne** upotrebe veza uključivanja radi prikaza dekompozicije sistema.

Ako želite da prikazete dekompoziciju sistema na manje podsysteme, možete koristiti pakete (slika 2.26). U paketima se mogu nalaziti drugi paketi, slučajevi korišćenja, učesnici i sve vrste veza.



Slika 2.26 Korišćenje paketa za grupisanje funkcionalnosti i dekompoziciju sistema. Paketi mogu sadržati druge pakete, učesnike, slučajeve korišćenja i veze.

Takođe, pogrešno je korišćenje veza uključivanja za modelovanje CRUD (*create, update, delete*) operacija, odnosno unosa, izmene i brisanja određene vrste podataka, kao na slici 2.27. Pod održavanjem podataka se podrazumevaju navedene operacije, ali se „održavanje“ kao takvo ne može pokrenuti, niti bi tada bilo potrebno svaki put izvršiti sve operacije tačno jednom.



Slika 2.27 Primer **pogrešne** upotrebe veza uključivanja radi modelovanja CRUD (*create, update, delete*) operacija

Modelovanje održavanja podataka je detaljnije objašnjeno u narednoj sekciji 2.4.

## 2.4 Modelovanje kompleksnih slučajeva korišćenja

Razmotrimo skicu S4 prikazanu na slici 2.28, koja je nacrtana tokom detaljne analize slučaja korišćenja Kreiranje testa, u saradnji sa naručiocima.



Kreiranje testova

Predmet Osnove programiranja

Nastavnik: Jovan Jovanović

Testovi

Dodaj...
Izmeni...
Obriši

Rbr.	Naziv testa	Ukupno poena
1	Dinamičke strukture	100
2	Rukovanje događajima	100

Pitanja

Dodaj...
Izmeni...
Obriši
Pretraga...

Rbr.	Pitanje
1	Pitanje 1
2	Pitanje 2
3	Pitanje 3
4	Pitanje 4

Odgovori

Dodaj...
Izmeni...
Obriši

Rbr.	Odgovor	Tačan?	Bodovi
1	Odgovor 1	da	3
2	Odgovor 2	ne	-1
3	Odgovor 3	ne	-1

Slika 2.28 Skica S4 – Stranica za kreiranje testa

Skicirana stranica bi trebalo da se koristi na sledeći način. Nastavnik bira predmet za koji želi da kreira test. Odmah po izboru predmeta se u prvoj tabeli prikazuju svi do sada kreirani testovi za taj predmet. Za izabrani test se prikazuju pitanja koja mu pripadaju. Za izabrano pitanje se prikazuju njegovi odgovori. Izbor reda u bilo kojoj tabeli se obavlja klikom miša. Svaka tabela, ako nije prazna, uvek mora imati tačno jedan izabrani red.

Tabele obezbeđuju pregled podataka. Unos, izmena i brisanje podataka se aktiviraju korišćenjem dugmadi smeštenih iznad odgovarajuće tabele. Uneti odgovori se vezuju za izabrano pitanje u tabeli sa pitanjima. Uneto pitanje se vezuje za izabrani test u tabeli testova. Uneti test se vezuje za izabrani predmet.

Izmena i brisanje se odnose na izabrani red u tabeli. Ako je tabela prazna, ova dugmad treba da budu blokirana.

Aktiviranjem dugmeta za dodavanje testa, prikazuje se forma kao na slici 2.29. Obavezna polja su označena znakom „\*” na kraju labele.

Korisnik popunjava tražene podatke i potvrđuje da je unos testa završen klikom na dugme „OK”. Sistem tada proverava podatke unete u poljima. Ako postoji neki problem (obavezno polje nije uneto ili podaci nisu validni iz nekog drugog razloga) korisnik dobija obaveštenje i fokus se postavlja na polje gde postoji problem.

Ako je sve u redu, podaci se trajno snimaju, uneti test se pojavljuje kao novi red u tabeli sa ostalim testovima, polja za unos u formi sa slike 2.29 se prazne od unetih podataka i fokus se postavlja na prvo polje, da bi korisnik mogao da unese još jedan test, ako želi. Ako ne želi, aktiviranjem dugmeta „Prekid” će forma biti zatvorena. Poslednji uneti test treba da bude izabran u tabeli na slici 2.28.

Unos testa

Predmet: Osnove programiranja

Naziv testa\*:

Ukupno poena\*:

OK Prekid

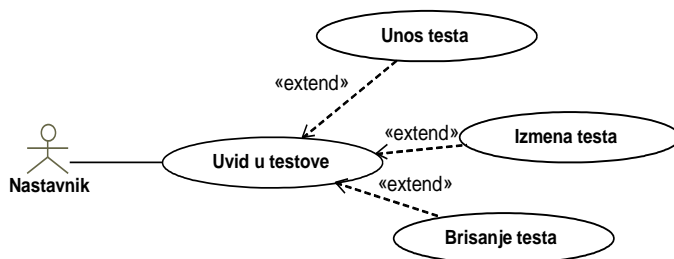
Slika 2.29 Skica S5 – Forma za unos i izmenu testa

Aktiviranjem dugmeta za izmenu se pojavljuje forma sa slike 2.29, popunjena podacima izabranog testa iz tabele testova. Naslov forme treba da bude „Izmena testa”. Korisnik menja podatke i potvrđuje da želi da ih trajno sačuva aktiviranjem dugmeta „OK”, posle čega se forma zatvara, a izmenjeni podaci snimaju na disk.

Ako je korisnik tokom izmene obrisao sadržaj nekog polja koje je obavezno za unos ili uneo nevalidne vrednosti, forma se, kada se aktivira dugme „OK”, ponaša na isti način kao kada postoji neki problem prilikom dodavanja.

Posle aktiviranja dugmeta za brisanje, sistem proverava da li postoje uneta pitanja za izabrani test. Ako postoje, korisnik se obaveštava o tome i brisanje se odbija. Ako pitanja ne postoje, sistem traži da korisnik potvrdi da zaista želi da obavi brisanje. Posle potvrde, izabrani red se briše iz tabele i sa diska.

Primećujemo da su uvid u testove, unos, izmena i brisanje testova slučajevi korišćenja, koji imaju svoje preduslove, posledice, osnovne i alternativne tokove. Njihov odnos je prikazan na slici 2.30.



Slika 2.30 Modelovanje osnovnih operacija za manipulaciju testovima (pregled, unos, izmena, brisanje)

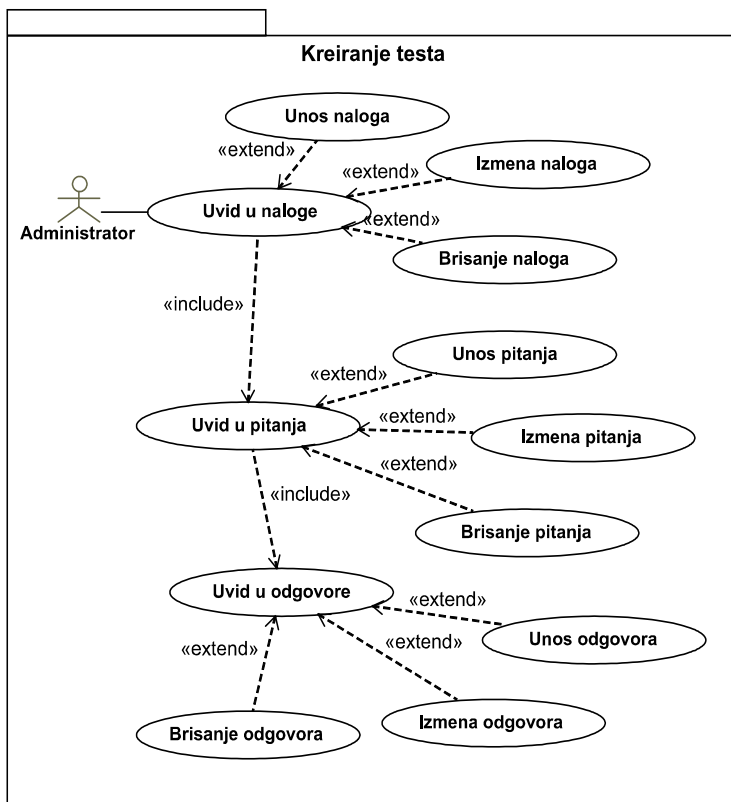
Na sličan način može da se modeluje uvid u podatke, unos, izmena i brisanje za pitanja i odgovore. Forme koje se pojavljuju prilikom unosa i izmene treba da imaju isti dizajn i ponašanje kao forma na slici 2.29, samo sa poljima za unos prilagođenim njihovim podacima. Kreiranje testa u celini bi moglo da se modeluje kao na slici 2.31.

U slučaju aplikacije sa velikim brojem klasa podataka, za svaku klasu bi trebalo obezbediti po četiri slučaja korišćenja, što bi dovelo do modela sastavljenog od stotina ili hiljada slučajeva korišćenja, pri čemu svi opisuju u osnovi istu funkcionalnost. Cilj analize zahteva je da se postigne uzajamno razumevanje razvojnog tima i naručioca, međutim, preobimni modeli tome ne doprinose, iako su ispravni sa stanovišta sintakse i semantike UML-a.

Da bi se izbeglo iscrpljivanje razvojnog tima i kreiranje nepreglednih modela, postoji više tehnika koje su navedene u nastavku.

### 2.4.1 Standardizacija korisničkog interfejsa

Standardizacija korisničkog interfejsa podrazumeva da se sa naručiocima izoluje, nacрта i specifikira ponašanje nekoliko tipova formi (panela, stranica, ekrana) koji će biti osnovni gradivni blokovi aplikacije. Primer za to je panel za rad sa jednom klasom podataka (tabela za pregled, sa pridruženim dugmadima za unos, izmenu i brisanje) i formom koja se pojavljuje kada se aktivira unos ili izmena. Takođe, tu je i forma za rad sa više hijerarhijski organizovanih klasa podataka (*parent-child* forma), prikazana na slici 2.28.



Slika 2.31 Modelovanje kreiranja testa, u skladu sa opisom ponašanja stranice sa slike 2.28. Iako korektan, ovakav način modelovanja se ne preporučuje, jer dovodi do pretrpanog i nečitkog modela.

Kada su svi upoznati kako određeni tipovi formi funkcionišu, dovoljno je nacrtati skicu i navesti je u slučaju korišćenja, kao na slici 2.32. U preduslovima i posledicama navodimo sve što je bitno za konkretne klase podataka, a što nije već navedeno prilikom opisa datog tipa forme.

<b>Identifikator:</b> SSC7
<b>Naziv:</b> Kreiranje testa
<b>Učesnik:</b> Nastavnik
<b>Opis:</b> Kreiranje, izmena i brisanje testova, kao i njihovih pitanja i odgovora.
<b>Preduslovi:</b> Nastavnik predaje predmet za koji želi da kreira test. Test, pitanja i odgovore nije moguće menjati ako je test rešen od strane studenata ili je

rešavanje u toku. Test nije moguće brisati ako su mu uneta pitanja. Prilikom brisanja pitanja, treba obrisati i njegove odgovore, ako postoje.

**Posledice:** Test je trajno sačuvan, sa svojim pitanjima i odgovorima.

**Osnovni tok izvršavanja:**

Pogledati skicu S4 (*parent-child* forma sastavljena od CRUD panela za manipulaciju testovima, pitanjima i odgovorima).

Slika 2.32 Slučaj korišćenja za kreiranje testa – bolji način u odnosu na primer prikazan na slici 2.31.

Složeni slučaj korišćenja, jezgrovito opisan i sa pridruženom skicom, jasniji je i efektniji od modela na slici 2.31, iz kojeg bi, bez naslova paketa, bilo teško zaključiti o čemu se radi.

Osim toga, unapred dogovorena specifikacija izgleda i ponašanja gradivnih blokova korisničkog interfejsa je bitna radi postizanja konzistentnog ponašanja aplikacije, što skraćuje vreme učenja korisnika kada dobiju gotovu aplikaciju, povećava im efikasnost i smanjuje broj grešaka u toku korišćenja. Sa stanovišta projeknog tima, pojednostavljuje se i ubrzava razvoj – smanjuje se potreba za opširnom dokumentacijom i olakšava stvaranje biblioteka i razvojnih okvira (*framework*).

Naravno, ako se pojavi situacija koja se ne može implementirati standardizovanim tipovima formi, tada se mora navesti detaljna specifikacija.

### 2.4.2 Korišćenje stereotipa

Drugi način za smanjivanje broja slučajeva korišćenja je korišćenje stereotipa. Stereotipi omogućavaju da se postojeći elementi UML modela obogate dodatnom semantikom (da dobiju dodatno značenje) koje je potrebno razvojnom timu da preciznije iskaže svoje potrebe. Stereotipi mogu imati proizvoljan naziv, a ako je potrebno i sopstvene attribute. Atributi stereotipa se zovu tagovi. Na elementima modela stereotipi se navode između znakova << >>, obično iznad naziva. Stereotipi se mogu pridružiti bilo kojem elementu UML modela, nezavisno kojoj vrsti dijagrama pripadaju.

Na slici 2.33 je upotrebljen CRUD stereotip, da naznači da manipulacija naložima podrazumeva skup slučajeva korišćenja koji specificiraju pregled, unos, izmenu i brisanje, na način kako je prikazano na slici 2.30 i opisano prilikom specifikacije standardizovanog panela za rad sa jednom klasom podataka.

Razvojni tim može da kreira proizvoljan broj stereotipa, za svaku složenu grupu funkcionalnosti za koju uoči da se ponavlja u projektu.



Slika 2.33 Korišćenje stereotipa za modelovanje osnovnih operacija nad korisničkim nalogima. Ovo je, u slučaju velikih sistema, bolji način u odnosu na primer prikazan na slici 2.30.

### 2.4.3 Korišćenje drugih vrsta dijagrama

U slučaju kada je neka funkcionalnost toliko složena da ju je teško opisati slučajevima korišćenja, mogu se koristiti druge vrste dijagrama koje opisuju ponašanje: dijagrami aktivnosti (poglavlje 3), dijagrami sekvenci (poglavlje 6), dijagrami komunikacije (poglavlje 7) i dijagrami prelaza stanja (poglavlje 8), kao i skice dijagrama klasa (poglavlje 4). Tada se u okviru osnovnog toka nekog složenog slučaja korišćenja samo navodi naziv i identifikator jednog ili više dijagrama kojima je odrađena specifikacija.

Cilj modelovanja nije primena određene vrste dijagrama po svaku cenu, već poboljšanje komunikacije i povećanje uzajamnog razumevanja razvojnog tima i korisnika. Za svaku namenu treba koristiti onu vrstu dijagrama, prototipa ili dokumenta koja pomaže da se najefikasnije postignu navedeni ciljevi [Ambler04].

## 2.5 Šta smo naučili

Slučajevi korišćenja se koriste za specifikaciju funkcionalnih zahteva. Skup slučajeva korišćenja predstavlja pogled na sistem sa stanovišta korisnika – koje servise treba da pruži, ne na koji način razvojni tim treba da ih implementira.

Prilikom prikupljanja skupa slučajeva korišćenja, preporučuju se sledeći koraci:

- odrediti učesnike,
- za svakog učesnika izolovati slučajeve korišćenja koji mu pripadaju i
- detaljno analizirati i opisati svaki slučaj korišćenja.

Razvojni tim kreće od minimalnog skupa slučajeva korišćenja koji modeluju osnovne funkcije sistema i postepeno, kroz dalju analizu, proširuje taj skup. Analiza se sprovodi komunikacijom sa naručiocima radi dogovora oko detalja funkcionisanja svakog uočenog slučaja korišćenja i pisanjem dokumenta koji ga opisuje. Dokument obično ima sledeće sekcije: identifikator, naziv, opis, preduslovi, posledice, osnovni tok, alternativni tokovi.

Prilikom pisanja dokumenta, može se specificirati esencijalni ili sistemski slučaj korišćenja. Esencijalni slučajevi korišćenja opisuju ponašanje nezavisno od implementacione platforme, a ako je moguće i od računara. Sistemski slučajevi se specificiraju tako da se uzima u obzir implementaciona platforma. Tada se obično sa naručiocima obavlja i skiciranje ekrana preko kojih će korisnik obavljati određene funkcije.



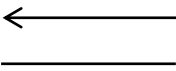
Slučajevi korišćenja se mogu povezivati vezama proširivanja (*extend*), generalizacije i uključivanja (*include*). Učesnici se mogu povezivati vezama generalizacije.

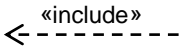
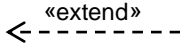

Radi izbegavanja kreiranja preopširnih i nečitljivih modela, može se koristiti:

- standardizacija korisničkog interfejsa,
- stereotipi sa unapred definisanim značenjem i
- druge vrste dijagrama koji mogu bolje modelovati određene aspekte ponašanja.

U tabeli 2.2 je dat pregled elemenata dijagrama slučajeva korišćenja koji su obrađivani u prethodnim sekcijama.

Tabela 2.2 Elementi dijagrama slučajeva korišćenja

Simbol	Naziv	Opis
	Učesnik	Učesnik modeluje korisničku ulogu, nekoga ko traži uslugu od sistema.
	Slučaj korišćenja	Slučaj korišćenja je funkcija ili servis kojim razmatrani sistem obezbeđuje merljivu vrednost nekom od učesnika.
	Veza asocijacije	<p>Veza asocijacije povezuje učesnika sa slučajem korišćenja. Ako je više učesnika povezano vezama asocijacije sa jednim slučajem korišćenja, tada:</p> <ul style="list-style-type: none"> <li>• Ako nijedna veza asocijacije nema strelicu ili sve imaju strelicu, smatra se da su svi učesnici ravnopravni i da mogu da pokrenu i izvrše dati slučaj korišćenja.</li> <li>• Ako veza samo jednog učesnika ima strelicu, znači da je on primarni učesnik (da pokreće taj slučaj</li> </ul>

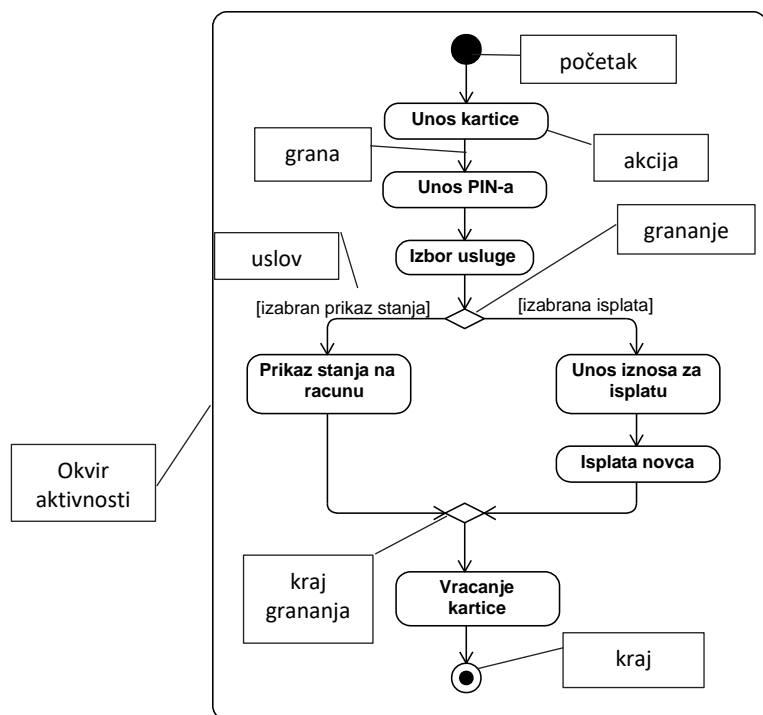
		korišćenja). Ostali izvršavaju neki od koraka datog slučaja korišćenja.
	Veza uključivanja	Veza uključivanja povezuje dva slučaja korišćenja. Ukoliko jedan slučaj korišćenja uključuje drugi, obavezno mora izvršiti korake uključenog tačno jednom, kao i svoje korake.
	Veza proširivanja	Veza proširivanja povezuje dva slučaja korišćenja. Koristi se kada se, prilikom izvršavanja osnovnog slučaja korišćenja, može, ali i ne mora, izvršiti dodatni slučaj korišćenja, u zavisnosti od određenog uslova. Uslov se navodi u tački proširenja ( <i>extension point</i> ) koja se nalazi u koracima osnovnog slučaja ili u preduslovima slučaja korišćenja koji treba da proširi osnovni. Prošireni slučaj korišćenja se može izvršiti nijednom, jednom ili više puta.
	Veza generalizacije (nasleđivanja)	<p>Veza generalizacije se koristi za specifikaciju nasleđivanja između dva učesnika ili dva slučaja korišćenja:</p> <ul style="list-style-type: none"> <li>• Ako je između dva učesnika, veza generalizacije znači da naslednik ima sve slučajeve korišćenja svog pretka, bez potrebe da sa njima bude direktno povezan.</li> <li>• Ako je između dva slučaja korišćenja, naslednik može redefinisati bilo koji element pretka: preduslove, posledice, opis, cele tokove kao i pojedninačne korake u okviru tokova. Ako neki deo specifikacije nije neophodno redefinisati kod naslednika, taj deo treba da ostane prazan ili da se čitalac uputi da pogleda odgovarajuću sekciju pretka.</li> </ul>



## Poglavlje 3

### Dijagram aktivnosti

Dijagrami aktivnosti se koriste za specifikaciju poslovnih procesa, radnih tokova (*workflow*), algoritama i koraka složenih slučajeva korišćenja. Obično se koriste u fazi analize zahteva ili specifikacije dizajna. Mogu se modelovati procesi koji se izvršavaju ručno ili na računaru. Notacija dijagrama aktivnosti podseća na notaciju blok-dijagrama algoritama (*flowcharts*) koja je dugo u primeni ali, za razliku od nje, omogućava da se modeluju i procesi čiji se potprocesi izvršavaju paralelno.



Slika 3.1 Aktivnost koja prikazuje osnovno ponašanja bankomata

Dijagram aktivnosti se crta kao usmereni graf uokviren pravougaonikom sa zaobljenim ivicama (okvirom aktivnosti). Čvorovi grafa mogu biti: različite vrste akcija, objekti (podaci) nad kojima akcije rade i elementi kontrole toka. Veze

između čvorova se nazivaju granama (*edge*) i modeluju se strelicama koje pokazuju smer izvršavanja. Okvir koji okružuje aktivnost ne mora da se crta.

Na slici 3.1 je modelovana aktivnost koja opisuje osnovnu funkcionalnost bankomata. Ta funkcionalnost odgovara sledećoj specifikaciji:

---

**Specifikacija ponašanja bankomata, verzija br. 1.** *Korisnik ubacuje karticu u bankomat, unosi PIN i bira vrstu usluge. Ako je izabrana usluga uvid u stanje na računu, bankomat mu daje traženu informaciju (koliko ima novca na računu u banci). Ako je izabrana isplata gotovine, korisnik unosi željeni iznos i bankomat isplaćuje novac u navedenom iznosu. Na kraju, bankomat vraća korisniku karticu.*

---

Na slici 3.1 vidimo osnovne čvorove: početak, akcije koje modeluju korake procesa, simbol za početak i kraj uslovnog izvršavanja (grananja) i kraj aktivnosti. Oni su povezani usmerenim granama koje pokazuju smer izvršavanja. Izvršavanje kreće od početnog čvora i ide redom, prateći smer strelice. Čim se završi izvršavanje tekućeg čvora, prelazi se na sledeći.

Navedeni elementi se najčešće koriste pri modelovanju aktivnosti. U nastavku ovog poglavlja ćemo postepeno uvoditi i ostale elemente. Više o njima se može naći u tabeli 3.1 na kraju poglavlja.

### 3.1 Dekompozicija složene aktivnosti na pod-aktivnosti

Primećujemo da je ponašanje bankomata koje opisuje dijagram na slici 3.1 krajnje ogoljeno. Koraci pojedinačnih akcija, moguće greške i situacije koje mogu sprečiti izvršenje traženih usluga nisu prikazani. Ovakav dijagram može pomoći da se stekne osnovno razumevanje ponašanja sistema koji se modeluje, ali nije dovoljno detaljan da bi se na osnovu njega moglo preći na implementaciju.

Razmotrimo sada sledeću verziju specifikacije koja je dobijena posle kreiranja inicijalnog dijagrama, kroz komunikaciju razvojnog tima i naručioca.

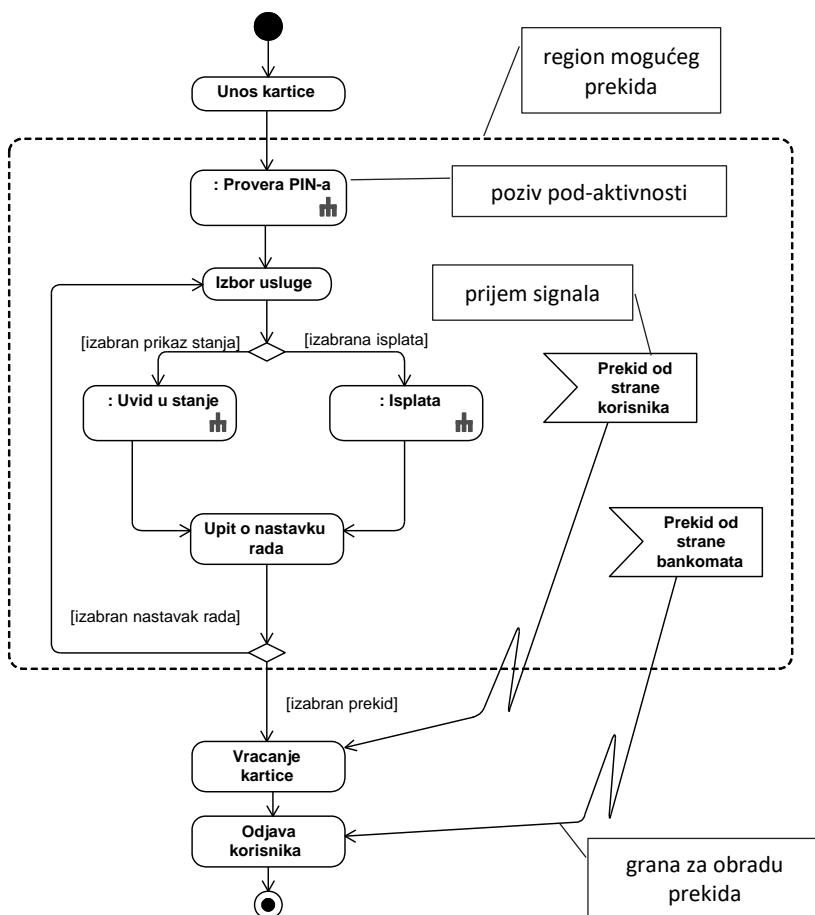
---

**Specifikacija ponašanja bankomata, verzija br. 2.**

**Unos i provera PIN-a.** *Kada korisnik unese karticu, bankomat prikazuje poruku kojom zahteva unos PIN-a. Kada korisnik obavi unos, bankomat proverava da li je uneti PIN ispravan. Provera se vrši tako što bankomat postavlja upit kartici, koja čuva ispravan PIN. Ukoliko uneti PIN nije korektan, bankomat obaveštava o tome korisnika i traži ponovni unos. Korisnik može da pogreši PIN ukupno tri puta, posle čega bankomat zadržava karticu i inicira prekid daljeg rada sa*

korisnikom. Na kartici se nalazi i brojač pokušaja, koji se uvećava posle svakog neuspelog unosa.

Posle ispravno unetog PIN-a, brojač pokušaja na kartici se postavlja na nulu, a korisnik dobija mogućnost da koristi usluge bankomata. Na raspolaganju su dve usluge: uvid u stanje na računu u banci i isplata gotovine.



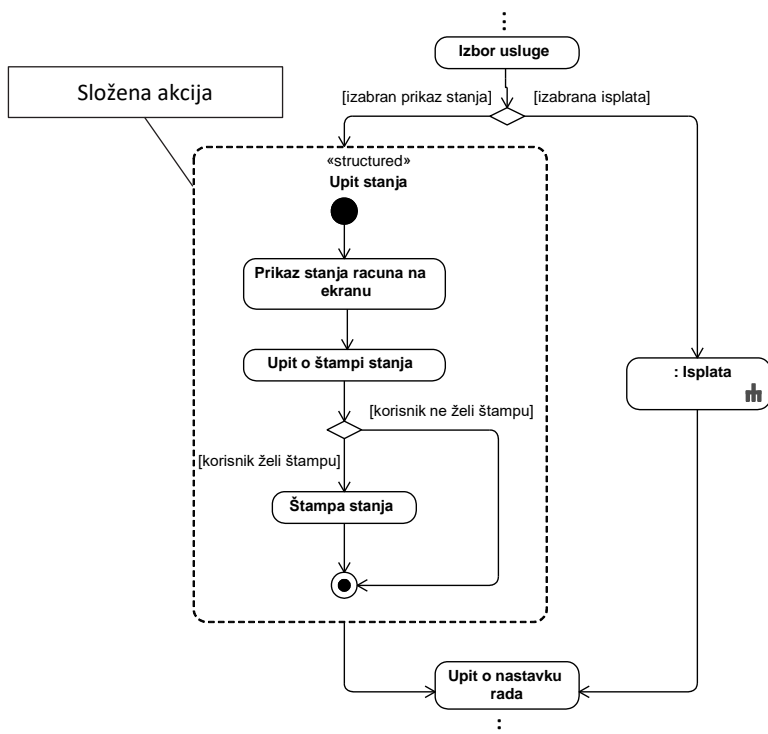
Slika 3.2. Modelovanje ponašanja bankomata prema drugoj verziji specifikacije. Uvedene su pod-aktivnosti i njihovi pozivi, kao i region mogućeg prekida, uz prijem dve vrste signala koji mogu izazvati prekid.

**Uvid u stanje računa.** Kada korisnik zatraži uvid u stanje računa, raspoloživi iznos na računu u banci treba da mu se prikaže na ekranu, uz pitanje da li želi i štampu tog iznosa. Štampa se obavlja ako korisnik odgovori pozitivno.

**Isplata gotovine.** Kod isplate gotovine može doći do sledećih poteškoća:

- bankomat nema dovoljno novca da isplati traženi iznos,
- bankomat ima dovoljno novca, ali nema odgovarajuće novčanice (npr. traženo je 1000 dinara, a bankomat ima samo novčanice od 2000),
- korisnik na svom računu u banci nema dovoljno novca.

U sva tri slučaja je potrebno dati odgovarajuće obaveštenje korisniku. Ako se ništa od navedenog ne desi, bankomat isplaćuje traženi iznos.

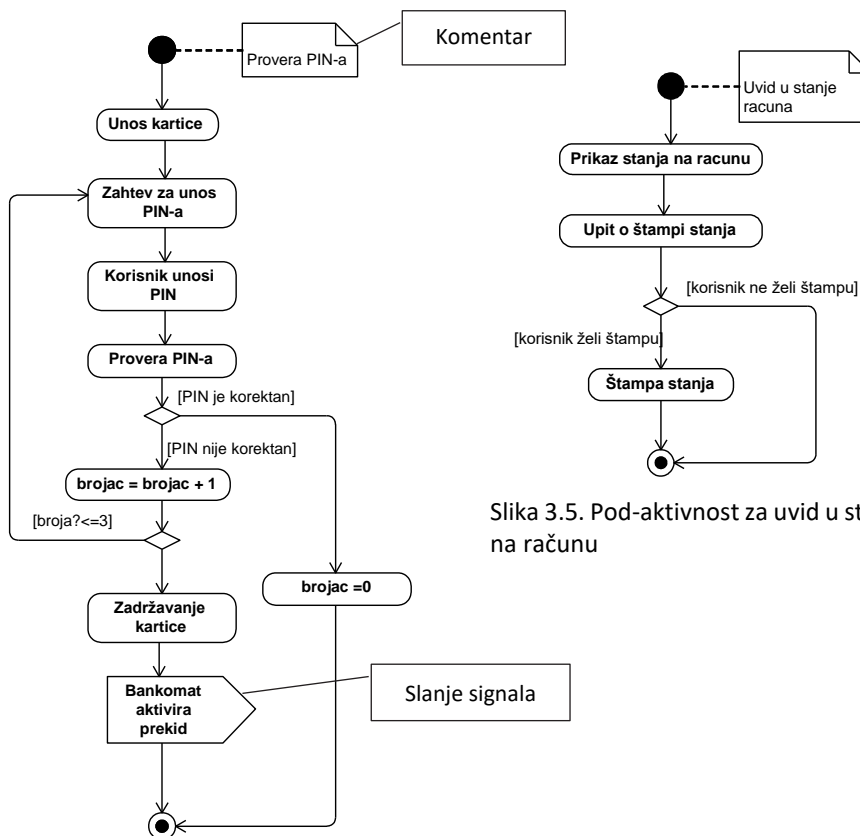


Slika 3.3 Pod-aktivnost za uvid u stanje je modelovana u okviru osnovnog dijagrama, kao složena akcija

**Dodatni zahtev 1.** U svakom trenutku, korisniku treba dozvoliti da odustane od korišćenja bankomata pritiskom na taster za prekid i kartica tada treba da mu bude vraćena.

**Dodatni zahtev 2.** Posle svake izvršene usluge, korisnika treba pitati da li želi da nastavi korišćenje bankomata ili da završi rad.

Ponašanja bankomata prema drugoj verziji specifikacije zahteva je modelovano na slici 3.2. Pošto se provera PIN-a, uvid u stanje na računu i isplata gotovine sastoje od više koraka, modelovani su u posebnim pod-aktivnostima na slikama 3.4, 3.5 i 3.6. Na dijagramu 3.2 se nalaze akcije koje omogućavaju njihov poziv (*call behavior action*). One imaju oznaku trozupca i naziv pridružene pod-aktivnosti. Na ovaj način i dalje imamo uvid u ponašanje bankomata u celini, ali i detalje potrebne za precizno razumevanje sistema i njegovu dalju implementaciju.

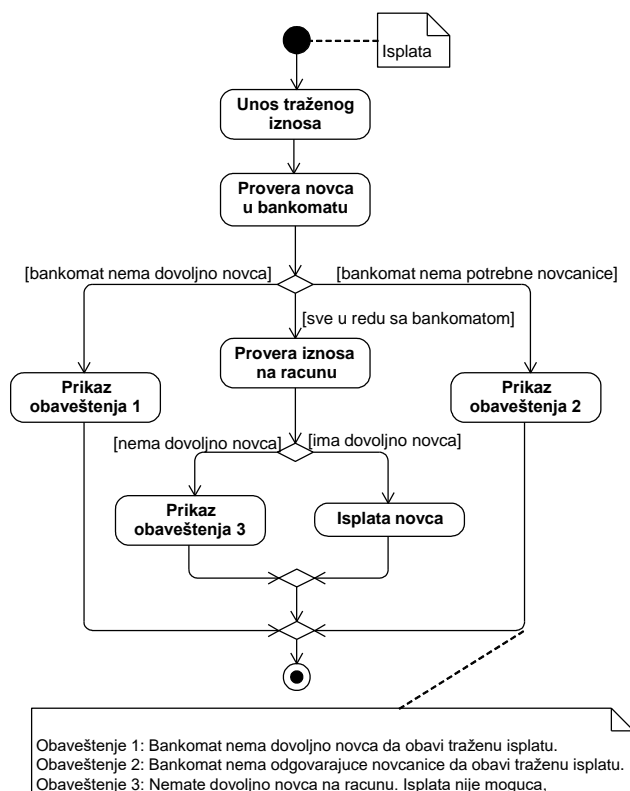


Slika 3.5. Pod-aktivnost za uvid u stanje na računu

Slika 3.4 Pod-aktivnost za proveru PIN-a. Uveden je simbol za slanje signala

Ovo je uobičajeni način dekomponovanja jednog složenog procesa primenom odozgo-na-dole (*top-down*) pristupa. Prvo se specificira osnovno ponašanje, radi sticanja celovite slike, a zatim se pojedinačni elementi dalje razrađuju u okviru posebnih modela.

U nekim situacijama je preglednije da se manje pod-aktivnosti prikažu direktno na osnovnom dijagramu, korišćenjem simbola za složenu (*structured*) aktivnost. Ovakav način modelovanja je ilustrovan na slici 3.3.



Slika 3.6 Pod-aktivnost za modelovanje isplate gotovine

### 3.2 Region mogućeg prekida

Na dijagramu 3.2 primećujemo i pravougaonik nacrtan isprekidanom linijom koji se zove *region mogućeg prekida*. U okviru njega su smeštene sve akcije koje treba trenutno zaustaviti ako korisnik pritisne dugme za prekid (posledica dodatnog zahteva br. 1) ili ako bankomat izazove prekid.

Čekanje signala za prekid je modelovano pomoću dva simbola za prijem signala: jedan čeka na pritisak tastera za prekid od strane korisnika, a drugi na signal prekida koji inicira bankomat, kada korisnik tri puta pogrešno unese PIN. Iz ovih simbola izlaze grane u obliku munje koje vode do akcija na koje se prelazi kada dođe do prekida.

Izvršavanje u regionu mogućeg prekida se odvija tako što se kreće od početka aktivnosti i redom izvršavaju modelovane akcije. Ako u bilo kom trenutku stigne neki od signala za prekid, izvršavanje se nastavlja od simbola koji čeka na prijem tog signala - analogno reagovanju na prekide (*interrupt*) i prelazak na izvršavanje prekidnih rutina u operativnim sistemima. Vidimo da, ako je prekid inicirao korisnik, bankomat vraća karticu, korisnik se odjavljuje i aktivnost se završava. Ako je prekid inicirao bankomat, kartica je ranije zadržana, tako da se korisnik samo odjavi i prelazi se na kraj.

### 3.3 Particije

Na slici 3.6 je modelovana pod-aktivnost za isplatu gotovine. Na slici 3.7 je navedena pod-aktivnost ponovo modelovana, ali sada su u dijagram uključeni i učesnici koji imaju uloge u funkcionisanju sistema: korisnik, bankomat i banka. Za svakog učesnika je odvojena jedna particija (neki autori je zovu i plivačka staza, *swim-lane*) u kojoj se nalaze akcije koje određeni učesnik izvršava.

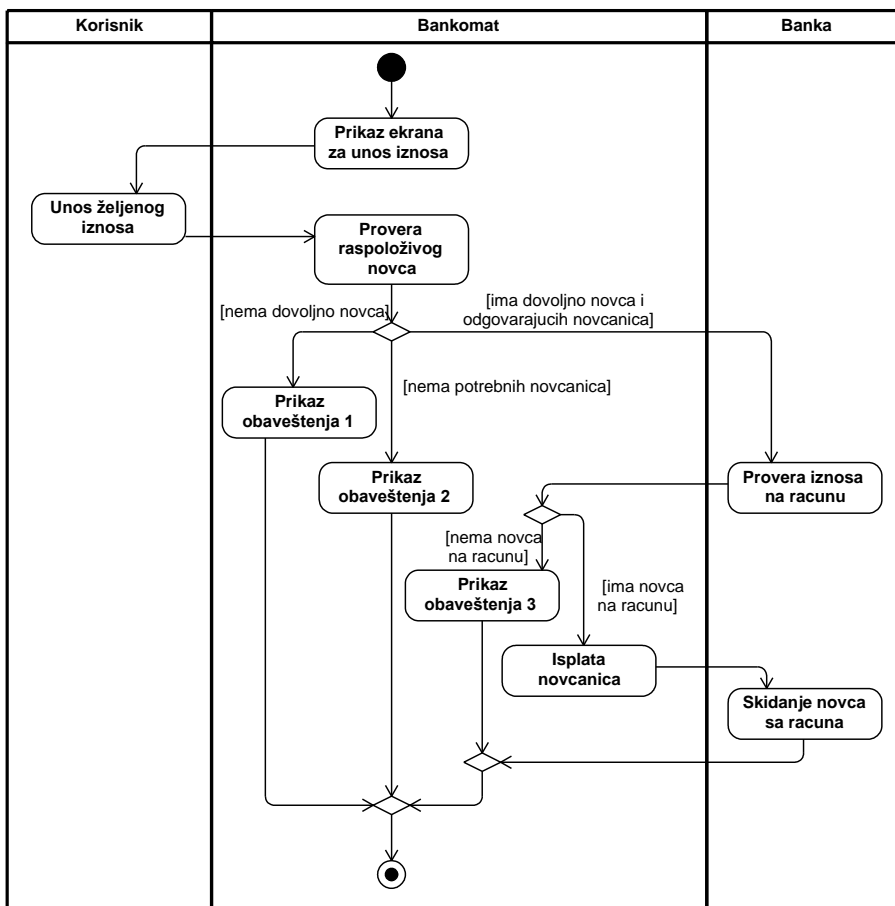
Particije se crtaju kada želimo da modelujemo odgovornosti u sistemu: ko šta izvršava ili gde se šta izvršava (npr. u kojim organizacionim jedinicama, u kojim uređajima).

### 3.4 Signali

Simbole za prijem signala smo videli na slici 3.2, u okviru regiona mogućeg prekida. Oni se mogu koristiti i za druge namene, svaki put kada je potrebno čekaње na neki signal ili događaj da bi se izvršavanje nastavilo.

Simbol za slanje signala se nalazi na kraju pod-aktivnosti za modelovanje unosa PIN-a, na slici 3.4. Posle slanja signala, prelazi se odmah na sledeću akciju u okviru date aktivnosti (ne čeka se da primalac signala završi obradu).

U okviru bankomata postoji više perifernih uređaja koji šalju i primaju signale: čitač kartice, tastatura, ekran, uređaj za isplatu novca i mali matrični štampač. Njima rukuje procesor koji izvršava osnovni proces modelovan na slici 3.2. Ako nam je cilj da akcenat stavimo na njihovo ponašanje i uzajamnu interakciju, neke segmente prethodnih dijagrama bismo mogli da modelujemo kao na slici 3.8.



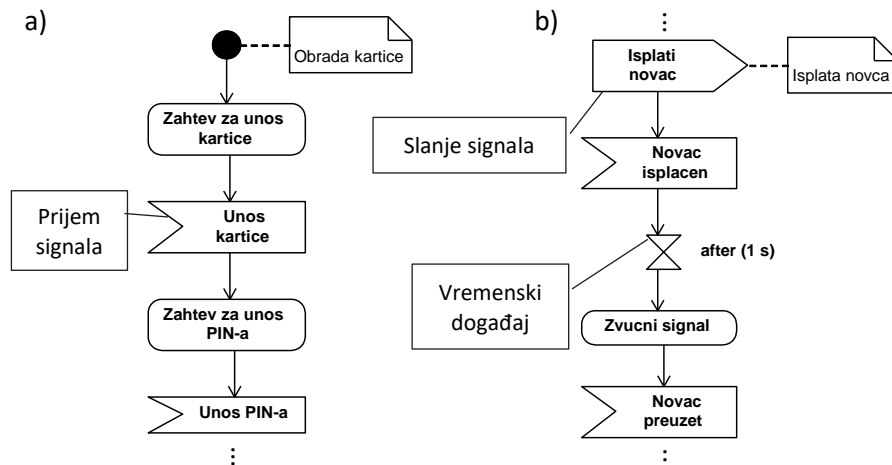
Slika 3.7 Pod-aktivnost za modelovanje isplate gotovine modelovana je korišćenjem particija („plivačkih staza“, *swim-lane*) koje omogućavaju da se akcije podele između učesnika u procesu.

Na slici 3.8a vidimo da je unos kartice sada modelovan tako da čeka prijem signala da je kartica uneta. Posle toga, bankomat prikazuje zahtev za unos PIN-a i čeka prijem signala da je PIN unet. Prvi signal šalje čitač kartice, a drugi dugme za potvrdu unosa na tastaturi. Ako iz konteksta nije jasno, može se u komentaru na dijagramu navesti od koga se očekuje signal.

Na slici 3.8b je detaljnije modelovan proces isplate. Procesor šalje signal uređaju za isplatu novca i čeka da od njega dobije potvrdu da je novac isplaćen. Posle toga, čeka jednu sekundu (simbol za vremenski događaj u obliku peščanog sata) i zatim aktivira zvučni signal da bi skrenuo pažnju korisniku da uzme novac. Na



kraju, čeka na potvrdu uređaja za isplatu da je novac zaista i preuzet, da bi nastavio sa daljim izvršavanjem.



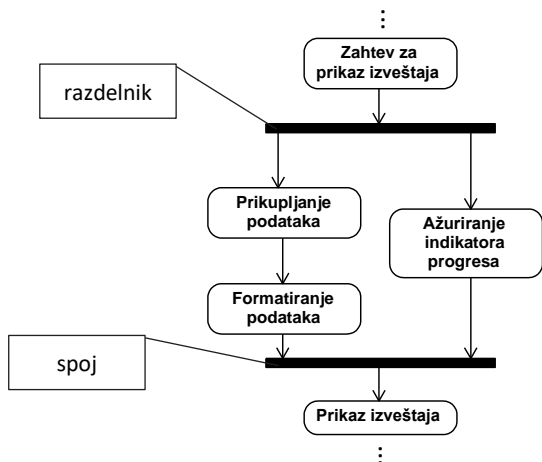
Slika 3.8 Primer modelovanja segmenata dijagrama 3.2 za a) obradu kartice i b) isplatu novca korišćenjem simbola za prijem i slanje signala

Vreme u okviru vremenskog događaja može da se specificira relativno u odnosu na vreme izvršavanja: *after* <broj>, kao na slici 3.8b, ili da se navede tačno vreme kada nešto treba da se izvrši: *at* <vreme> (na primer, tačno u ponoć treba aktivirati pravljenje sigurnosnih kopija).

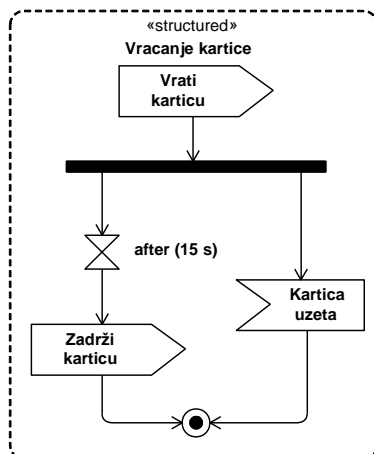
### 3.5 Paralelno izvršavanje

Dijagramima aktivnosti se mogu modelovati i procesi (niti) koje se paralelno izvršavaju. Početak paralelnog izvršavanja se modeluje simbolom koji se zove razdelnik (*fork*). On ima tačno jednu ulaznu granu i više izlaznih grana, koliko ima procesa koji se paralelno izvršavaju. Kraj paralelnog izvršavanja je obično spoj (*join*) - simbol koji izgleda kao razdelnik, samo što ima više ulaznih i tačno jednu izlaznu granu (slika 3.9). Izvršavanje iza spoja se nastavlja tek kada su svi paralelni procesi završili svoje akcije.

Vraćanje kartice, koje je na dijagramu 3.2 modelovano kao jedna akcija, sada je razloženo na više koraka. Procesor šalje signal čitaču kartica da vrati karticu i čeka dok ne dobije signal da je kartica uzeta. Paralelno sa čekanjem se aktivira i proces koji posle 15 sekundi šalje signal da čitač zadrži karticu, ako je u tom periodu vlasnik nije uzeo. Ovim se sprečava neovlašćen pristup, u slučaju zaboravljanja kartice u čitaču.



Slika 3.9 Primer modelovanja paralelnog izvršavanja dva procesa, koje počinje razdelnikom, a završava se spojem. Jedan proces izvršava potencijalno dugotrajnu obradu podataka, a drugi ažurira indikator progressa.

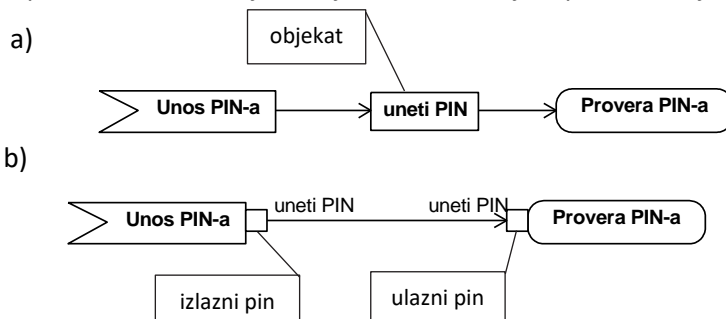


Slika 3.10 Primer modelovanja paralelnog izvršavanja koje se završava simbolom za kraj aktivnosti. Proces koji se prvi izvrši prekida paralelno izvršavanje. Ovo je uobičajeni način kako se modeluje *time-out*.

**Napomena:** Nemojte mešati grananje (uslovno izvršavanje) sa paralelnim izvršavanjem! Kod uslovnog izvršavanja se izvršava samo jedna grana koja zadovoljava zadati uslov. Kod paralelnog izvršavanja se istovremeno izvršavaju sve grane iza razdelnika.

### 3.6 Prenos podataka

Čvorovi dijagrama aktivnosti mogu biti i objekti (podaci) koji se razmenjuju između aktivnosti. Na slici 3.11 su prikazana dva ekvivalentna načina modelovanja prenosa podataka, korišćenjem objekta i korišćenjem pinova akcija.

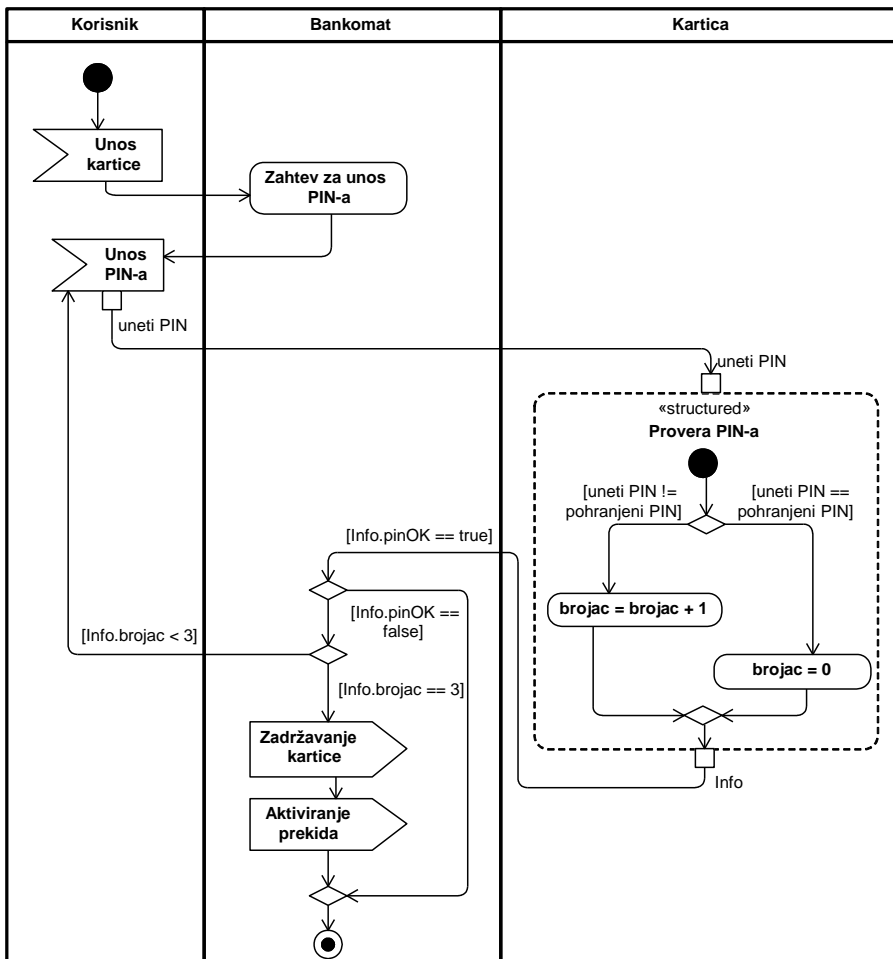


Slika 3.11 Primer modelovanja prenosa podataka između akcija a) korišćenjem objekta b) korišćenjem ulaznog i izlaznog parametra akcije. Parametri akcija se nazivaju pinovi.

Pinovi su parametri akcija. Mogu se modelovati neformalno, samo navođenjem naziva, ali im se može navesti i tip podataka, ako je to za neku primenu potrebno.

Na slici 3.12 vidimo pod-aktivnost za proveru PIN-a sa slike 3.4 koja je sada modelovana korišćenjem većine elemenata koje smo u međuvremenu naučili:

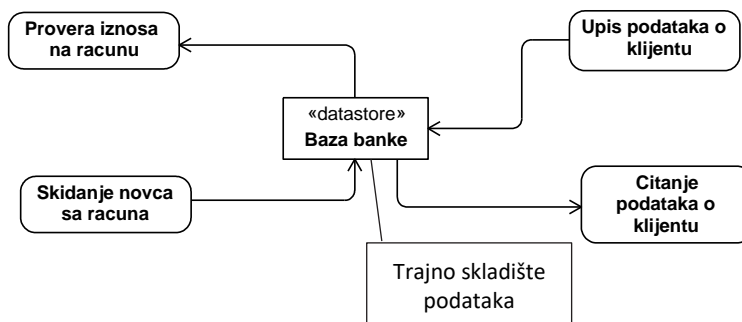
- Particije pokazuju ko je nadležan za izvršavanje određenih akcija. Vidimo da su učesnici: korisnik, bankomat i kartica. Kartica je aktivan element sistema koji sadrži pohranjen tačan PIN za datu karticu i brojač pokušaja. Ažuriranje brojača pokušaja je u nadležnosti kartice.
- Korišćeno je slanje i prijem signala za komunikaciju između učesnika.
- Za razmenu podataka između akcija korišćeni su pinovi. Vidimo da pinove mogu imati sve vrste akcija, uključujući složene akcije i akcije za prijem i slanje signala.
- Provera PIN-a je modelovana kao složena aktivnost, koja ima svoj ulazni i izlazni parametar (ulaz je PIN unet od strane korisnika, a izlaz je objekat klase Info koji sadrži attribute: da li je uneti PIN ispravan i trenutnu vrednost brojača pokušaja).



Slika 3.12 Aktivnost za proveru PIN-a sa slike 3.4 modelovana je sa više detalja. Korišćene su particije, prijem i slanje signala, složena aktivnost i pinovi za razmenu podataka između akcija.

### 3.7 Skladišta podataka

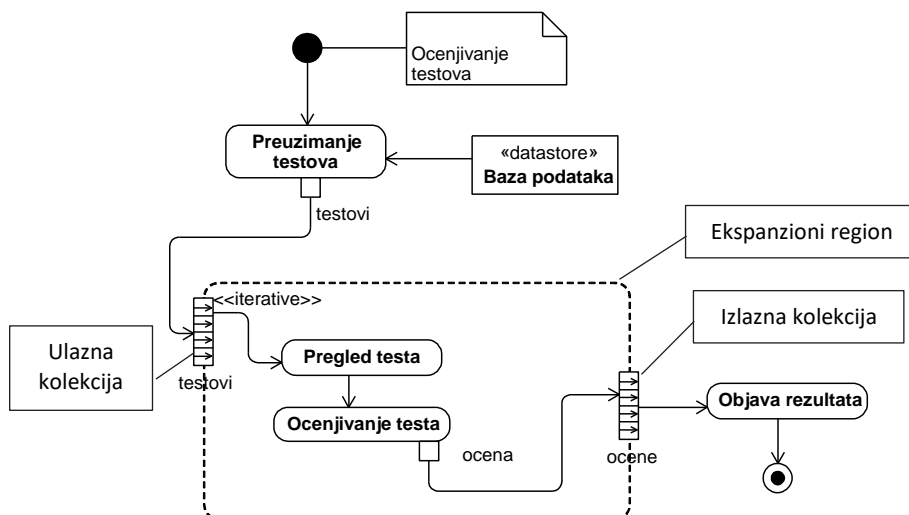
Skladišta podataka se koriste za modelovanje baza podataka ili datoteka na disku. Na slici 3.13 su prikazane četiri akcije koje, vremenski nezavisno jedna od druge, pristupaju skladištu. Po smeru strelice vidimo da akcije Provera iznosa na računu i Čitanje podataka o klijentu čitaju podatke, a Skidanje novca sa računa i Upis podataka o klijentu pišu u skladište podataka.



Slika 3.13 Modelovanje trajnog skladišta podataka. Akcije mu pristupaju nezavisno jedna od druge

### 3.8 Ekspanzioni region

Ekspanzioni region se koristi za modelovanje obrada nad kolekcijom elemenata (pandan foreach petlji u programskim jezicima). Mora imati minimalno jednu ulaznu kolekciju. Može i ne mora imati izlazne kolekcije.

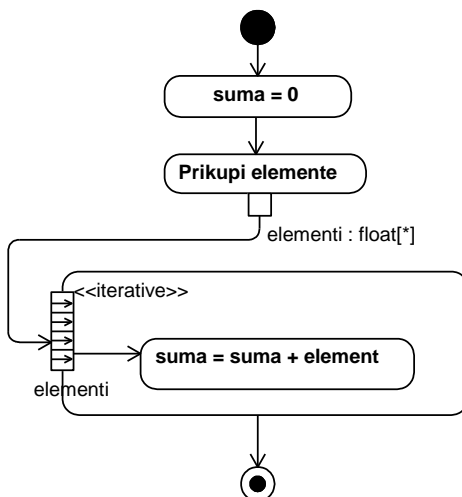


Slika 3.14 Modelovanje ekspanzionog regiona. Ulaz regiona je kolekcija testova koji se preuzimaju iz baze podataka, a izlaz je kolekcija ocena.

Obrada se nad elementima ulazne kolekcije može izvršavati sekvencijalno (iterative) ili istovremeno nad svim elementima (parallel).

Na slici 3.14 vidimo aktivnost za ocenjivanje testova u okviru sistema za elektronsko ocenjivanje. Kolekcija testova se preuzima iz baze podataka i kao ulaz prosleđuje ekspanzionom regionu. Nad svakim testom se vrše dve akcije: pregled i ocenjivanje. Rezultat (izlazni pin) akcije za ocenjivanje je ocena testa. Sve ocene se prikupljaju u kolekciju ocena koja je izlaz ekspanzionog regiona.

Ukoliko se samo vrši obrada nad elementima ulazne kolekcije, bez formiranja nove kolekcije, tada ekspanzioni region ne mora imati izlaz. Na slici 3.15 je ilustrovana ovakva situacija, na primeru računanja sume elemenata kolekcije.



Slika 3.15 Modelovanje ekspanzionog regiona koji nema izlaznu kolekciju. Obrada se vrši nad elementima postojeće kolekcije, bez kreiranja nove.

### 3.9 Šta smo naučili

Dijagrami aktivnosti se koriste za specifikaciju izvršavanja poslovnih procesa, radnih tokova, algoritama, koraka složenih slučajeva korišćenja. Modeluju se kao usmereni graf, čiji čvorovi mogu biti:

- različite vrste akcija (jednostavne i složene, prijem i slanje signala),
- elementi kontrole toka (grananje i kraj grananja, razdelnik i spoj, ekspanzioni region i region mogućeg prekida),
- podaci (objekti, skladišta podataka, parametri, odnosno pinovi akcija).

Čvorovi se povezuju usmerenim granama. Razlikujemo „obične” grane i grane koje vode od signala koji izaziva prekid, u okviru regiona mogućeg prekida.

Elementi mogu biti razvrstani u particije koje pokazuju nadležnosti učesnika sistema (ko šta izvršava).


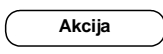
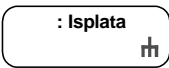
Primećujemo da smo krenuli od jednostavnog dijagrama aktivnosti koji modeluje osnovno ponašanje bankomata i postepeno dodavali detalje koji specifikiraju različite aspekte sistema.



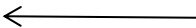
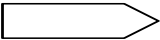

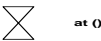
U realnim projektima razvojni tim odlučuje do kog nivoa detalja će se vršiti modelovanje, u zavisnosti od konkretnih potreba. Ako se model koristi za skiciranje poslovnih procesa u fazi analize zahteva, obično se prave jednostavni modeli kojima se proverava uzajamno razumevanje sa korisnikom. Tada se obično koristi samo minimalan skup simbola, da bi korisnik mogao da lakše razume proces.

U fazi specifikacije dizajna se mogu praviti detaljniji dijagrami, posebno ako ih crta softverski arhitekta kao specifikaciju implementacije ili za potrebe dokumentovanja. Važno je upamtiti da nije neophodno na svakom dijagramu koristiti sve simbole koji postoje u okviru dijagrama aktivnosti.


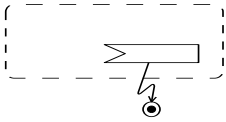


U tabeli 3.1 je data rekapitulacija elemenata koji su korišćeni u prethodnim sekcijama.

Tabela 3.1 Elementi dijagrama aktivnosti

Simbol	Naziv	Opis
	Početak aktivnosti	Aktivnost najčešće počinje ovim simbolom, mada može početi i simbolom za prijem signala.
	Akcija	Jedan korak aktivnosti koja se modeluje. Natpis može biti slobodan tekst, matematička formula ili iskaz na nekom programskom jeziku ili pseudo-jeziku, npr: vrednost = kolicina * cena, i++, itd.
	Poziv pod-aktivnosti koja se modeluje na zasebnom dijagramu	U slučaju modelovanja složenih poslovnih procesa koji se sastoje od puno koraka, korisno je izolovati pod-aktivnosti. Svaka pod-aktivnost se modeluje posebnim dijagramom, a na osnovnom dijagramu se specifikira njen poziv.

	Objekat	Objekat klase ili dokument koji se kreira od strane neke akcije, prosleđuje kao ulaz nekoj akciji ili mu akcija menja stanje.
	Uslovno izvršavanje ili kraj uslovnog izvršavanja	<p>Uslovno izvršavanje, odnosno grananje toka izvršavanja, u zavisnosti od uslova koji se navode na granama koje izlaze iz datog simbola. Uslove treba tako pisati da budu uzajamno isključivi.</p> <p>Isti simbol se može koristiti i za završetak grananja, kao na slici 3.1, ali se grane mogu direktno „uliti“ i u „običnu“ akciju, kao na slici 3.2. Uslovi na granama se mogu pisati kao slobodan tekst, matematička formula ili iskaz na nekom programskom jeziku ili pseudo-jeziku.</p>
	Grana ili tok	Grana koja vodi do sledećeg čvora dijagrama, po okončanju izvršavanja prethodnog.
	Slanje signala	Slanje signala (događaja) nekom uređaju, ili eksternom sistemu. Prijemnik signala može biti neki eksterni sistem, uređaj ili drugi proces u okviru istog sistema. Posle slanja, izvršavanje aktivnosti se normalno nastavlja - ne čeka se da prijemnik signala završi aktivnost koju dati signal inicira, osim ako se prijemnik signala nalazi u regionu mogućeg prekida.
	Prijem signala	Čekanje na prijem signala (događaja) od nekog uređaja ili eksternog sistema. Izvršavanje procesa je zaustavljeno do prijema očekivanog signala.
	Prijem vremenskog događaja	Specijalizacija elementa za prijem signala. Sistem čeka na zadati vremenski događaj, kojim se aktivira akcija koja mu je pridružena: (1) kada istekne specifikirani vremenski interval, (2) u zadato vreme.



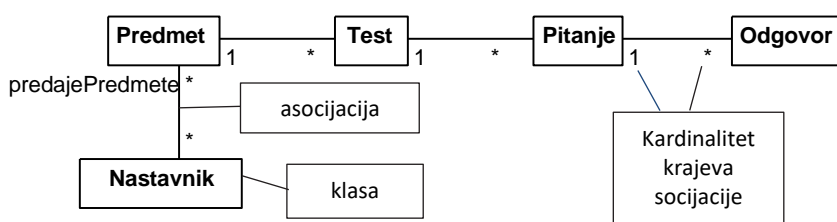
	Razdelnik ( <i>fork</i> ) ili spoj ( <i>join</i> ) paralelnog izvršavanja	Posle razdelnika se tok izvršavanja može razdvojiti na više tokova (procesa, niti) koji se istovremeno izvršavaju i koji iz njega „izviru“. Ako je potrebno, ovakvi tokovi se mogu ponovo sažeti u jedan tok „ulivanjem“ u spoj. Razdelnik ima jedan ulazni tok i više izlaznih, dok spoj ima više ulaznih tokova i jedan izlazni.
	Region mogućeg prekida	Region mogućeg prekida omogućava trenutni prekid izvršavanja akcija koje se nalaze u njemu i prelazak na obradu prekida, ukoliko se desi određeni događaj.
	Ekspanzioni region	Ekspanzioni region se koristi za modelovanje obrada nad kolekcijom elemenata (pandan <i>foreach</i> petlji u programskim jezicima).
	Kraj aktivnosti	Kraj aktivnosti



## Poglavlje 4

### Dijagram klasa

Dijagram klasa se može koristiti tokom celokupnog životnog ciklusa softverskog proizvoda. Tokom specifikacije zahteva se može koristiti za skiciranje strukture poslovnog domena za koji se softver implementira. Tada se uočavaju bitni koncepti i njihovi uzajamni odnosi i modeluju klasama i vezama između njih. Obeležja i metode se mogu u potpunosti izostaviti (slika 4.1), ili se mogu navesti samo imena najbitnijih obeležja i metoda koji se u datom trenutku razmatraju (slika 4.2).



Slika 4.1 Konceptualni dijagram klasa na visokom nivou apstrakcije koji modeluje strukturu testa u okviru sistema za elektronsko ocenjivanje. Obeležja i metode su izostavljeni.

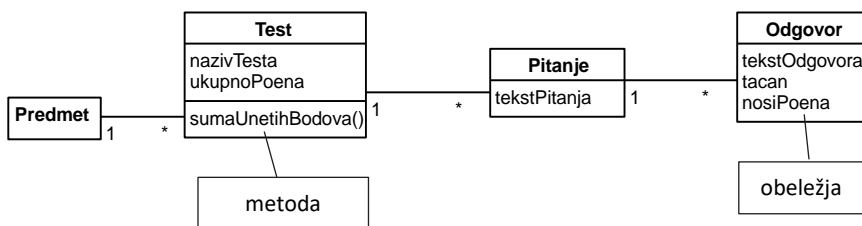
U narednim fazama se dijagram klasa može koristiti za komunikaciju ideja u okviru projektnog tima, specifikaciju implementacije, istraživanje različitih projektantskih odluka, generisanje koda, kao i za dokumentovanje rešenja. U zavisnosti od toga koja je namena modela, dijagram može biti više ili manje apstraktan, bliži pojmovima iz problemskog domena ili pojmovima implementacione platforme. *Konceptualni*, odnosno *domenski modeli*, se fokusiraju na modelovanje konceptata domena za koji se softver implementira i obično sadrže manje detalja od *implementacionih modela* koji se direktno mapiraju na programski kôd.

Na slici 4.1 je prikazan primer konceptualnog modela na visokom nivou apstrakcije koji skicira strukturu testa u okviru sistema za elektronsko ocenjivanje čiju specifikaciju zahteva smo odradili u okviru drugog poglavlja. Razvojni tim, kada prilikom analize zahteva sazna koji podaci su potrebni za određene slučajeve korišćenja, može kreirati skicu dela sistema i pridružiti je odgovaraju-

ćem slučaju korišćenja. Detaljno modelovanje se obavlja kasnije, tokom specifikacije dizajna i implementacije.

Model sa slike 4.1 odgovara slučaju korišćenja Kreiranje testa. Nastavni predmet može imati više pridruženih testova. Jedan test može pripadati samo jednom predmetu. Navedeni odnos je modelovan vezom asocijacije između klasa Predmet i Test. Kardinalitet na krajevima asocijacije označava sa koliko objekata (instanci) klase na suprotnom kraju asocijacije jedan objekat razmatrane klase može ili mora biti povezan. Broj 1 označava da objekat mora biti povezan sa tačno jednim objektom klase na suprotnom kraju asocijacije; \* označava proizvoljan broj, od nula do beskonačno. Kardinalitet se može navoditi i kao interval m..n, gde m označava minimalno potreban broj objekata, a n maksimalno dozvoljen broj.

Sada, kada znamo značenje asocijacija i kardinaliteta na njihovim krajevima, možemo sa modela pročitati i da test može imati više pitanja, a pitanje više odgovora. Jedno pitanje može pripadati samo jednom testu, a odgovor samo jednom pitanju.

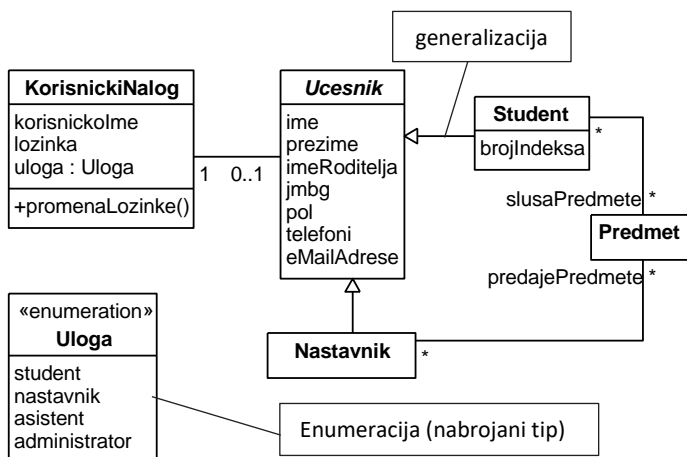


Slika 4.2 Konceptualni dijagram klasa sa više detalja u odnosu na sliku 4.1. Dodata su osnovna obeležja i jedna metoda. Metode se uglavnom izostavljaju u konceptualnim modelima ili se navode samo najbitnije.

Klasa može imati obeležja i metode koji se pišu u odvojenim odeljcima (slika 4.2). Ako nisu navedeni u konceptualnom modelu, to ne znači da ih nema, već da u datoj situaciji nisu bitni ili da nisu razmatrani. Iz istog razloga se često izostavljaju i tipovi podataka i modifikatori pristupa obeležjima i metodama, parametri metoda i sl. U početnim fazama je cilj modelovati brzo i skoncentrisati se na suštinu.

Na slici 4.3 je prikazan deo konceptualnog modela koji prikazuje učesnike u nastavnom procesu i njihov način prijave na sistem. Student i Nastavnik su učesnici u nastavnom procesu (nasleđuju apstraktnu klasu Ucesnik). Svaki učesnik u nastavi se prijavljuje korišćenjem tačno jednog korisničkog naloga. Korisnički nalog ne mora da bude vezan za učesnika u nastavi (kardinalitet 0..1)

zato što je administrator sistema osoba iz tehničke podrške - nije ni student ni nastavnik. Student može da sluša više predmeta; predmet može da bude slušan od strane više studenata. Nastavnik može da predaje više predmeta; predmet može da bude predavan od strane više nastavnika.

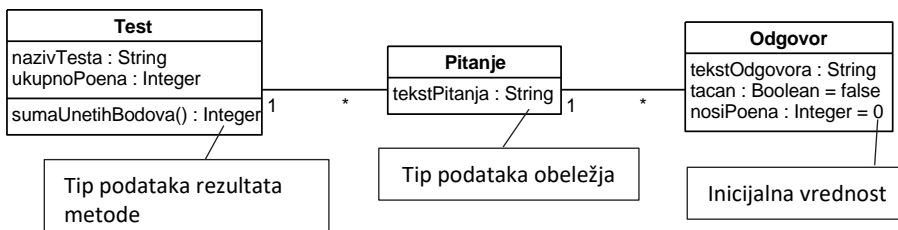


Slika 4.3 Konceptualni dijagram klasa koji modeluje učesnike u nastavnom procesu. Učesnici mogu biti nastavnici i studenti. Svaki učesnik ima jedan korisnički nalog kojim se prijavljuje na sistem.

Na slici 4.4 su dodati još neki detalji u konceptualni model sa slike 4.2. Obeležja klasa poseduju tipove podataka; obeležja *tacan* i *nosiPoena* u klasi *Odgovor* imaju specificirane inicijalne vrednosti *false* i *0*; metodi *sumaUnetihBodova* je dodeljen tip rezultata *Integer*.

Ovaj model je dovoljno precizan da bi se na osnovu njega mogla obaviti implementacija domenskih klasa. Instance domenskih klasa se trajno čuvaju u nekom skladištu podataka (u bazi podataka ili datotekama na disku). Ako je u pitanju relacionala baza, njena šema podataka se može dobiti na osnovu ovog modela primenom objektno-relacionog mapiranja. Podaci potrebni za kreiranje instanci klasa se dobijaju od korisnika, preko klasa koje pripadaju sloju korisničkog interfejsa (forme, dijalozi, web stranice i sl).

**Napomena:** Domenske klase treba da se projektuju tako da budu nezavisne od sloja korisničkog interfejsa – da nemaju znanje o njegovom izgledu i strukturi, čak ni vrsti. Projektovanje klasa korisničkog interfejsa i njihove saradnje sa domenskim klasama je obrađeno u sekciji 4.5.



Slika 4.4 Detaljan konceptualni dijagram klasa. Dodati su tipovi podataka za obeležja, inicijalne vrednosti, kao i tip rezultata za metodu.

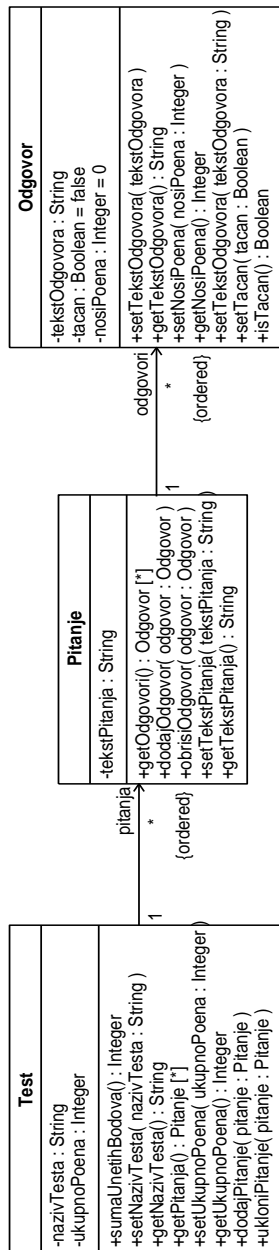
Na slici 4.5 se nalazi veoma detaljan konceptualni model koji se direktno preslikava na programski kod. Specificirana je vidljivost za obeležja i metode. Znak „-“ ispred imena označava da je u pitanju privatno obeležje ili metoda, kojem samo data klasa može da pristupi. Znak „+“ označava javna obeležja i metode. Dodate su i get i set metode za obeležja, sa parametrima i povratnim tipovima, kao i metode za dodavanje i brisanje elemenata kolekcija.

Moguća namena dijagrama sa slike 4.5 je detaljna specifikacija koju softverski arhitekta kreira programerima-početnicima, kao i ulaz za generator koda, ukoliko se koristi MDSE (videti sekciju 1.4).

Svi navedeni dijagrami klasa su korektni, sa stanovišta sintakse UML-a.

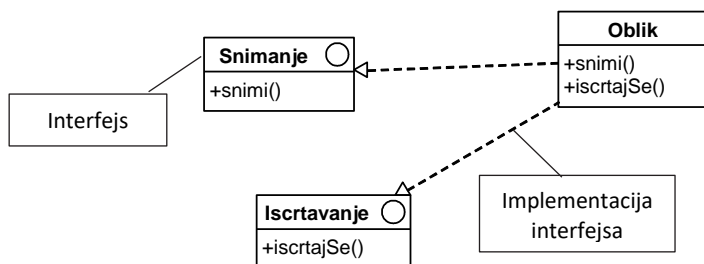
**Napomena:** Iako predstavlja potpunu specifikaciju, primećujemo da je model na slici 4.5 prekompleksan i težak za čitanje. Teško je primetiti metode koje implementiraju neku bitnu funkcionalnost od velikog broja get i set metoda.

Iz ovog razloga se u okviru softverskih timova obično kreiraju konvencije koje će se primenjivati tokom modelovanja, da modeli ne bi bili zatrpani podrazumevanim metodama i da bi oni koji ih koriste mogli da se skoncentrišu na suštinu. Česta je praksa da se u UML dijagramima klasa ne navode get i set metode, kao ni konstruktori klasa, već samo metode koje obavljaju neki složeniji zadatak karakterističan za tu klasu, kao na slici 4.4. Podrazumeva se da će programeri dodati konstruktore, get i set metode prilikom prevođenja dijagrama klasa na programski kod. Vidljivost se takođe ne navodi, ako su u pitanju privatna obeležja koja imaju javne get i set metode (što je u većini slučajeva). Ova konvencija će biti primenjivana u nastavku udžbenika.



Slika 4.5 Vrlo detaljan konceptualni dijagram klasa. Specificirana je vidljivost za obeležja i metode, dodate su get i set metode za obeležja, kao i metode za dodavanje i brisanje pitanja i odgovora

Na slici 4.6 su prikazana dva interfejsa, *Snimanje* i *Iscrtavanje*, kao i klasa *Oblik* koja ih implementira. Interfejs u UML-u može imati atribute i metode. Ovo je primer implementacionog modela – prikazujemo pojmove bliske implementacionoj platformi u okviru koje postoji koncept interfejsa (npr. programski jezik Java ili C#).



Slika 4.6. Primer implementacije interfejsa – klasa *Oblik* implementira interfejse *Snimanje* i *Iscrtavanje*

Svi prethodni pojmovi su navedeni radi sticanja brzog uvida u kreiranje dijagrama klasa i njegove najčešće elemente. Detalji su dati u nastavku. Modelovanje klasa je objašnjeno u sekciji 4.1. Pravila za specifikaciju obeležja klasa su opisana u sekciji 4.2. Metode su objašnjene u sekciji 4.3. Veze koje se mogu koristiti u okviru dijagrama klasa su detaljno obrađene u sekciji 4.4. Modelovanje korisničkog interfejsa i načini za njegovo povezivanje sa domenskim klasama su prikazani u sekciji 4.5.

## 4.1 Modelovanje klasa

Imena klasa se pišu prema *Upper Camel Case* konvenciji (ukoliko se ime sastoji od više reči, sve reči se pišu spojeno, pri čemu svaka reč počinje velikim slovom). Imena treba tako davati da se, ako je moguće, iz imena odmah vidi namena klase, bez potrebe za opširnim dokumentovanjem. Primeri validnih imena klasa su: *PozajmicaKnjige*, *GrafToka*, *DeoLukaFigure*, itd. Ukoliko je klasa apstraktna, ime klase se piše zakošenim slovima (videti klasu *Ucesnik* na slici 4.3). Imena klasa treba davati u jednini.

Kandidati za klase se u rečenicama specifikacije zahteva prepoznaju kao imenice (departman, student, predmet, osoba, merno mesto). Međutim, obeležja klasa su takođe imenice (ime, prezime, datum rođenja, naziv predmeta). U opštem slučaju, možemo zaključiti da je u pitanju klasa, a ne obeležje, ako pojam koji razmatramo ima svoje osobine (obeležja i metode) koje su bitne za sistem koji se razvija.



Pri određivanju šta klasa treba da sadrži od obeležja i metoda treba se voditi sledećim principima:

- Principom „jedne odgovornosti“ (*single responsibility principle*) – optimalno je da je klasa zadužena za jednu, jasno definisanu namenu, radi jednostavnije saradnje sa drugim klasama, kao i lakšeg testiranja i održavanja.
- Principom kohezije – optimalno je da svaka metoda klase radi sa većinom obeležja klase (ne računajući get i set metode). Ako imamo „klaster“ – odvojene skupove metoda koje rade sa odvojenim skupovima obeležja, verovatno bi datu klasu trebalo podeliti na više manjih.
- principom lokalizovane izmene – ako menjamo implementaciju jedne klase, u idealnom slučaju, ta promena ne bi trebalo da utiče na klase koje koriste njene usluge.

U narednim sekcijama ćemo, kroz primere, detaljnije razmatrati ove principe i vežbati njihovu primenu.

## 4.2 Modelovanje obeležja

Nazivi obeležja i metoda po UML konvenciji treba da počinju malim slovom i da se pišu u *Camel Case* notaciji (ukoliko se ime sastoji od više reči, sve reči su spojene i počinju velikim slovom, sem početne). Primer davanja imena obeležjima: imeOsobe, datumDokumenta, broj, itd.

Osnovne osobine obeležja u okviru klase su: vidljivost (*visibility*), naziv, tip podataka, podrazumevana (inicijalna) vrednost i kardinalitet (*multiplicity*). Takođe, može se specificirati da li je obeležje statičko, kao i niz dodatnih opcija<sup>12</sup>. Od navedenih osobina, jedino je obavezno navesti naziv obeležja. Format za specifikaciju obeležja u okviru klase je sledeći:

vidljivost naziv: tip-podataka kardinalitet = podrazumevana-vrednost {dodatne-opcije}

U okviru obeležja: + telefon: String[0..1] = ""

+ je vidljivost (javno obeležje), telefon je naziv, String je tip podataka, [0..1] je kardinalitet obeležja (broj vrednosti datog obeležja koji može da

---

<sup>12</sup> Pored navedenih, postoje i osobine koje nećemo obrađivati jer nadilaze potrebe ovog udžbenika (koriste se za specifikaciju meta-modela u okviru MDSE). Detaljan opis svih osobina obeležja se može naći u [Rumbaugh05].

postoji u okviru instance klase), a "" je podrazumevana, odnosno inicijalna vrednost koja se dodeljuje obeležju prilikom instanciranja klase.

### 4.2.1 Vidljivost

Vidljivost obeležja može biti: private, protected, public i package (tabela 4.1).

Tabela 4.1 Modelovanje vidljivosti obeležja i metoda u okviru klase

Simbol	Vidljivost obeležja	Značenje
-	private	Privatno obeležje – može mu se pristupati samo u okviru date klase. Naslednici klase ne mogu pristupati privatnim obeležjima pretka.
+	public	Javno obeležje – može mu se pristupati bez ograničenja.
#	protected	Zaštićeno obeležje – pristup ima klasa u kojoj je definisano i svi njeni naslednici
~	package	Obeležje je vidljivo na nivou paketa – mogu mu pristupati sve klase koje su u istom paketu sa klasom kojoj obeležje pripada.

Vidljivost obeležja može da se mapira na modifikatore pristupa u programskom kodu, ali i ne mora nužno da se poklapa sa njima - to zavisi od usvojene konvencije razvojnog tima i programskog jezika koji se koristi. Čest pristup je da se u programskom kodu sva obeležja implementiraju kao privatna, a da se njihova vidljivost na dijagramu klasa zapravo mapira na vidljivost modifikatora pristupa get i set metoda obeležja, pošto se oni obično ne modeluju. Na primer, ako je obeležje na modelu specificirano kao zaštićeno, to znači da se implementira kao privatno, ali da ima zaštićene get i set metode. Ako vidljivost obeležja nije specificirana, podrazumeva se da ima javne get i set metode.

### 4.2.2 Tip podataka

Tip podataka obeležja može biti:

- unapred definisan (npr. celobrojni – int; realni – real, float; logički – boolean; datumski – date i znakovni – string, char),
- nabrojani tip (enumeration),

- klasa i
- interfejs.

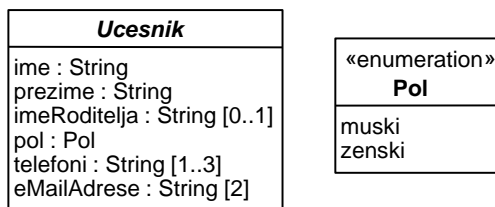
Na slici 4.7 vidimo klasu *Ucesnik* čija sva obeležja imaju tip *String*, sem obeležja *pol* koje ima nabrojani tip *Pol*. Nabrojani tipovi se koriste u situacijama kada je skup mogućih vrednosti obeležja unapred poznat i ne postoji mogućnost da će se menjati u budućnosti.

Modelovanje obeležja čiji je tip podataka klasa ili interfejs je obrađeno u sekcijama 4.4.1 i 4.4.3.

### 4.2.3 Kardinalitet

Kardinalitet omogućava da definišemo broj vrednosti određenog obeležja koji može ili mora da postoji u okviru instance klase. Ako se ne navede, podrazumeva se da je u pitanju 1 (obeležje sadrži tačno jednu vrednost koja ne sme biti nedefinisana).

Kardinalitet može da se zadaje kao interval (npr. 0..1, 3..5, 1..\*), ili kao jedan broj (npr. 1, 2, 3). Ako je u pitanju interval  $m..n$ , gde  $n$  mora biti  $\geq m$ , to znači da u okviru klase mora da postoji najmanje  $m$  vrednosti datog obeležja, a može da postoji najviše  $n$  vrednosti. Ako je kao kardinalitet zadat samo jedan broj, on definiše tačan broj vrednosti koje obeležje mora da ima (npr. 5 je ekvivalentno intervalu 5..5).



Slika 4.7 Definisanja kardinaliteta za obeležja klase *Ucesnik* iz sistema za automatsko testiranje studenata

U okviru klase *Ucesnik* na slici 4.7, *ime*, *prezime* i *pol* imaju podrazumevani kardinalitet 1. Obeležje *imeRoditelja* može, ali i ne mora imati unetu vrednost (kardinalitet 0..1). Obeležje *telefoni* ima kardinalitet 1..3, što znači da osoba mora imati najmanje 1 broj telefona, a najviše 3. Obeležje *eMailAdrese* ima kardinalitet 2, što znači da osoba mora imati unete tačno dve e-mail adrese. Konvencija je da se imena obeležja čiji je kardinalitet veći od 1 pišu u množini.

```

public enum Pol {
    MUSKI, ZENSKI
}

public abstract class Ucesnik {
    private String ime;
    private String prezime;
    private String imeRoditelja;
    private Pol pol;
    private String telefon1;
    private String telefon2;
    private String telefon3;
    private String eMailAdresa1;
    private String eMailAdresa2;

    public String getIme() {
        return ime;
    }

    //Set metoda za obeležje sa kardinalitetom 1
    public void setIme(String ime) {
        if (null == ime)
            throw new NullPointerException();
        ime = ime.trim();
        if (!"".equals(ime))
            this.ime = ime;
        else
            throw new IllegalArgumentException();
    }

    public String getImeRoditelja() {
        return imeRoditelja;
    }

    //Set metoda za obeležje sa kardinalitetom 0..1
    public void setImeRoditelja(String imeRoditelja) {
        this.imeRoditelja = imeRoditelja;
    }
    ...
}

```

Listing 4.1 Implementacija nabrojane vrednosti Pol i dela klase Ucesnik sa slike 4.7

**Napomena:** U programskom kodu se obeležja sa fiksnim kardinalitetom većim od 1 (npr. 1..3, 2) obično implementiraju kao više odvojenih obeležja, sa istim imenom i brojevnim sufiksima, radi lakšeg rukovanja pojedinačnim vrednostima (listing 4.1).

Set metodu obeležja sa kardinalitetom 1 treba tako implementirati da ne dozvoli unos „prazne“ vrednosti za dato obeležje. Na listingu 4.1 vidimo da metoda setIme ne dozvoljava da se za obeležje koje je tipa String unese null vrednost, niti prazan string. Kod set metode za obeležje imeRoditelja koje ima kardinalitet 0..1 to nije slučaj.

DnevnaTemperatura
datum : Date izmereneVrednosti : float [*] minVrednost : float maxVrednost : float
+srednjaVrednost() +dodajTemperaturu()

Slika 4.8 Definisanja kardinaliteta obeležja za klasu DnevnaTemperatura

U klasi DnevnaTemperatura (slika 4.8), obeležje izmereneVrednosti ima kardinalitet \*, čime se specificira da u okviru instance klase može da postoji proizvoljan broj vrednosti tog obeležja (nula, jedan ili više). Klasa DnevnaTemperatura se koristi za merenje dnevnih temperatura na jednom mernom mestu, a namena obeležja izmereneVrednosti je da skladišti sve vrednosti koje neki senzor izmeri u toku jednog dana.

**Napomena:** U programskom kodu se obeležje sa kardinalitetom \* implementira kao dinamička kolekcija. Za takva obeležja se preporučuje da se ne izlažu direktno spoljnom svetu, davanjem reference na kolekciju u get metodi, već da se implementiraju metode koje omogućavaju kontrolisan rad sa elementima kolekcije (listing 4.3):

- metoda za dodavanje, da bi se sprečio unos nedozvoljenih vrednosti
- metoda za brisanje elemenata, ako je brisanje dozvoljeno (za klasu DnevnaTemperatura to nije slučaj)
- get metoda treba da vraća zaštićenu (*readonly*) kopiju kolekcije koja dozvoljava pregled elemenata, ali ne i izmenu.

Kolekcije ne bi trebalo da imaju set metodu, osim u nekim veoma specifičnim slučajevima. Preporuka je da se kolekcije ne kreiraju odmah, prilikom instanciranja klase, već samo ako se ispostavi da je kolekcija zaista potrebna (tzv. „*lazy*“ inicijalizacija). Time se sprečava nepotrebno trošenje operativne memorije, u slučajevima kada kolekcije nemaju nijedan element.

Za kreiranje zaštićenih kopija različitih vrsta kolekcija na programskom jeziku Java se mogu koristiti metode klase `java.util.Collections` (listing 4.2).

```
Collections.unmodifiableCollection  
Collections.unmodifiableList  
Collections.unmodifiableSet  
Collections.unmodifiableMap  
Collections.unmodifiableSortedMap  
Collections.unmodifiableSortedSet
```

Listing 4.2 Metode klase `java.util.Collections` za kreiranje zaštićenih kopija različitih vrsta kolekcija na programskom jeziku Java

```
public class DnevnaTemperatura {  
    ...  
    private List<Float> izmereneVrednosti = null;  
  
    public List<Float> getIzmereneVrednosti() {  
        if (null == izmereneVrednosti)  
            return null;  
        return Collections.unmodifiableList(izmereneVrednosti);  
    }  
  
    public void dodajTemperaturu(float temperatura) {  
        //”lazy” inicijalizacija kolekcije  
        if (null == izmereneVrednosti) {  
            izmereneVrednosti = new ArrayList<Float>();  
        }  
        //... obavljanje različitih provera  
        // dodavanje vrednosti u kolekciju ako je sve u redu:  
        izmereneVrednosti.add(temperatura);  
    }  
    ...  
}
```

Listing 4.3 – Primer implementacije kolekcije `izmereneVrednosti` sa slike 4.8 i metoda koje obezbeđuju kontrolisani pristup njenim elementima

#### 4.2.4 Podrazumevana vrednost

Podrazumevana (*default*) vrednost je vrednost koju obeležje dobija prilikom instanciranja klase.

Odgovor
tekstOdgovora : String tacan : Boolean = false nosiPoena : Integer = 0

Slika 4.9 Klasa `Odgovor` – ilustracija definisanja podrazumevanih vrednosti

Obeležja `tacan` i `nosiPoena` na slici 4.9 imaju specificirane inicijalne vrednosti `false` i `0`, respektivno, iz razloga što u testovima obično ima više netačnih nego tačnih odgovora i netačni odgovori obično ne nose negativne poene. Na ovaj način se nastavnicima skraćuje posao pri kreiranju testova.

## 4.2.5 Statička obeležja

Statička obeležja (obeležja kojima se pristupa na nivou klase, ne instance klase) se označavaju podvlačenjem. Mogu se koristiti za skladištenje vrednosti koja je potrebna svim instancama neke klase (slika 4.10), za implementaciju globalno dostupnih resursa (slika 4.11), za specifikaciju konstanti (slika 4.12) i sl.

Element
-poslednjiBrojElementa : Integer = 0 idElementa : Integer {id, readOnly}

Slika 4.10 Klasa `Element` – `poslednjiBrojElementa` je privatno statičko obeležje

Na slici 4.10 je prikazana klasa `Element` koja u statičkom obeležju `poslednjiBrojElementa` čuva identifikacioni broj elementa (`id`) koji je poslednji dodeljen. `poslednjiBrojElementa` nema `get` i `set` metode i jedini način da mu se očita i promeni vrednost je u okviru konstruktora klase. Na ovaj način smo sigurni da će svaka instanca klase `Element` dobiti jedinstveni identifikacioni broj (listing 4.4).

```
public class Element {  
    private static int poslednjiBrojElementa = 1;  
    private int idElementa;  
  
    public Element() {  
        idElementa = poslednjiBrojElementa++;  
    }  
  
    public int getIdElementa() {  
        return idElementa;  
    }  
}
```

Listing 4.4 Korišćenje statičkog obeležja za dodelu jedinstvenog identifikacionog broja

Klasa `Konekcija` na slici 4.11 modeluje konekciju ka relacionoj bazi podataka. Pošto jedna aplikacija treba da otvori jednu konekciju koju će koristiti svi njeni delovi koji pristupaju bazi, navedena klasa je implementirana korišćenjem `Singleton` projektnog šablona [Gamma04]. Konstruktor klase je privatn, tako da niko „spolja“ ne može da je instancira. Za kreiranje i dobavljanje jedne instance je zadužena statička metoda `getKonekcija()`. Instanca se čuva u privatnom statičkom obeležju `konekcija` (listing 4.5).

Konekcija
konekcija : Konekcija{readOnly}
+getKonekcija() : Konekcija «constructor»-Konekcija()

Slika 4.11 Klasa Konekcija – primer implementacije Singleton projektnog šablona

```
public class Konekcija {
    private static Konekcija konekcija = null;

    private Konekcija() {
        // kod za otvaranje konekcije prema bazi podataka
    }

    public static Konekcija getKonekcija() {
        if (konekcija == null)
            konekcija = new Konekcija();
        return konekcija;
    }
}
```

Listing 4.5 Implementacija Singleton projektnog šablona na primeru klase Konekcija

**Napomena:** Prednost Singleton šablona je što omogućava da se na kontrolisan način implementira resurs kome svi mogu da pristupe, kao u slučaju konekcije ka bazi podataka. Mana je što omogućava „prečice“ u projektovanju, što za posledicu može imati lošu arhitekturu. Umesto pažljivog projektovanja enkapsulacije i saradnje klasa, tako da svaka klasa pristupa samo onome što joj je neophodno, korišćenjem Singleton obrasca se svakoj klasi obezbeđuje globalni pristup određenim resursima aplikacije, bilo da su joj potrebni ili nisu. Ako se nekritički primenjuje, ovim se može povećati uzajamna zavisnost klasa i modula, što može dovesti do težeg održavanje koda i lakšeg uvođenja grešaka. Generalna preporuka je da se Singleton ne koristi, ako postoji drugi način da se postigne isti efekat.

U klasi DnevnaTemperatura na slici 4.12, statička obeležja `min_temp` i `max_temp` su iskorišćena za definisanje konstanti – čuvaju minimalnu i maksimalnu temperaturu za koje se smatra da se mogu izmeriti u regionu gde se instaliraju date merne stanice (listing 4.6).

#### 4.2.6 Dodatne opcije

Dodatne opcije obeležja omogućavaju da se razvojnom timu ili generatoru koda daju precizne instrukcije kako programski kod treba da se implementira, na osnovu analize problemskog domena i zahteva korisnika (tabela 4.2).



Tabela 4.2 Dodatne opcije obeležja klase

Opcija	Značenje
id	Obeležje je jedinstveni identifikator ili deo jedinstvenog identifikatora klase. Ako se instance date klasa smeštaju u kolekciju, sortiranje u kolekciji i sprečavanje pojave duplikata se vrši na osnovu vrednosti ovog obeležja. Ako se koristi relacionala baza podataka, id označava obeležje koje treba da se mapira na primarni ključ ili deo primarnog ključa tabele na koju se klasa mapira. Primer korišćenja je dat na slici 4.10.
readOnly	Obeležje ne može da se menja posle inicijalizacije. U širem smislu, ovom opcijom se može specificirati da obeležje ne sme imati set metodu. Primer korišćenja je dat na slikama 4.10, 4.11 i 4.12.
derived	Obeležje je izvedeno (njegova vrednost se računa na osnovu vrednosti drugih obeležja). Izvedeno obeležje se označava znakom „/“ ispred imena (slika 4.13).
ordered	ordered se može primenjivati na obeležja koja se mapiraju na kolekciju u programskom kodu (čiji je kardinalitet *). Specificira da kolekcija mora biti uređena, odnosno da je bitan redosled unosa elemenata u kolekciju (slika 4.12).
unique	unique se može primenjivati na obeležja koja se mapiraju na kolekciju. Specificira da ne može postojati više elemenata sa istom vrednošću u kolekciji.
nonunique	nonunique se može primenjivati na obeležja koja se mapiraju na kolekciju. Specificira da može postojati više elemenata sa istom vrednošću u kolekciji (slika 4.12).

Na slici 4.12 je prikazana detaljna specifikacija klase `DnevnaTemperatura` čija je implementacija data na listingu 4.6. Pošto su sva obeležja sa kardinalitetom 1 označena kao `readOnly`, na listingu možemo primetiti da nemaju implementirane `set` metode. Obeležje `izmereneVrednosti` je u programskom kodu implementirano kao `ListArray`, zbog opcije `ordered` koja specificira da je bitno

očuvati redosled unosa elemenata u kolekciju. Metoda dodajTemperaturu ne proverava da li element sa istom vrednošću već postoji, jer je nonunique opcijom specificirano da može postojati više elemenata sa istom vrednošću.

DnevnaTemperatura
«const»max_vrednost : float = 50.0 «const»min_vrednost : float = -50.0 datum : Date{readOnly} izmereneVrednosti : float [*] = null{ordered,nonunique} minVrednost : float = max_vrednost{readOnly} maxVrednost : float = min_vrednost{readOnly}
+srednjaVrednost() +dodajTemperaturu( float temp )

Slika 4.12 Detaljna specifikacija klase DnevnaTemperatura

Na slici 4.13 je dat primer specifikacije izvedenog obeležja vrednost, koje se računa kao količina \* cena. Možemo primetiti da ima oznaku „/“ ispred imena.

Racun
brojRacuna : int{readOnly,id} kolicina : int cena : int{readOnly} /vrednost : int{readOnly}

Slika 4.13 Primer specifikacije izvedenog obeležja: vrednost je izvedeno obeležje koje se računa kao kolicina \* cena

Srednja vrednost dnevne temperature na slici 4.12 je takođe mogla da se modeluje kao izvedena vrednost. U tom slučaju ne bismo u modelu posebno navodili metodu srednjaVrednost().

#### 4.2.7 Stereotipi

Na slici 4.12 možemo primetiti i korišćenje stereotipa <<const>>, kao specifikaciju programerima da statička obeležja max\_vrednost i min\_vrednost treba da budu implementirana kao konstante. Ovaj stereotip nije deo UML-a. Razvojni tim može uvesti proizvoljan broj stereotipa i konvenciju kako se mapiraju na programski kod, da bi mogle da se precizno modeluju situacije koje nisu predviđene postojećim elementima UML-a.

```

public class DnevnaTemperatura {
    public static final float MAX_TEMP = 50;
    public static final float MIN_TEMP = -50;
    private Date datum;
    private List<Float> izmereneVrednosti = null;
    private float minVrednost = MAX_TEMP;
    private float maxVrednost = MIN_TEMP;

    public DnevnaTemperatura() {
        datum = new Date(); // današnji datum
    }

    public Date getDatum() {
        return datum;
    }

    public List<Float> getIzmereneVrednosti() {
        if (null == izmereneVrednosti)
            return null;
        return Collections.unmodifiableList(izmereneVrednosti);
    }

    public float getMinVrednost() {
        return minVrednost;
    }

    public float getMaxVrednost() {
        return maxVrednost;
    }

    public void dodajTemperaturu(float temperatura) {
        if (null == izmereneVrednosti) {
            izmereneVrednosti = new ArrayList<Float>();
        }
        if (temperatura > MAX_TEMP || temperatura < MIN_TEMP)
            throw new IllegalArgumentException(String.format("Temperatura mora biti u opsegu [%f, %f]", MIN_TEMP, MAX_TEMP));
        if (temperatura < minVrednost)
            minVrednost = temperatura;
        if (temperatura > maxVrednost)
            maxVrednost = temperatura;
        izmereneVrednosti.add(temperatura);
    }

    public float srednjaVrednost() {
        if (null == izmereneVrednosti)
            throw new NullPointerException();
        int brojElemenata = izmereneVrednosti.size();
        float suma = 0;
        for (int i = 0; i < brojElemenata; i++) {
            suma += izmereneVrednosti.get(i);
        }
        return suma / brojElemenata;
    }
}

```

Listing 4.6 Implementacija klase DnevnaTemperatura sa slike 4.12.

## 4.3 Modelovanje metoda

Nazivi metoda klasa i njihovih parametara treba da se pišu po istoj konvenciji kao i nazivi obeležja klasa (*Camel Case* konvencija) i da daju jasnu informaciju o nameni metode, odnosno parametara. Primer davanja imena metodama: `ocitajVrednost`, `stampaPlatnihListica`, `izvestajORadnicima`, itd.

Format za specifikaciju metoda u okviru klase je sledeći:

```
vidljivost naziv (lista-parametara): tip-povratne-vrednosti  
{dodatna-podešavanja}
```

Vidljivost metode se specificira kao kod obeležja (sekcija 4.2.1).

Lista parametara može biti prazna. Ako metoda ima parametre, format za specifikaciju jednog parametra je sledeći:

smer naziv-parametra: tip kardinalitet = podrazumevana-vrednost

Smer može biti: `in` (ulazni parametar), `out` (izlazni parametar) `inout` (ulazno-izlazni parametar). Ako se ne navede smer, podrazumeva se da je u pitanju ulazni parametar. Sve ostale osobine parametra (naziv, tip podataka, kardinalitet, podrazumevana vrednost) specificiraju se kao kod obeležja klase (sekcija 4.2).

Primeri specifikacije metoda klase su:

```
+stampaPozajmica(odDatum: date, doDatum: date): void  
+suma(): int  
+statistika(out min: float, out max: float, out srednjaVrednost:  
float): void  
+dodajVrednosti(vrednosti int[0..10]): void
```

Takođe, može se specificirati da li je metoda statička. Statičke metode su podvučene u okviru klase, kao i statička obeležja (slika 4.11).

Apstraktne metode, za koje ne želimo da obezbedimo implementaciju, označavaju se zakošenim slovima.

Pri modelovanju metoda, treba se pridržavati pravila dobre prakse:

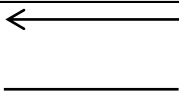
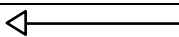

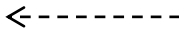
- Metodu staviti u onu klasu koja ima obeležja nad kojima metoda radi. Ako metoda većinu vremena radi nad obeležjima neke druge klase kojoj iz svoje klase pristupa preko reference, trebalo bi je premestiti u tu drugu klasu.

- Ako metoda ima povratnu vrednost („funkcija“), ne očekuje se da menja stanje sistema (tj. vrednosti obeležja svoje ili neke druge klase).
- Ako metoda nema povratnu vrednost (void metoda, „procedura“), očekuje se da može menjati stanje sistema.
- Ako metoda implementira neki složeni algoritam za koji je potreban veliki broj programskih linija, podeliti je na više manjih metoda od kojih svaka obavlja jedan jasno definisan deo algoritma. Ovim se povećava čitljivost koda i olakšava testiranje i održavanje. Generalna preporuka je da programski kod metode ne bi trebalo da pređe veličinu jednog ekrana.

## 4.4 Veze

Na dijagramu klasa se najčešće koriste vrste veza koje su navedene u tabeli 4.3. U nastavku su opisana pravila za njihovo korišćenje na dijagramima i način preslikavanja na programski kod.

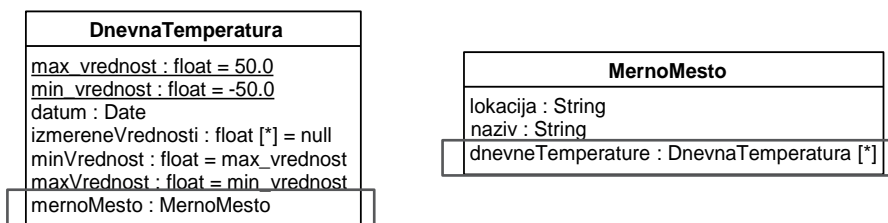
Tabela 4.3 Vrste veza koje se najviše koriste na dijagramu klasa

Vrsta veze	Ilustracija
Asocijacija	
Generalizacija	
Implementacija interfejsa	
Veza zavisnosti	

### 4.4.1 Veza asocijacije

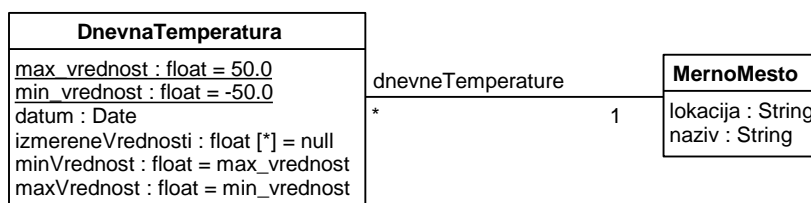
U sekciji 4.2.3 je uvedena klasa DnevnaTemperatura koja se koristi za merenje dnevnih temperatura na jednom mernom mestu. Da bi podaci koje meteo-rološka služba prikuplja bili potpuni, potrebno je znati i na kojem mernom mestu se vrši merenje. Zbog toga uvodimo klasu MernoMesto koja ima kolekciju instanci klase DnevnaTemperatura (obeležje dnevneTemperature), slika 4.14. Merno mesto ima u kolekciji po jednu instancu klase DnevnaTemperatura za svaki dan kada je vršeno merenje. U klasu DnevnaTemperatura uvodimo obeležje mernoMesto koje je referenca na merno mesto kojem očitane temperature pripadaju.

Navedene klase se uzajamno referenciraju, ali se to na slici 4.14 teško primećuje - moramo pažljivo pogledati tip svakog obeležja.



Slika 4.14 Direktna specifikacija obeležja čiji je tip referenca na klasu

Mada je navedeni dijagram ispravan sa stanovišta sintakse UML-a, bolji način modelovanja je da se u ovakvim situacijama koristi *veza asocijacije* (slika 4.15). Veze asocijacije pomažu da se bolje sagleda struktura sistema i zavisnosti između njegovih delova.



Slika 4.15 Korišćenje asocijacije umesto direktne specifikacije obeležja čiji je tip referenca na klasu

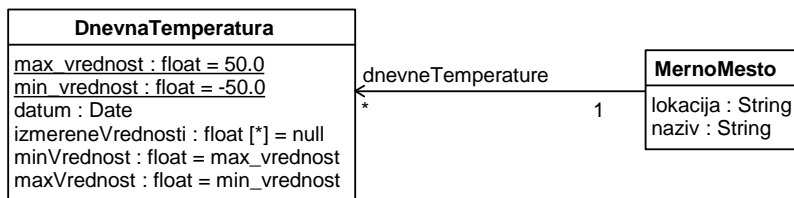
```

public class MernoMesto {
    private String lokacija;
    private String naziv;
    private List<DnevnaTemperatura> dnevneTemperature;
    ...

    public class DnevnaTemperatura {
        public static final float MAX_TEMP = 50;
        public static final float MIN_TEMP = -50;
        private Date datum;
        private List<Float> izmereneVrednosti = null;
        private float minVrednost = MAX_TEMP;
        private float maxVrednost = MIN_TEMP;
        private MernoMesto mernoMesto;
        ...
    }
  
```

Listing 4.7 Implementacija obeležja klasa MernoMesto i DnevnaTemperatura sa slika 4.14 i 4.15.

Sa stanovišta implementacije, modeli na slikama 4.14 i 4.15 su ekvivalentni (listing 4.7), osim što je model na slici 4.15 čitljiviji.



Slika 4.16 Asocijacija je navigabilna u jednom smeru – samo MernoMesto „vidi“ klasu DnevnaTemperatura

U slučaju kada nam je potrebno da samo jedna klasa referencira drugu (npr. želimo da samo MernoMesto ima listu referenci na klasu DnevnaTemperatura), asocijacija dobija strelicu koja specificira da je asocijacija *navigabilna* od klase MernoMesto prema klasi DnevnaTemperatura (slika 4.16). U žargonu se kaže da klasa MernoMesto „vidi“ klasu DnevnaTemperatura. Zbog smera asocijacije, DnevnaTemperatura „ne vidi“ MernoMesto. Ako asocijacija nema strelicu, kao na slici 4.15, podrazumeva se da je navigabilna u oba smera, odnosno bidi-rekciona. Tada se klase uzajamno „vide“.

```

public class MernoMesto {
    private String lokacija;
    private String naziv;
    private List<DnevnaTemperatura> dnevneTemperature;
    ...

    public class DnevnaTemperatura {
        public static final float MAX_TEMP = 50;
        public static final float MIN_TEMP = -50;
        private Date datum;
        private List<Float> izmereneVrednosti = null;
        private float minVrednost = MAX_TEMP;
        private float maxVrednost = MIN_TEMP;
        ...
    }
}
    
```

Listing 4.8 Implementacija obeležja klasa MernoMesto i DnevnaTemperatura sa slike 4.16. Asocijacija je navigabilna u jednom smeru, tako da DnevnaTemperatura nema referencu na MernoMesto.

Primitimo da na slici 4.15 u klasama nisu navedena obeležja mernoMesto i dnevneTemperature sa slike 4.14. Ona se izvode na osnovu osobina suprotnih *krajeva asocijacije*. Krajevi asocijacije se zovu uloge (*role*). Klasa ima obeležje koje je tipa klase na suprotnom kraju, pod uslovom da „vidi“ tu klasu. Kardinalitet tog obeležja odgovara kardinalitetu navedenom na suprotnom kraju asocijacije. Ako kardinalitet nije naveden, podrazumeva se da je 1.

Naziv obeležja odgovara nazivu uloge suprotnog kraja asocijacije, ako je taj naziv naveden (npr. `dnevneTemperature: DnevnaTemperatura[ * ]`). Ako naziv uloge nije naveden, obeležje dobija ime kao klasa na suprotnom kraju, sa malim početnim slovom (npr. `mernoMesto: MernoMesto`). Vidljivost se takođe može naznačiti. Ako nije naznačena, podrazumeva se `private`.

Sada možemo sa dijagrama na slici 4.4, prikazanog u uvodnoj sekciji, pročitati da klase `Test`, `Pitanje` i `Odgovor` imaju obeležja kao na listingu 4.9.

```
public class Test {  
    private String nazivTesta;  
    private int ukupnoPoena;  
    private List<Pitanje> pitanja = null;  
    ...  
  
    public class Pitanje {  
        private String tekstPitanja;  
        private List<Odgovor> odgovori = null;  
        private Test test;  
        ...  
  
        public class Odgovor {  
            private String tekstOdgovora;  
            private Boolean tacan = false;  
            private int nosiPoena = 0;  
            private Pitanje pitanje;  
            ...  
        }  
    }  
}
```

Listing 4.9 Implementacija obeležja klase `Test`, `Pitanje` i `Odgovor` sa slike 4.4, gde su korišćene navigabilne asocijacije u oba smera

Na dijagramu sa slike 4.5, gde su korišćene navigabilne asocijacije u jednom smeru, primećujemo da `test` „vidi“ svoja pitanja, kao i pitanja odgovore, ali da obrnuto nije slučaj (implementacija obeležja je prikazana na listingu 4.10).

**Napomena:** Ako je asocijacija navigabilna u oba smera, lakše i brže je kretanje po grafu koji formiraju objekti (instance) tih klasa kada aplikacija počne da se izvršava. Međutim, kreiranje i brisanje objekata se usložnjava i postaje podložnije greškama zbog potrebe održavanja uzajamnog referenciranja. Iz navedenog razloga, preporuka je da se asocijacije inicijalno modeluju da budu navigabilne u jednom smeru, koji u tom trenutku izgleda logično. Kasnije, detaljnom analizom slučajeva korišćenja koje softver treba da podrži, neke veze će biti promenjene u bidirekzione, radi efikasnijeg izvršavanja određenih algoritama i njihove jednostavnije implementacije. Implementacija klase i metoda za podršku rada sa asocijacijama se detaljnije obrađuje u sekciji 4.4.1.5.



```

public class Test {
    private String nazivTesta;
    private int ukupnoPoena;
    private List<Pitanje> pitanja = null;
    ...

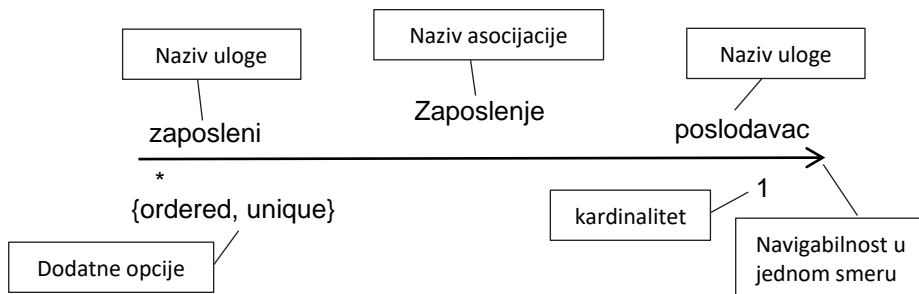
    public class Pitanje {
        private String tekstPitanja;
        private List<Odgovor> odgovori = null;
        ...

        public class Odgovor {
            private String tekstOdgovora;
            private Boolean tacan = false;
            private int nosiPoena = 0;
            ...

```

Listing 4.10 Implementacija obeležja klasa Test, Pitanje i Odgovor sa slike 4.5, gde su korišćene navigabilne asocijacije u jednom smeru

Da rezimiramo, na slici 4.17 su prikazane najvažnije osobine asocijacije i krajeva asocijacije (naziv asocijacije, naziv uloge, kardinalitet, navigabilnost, dodatne opcije). Naziv asocijacije se piše po pravilima za pisanje imena klase i nije ga obavezno navesti, kao ni ostale osobine. Naziv uloge se piše po pravilima za pisanje obeležja, pošto se na osnovu njih imenuju obeležja koja se preuzimaju preko navigabilnih krajeva asocijacije. Naziv asocijacije se navodi u situacijama kada se iz naziva uloga ne može razumeti svrha asocijacije, ili ako nazivi uloga nisu navedeni.

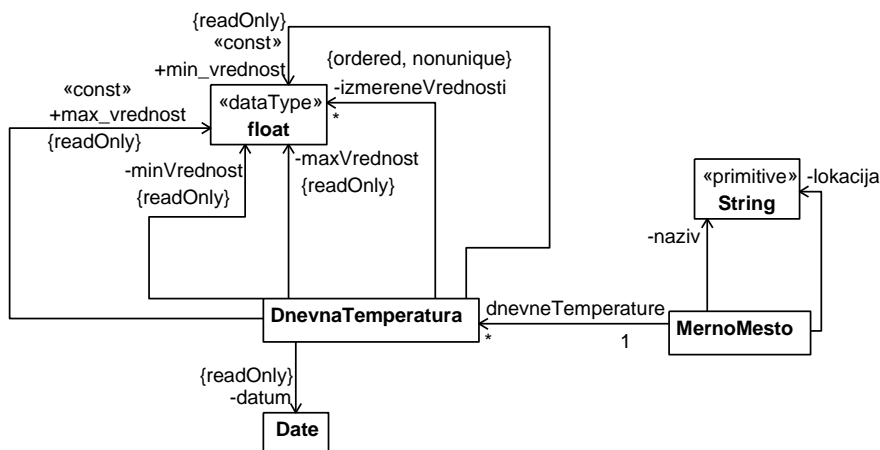


Slika 4.17 Osnovne osobine asocijacije

Primećujemo da se sve osobine koje može imati obeležje mogu specificirati i pri modelovanju krajeva asocijacije. Po UML sintaksi, klase se mogu prikazati i kao na slici 4.18, ali se to ne primenjuje – modeli bi postali izrazito nečitki.

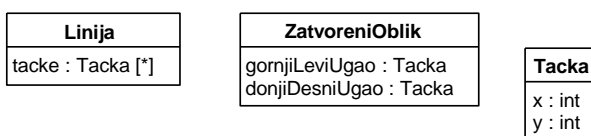
Preporuka koja se tiče „pravila lepog ponašanja“ pri modelovanju je:

1. Obeležja koja su unapred definisanih ili nabrojanih tipova navode se direktno, u okviru klase.
2. Obeležja čiji su tipovi reference na klase se navode korišćenjem asocijacija.



Slika 4.18 Sva obeležja klase sa dijagrama na slici 4.16. su modelovana preko asocijacija. Ovakav način modelovanja se ne preporučuje!

Od druge preporuke se može odstupiti, ako su u pitanju obeležja čiji je tip mala, pomoćna klasa koja se često koristi. Takva obeležja se direktno navode u okviru klase, kao da imaju unapred definisani tip, da se model ne bi zagušio velikim brojem asocijacija. Primer je dat na slici 4.19, gde se klasa Tacka navodi direktno kao tip obeležja različitih vrsta geometrijskih oblika.



Slika 4.19 Primer kada je opravdano direktno navoditi klasu kao tip obeležja - videti obeležja u okviru klase Linija i ZatvoreniOblik čiji je tip klasa Tacka

#### 4.4.1.1 Izolovanje asocijacija iz specifikacije zahteva

Kada slušamo zahteve naručioca ili čitamo tekst specifikacije zahteva, kandidati za asocijacije su glagoli u rečenicama:

Preduzeće **se sastoji** od sektora.

Student **bira** predmete.

Član biblioteke **iznajmljuje** primerke knjige.

Dnevne temperature **se mere** na mernom mestu.

Osoba **ima** mesto stanovanja.

Uslov da se na osnovu glagola kreira asocijacija je da su imenice između kojih se nalazi glagol nazivi klasa, a ne nazivi klase i njenog obeležja koje je unapred definisanog tipa. Na primer, u ovom slučaju nećemo kreirati asocijacije (videti sliku 4.18):

Osoba **ima** ime, prezime i datum rođenja.

Pored toga, uslov je i da je asocijacija potrebna za podršku nekog korisničkog zahteva.

Pri određivanju kardinaliteta, potrebno je saznati koja je donja i gornja granica oba kraja asocijacije. Na primer:

Koliko test može imati pitanja? Da li pitanje može da pripada samo jednom testu ili se može uključiti u više testova? Da li pitanje mora da pripada testu (tj. da li može postojati nezavisno od testa)?

Razmotrimo primer 4.1 dat u nastavku.

---

**Primer 4.1** *Potrebno je kreirati konceptualni model za deo informacionog sistema fakulteta. Fakultet se sastoji od departmana. Na fakultetu su zaposleni nastavnici. Podaci koje je potrebno voditi o fakultetu su: naziv fakulteta, matični broj, web adresa, rukovodilac (dekan). Dekan je nastavnik koji je zaposlen na fakultetu. Podaci koje je potrebno voditi o departmanu su: naziv i telefon. Podaci koje je potrebno voditi o nastavniku su: ime, prezime, datum rođenja, telefon.*

---

Primećujemo da se Nastavnik, Fakultet i Departman mogu modelovati kao klase. Izdvajamo rečenice iz specifikacije koje nam daju kandidate za asocijacije:

Fakultet **se sastoji** od departmana.

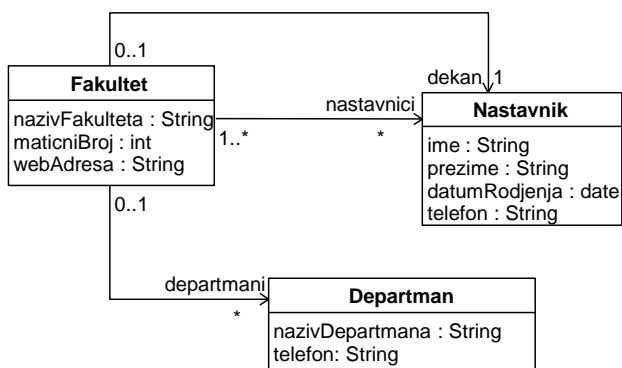
Na fakultetu **su zaposleni** nastavnici.

Da bismo mogli da ih precizno modelujemo, treba od naručioca da dobijemo sledeće odgovore:

- Da li departman pripada jednom fakultetu ili može biti u okviru više njih?  
*Može biti u okviru jednog fakulteta.*
- Da li departman mora pripadati nekom fakultetu?  
*Ne – može biti i u okviru univerziteta.*
- Da li nastavnik može da radi na više fakulteta?  
*Da.*

- Da li nastavnik mora da radi na nekom fakultetu?  
*Da, inače nije nastavnik.*
- Da li jedan nastavnik može biti dekan više fakulteta?  
*Ne.*

Na osnovu inicijalne specifikacije i dobijenih odgovora, modelujemo konceptualni dijagram klasa na slici 4.20.



Slika 4.20 Konceptualni dijagram klasa za primer 4.1

Obeležja klase Fakultet su prikazana na listingu 4.11. Pošto u ovakvim sistemima nije dozvoljeno postojanje duplikata, a redosled unosa nije bitan, objekti klase Departman i Fakultet se smeštaju u kolekciju koja implementira Set interfejs.

```

public class Fakultet {
    private String nazivFakulteta;
    private int maticniBroj;
    private String webAdresa;
    private Set<Nastavnik> nastavnici = null;
    private Nastavnik dekan;
    private Set<Nastavnik> departmani = null;
    ...
}
  
```

Listing 4.11 Implementacija obeležja klase Fakultet. Klase Departman i Nastavnik imaju samo obeležja predefinisanih tipova, pošto ne „vide“ klasu Fakultet.

#### 4.4.1.2 Agregacija i kompozicija

Pored osobina asocijacije navedenih na slici 4.17, na dijagramima klasa se mogu primetiti i asocijacije koje imaju prazan ili popunjen romb na jednom kraju.

Na slici 4.21 klase su povezane asocijacijom koja se zove *agregacija*. Agregacija je *vrsta asocijacije* kojom se modeluje odnos „celina-deo“ između dve klase. Na

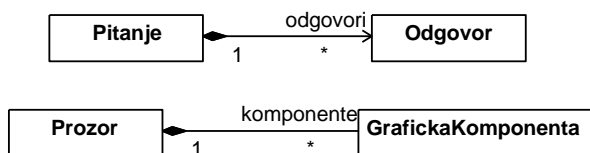
slici 4.21 vidimo da se računar sastoji od delova a razvojni timovi od programera. Romb se stavlja kod klase koja predstavlja celinu.



Slika 4.21 Primeri agregacije

Kod agregacije, klase na oba kraja su „ravnopravne“ – celina se formira grupisanjem (agregiranjem) delova, ali delovi mogu postojati i bez celine. Na primer, deo ne mora nužno pripadati nekom računaru (otuda kardinalitet 0..1). Ako se računar pokvari, njegovi ispravni delovi ostaju i mogu se ugraditi u druge računare. Pri tome, deo može pripadati najviše jednom računaru. Ako ne pripada nijednom, znači da je slobodan i čeka na ugradnju.

Razvojni timovi se sastoje od programera. Programer može istovremeno raditi u okviru više timova. Kada neki tim prestane da postoji, programeri nastavljaju da rade u drugim timovima. Na slici 4.21 vidimo da je kardinalitet agregacije „više prema više“ (\* na oba kraja) i da je bidirekciona (programer „vidi“ timove u kojima učestvuje i timovi „vide“ svoje članove).



Slika 4.22 Primeri kompozicije

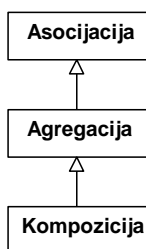
*Kompozicija je vrsta agregacije* - njome takođe modelujemo odnos „celina-deo“, ali klase nisu „ravnopravne“: delovi ne mogu postojati bez celine, a njihov životni ciklus je čvrsto povezan (slika 4.22). Klasa koja modeluje celinu kreira svoje delove i njihov je jedini vlasnik. Kada se celina briše, brišu se i njeni delovi. Kardinalitet na strani klase koja predstavlja celinu je uvek 1, tako da je greška korišćenje drugačijeg kardinaliteta. Kompozicija je obično usmerena od celine ka delovima.

Na slici 4.22 vidimo da se prozor sastoji od grafičkih komponenti (polja za unos, dugmići i sl). Objekat klase Prozor instancira komponente od kojih se sastoji. Kada se prozor zatvori i prestane da postoji, oslobađaju se i grafičke komponente.

te koje su mu pripadale. Slična je situacija i sa drugim primerom sa slike 4.22. Pitanje se sastoji od odgovora; kada se briše pitanje, brišu se i svi njegovi odgovori.

Kod programskih jezika kao što je C++, gde programeri moraju da paze na oslobađanje zauzete memorije, u destruktora klasa koje predstavljaju celinu je obavezno implementirati oslobađanje (dealokaciju) delova. Prilikom uklanjanja dela iz kolekcije delova, poziva se i oslobađanje memorije koju zauzima dati deo. Kod agregacije to nije slučaj – deo nastavlja da „živi“ i posle brisanja celine, a verovatno je „živeo“ i pre kreiranja celine, samostalno ili u okviru nekog drugog sklopa.

**Napomena:** Agregacija je vrsta (specijalizacija) asocijacije. Kompozicija je vrsta (specijalizacija) agregacije. Njihov odnos bi se mogao modelovati kao na slici 4.23.



Slika 4.23 Odnos asocijacije, agregacije i kompozicije: Agregacija je vrsta (specijalizacija) asocijacije. Kompozicija je vrsta (specijalizacija) agregacije.

Dakle, agregacija i kompozicija imaju sve osobine osnovne asocijacije. Ako nismo sigurni koju vrstu asocijacije treba da upotrebimo (nejasna specifikacija zahteva, nije moguće dobiti sve detalje), bolje je koristiti osnovnu asocijaciju, nego pogrešno upotrebiti njene specijalizacije. U tom smislu, nije greška što je upotrebljena asocijacija za modelovanje odnosa fakulteta i departmana na slici 4.20, iako bi model bio semantički bogatiji da je upotrebljena agregacija. Slično važi i za modelovanje odnosa klasa Test, Pitanje i Odgovor u uvodnoj sekciji – bilo bi preciznije da smo upotrebili kompoziciju, ali je model korektan i sa „običnim“ asocicijama.

Česta greška kod modelovanja je da se agregacija ili kompozicija koriste u situacijama koje nisu „celina-deo“, odnosno, gde se ne može iskoristiti glagol „sastoji se od“. U sledećim primerima bi bilo pogrešno koristiti agregaciju:

Departman **ima** studente.  
Preduzeće **ima** zaposlene i robu.

Studenti nisu delovi departmana, niti su zaposleni i roba delovi preduzeća, tako da treba koristiti osnovnu asocijaciju. Agregacija se može koristiti u sledećim primerima:

Departman **se sastoji od** katedri (ima katedre).  
Preduzeće **se sastoji od** poslovnica (ima poslovnice).

Ako smo sigurni da je u pitanju odnos „celina-deo“, agregaciju od kompozicije možemo razlikovati postavljanjem pitanja 1 ili 2:

1. Da li deo može postojati bez celine?
  - da – agregacija,
  - ne – kompozicija.
2. Šta se dešava sa delovima kada se briše celina?
  - brišu se i delovi – kompozicija,
  - delovi nastavljaju da postoje – agregacija.

Neki autori preporučuju da se agregacija uopšte ne koristi jer nema efekta na programski kod, a njeno korišćenje, zbog nejasne semantike i čestih grešaka, može uneti konfuziju u model, umesto da ga učini preciznijim.

#### 4.4.1.3 Asocijativne klase

Razmotrimo primer 4.2, kao dodatak specifikaciji zahteva iz primera 4.1.

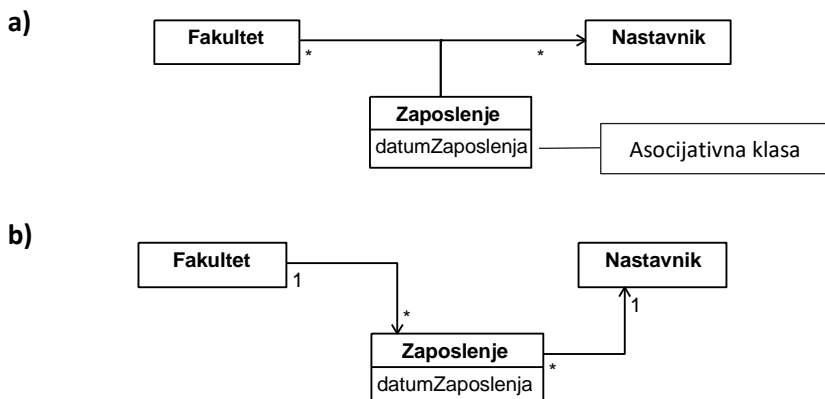
---

**Primer 4.2** *Na fakultetu su zaposleni nastavnici. Jedan nastavnik može da radi na više fakulteta. Bitno je znati kada je nastavnik počeo da radi na svakom od fakulteta na kojem je zaposlen.*

---

Primećujemo da obeležje datumZaposlenja ne može da se stavi ni u klasu Nastavnik (na koji bi se fakultet datum odnosio?), ni u klasu Fakultet (na kojeg nastavnika bi se datum odnosio?). Datum je vezan za radni odnos koji je određeni nastavnik uspostavio sa određenim fakultetom, odnosno, vezuje se za par: jedan fakultet – jedan nastavnik.

Obeležja koja dodatno opisuju odnos dve klase povezane asocijacijom mogu se smestiti u *asocijativnu klasu* pridruženu toj asocijaciji. To je klasa Zaposlenje na slici 4.24a. Može se reći da asocijativna klasa sadrži dodatne osobine asocijacije između dve klase.



Slika 4.24 a) Korišćenje asocijativne klase radi pridruživanja datuma zaposlenja asocijaciji koja modeluje radni odnos jednog nastavnika u okviru jednog fakulteta b) Ekvivalentan model bez korišćenja asocijativne klase

Ekvivalentan model, bez korišćenja asocijativne klase, je prikazan na slici 4.24b. Implementacija na nekom programskom jeziku se vrši prema ovom modelu (listing 4.12).

```

public class Fakultet {
    private String nazivFakulteta;
    private int maticniBroj;
    private String webAdresa;
    private Set<Zaposlenje> zaposlenje = null;
    ...
}

public class Zaposlenje {
    private Date datumZaposlenja;
    private Nastavnik nastavnik;
    ...
}

public class Nastavnik {
    private String ime;
    private String prezime;
    private Date datumRodjenja;
    private String telefon;
    ...
}

```

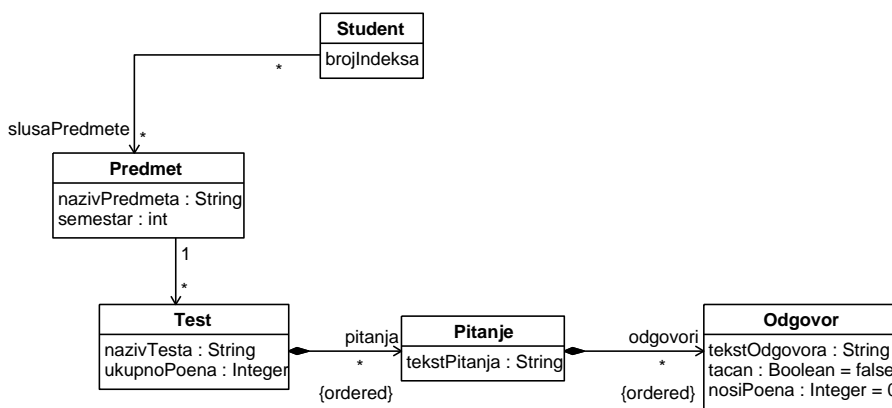
Listing 4.12 Implementacija obeležja klase sa slike 4.24

Na slici 4.25 je dat deo modela sistema za elektronsko ocenjivanje koji je do sada razmatran. Klase Test, Pitanje i Odgovor su podloga za slučaj korišćenja Kreiranja testa od strane nastavnika. Asocijacije između klasa Student, Predmet i Test omogućavaju da se studentu ponude za rešavanje samo testovi



iz predmeta koje sluša. Slučajevi korišćenja za rešavanje testa od strane studenta, ocenjivanje rešenih testova i uvid u rešene testove još nisu podržani u dijagramu klasa.

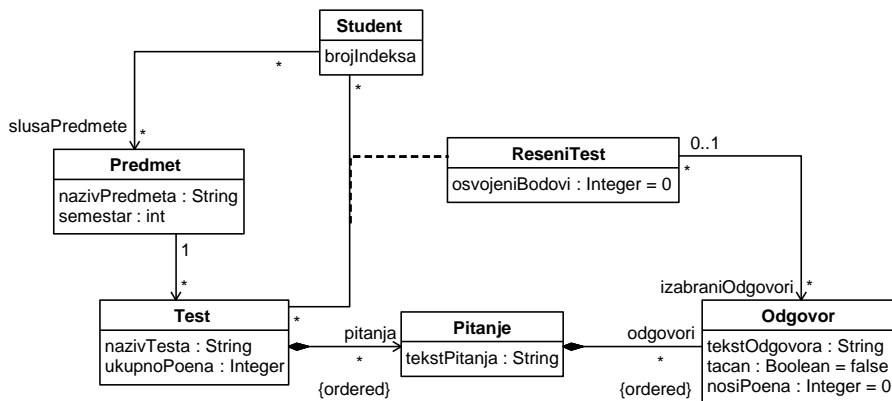
Student tokom školovanja rešava više testova. Svaki test se može rešavati od strane više studenata. Ova činjenica se može specificirati asocijacijom „više na više“ između klasa. Međutim, potrebno je znati i koliko poena je student osvojio na svakom testu koji je rešavao. Te poene treba da dodeli sistem za automatsko ocenjivanje, na osnovu odgovora koje je student dao na pitanja u okviru testa.



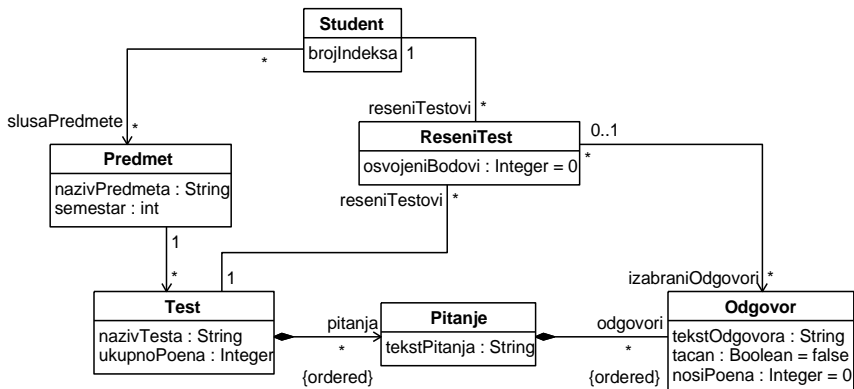
Slika 4.25 Deo konceptualnog modela sistema za elektronsko ocenjivanje

Sve navedeno se može modelovati asocijativnom klasom `ReseniTest` koja je pridružena asocijaciji između klasa `Student` i `Test` (slika 4.26). Ona sadrži broj bodova koje je jedan student osvojio rešavanjem jednog testa. Asocijacijom sa klasom `Odgovor`, klasa `ReseniTest` dobija kolekciju u koju će se smeštati odgovori koje student izabere i na osnovu kojih će se računati osvojeni broj bodova. Ekvivalentan model, bez korišćenja asocijativne klase, je prikazan na slici 4.27.

Instanca klase `ReseniTest` se kreira kada student izabere test koji želi da rešava. Inicijalni broj bodova je 0, a kolekcija izabranih odgovora je prazna. Izborom ponuđenih odgovora na pitanja iz testa, reference na odgovore se dodaju u kolekciju. Kada se polaganje završi, sistem, na osnovu sadržaja kolekcije `izabraniOdgovori` i poena koje svaki odgovor nosi, računa osvojeni broj bodova koji upisuje kao vrednost obeležja `osvojeniBodovi` klase `ReseniTest`.



Slika 4.26 Asocijativna klasa ReseniTest obezbeđuje podlogu za rešavanje i ocenjivanje testa, kao i uvid u rešene testove



Slika 4.27 Ekvivalentan model kao na slici 4.26, bez korišćenja asocijativne klase

Radi podrške uvida u rešene testove, gde student treba da vidi pitanja sa označenim tačnim odgovorima i odgovorima koje je dao, asocijacija između klase Odgovor i Pitanje je promenjena u bidirekcionu, da bi se omogućilo kretanje od rešenog testa, preko izabranih odgovora, do pitanja sa svim odgovorima. Asocijacija između klasa Student i Test je takođe bidirekciona: student može da vidi sve testove koje je rešavao tokom studija (podloga za slučaj korišćenja Uvid u rešene testove), a klasa Test ima pristup osvojenim bodovima svih studenata (podloga za Slanje rezultata nastavniku).

**Napomena:** Obratimo pažnju na način kako smo stigli do dijagrama klasa na slici 4.26. Uočavamo osnovne pojmove u specifikaciji zahteva koji čine kostur sistema (student, nastavnik, predmet, test, pitanje, itd). Kroz analizu njihovih uzajamnih odnosa, dolazimo do asocijacija sa odgovarajućim kardinalitetima. Razmatranjem kako će se pojedinačni slučajevi korišćenja izvršavati, otkrivamo nove klase (asocijativne ili „obične“), nova obeležja i metode postojećih klasa, a po potrebi menjamo i navigabilnost asocijacija.

U procesu projektovanja dijagrama klasa koji treba da posluži kao podloga za izvršavanje uočenih slučajeva korišćenja, od pomoći može biti i dijagram sekvence koji je obrađen u šestom poglavlju.

Na listingu 4.13 su implementirana obeležja klasa Student, Test i ReseniTest, sa slika 4.26 i 4.27.

```
public class Test {
    private String nazivTesta;
    private int ukupnoPoena;
    private Set<ReseniTest> reseniTestovi = null;
    ...

    public class Student {
        private int brojIndeksa;
        private Set<ReseniTest> reseniTestovi = null;
        ...

        public class ReseniTest {
            private Student student;
            private Test test;
            private List<Odgovor> izabraniOdgovori = null;
            private int osvojeniBodovi = 0;
            ...
        }
    }
}
```

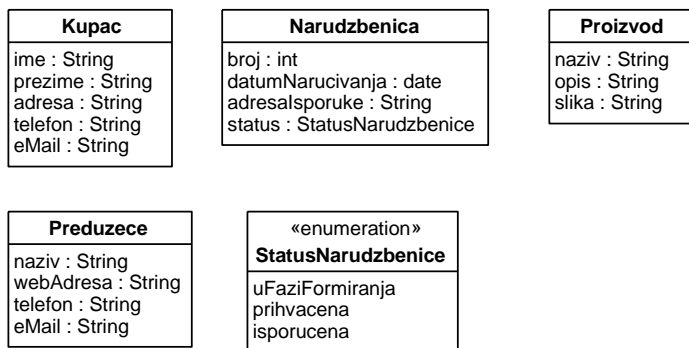
Listing 4.13 Implementacija obeležja klasa Student, Test i ReseniTest sa slika 4.26 i 4.27

Razmotrimo sada malo složeniji primer 4.3.

**Primer 4.3** Prilikom formiranja narudžbenice za neko preduzeće, kupac bira proizvode iz skupa raspoloživih proizvoda nekog preduzeća i unosi željenu količinu za svaki od njih. Podaci koje je bitno znati o kupcu su: ime, prezime, adresa, telefon, e-mail. Podaci koje je potrebno znati o proizvodu su: naziv, opis, slika proizvoda. Podaci koje treba da ima narudžbenica su: broj narudžbenice, datum naručivanja, kupac, spisak naručenih proizvoda sa količinama, adresa isporuke (ne mora biti ista kao adresa kupca), status narudžbenice (u fazi formiranja, prihvaćena, isporučena). Podaci koje treba znati o preduzeću su: naziv, web adresa, telefon, e-mail. Za svakog kupca je bitno imati uvid u sve njegove narudžbenice.

*Potrebno je obezbediti i sledeće izveštaje: spisak isporučenih narudžbenica u zadanom vremenskom periodu, spisak svih kupaca koji imaju isporučene narudžbenice u zadanom vremenskom periodu, spisak narudžbenica za sve kupce iz zadanog mesta u zadanom vremenskom periodu.*

U tekstu specifikacije uočavamo da su kandidati za klase: Preduzece, Kupac, Narudzbenica i Proizvod. Možemo odmah da ih stavimo na dijagram klasa, zajedno sa uočenim obeležjima (slika 4.27). Status narudžbenice koji ima fiksni skup vrednosti modelujemo kao nabrojani tip.



Slika 4.27 Početak modelovanja dijagrama klasa za primer 4.3 – postavljene su klase koje čine osnovu sistema

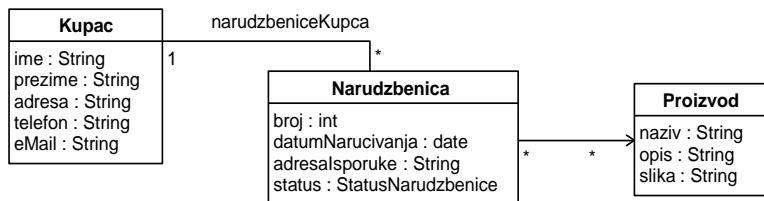
Možemo primetiti da smo u navedene klase uneli samo osnovna obeležja koja opisuju date koncepte iz poslovnog domena. Sve ostalo što proizilazi iz zahteva navedenih u specifikaciji treba obezbediti preko asocijacija i dodatnih klasa.

Obratimo pažnju na sledeće rečenice:

*Podaci koje treba da ima narudžbenica su: broj narudžbenice, datum naručivanja, kupac, spisak naručenih proizvoda sa količinama. Za svakog kupca je bitno imati uvid u sve njegove narudžbenice.*

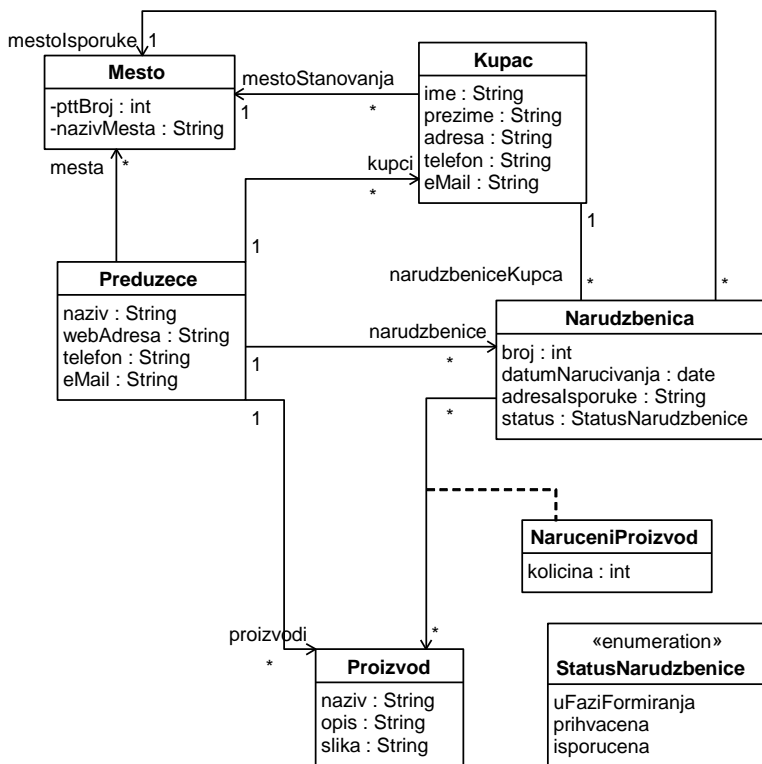
Zahtev da narudžbenica ima podatke o kupcu i da kupac treba da vidi sve svoje narudžbenice modelujemo bidirekcionom asocijacijom između klasa Kupac i Narudzbenica.

Narudžbenica ima spisak naručenih proizvoda. Spisak nećemo modelovati kao klasu, već ćemo ga dobiti kao posledicu asocijacije sa klasom Proizvod, sa kardinalitetom \* na strani proizvoda. Pošto jedan proizvod može biti naručen u okviru više narudžbenica, kardinalitet na strani klase Narudzbenica je takođe \* (slika 4.28).



Slika 4.28 Početak modelovanja narudžbenice iz primera 4.3

Ostaje pitanje: gde staviti količinu? Ona može biti različita za svaki proizvod u okviru svake narudžbenice, tako da se ne može staviti ni u klasu Proizvod, ni u klasu Narudžbenica. Zaključak je da nam je potrebna asocijativna klasa - videti klasu NaruceniProizvod na slici 4.29.



Slika 4.29 Konceptualni dijagram klasa za primer 4.3. Podrška za prikaz izveštaja je modelovana na slici 4.30.

Klasa Mesto je uvedena da bismo mogli da napravimo izveštaj o svim narudžbenicama koje su realizovali kupci iz zadatog mesta. Ukoliko bi naziv mesta mode-

lovali kao String obeležje klase Kupac, korisnici bi mogli da unose proizvoljne vrednosti, a dešavale bi se i greške u kucanju. *Tačan izveštaj koji je baziran na poređenju proizvoljno unetih stringova nije moguće dobiti!* Kreiranjem odvojene kolekcije mesta koja se održava na nivou preduzeća i spajanjem instance klase Kupac sa željenom instancom klase Mesto, možemo tačno utvrditi koji su kupci iz određenog mesta (svi koji imaju referencu na dati objekat).

Da bi se ovo postiglo, potrebno je implementirati i odgovarajući korisnički interfejs koji sprečava proizvoljan unos: u okviru forme za unos kupaca, mesto bi trebalo birati preko padajućih listi (*combo-box*), popunjenih elementima kolekcije mesta u okviru preduzeća.

Raspoložive proizvode nekog preduzeća modelujemo asocijacijom između klasa Preduzece i Proizvod. Time dobijamo kolekciju u koju možemo da unosimo nove proizvode. Ispisivanjem njenih elemenata možemo da prikazemo spisak proizvoda koje preduzeće nudi na prodaju.

Da bismo znali ko su kupci koji posluju sa datim preduzećem, uvodimo asocijaciju između klasa Preduzece i Kupac. Iako kupac ima kolekciju svojih narudžbenica, preduzeće je takođe povezano sa klasom Narudzbenica, radi mogućnosti direktnog pretraživanja svih narudžbenica, nezavisno od kupca.

Uvođenjem ove asocijacije se komplikuje unos novih narudžbenica (potrebno je održavati kolekcije narudžbenica u okviru i kupca i preduzeća), ali se ubrzava izveštavanje. Ovakve odluke se često moraju donositi tokom modelovanja: usložnjavanjem strukture se obično optimizuje izvršavanje različitih algoritama i pojednostavljuje njihova implementacija, ali se otežava održavanje (unos, izmena, brisanje) podataka. Idealno rešenje retko postoji, tako da treba biti svestan dobrih i loših strana svojih projektantskih odluka i naći optimalno rešenje za određenu situaciju. Model sa slike 4.29 bi bio mogao da podrži sve iznete korisničke zahteve i bez asocijacije između klasa Preduzece i Narudzbenica, samo što bi kreiranje izveštaja sporije radilo u slučaju većeg broja narudžbenica.

Ostalo je još da modelujemo podršku za prikaz izveštaja. Domenske klase treba da obezbede metode koje pribavljaju tražene podatke, a klase iz sloja korisničkog interfejsa se koriste da te podatke formatiraju na odgovarajući način i prikažu korisniku.

U konceptualnom modelu, metode stavljamo u onu klasu koja ima potrebne podatke nad kojima metoda radi. Pošto preduzeće ima pristup svim kupcima i narudžbenicama, odgovarajuće mesto je klasa Preduzece (slika 4.30).

Preduzece
naziv : String webAdresa : String telefon : String eMail : String
+isporuceneNarudzbenice( Date odDatuma, Date doDatuma ) : Narudzbenica [*] +kupciSaNarudzbenicama( Date odDatuma, Date doDatuma ) : Kupac [*] +narudzbeniceIzMesta( Mesto mesto, Date odDatuma, Date doDatuma ) : Narudzbenica [*]

Slika 4.30 Klasa Preduzece iz primera 4.3 sa dodatim metodama potrebnim za prikaz traženih izveštaja

Često se postavlja pitanje zašto nam je potrebna klasa preduzeće? Postoji više razloga:

- String obeležja klase Preduzece se koriste za ispis podataka u zaglavlju dokumenata koji se šalju poslovnim partnerima (tzv. memorandum).
- Pošto instanca klase Preduzece ima pristup kolekcijama instanci svih klasa potrebnih za poslovanje, imamo jednu ulaznu tačku (*entry point*) koja može da primi i preusmerava zahteve koje korisnik prosleđuje putem korisničkog interfejsa.
- Možemo primetiti i da imamo mogućnost postojanja više instanci klase Preduzece, gde svaka instanca ima u kolekcijama svoje podatke koji se ne mešaju sa podacima drugih instanci. Postojanje više instanci preduzeća omogućava kreiranje tzv. *multi tenant*<sup>13</sup> arhitektura: korisnici jednog preduzeća vide samo svoje podatke, iako se ista softverska infrastruktura koristi za podršku svih instanci preduzeća – čest pristup za rešenja u „računarskom oblaku“ (*cloud*).

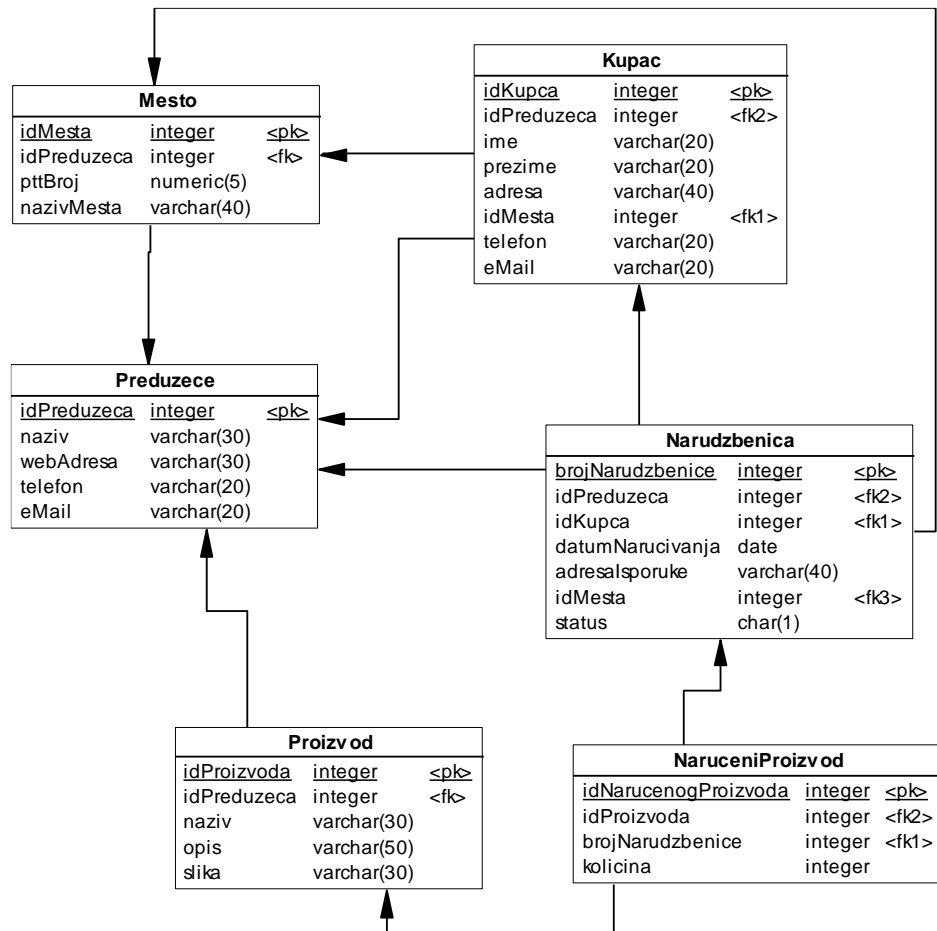
Šema relacione baze podataka sa surogat ključevima, koja se može dobiti na osnovu dijagrama sa slike 4.29 je prikazana na slici 4.31.

S obzirom da izgleda kao da je većina funkcionalnosti skoncentrisana u klasu Preduzece, postavlja se pitanje da li smo dobili tzv. „God“ klasu? „God“ klasa je termin koji označava klasu koja implementira veliki deo funkcionalnosti nekog rešenja i time narušava principe projektovanja kvalitetne OO arhitekture (preporuka je da se kreira veći broj dobro enkapsuliranih klasa koje obavljaju jedan, jasno definisan zadatak, a tražene korisničke zahteve obezbeđuju kroz međusobnu saradnju).

Odgovor je: Preduzece neće biti „God“ klasa ako se korektno obavi implementacija. Treba imati u vidu da je dijagram sa slike 4.29 konceptualni

<sup>13</sup> Videti npr. [www.ibm.com/cloud/learn/multi-tenant](http://www.ibm.com/cloud/learn/multi-tenant)

model, kojim prikazujemo pojmove iz poslovnog domena i njihove uzajamne odnose i koji se preslikava na šemu baze podataka. U okviru implementacionog modela koji se preslikava na programski kod, uvode se dodatne klase koje omogućavaju ravnomernu raspodelu odgovornosti, a Preduzece iz svojih metoda samo treba da im „preusmerava“ pozive. Ovim se detaljno bavi sekcija 4.4.1.5.



Slika 4.31 Relacioni model za primer 4.3 koji je izveden na osnovu konceptualnog dijagrama klasa sa slike 4.29. Na dijagramu vidimo tabele povezane relacijama. Relacija je usmerena prema tabeli iz koje razmatrana tabela preuzima strani ključ.

Za kraj, pogledajmo još primer 4.4.



---

**Primer 4.4** Projektovati informacijski sistem bioskopa koji omogućava kupovinu karata za projekcije filmova i praćenje zarade od projekcija. Bioskop ima više sala za projekcije.

Administrator unosi podatke o salama i filmovima. Za svaku salu je potrebno da unese: naziv sale, broj redova i broj sedišta u redu (svaki red ima isti broj sedišta u jednoj sali). Za film unosi podatke: naziv, žanr (npr. komedija, akcioni, sf, horor i sl), jednog ili više režisera, jednog ili više scenarista, glumce, jednog ili više autora muzike, jednog ili više scenografa, vreme trajanja i godinu proizvodnje. Za glumce, režisere, scenariste, autore muzike i scenografe je potrebno uneti samo ime i prezime.

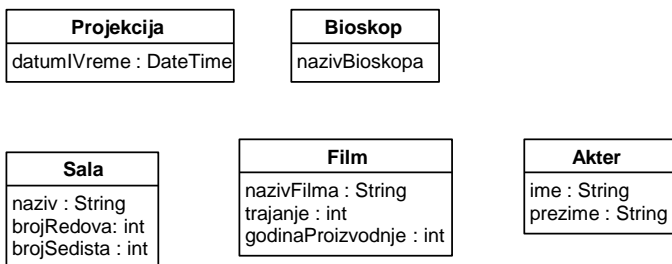
Planer kreira raspored projekcija filmova (kada se i u kojoj sali prikazuje određeni film). Pored toga, određuje i cene karata koje će važiti u narednom periodu. Sve projekcije u jednom danu imaju istu cenu karte.

Planeru su potrebni sledeći izveštaji: 1) izveštaj o zaradi za datu projekciju, 2) izveštaj o zaradi određenog filma u svim projekcijama koje je dati film imao u zadatom vremenskom intervalu i 3) izveštaj o zaradi bioskopa od projekcija svih filmova u zadatom vremenskom intervalu.

Za projekcije koje je planer uneo, a koje još nisu održane, radnik bioskopa može da prodaje karte. Da bi mogao da obavlja svoj posao, potrebno je da za svaku projekciju zna koja sedišta su prodana, a koja slobodna. Takođe, treba da ima podršku da daje odgovore na pitanja: 1) kada su projekcije određenog filma 2) kada su projekcije filmova određenog glumca, režisera ili scenariste i 3) kada su projekcije filmova određenog žanra i koji su filmovi u pitanju?

---

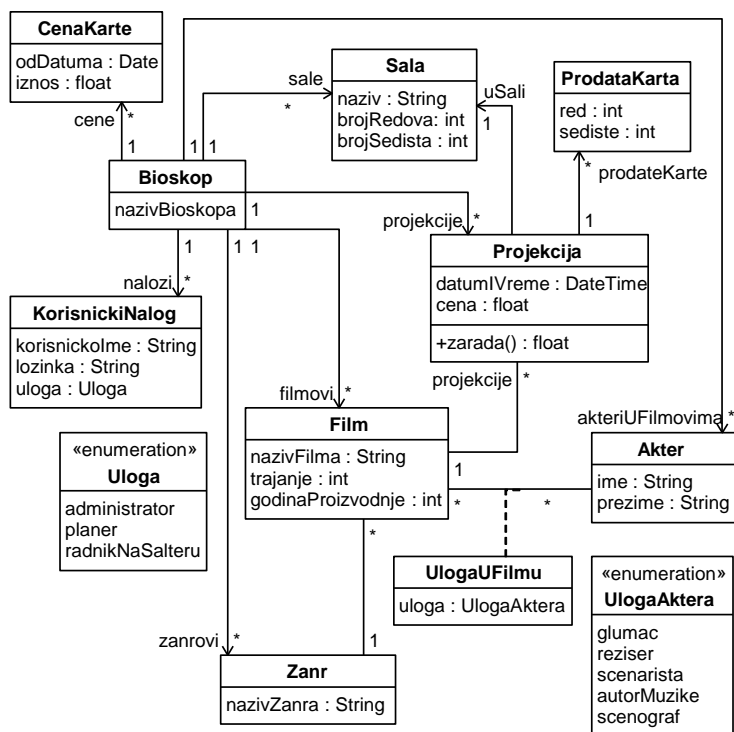
Modelovanje počinjemo tako što iz specifikacije izdvajamo moguće kandidate za klase sa njihovim osnovnim obeležjima i postavljamo na dijagram (slika 4.32).



Slika 4.32 Početak modelovanja dijagrama klasa za primer 4.4 – postavljena je osnova sistema sa mogućim kandidatima za klase

Akter je zajednička klasa za sve koji obavljaju neki posao u okviru filma (glumac, režiser, scenarista, itd).

Sada možemo da krenemo sa povezivanjem učenih klasa i dodavanjem novih klasa, obeležja i metoda, u zavisnosti od zahteva koje sistem treba da podrži.



Slika 4.33 Konceptualni dijagram klasa za primer 4.4. Podrška za prikaz izveštaja i davanje obaveštenja je modelovana na slici 4.34.

Na osnovu dodatnih pitanja naručiocima, saznajemo:

- Jedan akter može da ima više uloga u filmu (npr. može da bude istovremeno glumac, režiser i scenarista), kao i da ima različite uloge u različitim filmovima (npr. u jednom filmu je glumac, a u drugom režiser).
- Konačan spisak žanrova nije poznat unapred – treba omogućiti administratoru da ga dopuni po potrebi.
- Prilikom unosa podataka o filmu, potrebno je uneti samo glumce, režisere, scenariste, autore muzike i scenografe, ne i ostale moguće profesije koje imaju ulogu u kreiranju filma.

Film modelujemo tako što ga povezujemo sa klasom Akter asocijacijom „više prema više“, a u pridruženu asocijativnu klasu UlogaUFilmu stavljamo ulogu koju dati akter ima u datom filmu (slika 4.33). Ako neki akter ima više uloga u jednom filmu, imaćemo više instanci klase UlogaUFilmu sa istim filmom i akterom, ali različitim vrednostima obeležja uloga. UlogaAktera je modelovana kao nabrojani tip, zato što je spisak uloga poznat unapred i konačan. Asocijacija je bidirekciona da bismo mogli da na jednostavan način prikazujemo podatke o filmu, kao i da dobijemo odgovor na upite koji akter učestvuje u kojim filmovima.

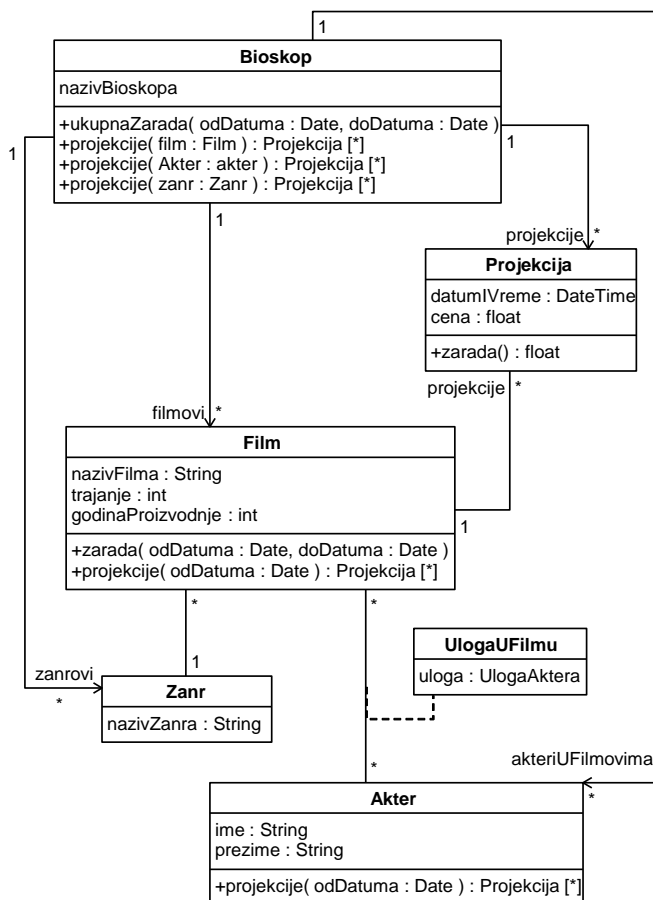
Klasa Film je povezana i sa klasom Završ. Žanr bi mogao da bude String obeležje klase Film, ali je izdvojen u posebnu klasu zato što postoji zahtev za pretragom projekcija po žanru filma. Bez posebne klase, korisnici bi, prilikom unosa podataka o filmu, mogli da unose proizvoljne vrednosti za žanr, na osnovu kojih ne bismo mogli da dobijemo tačne podatke prilikom pretrage. Kada bi spisak žanrova bio unapred poznat i konačan, modelovali bismo ga kao nabrojani tip.

Klasa Projekcija se povezuje asocijacijama sa filmom koji se prikazuje, salom u kojoj se održava projekcija i sa klasom ProdataKarta, da bismo znali koja sedišta su zauzeta i koliko karata je prodato. Asocijacija između klase Projekcija i Film je bidirekciona, da bismo mogli jednostavno da utvrdimo kada su projekcije nekog filma, kao i koji se film prikazuje u određenoj projekciji.

Cena karte koja važi počevši od nekog datuma je specificirana kao klasa CenaKarte. Bioskop ima kolekciju svih cena karata koje su ranije važile, kao i koje će se primenjivati za buduće projekcije. Prilikom kreiranja instance klase Projekcija, kao argument konstruktora treba da joj se prosledi cena koja se određuje na osnovu datuma projekcije. Na osnovu prosleđene cene i kolekcije prodatih karata, metoda zarada() klase Projekcija može da izračuna koliko je novca dobijeno na toj projekciji.

**Napomena:** Cenu treba držati u posebnoj klasi, sa naznačenim datumom od kada važi, da bi preduzeće moglo da ima istoriju promene cena, kao i mogućnost da pravi cenovnike unapred. Ukoliko bismo cenu modelovali kao obeležje klase Bioskop, imali bismo samo cenu koja trenutno važi – svaka izmena bi poništavala prethodnu vrednost. Ako preduzeće prodaje više proizvoda (kao u primeru 4.3), za svaki proizvod treba obezbediti mogućnost unosa cene po sličnom principu. Ako bi se cena čuvala kao obeležje klase Proizvod (česta greška pri modelovanju), mogli bismo da imamo samo trenutnu cenu proizvoda.

Klasa `KorisnickiNalog` je kreirana da bi korisnici, koji mogu biti administratori, planeri i radnici na šalteru, mogli da se prijave svojim korisničkim imenom i lozinkom i obavljaju svoj posao.



Slika 4.34 Klase iz primera 4.4 sa dodatim metodama potrebnim za prikaz izveštaja i davanje obaveštenja

Pitanje koje se obično postavlja je: zašto nije kreirana posebna klasa za svaku vrstu korisnika, sa metodama koje odgovaraju poslovima koje dati korisnik obavlja? Razlog je što u postojećem modelu već imamo sve što je potrebno za obavljanje posla svih vrsta korisnika. Metode su raspoređene u one klase koje imaju potrebne podatke, prema pravilima OO projektovanja (slika 4.34). Klasa `Projekcija` ima metodu za računanje zarade od jedne projekcije. `Film` ima metodu za računanje zarade od svojih projekcija u zadatom vremenskom intervalu. Svaka klasa koja ima neku kolekciju, treba da ima i mogućnost

dodavanja i brisanja elemenata kolekcije. Do svih instanci klasa se može stići preko klase Bioskop. Ukoliko bismo kreirali klase za svaku vrstu korisnika, njihove metode bi mogle samo da se obraćaju metodama klase Bioskop za uslugu, pošto nemaju potrebne podatke za rad (osim ako i njih spojimo sa svim klasama kao i Bioskop, kreirajući dodatnu kompleksnost u sistemu).

Treba imati u vidu i da postoji sloj korisničkog interfejsa koji radi nad modelovanim domenskim klasama. Kada se korisnik prijavi, korisnički interfejs treba da se dinamički prilagodi i prikaže samo opcije koje odgovaraju njegovim pravima korišćenja. Zahtevi korisnika se prosleđuju klasi Bioskop, koja ih dalje prosleđuje odgovarajućim domenskim klasama.

Domenske klase se projektuju tako da podrže poslove svih korisnika, a pravima pristupa se podešava da određena vrsta korisnika ne može da koristi nešto što ne bi trebalo. Prava pristupa se podaševaju i u okviru baze podataka, kao „poslednja linija odbrane“, za slučaj da neko direktno pristupa bazi podataka, bez korišćenja razvijene aplikacije.

#### **4.4.1.4 N-arna asocijacija**

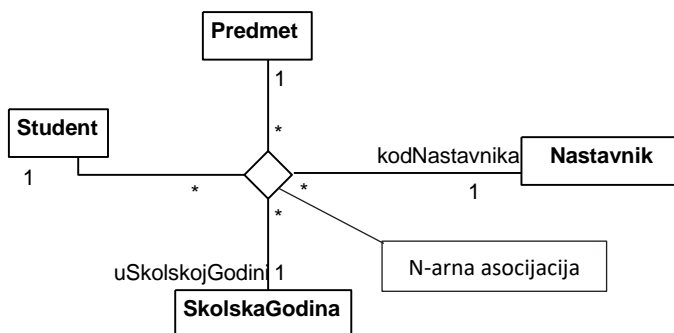
U okviru UML-a 1.0 asocijacije su mogle da povežu samo dve klase (odnosno, da imaju samo dva kraja), kao što je prikazano u prethodnim sekcijama. U UML-u 2.0 asocijacija može da poveže proizvoljan broj klasa. Na slici 4.35a je dat primer takve asocijacije za sledeću specifikaciju:

*Student sluša predmete, svaki kod određenog nastavnika, u određenoj školskoj godini.*

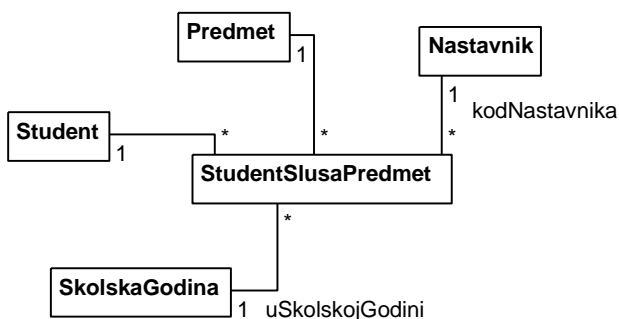
Za svaki kraj asocijacije se mogu navesti sve osobine koje smo do sada imali prilike da vidimo: kardinalitet, navigabilnost, dodatne opcije i sl.

Ovakav način modelovanja asocijacije nije široko prihvaćen, ni od alata za modelovanje, ni od projekatara, tako da je bolje primenjivati ekvivalentan način modelovanja kao na slici 4.35b, na osnovu kojeg se implementira programski kod (listing 4.14).

a)



b)



Slika 4.35 a) Primer n-arne asocijacije b) Ekvivalentan način modelovanja: umesto n-arne asocijacije je specificirana klasa StudentSlusaPredmet

```

// Klasa koja nastaje kao posledica n-arne asocijacije:
public class StudentSlusaPredmet {
    private Student student;
    private Predmet predmet;
    private SkolskaGodina uSkolskojGodini;
    private Nastavnik kodNastavnika;
    ...
}

```

Listing 4.14 Implementacija obeležja n-arne asocijacije sa slike 4.35

#### 4.4.1.5 Preslikavanje asocijacija na programski kod

Na slici 4.20 smo modelovali deo informacionog sistema nekog fakulteta, za specifikaciju datu u okviru primera 4.1. Fakultet ima departmane i nastavnike. Jedan od nastavnika je dekan. Asocijacije su usmerene od klase Fakultet, prema klasama Nastavnik i Departman i imaju kardinalitet \*. Ovo za posledicu ima da klasa Fakultet ima skup instanci klase Nastavnik i Departman. Rad sa skupom

departmana u okviru klase Fakultet je prikazan na listingu 15. Slično se implementira i rad sa skupom nastavnika.

```
public class Fakultet {
    private String nazivFakulteta;
    private int maticniBroj;
    private String webAdresa;
    private Set<Nastavnik> nastavnici = null;
    private Nastavnik dekan;
    private Set<Nastavnik> departmani = null;
    ...

    public Set<Departman> getDepartmani() {
        if (null == departmani)
            return null;
        return Collections.unmodifiableSet(departmani);
    }

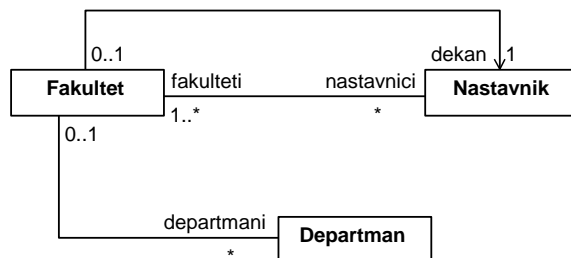
    public void dodajDepartman(Departman departman) {
        if (null == departman)
            throw new NullPointerException();
        if (null == departmani)
            departmani = new HashSet<Departman>();
        if (!departmani.contains(departman)) {
            departmani.add(departman);
        }
    }

    public void ukloniDepartman(Departman departman) {
        if (null == departman)
            throw new NullPointerException();
        if (null == departmani)
            throw new IllegalArgumentException();
        if (departmani.contains(departman)) {
            departmani.remove(departman);
        } else
            throw new IllegalArgumentException();
    }
    ...
}
```

Listing 4.15 Implementacija dela koda klase Fakultet koja ilustruje saradnju sa klasom Departman, sa slike 4.20. Asocijacija je usmerena od klase Fakultet ka klasi Departman

Klase Departman i Nastavnik „ne vide“ Fakultet, tako da treba da implementiraju samo get i set metode za svoja obeležja unapred definisanih tipova (String i date).

Da bismo razumeli zašto bidirekzione asocijacije komplikuju dodavanje i brisanje objekata, pretpostavimo da smo situaciju opisanu u primeru 4.1 modelovali korišćenjem bidirekcionih veza, kao na slici 4.36. Sada se klase Fakultet i Nastavnik uzajamno „vide“, kao i klase Fakultet i Departman.



Slika 4.36 Konceptualni dijagram klasa za primer 4.1, kreiran korišćenjem bidirekcionih asocijacija između klasa Fakultet i Nastavnik, kao i Fakultet i Departman

```

public class Departman
{
    private String nazivDepartmana;
    private String telefon;
    private Fakultet fakultet;
    ...
    public Fakultet getFakultet() {
        return fakultet;
    }

    public void setFakultet(Fakultet fakultet) {
        if (null != this.fakultet && !this.fakultet.equals(fakultet))
            this.fakultet.ukloniDepartman(this);
        this.fakultet = fakultet;
        if (null != fakultet)
            fakultet.dodajDepartman(this);
    }
    ...
}
  
```

Listing 4.16 Implementacija dela koda klase Departman u slučaju bidirekcionih asocijacija sa klasom Fakultet (slika 4.36)

Razmotrimo prvo kako izgleda programski kod koji implementira saradnju klasa Fakultet i Departman (fakultet ima više departmana, departman je pridružen najviše jednom fakultetu). Na listingu 4.16 je dat deo programskog koda klase Departman. U okviru metode setFakultet možemo primetiti da se, prilikom dodele vrednosti obeležju fakultet određenog objekta klase Departman, dati objekat istovremeno dodaje u kolekciju departmana fakulteta koji je prosleđen kao parametar (pod uslovom da nije prosleđena null vrednost). Ako je kao argument prosleđena null vrednost, ili ako se prosleđeni fakultet razlikuje od postojećeg, objekat departmana se uklanja iz kolekcije departmana fakulteta kojem trenutno pripada.



```

public class Fakultet {
    ...
    public void dodajDepartman(Departman departman) {
        if (null == departman)
            throw new NullPointerException();
        if (null == departmani)
            departmani = new HashSet<Departman>();
        if (!departmani.contains(departman)) {
            departmani.add(departman);
            departman.setFakultet(this);
        }
    }

    public void ukloniDepartman(Departman departman) {
        if (null == departman)
            throw new NullPointerException();
        if (null == departmani)
            throw new IllegalArgumentException();
        if (departmani.contains(departman)) {
            departmani.remove(departman);
            departman.setFakultet(null);
        } else
            throw new IllegalArgumentException();
    }
    ...
}

```

Listing 4.17 Implementacija dela koda klase Fakultet koja ilustruje saradnju sa klasom Departman, kada su povezani bidirekcionom asocijacijom (slika 4.36)

Na listingu 4.17 vidimo da se, prilikom dodavanja i uklanjanja departmana u okviru metoda `dodajDepartman` i `ukloniDepartman` klase `Fakultet`, istovremeno poziva metoda `setFakultet` departmana koji je prosleđen kao parametar.

Možemo primetiti da se `departman` može uključiti u fakultet ili isključiti iz njega, bilo od strane fakulteta, pozivom metoda `dodajDepartman` i `ukloniDepartman`, bilo od strane departmana, pozivom metode `setFakultet`. Pošto se navedene metode uzajamno pozivaju, moramo paziti da obradimo sve situacije, kao i da sprečimo da dođe do beskonačne petlje usled nedovoljno dobro implementirane uzajamne rekurzije.

U slučaju bidirekcionе asocijacije između klasa `Nastavnik` i `Fakultet`, možemo primetiti da, kada nastavnika dodajemo u kolekciju nastavnika nekog fakulteta, istovremeno dodajemo fakultet u listu fakulteta na kojima nastavnik radi. Slično važi i za uklanjanje.

```

public class Fakultet {
    ...
    public Set<Nastavnik> getNastavnici() {
        if (null == nastavnici)
            return null;
        return Collections.unmodifiableSet(nastavnici);
    }

    public void dodajNastavnika(Nastavnik nastavnik) {
        if (null == nastavnik)
            throw new NullPointerException();
        if (null == nastavnici)
            nastavnici = new HashSet<Nastavnik>();
        if (!nastavnici.contains(nastavnik))
        {
            nastavnik.add(nastavnik);
            nastavnik.dodajFakultet(this);
        }
    }

    public void ukloniNastavnika(Nastavnik nastavnik) {
        if (null == nastavnik)
            throw new NullPointerException();
        if (null == nastavnici)
            throw new IllegalArgumentException();
        if (nastavnici.contains(nastavnik))
        {
            nastavnik.remove(nastavnik);
            nastavnik.ukloniFakultet(this);
        } else
            throw new IllegalArgumentException();
    }
    ...
}

```

Listing 4.18 Implementacija dela koda klase Fakultet koja ilustruje saradnju sa klasom Nastavnik, kada su povezani bidirekcionom asocijacijom (slika 4.36)

Iz svega navedenog se može primetiti da bidirekzione asocijacije izazivaju jako sprezanje klasa i komplikuju implementaciju. U slučaju usmerenih asocijacija sa slike 4.20, programski kod koji implementira njihovu saradnju je mnogo jednostavniji: sav potreban programski kod se nalazi jedino u okviru klase Fakultet.

Pored toga, možemo primetiti da je programski kod koji implementira rad sa kolekcijama veoma sličan. U slučaju usmerenih asocijacija, na isti način se rukuje elementima skupa nastavnika i skupa departmana (listing 4.15) - razlike su samo u imenima elemenata kolekcije. U slučaju bidirekcionih asocijacija, programski kod sa listinga 4.15 se proširuje radi održavanja drugog kraja asocijacije (listinzi 4.17 i 4.18). Klasa Nastavnik na isti način rukuje elementima skupa fakulteta, kao što Fakultet rukuje elementima skupa nastavnika (listing 4.19). Za jednostavan primer sa tri klase, skoro identičnu funkcionalnost smo implementirali

četiri puta. Zamislimo kako bi izgledao programski kod za realan sistem sa stotinama domenskih klasa!

```
public class Nastavnik {
    ...

    public Set<Fakultet> getFakulteti() {
        if (null == fakulteti)
            return null;
        return Collections.unmodifiableSet(fakulteti);
    }

    public void dodajFakultet(Fakultet fakultet) {
        if (null == fakultet)
            throw new NullPointerException();
        if (null == fakulteti)
            fakulteti = new HashSet<Fakultet>();
        if (!fakulteti.contains(fakultet))
        {
            fakulteti.add(fakultet);
            fakultet.dodajNastavnika(this);
        }
    }

    public void ukloniFakultet(Fakultet fakultet) {
        if (null == fakultet)
            throw new NullPointerException();
        if (null == fakulteti)
            throw new IllegalArgumentException();
        if (fakulteti.contains(fakultet))
        {
            fakulteti.remove(fakultet);
            fakultet.ukloniNastavnika(this);
        } else
            throw new IllegalArgumentException();
    }
    ...
}
```

Listing 4.19 Implementacija dela koda klase Nastavnik koja ilustruje saradnju sa klasom Fakultet, kada su povezani bidirekcionom asocijacijom (slika 4.36)

Prema pravilima pisanja „čistog koda“ (*clean code*), uočene redundanse u programskom kodu bi trebalo izdvajati, implementirati u okviru bibliotečkih klasa ili razvojnih okvira (*framework*) i zatim koristiti na svim mestima gde je to potrebno (tzv. *DRY – don't repeat yourself* princip). Ako se ukaže potreba za promenom programskog koda, zbog ispravke grešaka ili drugih razloga, ispravku je potrebno izvršiti na samo jednom mestu, umesto na svim kopijama koje postoje u kodu.

Na listingu 4.20 je prikazana implementacija klase SetProperty koja može da se iskoristi za implementaciju krajeva asocijacije sa kardinalitetom \* i {unique}

opcijom. Data klasa radi sa generičkim tipovima<sup>14</sup>, koji se zamenjuju konkretnim tipovima podataka tokom kompilacije.

```
public class SetProperty<T> {
    protected Set<T> values = null;

    public Set<T> get() {
        if (null == values)
            return null;
        return Collections.unmodifiableSet(values);
    }

    public Boolean add(T t) {
        Boolean added = false;
        if (null == t)
            throw new NullPointerException();
        if (null == values)
            values = new HashSet<T>();
        if (!values.contains(t)) {
            added = values.add(t);
        }
        return added;
    }

    public Boolean remove (T t) {
        Boolean removed = false;
        if (null == t)
            throw new NullPointerException();
        if (null == values)
            throw new IllegalArgumentException();
        if (values.contains(t)) {
            removed = values.remove(t);
        } else
            throw new IllegalArgumentException();
        return removed;
    }
}
```

Listing 4.20 Generička klasa SetProperty koja se može iskoristi za implementaciju krajeva asocijacije sa kardinalitetom \* i {unique} opcijom

Programski kod sa listinga 4.15, ponovo implementiran korišćenjem ove klase je prikazan na listingu 4.21. Implementirana je podrška za asocijaciju koja nije bidirekciona, tako da metode klase Fakultet samo pozivaju odgovarajuće metode generičke klase.

---

<sup>14</sup> <https://docs.oracle.com/javase/tutorial/java/generics/>

```

public class Fakultet {
    ...
    private SetProperty<Departman> departmani = new SetProperty<Departman>();
    public void dodajDepartman(Departman departman) {
        departmani.add(departman);
    }
    public void ukloniDepartman(Departman departman) {
        departmani.remove(departman);
    }
    public Set<Departman> getDepartmani() {
        return departmani.get();
    }
    ...
}

```

Listing 4.21 Programski kod sa listinga 4.15, ponovo implementiran korišćenjem generičke klase SetProperty

U slučaju bidirekcionе asocijacije, potrebno je naslediti klasu SetProperty i dodati joj programski kod za održavanje suprotnog kraja asocijacije (listing 4.22).

```

public class SetNastavnika extends SetProperty<Nastavnik> {
    private Fakultet fakultet;

    public SetNastavnika(Fakultet fakultet) {
        this.fakultet = fakultet;
    }

    public Boolean add(Nastavnik nastavnik) {
        Boolean dodat = super.add(nastavnik);
        if (dodat)
            nastavnik.dodajFakultet(fakultet);
        return dodat;
    }

    public Boolean remove (Nastavnik nastavnik) {
        Boolean uklonjen = super.remove(nastavnik);
        if (uklonjen)
            nastavnik.ukloniFakultet(fakultet);
        return uklonjen;
    }

    public Set<Nastavnik> rodjeniGodine(int godina) {
        //Kod koji implementira izdvajanje skupa nastavnika koji su rodjeni
        //odredjene godine
    }
}

```

Listing 4.22 Klasa NastavnikSet je naslednica generičke klase SetProperty koja obezbeđuje održavanje oba kraja bidirekcionе asocijacije između klasa Fakultet i Nastavnik.

Ukoliko postoji zahtev da se implementiraju metode koje treba da rade nad kolekcijom nastavnika, treba ih staviti u ovu klasu. Kao primer je data skica metode `rodjeniUGodini` koja kao rezultat vraća skup svih nastavnika koji su rođeni određene godine.

```
public class Fakultet {  
    ...  
    private SetNastavnika nastavnici = new SetNastavnika(this);  
  
    public void dodajNastavnika(Nastavnik nastavnik) {  
        nastavnici.add(nastavnik);  
    }  
  
    public void ukloniNastavnika(Nastavnik nastavnik) {  
        nastavnici.remove(nastavnik);  
    }  
  
    public Set<Nastavnik> getNastavnici() {  
        return nastavnici.get();  
    }  
  
    public Set<Nastavnik> nastavniciRodjeniGodine(int godina) {  
        return nastavnici.rodjeniGodine(godina);  
    }  
    ...  
}
```

Listing 4.23 Korišćenje klase `NastavnikSet` od strane klase `Fakultet`

Na listingu 4.23 je prikazano korišćenje klase `SetNastavnika` od strane klase `Fakultet`. Instanca klase `SetNastavnika` dobija kao parameter konstruktora referencu na instancu fakulteta kojoj je pridružena, tako da, prilikom dodavanja i brisanja nastavnika, može da održava i reference nastavnika prema fakultetu.

Metoda `nastavniciRodjeniUGodini` klase `Fakultet` se implementira tako što poziva metodu `rodjeniUGodini` klase `SetNastavnika`. Na ovaj način, klasa `Fakultet` obebeđuje potrebne usluge sloju korisničkog interfejsa, a pri tome ne izlaže detalje svoje unutrašnje strukture (ne postoji potreba da se neko spolja direktno obraća klasi `SetNastavnika`). Ako nađemo bolji način da implementiramo bidirekzione asocijacije (npr. korišćenjem refleksije<sup>15</sup> ili aspekt-orijentisanog programiranja<sup>16</sup>), klasa `Fakultet` se i dalje može koristiti na isti način.

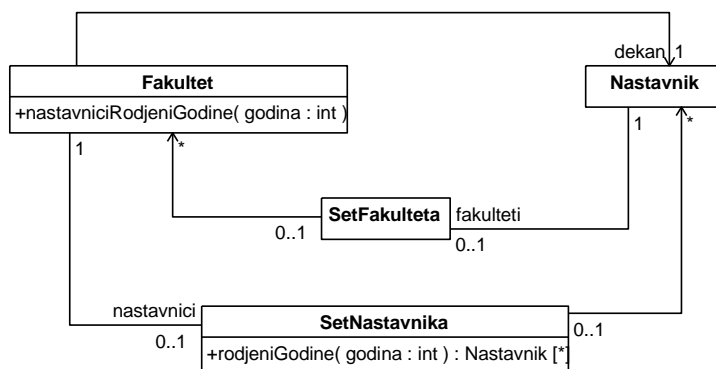
Možemo primetiti i da `Fakultet` ne implementira svu funkcionalnost, odnosno, nije „God“ klasa, iako spolja tako izgleda. Metode su implementirane u okviru

---

<sup>15</sup> <https://www.oracle.com/technical-resources/articles/java/javareflection.html>

<sup>16</sup> <https://www.eclipse.org/aspectj/>

onih klasa koje imaju podatke potrebne za njihovo izvršavanje i samo se pozivaju od strane klase Fakultet koja im preusmerava pozive.



Slika 4.37 Implementacioni dijagram klasa koji ilustruje način saradnje klasa Fakultet i Nastavnik preko bidirekcionne asocijacije

Implementacioni model koji prikazuje saradnju klasa Fakultet i Nastavnik u slučaju bidirekcionne asocijacije je prikazan na slici 4.37. Ovakvi modeli se ne moraju specificirati, ako postoji konvencija u okviru razvojnog tima kako se konceptualni model implementira u programskom kodu.

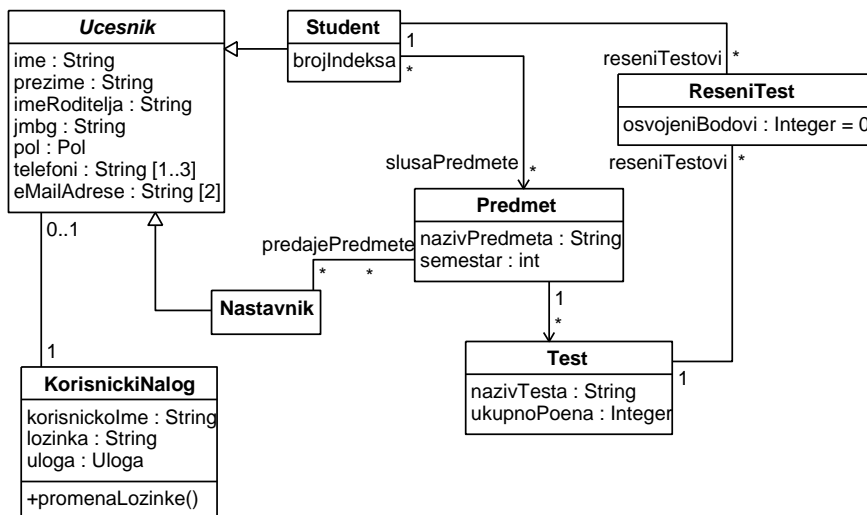
Domenske klase se mogu koristiti na način kako je specificirano u konceptualnom modelu, tako da implementacioni model nije neophodan, osim možda razvojnom timu koji nema dovoljno iskustva, pa softverski arhitekta mora da im specificira sve detalje.

#### 4.4.2 Veza generalizacije

Veza generalizacije (nasleđivanja) povezuje opštije elemente modela (predak, roditelj) sa specijalizovanim (naslednik, dete). Može se koristiti između klasa koje su iste vrste i dele zajedničke osobine i ponašanje. Specijalizovana klasa nasleđuje sva obeležja i metode opštije klase (pretka), pri čemu može dodavati svoja obeležja i redefinisati nasleđene metode. Na svim mestima gde stoji predak kao tip obeležja klase ili parametra metode, tokom izvršavanja se može proslediti instanca njegovog naslednika. U UML-u je podržano višestruko nasleđivanje.

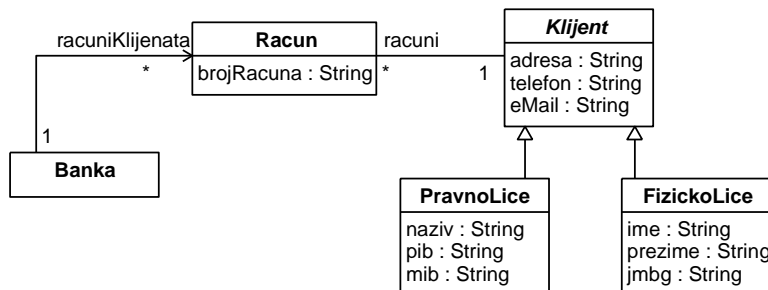
Na slici 4.38 i 4.39 vidimo primere korišćenja veze generalizacije u okviru konceptualnih modela. Na slici 4.38 klase Student i Nastavnik nasleđuju apstraktnu klasu Ucesnik, koja modeluje učesnika u nastavnom procesu u

okviru sistema za elektronsko ocenjivanje. Oni od učesnika nasleđuju njegova lična obeležja (ime, prezime, itd) i korisnički nalog za prijavu na sistem.



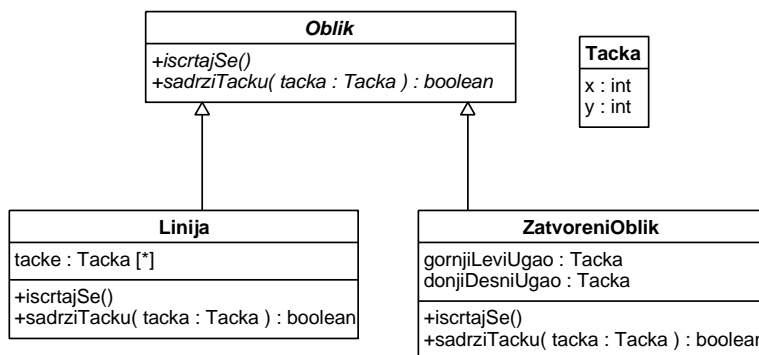
Slika 4.38 Klase Student i Nastavnik nasleđuju apstraktnu klasu Ucesnik koja modeluje učesnika u nastavnom procesu - primer korišćenja generalizacije u konceptualnom modelu

Na slici 4.39 vidimo da, u okviru informacionog sistema banke, **PravnoLice** (preduzeće) i **FizickoLice** (osoba) nasleđuju apstraktnu klasu **Klijent**, koja modeluje klijenta banke. Njihov odnos bi se mogao pročitati na sledeći način: klijent banke može biti fizičko ili pravno lice. Klijent može imati više računa u datoj banci. Nezavisno od vrste klijenta, banka radi sa njegovim računom na isti način: beleži uplate i isplate, obaveštava ga o stanju na računu i sl.



Slika 4.39 Klase PravnoLice i FizickoLice nasleđuju apstraktnu klasu Klijent koja modeluje klijenta banke - primer korišćenja generalizacije u konceptualnom modelu





Slika 4.40 Klase Linija i ZatvoreniOblik nasleđuju apstraktnu klasu Oblik – primer korišćenja generalizacije u implementacionom modelu

Na slici 4.40 vidimo primer veze generalizacije u okviru implementacionog modela. Klase Linija i ZatvoreniOblik nasleđuju apstraktnu klasu Oblik koja modeluje geometrijski oblik. Klasa Oblik uvodi dve metode, `iscrtajSe()` i `sadrziTacku(tacka: Tacka): boolean`, koje svaki naslednik treba da implementira u skladu sa svojom geometrijom. Metode klase Oblik su ispisane zakošenim slovima, što znači da su apstraktne (nemaju implementaciju).

**Napomena:** Smer generalizacije je od naslednika ka pretku. Česta je greška da se obrne smer!

#### 4.4.2.1 Prepoznavanje generalizacija u specifikaciji zahteva

Veza generalizacije se obično povlači između dve klase za koje primetimo da dele zajedničke osobine ili ponašanje. Međutim, to nije dovoljno: potrebno je i da date klase budu *iste vrste*, odnosno, da jedna bude *specijalizacija* druge. Za primere sa slika 4.38, 4.39 i 4.40 bismo mogli reći sledeće:

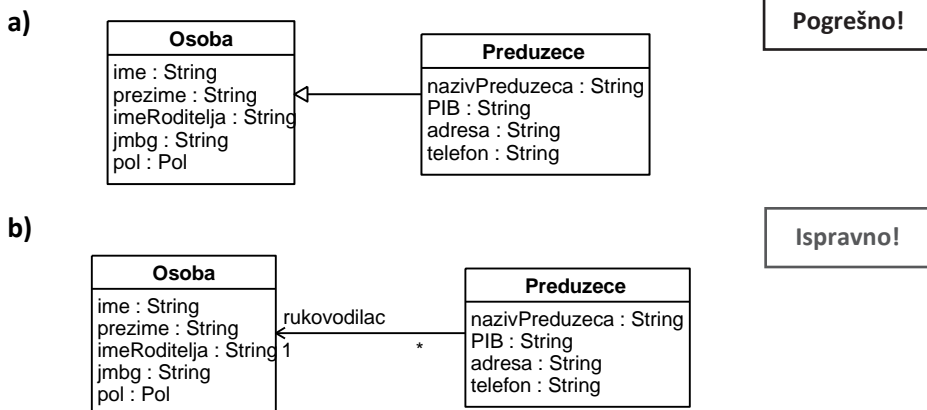
- Nastavnik i student **su** učesnici u nastavnom procesu.
- Klijenti banke **mogu biti** fizička i pravna lica
- Linija i zatvoreni oblik **su** geometrijski oblici.

Razmotrimo sledeću specifikaciju zahteva u okviru primera 4.5.

**Primer 4.5** Podaci koje je potrebno voditi o zaposlenima su: prezime, ime, ime roditelja, jedinstveni matični broj građana (JMBG) i pol. Podaci koje je potrebno voditi o preduzeću su: naziv, poreski broj (PIB), adresa, telefon, prezime, ime i JMBG rukovodioca. Podaci koje je potrebno voditi o klijentima koji su fizička lica su: prezime, ime, ime roditelja, JMBG, pol, adresa, telefon.

Na slici 4.41a vidimo da je razvojni tim zajedničke podatke zaposlenih i klijenata stavio u klasu *Osoba*. Klasa *Preduzece* je modelovana kao naslednik klase *Osoba*, zato što sa njom deli neka obeležja (ime, prezime i JMBG).

Ovo nije dobar način modelovanja zato što preduzeće nije vrsta osobe. Ispravan način je prikazan na slici 4.41b, gde je iskorišćena veza asocijacije: preduzeće ima jednu osobu koja je rukovodilac.



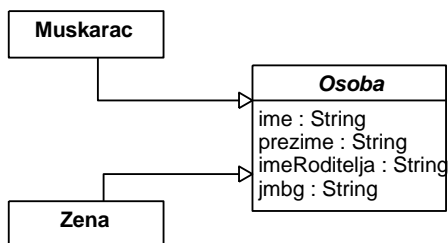
Slika 4.41 a) Primer pogrešno upotrebljenog nasleđivanja klase *Osoba* radi modelovanja rukovodioca preduzeća b) Ispravan način da se modeluje rukovodilac

Veza generalizacije kreira jaku zavisnost između pretka i naslednika: svaka promena pretka direktno utiče na naslednika. Naslednik mora preuzeti sve osobine pretka – ne može izostaviti one koje ne želi. Ako nam nisu potrebne sve osobine neke klase, verovatno je u datoj situaciji primerenija veza asocijacije ili veza implementacije interfejsa (videti sekciju 4.4.3). Ako za naslednika kojeg želimo da modelujemo možemo da odredimo kardinalitet (npr. preduzeće ima tačno jednog rukovodioca), umesto nasleđivanja treba upotrebiti asocijaciju.

Na slici 4.42a vidimo još jedan primer loše organizovane hijerarhije klasa. Iako muškarac i žena jesu osobe, ako nemaju nijedno dodatno obeležje ili metodu koja je bitna za sistem koji se implementira, ne bi ih trebalo modelovati kao posebne klase.

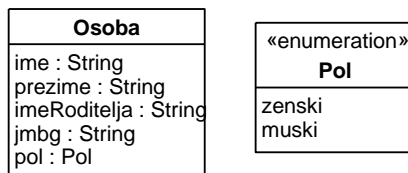
Klasifikacija (podela) osoba po nekom kriterijumu se u ovakvim slučajevima može modelovati dodavanjem obeležja nabrojanog tipa (slika 4.42b). Da bismo koristili vezu generalizacije, potrebno je i da razmatrane klase imaju dovoljno razlika da bi nasleđivanje imalo smisla, nije dovoljno samo da budu iste vrste.

a)



Pogrešno!

b)

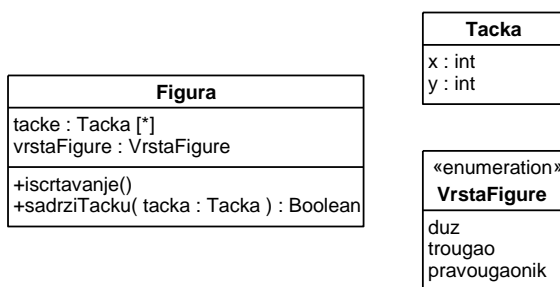


Ispravno!

Slika 4.42 a) Primer pogrešno upotrebljene generalizacije radi uvođenja podele (klasifikacije) osoba na muškarce i žene b) Ispravan način za klasifikaciju osoba uvođenjem obeležja pol nabrojanog tipa

#### 4.4.2.2 Prepoznavanje generalizacija u programskom kodu

Agilni način razvoja podrazumeva česte iteracije između kreiranja modela i programskog koda. U nekim situacijama, ako ne primetimo tokom modelovanja da nam je potrebna hijerarhija klasa, možemo je kasnije otkriti analizom programskog koda.



Slika 4.43 Vrste figura se specificiraju obeležjem nabrojanog tipa vrstaFigure, na osnovu čije vrednosti se vrši izbor algoritma za određivanje da li tačka pripada figuri

Na slici 4.43 je prikazana klasa Figura koja može biti duž, trougao ili pravougaonik. Ona poseduje listu tačaka koje je potrebno spojiti dužima, radi iscrtavanja figure. Tačke su poredane po redosledu kako je potrebno obaviti iscrtavanje. Na osnovu vrednosti obeležja vrstaFigure se bira algoritam za određivanje da li se zadata tačka nalazi unutar figure (listing 4.24).

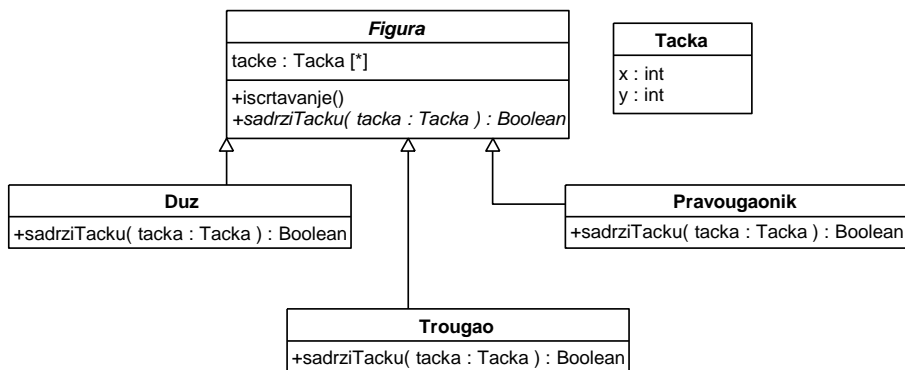
```

public Boolean sadrziTacku(Tacka tacka) {
    switch(vrstaFigure) {
        case PRAVOUGAONIK:
            // provera da li pravougaonik sadrži tačku
            break;
        case DUZ:
            // provera da li duž sadrži tačku
            break;
        case TROUGAO:
            // provera da li trougao sadrži tačku
            break;
        default:
            // ...
    }
    ...
}

```

Listing 4.24 Ilustracija implementacije metode za određivanje da li figura sadrži prosleđenu tačku, za model sa slike 4.43

U programskom kodu, indikator da nam nedostaje nasleđivanje može biti postojanje glomaznih switch ili if... else if... else if... iskaza, koji se izvršavaju u zavisnosti od vrednosti obeležja kojim se vrši klasifikacija (obično je u pitanju obeležje nabrojanog tipa).



Slika 4.44 Model sa slike 4.43 je promenjen tako da se, umesto jedne klase sa obeležjem nabrojanog tipa, kreira hijerarhija klasa

Problem sa switch iskazima, a još više sa if... else if... else if... iskazima, je što su teški za održavanje. Svaka izmena u bilo kojem njihovom segmentu potencijalno može ugroziti funkcionisanje ostalih segmenata. Uvođenje nove vrste figure zahteva pažljivo testiranje i novog i postojećeg koda.

U ovakvim situacijama, bolji način je da se, umesto obeležja nabrojanog tipa, kreiraju naslednici, koji će implementirati traženo ponašanje u skladu sa svojom prirodom. Na ovaj način, izmena je lokalizovana, prethodno testiran kod u

ostalim klasama ostaje netaknut, a metoda svake klase implementira samo jedan, jasno definisan zadatak.

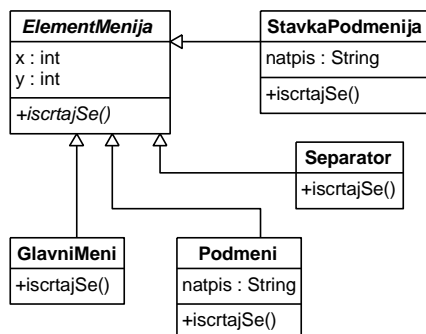
#### 4.4.2.3 Kompozitni (*Composite*) projektni šablon

Razmotrimo specifikaciju zahteva datu u primeru 4.6.

**Primer 4.6** *Potrebno je projektovati biblioteku grafičkih komponenti za realizaciju menija aplikacije. Glavni meni aplikacije se može sastojati od podmenija. Podmeni se može sastojati od drugih podmenija, stavki menija i separatora.*

*Glavni meni, podmeni, stavka menija i separatori imaju poziciju na kojoj se iscrtavaju (x i y koordinata). Podmeni i stavka menija imaju i natpis. Sve navedene komponente se mogu iscrtati.*

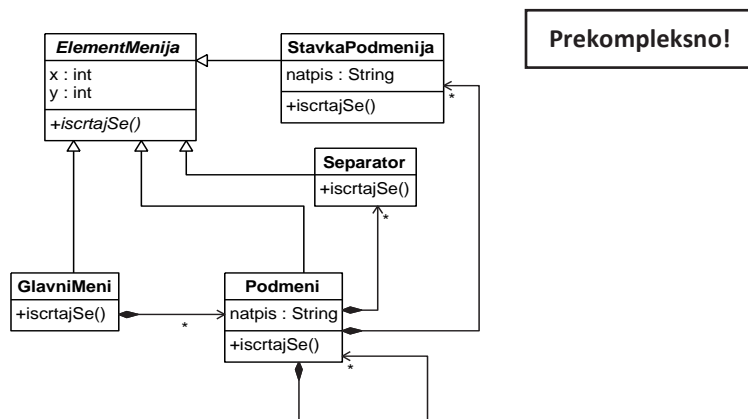
Iz specifikacije se može zaključiti da je potrebno kreirati jednu apstraktnu klasu `ElementMenija` koja sadrži zajednička obeležja x i y i metodu za iscrtavanje. Nju nasleđuju klase koje modeluju stavku menija, separator, podmeni i glavni meni. Naslednici implementiraju metodu za iscrtavanje u skladu sa svojim izgledom (slika 4.45).



Slika 4.45 Početak modelovanja biblioteke menija – uočene su osobine pretka hijerarhije i njegovih naslednika

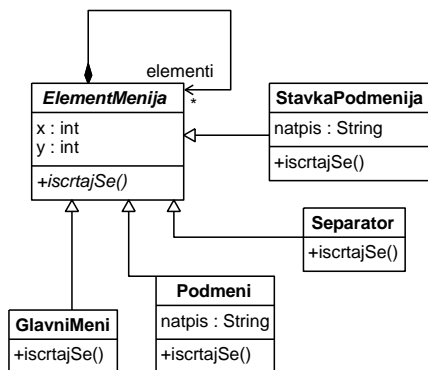
Preostalo je da se modeluju veze između učenih klasa. Direktnim prevođenjem specifikacije zahteva na asocijacije dobijamo dijagram na slici 4.46 (glavni meni se sastoji od podmenija, a podmeni od drugih podmenija, stavki menija i separatora). Ovakav dijagram je previše komplikovan za implementaciju, pošto

kao posledicu svake asocijacije u programskom kodu dobijamo jednu kolekciju koju treba da održavamo.



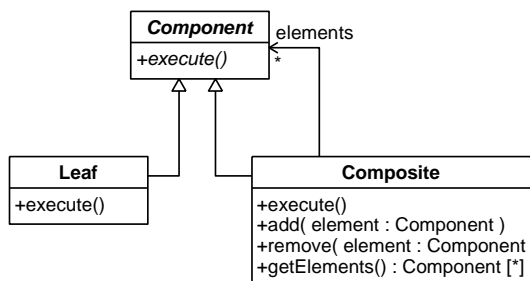
Slika 4.46 Povezivanje uočenih klasa direktnim prevođenjem specifikacije zahteva na asocijacije. Rezultat je prekompleksan model težak za održavanje.

Na slici 4.47 je prikazano bolje rešenje u odnosu na prethodno: uvedena je rekurzivna asocijacija na nivou pretka hijerarhije koja omogućava da svaki element menija može imati kolekciju elemenata menija (odnosno, njegovih naslednika). Rešenje je znatno jednostavnije, ali u ovom slučaju Stavka-Podmenija i Separator takođe imaju kolekciju elemenata, koja im nije potrebna.



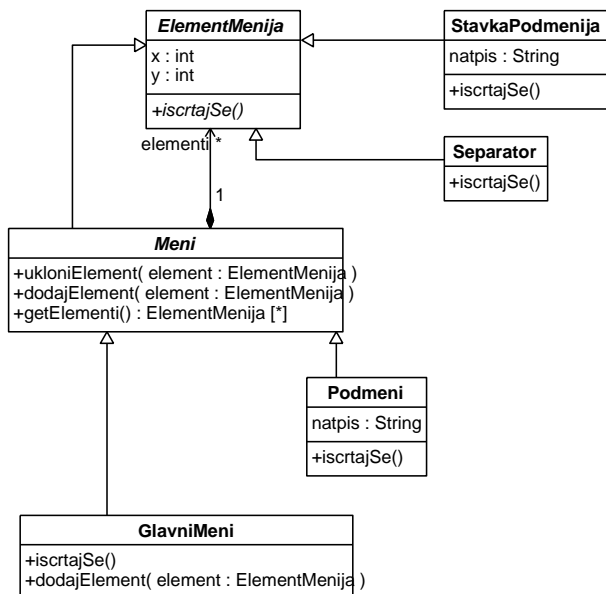
Slika 4.47 Asocijacije između različitih elemenata menija sa slike 4.46 su zamenjene jednom rekurzivnom asocijacijom na nivou pretka hijerarhije, što omogućava da svaki element menija ima kolekciju drugih elemenata menija. Problem je što stavci menija i separatoru nije potrebna data kolekcija.

Optimalno rešenje je bazirano na kompozitnom projektnom šablonu [Gamma94]. Dati šablon se primenjuje u slučaju kada komponente neke hijerarhije dele zajedničke osobine, pri čemu su neke komponente kompozitne: sastoje se od drugih komponenti iste vrste (slika 4.48). Kompozitna komponenta treba da poseduje metode za rukovanje elementima kolekcije. Metoda `execute()` implementira ponašanje koje je karakteristično za svaku komponentu.



Slika 4.48 Kompozitni projektni šablon (*Composite*)

Na slici 4.49 vidimo primenu kompozitnog projektnog šablona na hijerarhiju klasa iz primera 4.6. Apstraktna klasa `Meni` je uvedena kao predak za dve kompozitne komponente, klase `Podmeni` i `GlavniMeni`. Ona implementira



Slika 4.49 Biblioteka menija je projektovana primenom kompozitnog projektnog šablona. Kolekciju elemenata imaju samo one klase kojima je to potrebno.

metode za rukovanje elementima kolekcije. Metoda za dodavanje dozvoljava da se u kolekciju uključi samo separator, stavka menija i podmeni, ne i glavni meni. Metoda za dodavanje klase GlavniMeni redefiniše tu metodu tako da dozvoli samo unos podmenija.

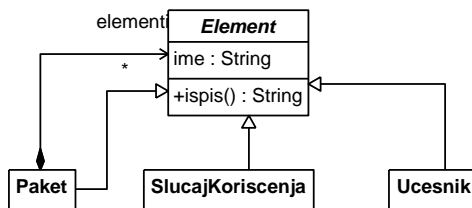
Na ovaj način, dobijamo jednostavan dijagram, kod kojeg kolekciju elemenata imaju samo one klase kojima je to potrebno (kompozitni elementi hijerarhije).

Razmotrimo još primer 4.7.

**Primer 4.7** Potrebno je projektovati strukturu podataka za čuvanje informacija o pojednostavljenim dijagramima slučajeva korišćenja. Svaki dijagram se sastoji od proizvoljno mnogo elemenata. Elementi su: učesnici (actor), slučajevi korišćenja, veze i paketi. Veze su: asocijacija (između učesnika i slučaja korišćenja), generalizacija (između dva učesnika ili između dva slučaja korišćenja) i veza zavisnosti (između dva slučaja korišćenja). Paketi se koriste za grupisanje učesnika, slučajeva korišćenja, veza i potpaketa.

Svi navedeni elementi imaju ime. Dijagram takođe ima ime. Slučaj korišćenja ima i korake, preduslove i posledice (stringovi). Svi elementi dijagrama slučaja korišćenja treba da imaju mogućnost ispisa. Paket i dijagram se ispisuju tako što ispišu svoje ime i tip a zatim i sve svoje elemente u obliku stabla. Elementi koji nisu paketi ispisuju samo svoje ime i tip. Ukoliko je element nekog paketa drugi paket, potrebno je prikazati i njegov sadržaj.

Možemo primetiti da svi pomenuti elementi dijagrama slučajeva korišćenja imaju zajedničke osobine i ponašanje: ime, mogućnost ispisa i mogućnost da budu elementi paketa. Paket može biti element drugog paketa. Jedino dijagram ne može biti ničiji element.



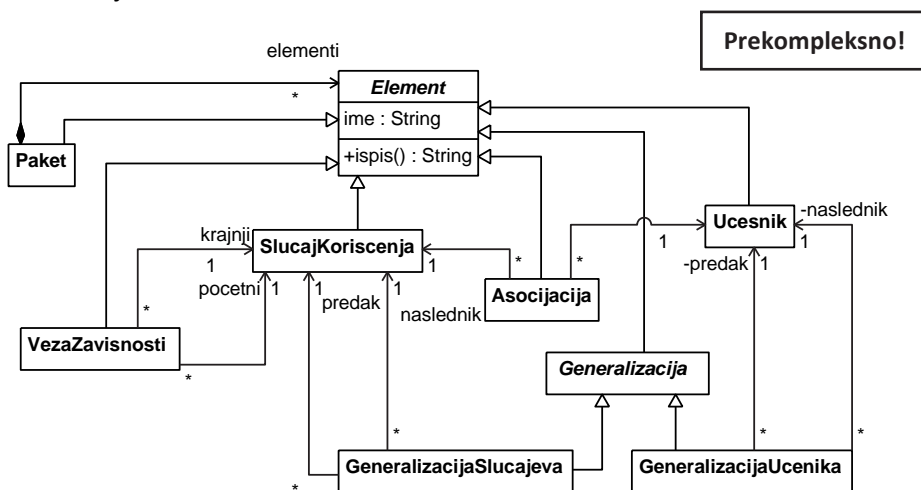
Slika 4.50 Početak modelovanja elemenata dijagrama slučajeva korišćenja na bazi kompozitnog projektnog šablona

Sve navedeno sugeriše da je potrebno uvesti apstraktnu klasu `Element` koja uvodi osnovne osobine, obeležje `ime` i metodu `ispis()`, koju bi ostali elementi dijagrama trebalo da naslede i implementiraju u skladu sa svojom prirodom.



Pošto paket ima zajedničke osobine sa drugim elementima dijagrama, a pri tome se sastoji od drugih elemenata, nameće se korišćenje kompozitnog šablona (slika 4.50).

Ostaje još da modelujemo veze. Veze su takođe elementi dijagrama, tako da je logično da naslede Element. Direktnim prevođenjem specifikacije zahteva na dijagram klasa, dobija se veoma komplikovano rešenje, sa mnoštvom asocijacija koje opisuju koja veza se može naći između kojih tipova elemenata dijagrama (slika 4.51). Ovakva rešenja se često viđaju kod osoba koje počinju da uče modelovanje.



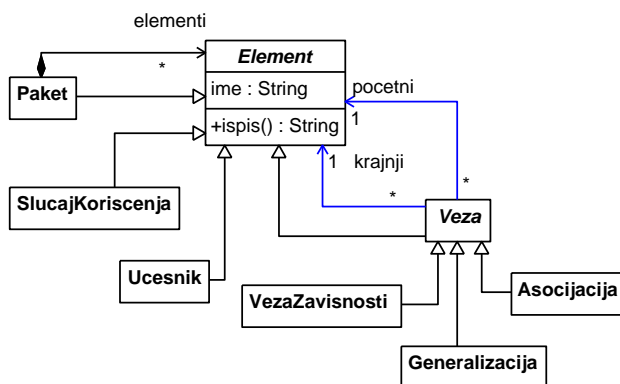
Slika 4.51 Model koji nastaje direktnim prevođenjem specifikacije zahteva na asocijacije. Rezultat je prekompleksan model težak za implementaciju i održavanje.

Dijagram klasa na slici 4.51 zahteva puno napora da bi se implementirao, testirao i održavao. Pored prikazanog domenskog sloja, treba implementirati i sloj korisničkog interfejsa koji komunicira sa korisnikom, kao i procedure za skladištenje i učitavanje, tako da se kompleksnost domenskog modela preslikava i na njih. Model koji izgleda kao na slici 4.51 bi trebalo da nas podstakne da nađemo način kako da ga pojednostavimo.

Pošto se sve veze povlače između tačno dva elementa, možemo značajno pojednostaviti model ako uvedemo apstraktnu klasu Veza, koja ima početni i krajnji element koje spaja (slika 4.52). Bitno je znati koji je početni a koji krajnji element, zato što su generalizacija i veza zavisnosti usmerene.

Ovakvo rešenje je puno jednostavnije, ali je sada odnos između klasa Veza i Element takav da instanca svakog naslednika klase Element može postati

početni ili krajnji element, uključujući i veze i pakete. Na modelu sa slike 4.51 to nije bio slučaj, jer je svaki tip veze tačno specificirao tipove elemenata koje povezuje.

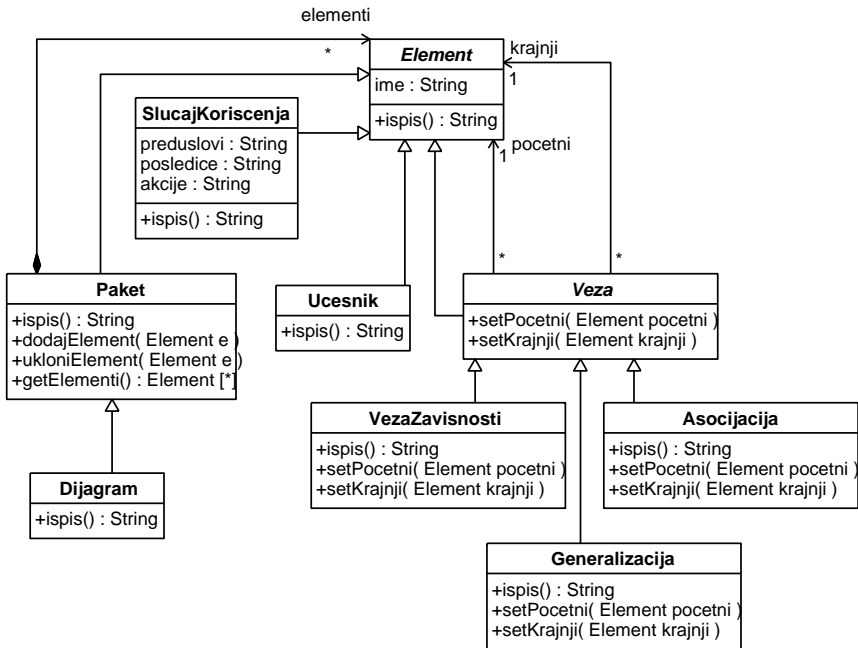


Slika 4.52 Model sa slike 4.51 je značajno pojednostavljen uvođenjem apstraktne klase Veza, koja može spojiti bilo koja dva elementa dijagrama slučaja korišćenja

Ovo je česta posledica uvođenja generičkih rešenja koje treba biti svestan. Rešava se metodama koje vrše proveru pri dodeli vrednosti obeležja ili pri unosu elementa u kolekcije, tako da se ne dozvole neregularne situacije.

Na slici 4.52 se nalazi završen model, sa obeležjima i metodama. Metode za dodelu vrednosti za početni i krajnji element veze, `setPocetni()` i `setKrajnji()`, dozvoljavaju unos samo odgovarajućih elemenata za tu vezu, inače aktiviraju izuzetke. Ako je dodeljen početni element, metoda za dodelu krajnjeg elementa proverava i njegov tip. Na primer, ako je početni element klase `Generalizacija` instanca klase `Ucesnik`, to mora biti i krajnji element. Metoda `setKrajnji()` u okviru klase `Generalizacija` ne sme da dozvoli da početni i krajnji element budu isti (element ne sme da nasledi sam sebe). Ako je početni element klase `Asocijacija` instance klase `Ucesnik`, krajnji mora biti instance klase `SlucajKoriscenja` i obrnuto. Oba elementa klase `VezaZavisnosti` moraju biti slučajevi korišćenja.

Sličnu proveru treba da sprovodi i metoda za dodavanje elementa u okviru paketa. Dijagram se ne može uključiti u paket. Paket se ne može uključiti sam u sebe.



Slika 4.53 Završen dijagram klasa koji modeluje elemente dijagrama slučajeva korišćenja. Metode se koriste da spreče nedozvoljeno povezivanje elemenata.

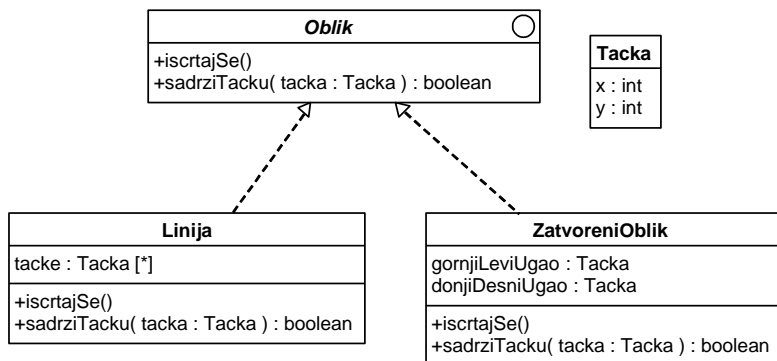
**Napomena:** Model sa slike 4.53, koji modeluje strukturu nekog jezika (u ovom slučaju dijagrama slučajeva korišćenja), naziva se meta-model. Meta-model UML-a se sastoji od više stotina meta-klasa i specificira strukturu UML-a koju proizvođači alata za modelovanje koriste kao ulaznu specifikaciju [UML2.5.1]

#### 4.4.3 Implementacija interfejsa

U programskim jezicima poput Java ili C#, interfejsi omogućavaju da se propiše željeno ponašanje. Oni specificiraju skup metoda koje svaka klasa, koja implementira dati interfejs, mora da podrži na sebi svojstven način. S obzirom da klasa može implementirati više interfejsa, dobijamo efekat kao kod višestrukog nasleđivanja.

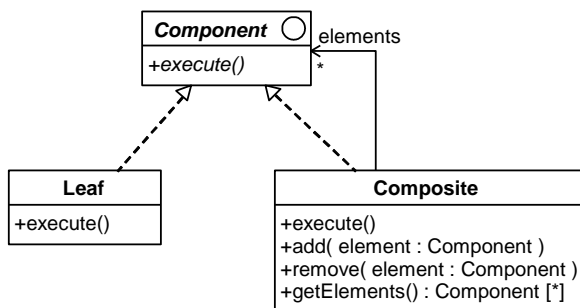
Implementacija interfejsa je manje striktna od veze generalizacije: klase koje implementiraju neki interfejs ne moraju imati ništa zajedničko, osim ponašanja

propisanog interfejsom. Generalizacija se sme koristiti isključivo između klasa iste vrste, gde naslednik predstavlja specijalizaciju pretka.



Slika 4.54 Dijagram sa slike 4.40 u okviru kojeg je, umesto apstraktne klase Oblik upotrebljen interfejs.

Na slici 4.54 je situacija sa dijagrama na slici 4.40 ponovo modelovana korišćenjem interfejsa, umesto apstraktne klase Oblik. Slično, na slici 4.55 vidimo kompozitni projektni šablon modelovan korišćenjem interfejsa, umesto apstraktne klase, kao na slici 4.48.



Slika 4.55 Kompozitni projektni šablon sa slike 4.48 koji je modelovan korišćenjem interfejsa, umesto apstraktne klase Component

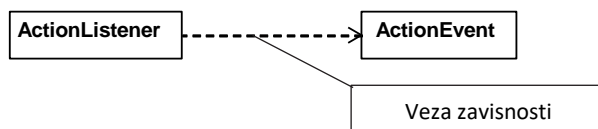
Primenu interfejsa ćemo videti i u sekciji 4.5, za potrebe povezivanja domenskih klasa sa klasama korisničkog interfejsa.

#### 4.4.4 Veza zavisnosti

Veza zavisnosti označava da jedna klasa zavisi od druge. Ako se implementacija nezavisne klase promeni, verovatno će biti potrebna i promena implementacije zavisne klase koja koristi njene usluge.

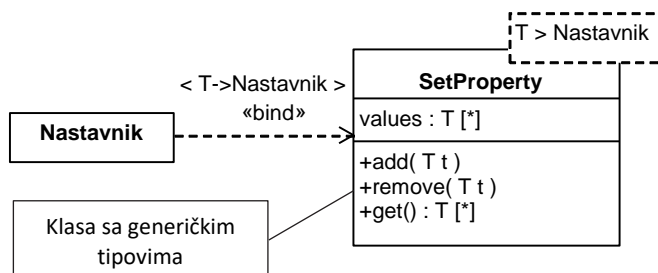
Veze koje smo do sada videli (asocijacija, generalizacija, implementacija interfejsa) takođe označavaju da između klasa koje povezuju postoji zavisnost. Veza prikazana na slici 4.56 se može koristiti ako želimo da na modelu specificiramo da između dve klase postoji zavisnost, a pri tome nijedna od do sada korišćenih veza nije odgovarajuća. Primer može biti situacija kada jedna klasa kreira instance druge klase, ili kada se instanca jedne klase prenosi kao argument metode druge klase, pri čemu se date instance ne čuvaju u obeležjima zavisne klase (što bi rezultovalo vezom asocijacije).

Na slici 4.56 je dat primer veze zavisnosti između klase `ActionListener` i klase `ActionEvent`. Klasa `ActionListener` dobija instancu klase `ActionEvent` kao parametar svoje metode `actionPerformed`, tako da zavisi od klase `ActionEvent`.



Slika 4.56 Primer veze zavisnosti između klase `ActionListener` i klase `ActionEvent`.

Na slici 4.57 je prikazana veza zavisnosti koja se koristi za povezivanje generičkog tipa `T` klase `SetProperty` sa konkretnom klasom `Nastavnik`. Ovim dobijamo kolekciju instanci klase `Nastavnik` i metode za rad sa tom kolekcijom.



Slika 4.57 Primer specijalne veze zavisnosti koja se koristi za povezivanje generičkog tipa `T` klase `SetProperty` sa konkretnim tipom `Nastavnik`

Veze zavisnosti treba stavljati na dijagram samo ako želimo da posebno naglasimo zavisnost dve klase između kojih nema drugih vrsta veza. Ako se previše koriste, mogu učiniti dijagram nečitkim, pošto odvlače pažnju od veza koje imaju direktan uticaj na programski kod, odnosno na šemu baze podataka.

## 4.5 Modelovanje korisničkog interfejsa

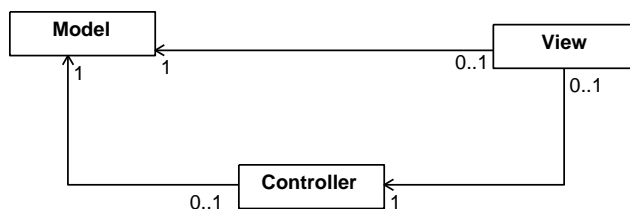
U prethodnim sekcijama smo se većinom bavili projektovanjem konceptualnih modela, na osnovu kojih se vrši implementacija domenskih klasa i šeme baze podataka. Da bismo zaokružili priču, ostalo je da proučimo projektovanje klasa korisničkog interfejsa i načine za njihovo sprezanje sa domenskim klasama.

Klase korisničkog interfejsa treba projektovati tako da ostvarimo striktnu podelu nadležnosti između njih i domenskih klasa. Podela nadležnosti znači da svaka klasa obavlja samo onaj posao koji je za nju karakterističan, a sa ostatkom klasa iz sistema, potrebnih za implementaciju nekog korisničkog zahteva, komunicira putem uspostavljenih protokola.

U stručnoj literaturi je široko zastupljena preporuka da se saradnja klasa korisničkog interfejsa i ostatka sistema organizuje prema MVC (*Model-View-Controller*) arhitektonskom šablonu. U nastavku je obrađen osnovni MVC šablon, kao i MVC proširen *Observer* projektnim šablonom. Pomenuti šabloni su ilustrovani programskim kodom za implementaciju aplikacije za merenje dnevnih temperatura na jednom mernom mestu.

### 4.5.1 *Model-View-Controller* (MVC) arhitektonski šablon

MVC je tako organizovan da omogućiti striktnu podelu nadležnosti između tri segmenta: modela (*Model*), pogleda (*View*) i kontrolera (*Controller*). Cilj je da se dobije slabo spregnuta arhitektura, koja olakšava održavanje i testiranje (slika 4.58). U literaturi postoji više varijanti kako ova tri segmenta mogu sarađivati. Arhitektura prikazana u nastavku je bazirana na [Potel96]. Iscrpan opis različitih načina povezivanja korisničkog interfejsa sa klasama koje implementiraju poslovnu logiku se može naći u [Fowler06].



Slika 4.58 Osnovni MVC arhitektonski šablon

Pogled se sastoji od jedne ili više klasa korisničkog interfejsa, obično nasljednika neke grafičke biblioteke, kao što je npr. Swing<sup>17</sup> u Javi. Nadležnost pogleda je da obezbedi sredstvo za komunikaciju sa krajnjim korisnikom: treba da konstruiše prikaz na ekranu u kojem korisnik može da unosi i gleda podatke i pokreće akcije.

Kao odgovor na akciju korisnika, pogled poziva odgovarajuću metodu kontrolera i preko parametara joj prosleđuje podatke koje je korisnik uneo. Zadatak kontrolera je da izvrši proveru unetih podataka, po potrebi ih konvertuje u drugi format i pozove metodu modela koja implementira odgovor na datu akciju. Ako pri proveri otkrije greške, kontroler ih javlja pogledu aktiviranjem izuzetaka, koje pogled „hvata“ i prikazuje korisniku. Kontroler ne treba ništa da zna o pogledu, niti da mu se direktno „obraća“ (poziva njegove metode).

Klase koje čine model su domenske klase. Njihova nadležnost je da u okviru svojih obeležja čuvaju podatke potrebne za rad aplikacije, kao i da obezbede operacije za rad nad podacima (jednostavne get i set metode, kao i složenije operacije iz domena poslovne logike). Model se projektuje tako da ne zavisi od pogleda i kontrolera. Na slici 4.58 možemo primetiti da je asocijacija usmerena od pogleda ka modelu, odnosno od kontrolera ka modelu i da je kardinalitet na suprotnom kraju asocijacije u odnosu na model 0..1.

Pogled se modelu direktno „obraća“ jedino radi čitanja podataka koje treba da prikaže korisniku - nikada direktno ne poziva operacije nad modelom koje mu menjaju stanje, već za to koristi kontroler, kao posrednika.

Ako model treba da „nešto javi“ pogledu, može to da uradi aktiviranjem događaja i izuzetaka. Zadatak pogleda je da reaguje na primljeni događaj (obično osvežavanjem prikaza) i da „uhvati“ izuzetak i prikaže odgovarajuću poruku korisniku. Direktno pozivanje metoda pogleda od strane modela nije dozvoljeno.

Često se postavlja pitanje zašto nam je potreban kontroler? Provera i priprema unetih podataka bi mogle da se implementiraju i u okviru pogleda (svakako ima te podatke u svojim grafičkim komponentama), a mogao bi i da direktno pozove metode modela, jer ima referencu na njega. Izgleda kao da kontroler nepotrebno usložnjava arhitekturu.

Jedan razlog za postojanje kontrolera je što će u nekim situacijama više različitih pogleda raditi nad istim modelom (npr. podaci se mogu unositi kroz dijaloge i preko tabele). U ovom slučaju, oba pogleda mogu koristiti isti kontroler, umesto

---

<sup>17</sup> <https://docs.oracle.com/javase/8/docs/technotes/guides/swing/>

da svaki implementira svoju verziju koda za proveru i prilagođavanje unetih podataka i komunikaciju sa modelom.

Drugi razlog je radi podrške za automatsko testiranje, korišćenjem okvira za testiranje kao što je npr. JUnit<sup>18</sup>. Odvajanjem funkcionalnosti u kontroler, dobijamo mogućnost da testiramo veći procenat koda aplikacije na jednostavniji način. Pošto kontroler nema elemente korisničkog interfejsa, može da se instancira u okviru JUnit testa i da se obavi provera rada svih njegovih metoda. S obzirom da metode kontrolera obično pozivaju metode modela, ovim se testira i njihova saradnja. Naravno, model treba da ima i svoje testove, nezavisno od kontrolera.

**Napomena:** Kada se pišu testovi, treba proveriti ponašanje u regularnim situacijama (tzv. *happy flow*) kao i u neregularnim, kada se testira da li će biti aktiviran baš onaj izuzetak koji se u određenoj situaciji očekuje.

Za automatsko testiranje korisničkog interfejsa su potrebna specijalizovana okruženja koja imaju mogućnost da instanciraju klasu korisničkog interfejsa i simuliraju akcije miša i tastature na osnovu snimljenih sesija korišćenja. Problem je što, ako promenimo geometriju prozora ili raspored komponenti, testovi prestaju da funkcionišu. Bez ovakvih okruženja, testiranje pogleda mora da se obavlja ručno, od strane razvojnog tima, što je spor i nepouzdan proces. Ni u jednom slučaju, testiranje korisničkog interfejsa nije jednostavno.

Iz navedenih razloga, preporuka je da se klase korisničkog interfejsa implementiraju tako da sadrže samo neophodan programski kod, koji se ne može smestiti u model ili kontroler. Ideja je da se poveća verovatnoća da neće biti grešaka u korisničkom interfejsu, ako i ne postoji mogućnost za njegovo automatsko testiranje.

Treći razlog za postojanje kontrolera je radi podrške troslojnih aplikacija. Sva poslovna logika se smešta u model i kontroler i izvršava se na serveru (*backend* deo), a klijentski deo aplikacije, koji može biti implementiran za različite platforme (web, mobilne aplikacije, desktop aplikacije), bavi se samo konstrukcijom prikaza na datoj platformi, prosleđivanjem zahteva korisnika i prikazom dobijenih odgovora od serverske strane.

---

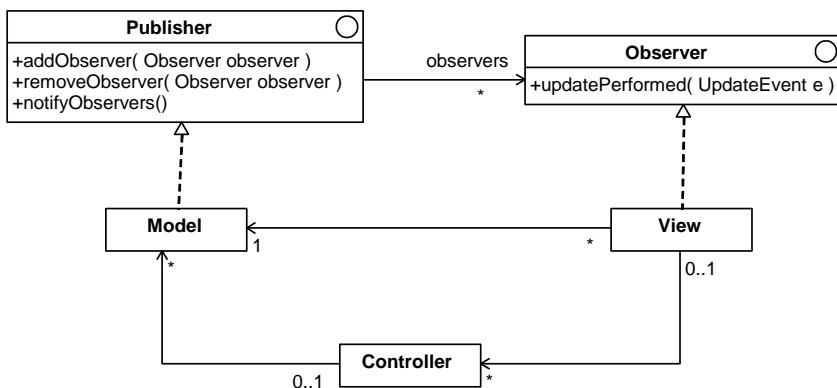
<sup>18</sup> [https://download.oracle.com/otn\\_hosted\\_doc/jdeveloper/904preview/junit\\_doc/junit\\_main.html](https://download.oracle.com/otn_hosted_doc/jdeveloper/904preview/junit_doc/junit_main.html)



## 4.5.2 MVC sa *Observer* šablonom

Na slici 4.58 je prikazana situacija kada imamo jedan model, koji se prikazuje u okviru jednog pogleda, koji prosleđuje akcije i podatke korisnika jednom kontroleru. Međutim, nekad postoji potreba da se podaci iz modela prikazuju i menjaju na različite načine, na primer: korišćenjem površine za crtanje, u okviru tabele, preko komponente tipa stabla (*tree-view*), kroz dijaloge i sl. Ako se kroz bilo koji od navedenih pogleda promeni model, preostali pogledi treba da trenutno prikažu promenu, ako su vidljivi na ekranu.

Da bismo mogli da podržimo ovakvo ponašanje, potrebno je da osnovni MVC šablon proširimo *Observer* šablonom (slika 4.59).



Slika 4.59 MVC arhitektonski šablon proširen *Observer* projektnim šablonom

*Observer* šablon se koristi u situacijama kada je potrebno omogućiti da više klasa prate promene stanja jedne klase, a da pri tome ne dođe do čvrstog sprežanja između njih [Gamma04]. Da bi se to postiglo, šablon specificira dva interfejsa: *Publisher* i *Observer* (listing 4.25).

```
public interface Publisher {
    public void addObserver(Observer observer);
    public void removeObserver(Observer observer);
    public void notifyObservers();
}

public interface Observer {
    public void updatePerformed(EventObject e);
}
```

Listing 4.25 Interfejsi koji omogućavaju implementaciju *Observer* šablona

*Publisher* interfejs implementira klasa koja treba da omogući praćenje promena svojih podataka. To postiže:

- kreiranjem kolekcije `observers` u kojoj će se nalaziti „posmatrači“ (sve klase koje implementiraju `Observer` interfejs)
- metodama koje omogućavaju da se „posmatrači“ dodaju u `observers` kolekciju i uklone iz nje (`addObserver` i `removeObserver`) i
- metodom `notifyObservers` kojom obaveštava sve „posmatrače“ da je došlo do promene (listing 4.26).

Obaveštavanje se odvija tako što metoda `notifyObservers` poziva metodu `updatePerformed` koju implementiraju „posmatrači“ i prosleđuje joj događaj koji opisuje vrstu promene koja se desila.

```
public class KlasaKojaSePrati implements Publisher {
    ...
    private List<Observer> observers;

    public void addObserver(Observer observer) {
        if (null == observers)
            observers = new ArrayList<Observer>();
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        if (null == observers)
            return;
        observers.remove(observer);
    }

    public void notifyObservers() {
        UpdateEvent e = new EventObject(this);
        for (Observer observer : observers) {
            observer.updatePerformed(e);
        }
    }

    public void setX(int x) {
        //promena podataka koja se prati
        ...
        notifyObservers();
    }
    ...
}
```

Listing 4.26 Primer implementacije `Publisher` interfejsa od strane klase koja treba da omogući praćenje promena svojih podataka

`Observer` interfejs implementiraju sve klase koje „žele“ da prate promenu stanja klase koja implementira `Publisher` interfejs. To podrazumeva implementaciju metode `updatePerformed` koja se poziva od strane metode `notifyObservers` kada dođe do promene. Svaki „posmatrač“ implementira tu metodu tako da na sebi svojstven način reaguje na promenu.

```

public class KlasaPosmatrac implements Observer {
    ...
    public void updatePerformed(EventObject e) {
        // reakcija na promenu
    }
    ...
}

```

Listing 4.27 Primer implementacije Observer interfejsa od strane klase koja treba da prati promene podataka klase sa listinga 4.26

Kada se *Observer* šablon pridruži osnovnom MVC šablonu, tada model (domenska klasa) implementira *Publisher* interfejs, a svi pogledi koji treba da prate izmene na njemu implementiraju *Observer* interfejs (primer je dat na slici 4.61). U nekim situacijama, ako je kontroler veoma kompleksan i ima svoje podatke čiju promenu pogledi treba da prate, kontroler takođe može da implementira *Publisher* interfejs.

Umesto događaja koji je tipa *EventObject* na listingu 4.26, može se proslediti bilo koji njegov naslednik. Ako *Publisher* klasa treba da obaveštava posmatrača o različitim vrstama promena koje se mogu desiti, po potrebi se može kreirati hijerarhija događaja, gde svaki događaj u svojim obeležjima može prenositi informacije karakteristične za tu vrstu promene.

### 4.5.3 Primer primene MVC i *Observer* šablona

Radi ilustracije primene navedenih šablona, implementirana je klasa *Prozor-MeteoroloskeStanice*, naslednica klase *JDialog*. Dati prozor se koristi za unos temperatura koje su izmerene u toku jednog dana (klasa *DnevnaTemperatura*) na jednom mernom mestu.

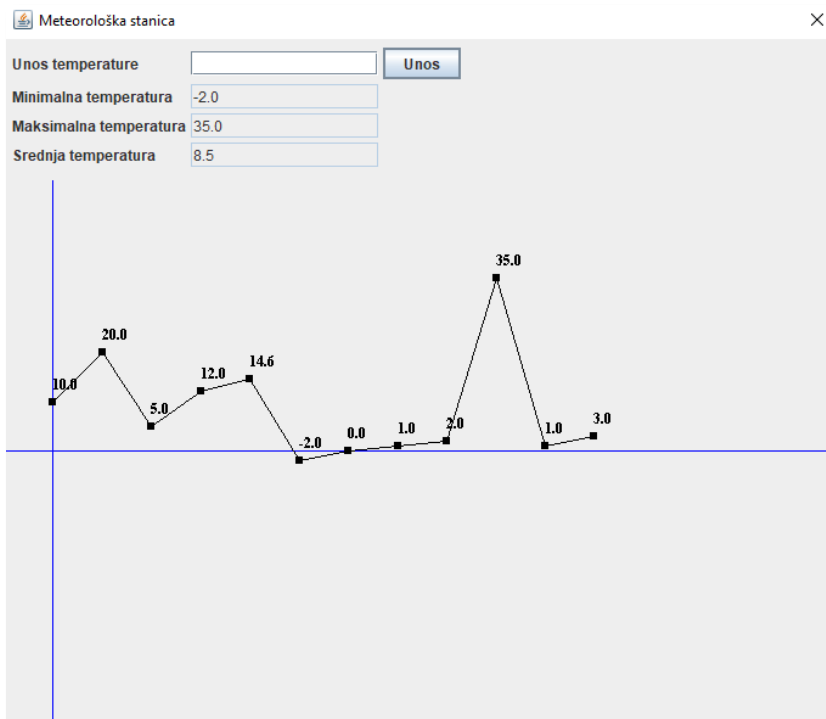
Korisnik unosi željenu vrednost u polje za unos temperature i pritiskom na dugme Unos pokreće akciju koja je pridružena dugmetu (slika 4.60). Uneta temperatura se prosleđuje kontroleru, kojeg implementira klasa *Kontroler-Stanice* (listing 4.29).

Posle svakog unosa temperature, računaju se i prikazuju minimalna, maksimalna i srednja vrednost temperature za taj dan, a novouneta temperatura se dodaje na grafik dnevnih temperatura. Za iscrtavanje grafika je zadužena klasa *PanelGrafik* koja nasleđuje klasu *JPanel*. *JDialog* i *JPanel* su klase *Swing* biblioteke u Javi.

Klasa *DnevnaTemperatura* ima ulogu modela u okviru MVC šablona. Ona se brine o podacima i obezbeđuje metode za kontrolisan rad sa njima. U odnosu na programski kod na listingu 4.6, proširena je metodama za implementaciju

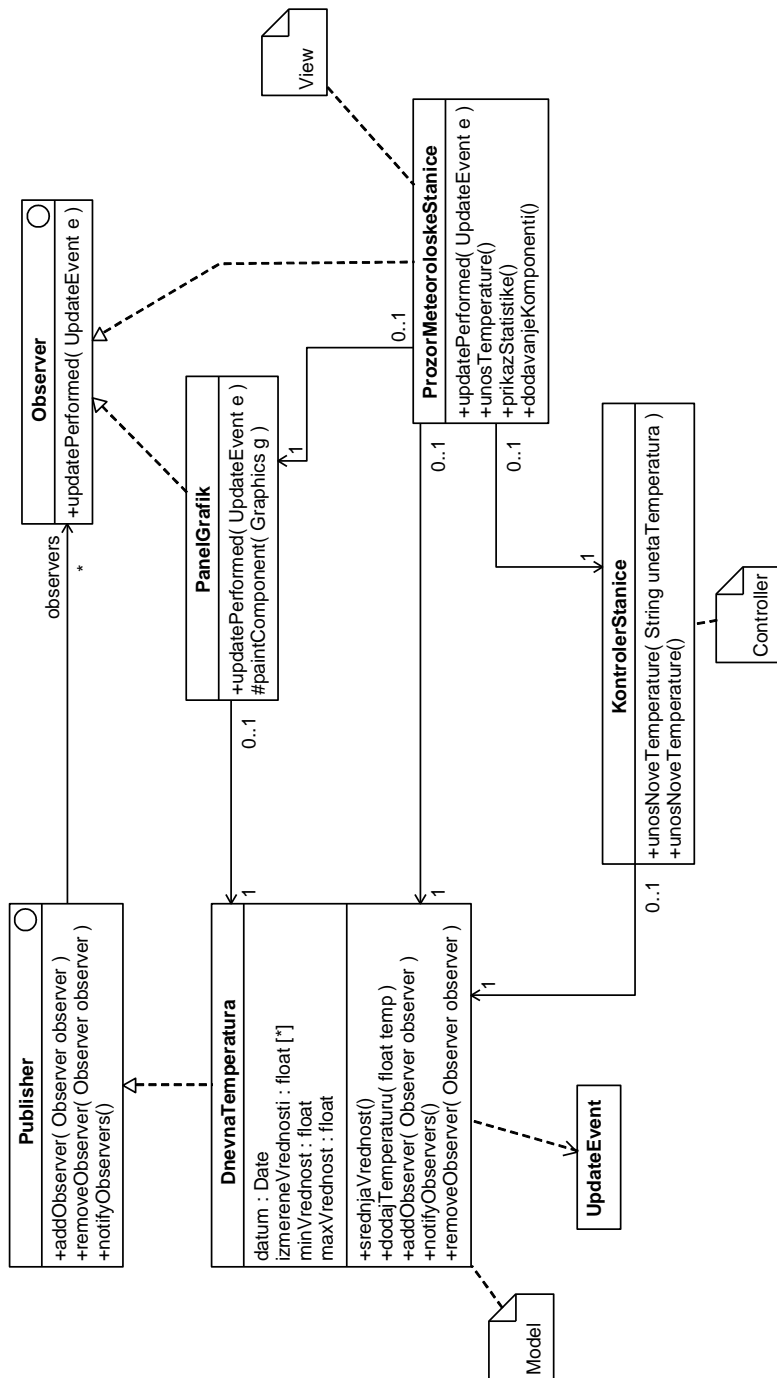
Publisher interfejsa i observers kolekcijom u kojoj će se nalaziti reference na sve koji „žele“ da prate promenu njenih podataka (listing 4.28).

Primetimo da DnevnaTemperatura, iako obezbeđuje kolekciju za registrovanje pogleda koji implementiraju Observer interfejs, ne zavisi direktno od njih: može se nezavisno instancirati i „ne zna“ ništa o njihovoj vrsti i načinu funkcionisanja. Kao „posmatrači“ se mogu dodati i pogledi koji još nisu bili implementirani u trenutku projektovanja date klase, što podržava budući razvoj sistema.



Slika 4.60 Prozor, naslednik klase `JDialog`, koji omogućava unos i prikaz izmerenih temperatura u toku jednog dana

`ProzorMeteoroloskeStanice` (listing 30) i `PanelGrafik` (listing 31) su dva pogleda koji na različite načine prikazuju podatke koje dobijaju od klase `DnevnaTemperatura`. `PanelGrafik` crta grafik dnevnih temperatura na osnovu podataka koje preuzima iz liste `izmereneVrednosti` u okviru klase `DnevnaTemperatura`. `ProzorMeteoroloskeStanice` prikazuju minimalnu, maksimalnu i srednju vrednost temperature koje dobija pozivima odgovarajućih metoda klase `DnevnaTemperatura`. Oba pogleda osvežavaju svoj prikaz svaki put kada ih klasa `DnevnaTemperatura` obavesti da je došlo do promene metodom `notifyObservers` koja poziva njihovu metodu `updatePerformed`.



Slika 4.61 Primer implementacije MVC šablona proširenog *Observer* šablonom. Domenska klasa *DnevnaTemperatura* je model u okviru MVC-a i implementira *Publisher* interfejs. *PanelGrafik* i *ProzorMeteoroloskeStanice* su pogledi i implementiraju *Observer* interfejs.

```

/**
 * DnevnaTemperatura čuva izmerene dnevne temperaturu na jednom
 * mernom mestu u jednom danu
 */
public class DnevnaTemperatura implements Publisher {
    public static final float MAX_TEMP = 50;
    public static final float MIN_TEMP = -50;

    private Date datum;
    private List<Float> izmereneVrednosti = null;
    private float minVrednost = MAX_TEMP;
    private float maxVrednost = MIN_TEMP;

    public DnevnaTemperatura() {
        datum = new Date(); // današnji datum
    }

    public Date getDatum() {
        return datum;
    }

    public List<Float> getIzmereneVrednosti() {
        if (null == izmereneVrednosti)
            return null;
        return Collections.unmodifiableList(izmereneVrednosti);
    }

    public float getMinVrednost() {
        return minVrednost;
    }

    public float getMaxVrednost() {
        return maxVrednost;
    }

    public void dodajTemperaturu(float temperatura) {
        if (null == izmereneVrednosti) {
            izmereneVrednosti = new ArrayList<Float>();
        }
        if (temperatura > MAX_TEMP || temperatura < MIN_TEMP)
            throw new IllegalArgumentException(String.format("Temperatura mora
                biti u opsegu [%f, %f]", MIN_TEMP, MAX_TEMP));
        if (temperatura < minVrednost)
            minVrednost = temperatura;
        if (temperatura > maxVrednost)
            maxVrednost = temperatura;
        izmereneVrednosti.add(temperatura);

        //Obaveštavanje da je došlo do promena
        notifyObservers();
    }
}

```

Listing 4.28 (prvi deo) Domenska klasa DnevnaTemperatura implementira metode Publisher interfejsa da bi omogućila pogledima da prate i reaguju na izmenu njenih podataka

```

public float srednjaVrednost() {
    if (null == izmereneVrednosti)
        throw new NullPointerException();
    int brojElemenata = izmereneVrednosti.size();
    float suma = 0;
    for (int i = 0; i < brojElemenata; i++) {
        suma += izmereneVrednosti.get(i);
    }
    return suma / brojElemenata;
}

/**
 * List<Observer> observers - lista onih koji posmatraju promenu podataka
 * Observer pattern). Više različitih elemenata korisničkog interfejsa (čak i
 * oni koji nisu bili poznati u trenutku inicijalnog dizajna aplikacije) se
 * mogu registrovati da prate promene.
 */
private List<Observer> observers;

public void addObserver(Observer observer) {
    if (null == observers)
        observers = new ArrayList<Observer>();
    observers.add(observer);
}

public void removeObserver(Observer observer) {
    if (null == observers)
        return;
    observers.remove(observer);
}

/**
 * notifyObservers - Metoda za slanje događaja da se desila promena svima
 * koji su se registrovali za praćenje promena
 */
public void notifyObservers() {
    UpdateEvent e = new UpdateEvent(this);
    for (Observer observer : observers) {
        observer.updatePerformed(e);
    }
}
}

```

Listing 4.28 (drugi deo)<sup>19</sup>

---

<sup>19</sup> Kompletan programski kod se može naći na <https://github.com/milosavljevicg/SIMS-primeri>.

```

/**
 * KontrolerStanice - klasa koja implementira kontroler deo MVC paterna
 */
public class KontrolerStanice {
    private DnevnaTemperatura dnevnaTemperatura;    // model

    public KontrolerStanice(DnevnaTemperatura dnevnaTemperatura) {
        this.dnevnaTemperatura = dnevnaTemperatura;
    }

    /**
     * unosNoveTemperature - prima ulaz od prozora (sadržaj unetog stringa), vrši
     * provere, prevodi ga u broj i poziva metodu iz modela
     */
    public void unosNoveTemperature(String unetaTemperatura) {
        if (unetaTemperatura == null)
            throw new NullPointerException();
        // parseFloat potencijalno baca exception, koji hvata metoda
        // unosTemperature klase prozor
        float temperatura = Float.parseFloat(unetaTemperatura);
        dnevnaTemperatura.dodajTemperaturu(temperatura);
    }
}

```

Listing 4.29 KontrolerStanice je klasa koja implementira kontroler u okviru MVC šablona.

```

/**
 * ProzorMeteoroloskeStanice - prozor koji omogućava unos i pregled dnevnih
 * temperatura i njihovih statistika. Demonstrira upotrebu Model-View-Controller
 * šablona kombinovanog sa Observer šablonom
 */
public class ProzorMeteoroloskeStanice extends JDialog implements Observer {
    private KontrolerStanice kontrolerStanice;
    private DnevnaTemperatura dnevnaTemperatura;

    private JLabel lblUnosTemperature = new JLabel("Unos temperature");
    private JTextField tfUnosTemperature = new JTextField(15);
    private JButton btnUnosTemperature = new JButton("Unos");
    private JLabel lblMinTemperatura = new JLabel("Minimalna temperatura");
    private JLabel lblMaxTemperatura = new JLabel("Maksimalna temperatura");
    private JLabel lblSrednjaTemperatura = new JLabel("Srednja temperatura");
    private JTextField tfMinTemperatura = new JTextField(15);
    private JTextField tfMaxTemperatura = new JTextField(15);
    private JTextField tfSrednjaTemperatura = new JTextField(15);

    public ProzorMeteoroloskeStanice(DnevnaTemperatura dnevnaTemperatura,
        KontrolerStanice kontrolerStanice) {
        this.dnevnaTemperatura = dnevnaTemperatura;
    }
}

```

Listing 4.30 (prvi deo) ProzorMeteoroloskeStanice je klasa korisničkog interfejsa koja implementira pogled u okviru MVC šablona. Njena nadležnost je da konstruiše izgled, prima zahteve od korisnika i prosleđuje ih kontroleru. Osvežava prikaz kada, pozivom metode updatePerformed od strane klase DnevnaTemperatura, dobije informaciju da je došlo do izmene podataka.



```

        this.kontrolerStanice = kontrolerStanice;
        //Prozor se registruje za praćenje događaja o promeni podataka u modelu
        this.dnevnaTemperatura.addObserver(this);
        setTitle("Meteorološka stanica");
        setSize(700, 600);
        setLayout(new BorderLayout());
        dodajKomponente();
    }

    public void dodajKomponente() {
        JPanel panel = new JPanel();
        panel.setLayout(new MigLayout());
        panel.add(lblUnosTemperature);
        panel.add(tfUnosTemperature);
        panel.add(btnUnosTemperature, "wrap");
        tfMinTemperatura.setEditable(false);
        tfMaxTemperatura.setEditable(false);
        tfSrednjaTemperatura.setEditable(false);
        panel.add(lblMinTemperatura);
        panel.add(tfMinTemperatura, "wrap");
        panel.add(lblMaxTemperatura);
        panel.add(tfMaxTemperatura, "wrap");
        panel.add(lblSrednjaTemperatura);
        panel.add(tfSrednjaTemperatura, "wrap");
        add(panel, BorderLayout.NORTH);

        PanelGrafik pnlGrafik = new PanelGrafik(dnevnaTemperatura);
        add(pnlGrafik, BorderLayout.CENTER);

        btnUnosTemperature.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                unosTemperature();
            }
        });

        getRootPane().setDefaultButton(btnUnosTemperature);
    }

    /**
     * unosTemperature - metoda koja se poziva kada korisnik klikne na dugme
     * "Unos"
     */
    public void unosTemperature () {
        try {
            kontrolerStanice.unosNoveTemperature(tfUnosTemperature.getText());
            tfUnosTemperature.setText(null);
        }
        catch (NullPointerException ex){
            JOptionPane.showMessageDialog(this, "Temperatura nije uneta!",
                "Greška", JOptionPane.ERROR_MESSAGE);
        }
        catch (NumberFormatException ex){
            JOptionPane.showMessageDialog(this, "Temperatura mora biti uneta kao
                broj!", "Greška", JOptionPane.ERROR_MESSAGE);
        }
    }

```

Listing 4.30 (drugi deo)

```

        catch(IllegalArgumentException ex){
            JOptionPane.showMessageDialog(this, ex.getMessage(), "Greška",
                JOptionPane.ERROR_MESSAGE);
        }
        finally {
            tfUnosTemperature.grabFocus();
        }
    }

    /**
     * updatePerformed - metoda koja implementira Observer interfejs. Nju poziva
     * Publisher kada dođe do promene njegovih podataka ili stanja
     */
    public void updatePerformed(UpdateEvent e) {
        prikaziStatistiku();
    }

    /**
     * prikaziStatistiku - metoda koja osvežava vrednosti minimalne, maksimalne i
     * prosečne temperature na prozoru
     */
    private void prikaziStatistiku() {
        try {
            DecimalFormat df = new DecimalFormat("0.0");
            tfMaxTemperatura.setText(df.format(dnevnaTemperatura.
                getMaxVrednost()));
            tfMinTemperatura.setText(df.format(dnevnaTemperatura.
                getMinVrednost()));
            tfSrednjaTemperatura.setText(df.format(dnevnaTemperatura.
                srednjaVrednost()));
        }
        catch(NullPointerException ex){
            JOptionPane.showMessageDialog(this, "Nema unetih temperatura!",
                "Greška", JOptionPane.ERROR_MESSAGE);
        }
    }
}

```

Listing 4.30 (treći deo)

```

/**
 * PanelGrafik mogućava crtanje grafika promene temperature u toku dana
 */
public class PanelGrafik extends JPanel implements Observer {
    private DnevnaTemperatura dnevnaTemperatura;
    private float razmakX;
    private float razmakY;

    public PanelGrafik(DnevnaTemperatura dnevnaTemperatura) {
        this.dnevnaTemperatura = dnevnaTemperatura;
        dnevnaTemperatura.addObserver(this);
    }
}

```

Listing 4.31 (prvi deo) Klasa PanelGrafik je klasa korisničkog interfejsa koja je još jedan pogled u okviru MVC šablona. Obezbeđuje iscrtavanje grafika izmerenih temperatura. Osvežava prikaz kada se podaci u okviru klase DnevnaTemperatura promene.

```

public void updatePerformed(UpdateEvent e) {
    // Poziv metode pretka za ponovno iscrtavanje komponente. Ona poziva
    // metodu paintComponent. paintComponent se nikad ne poziva direktno!
    repaint();
}

protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;
    List<Float> temperature = dnevnaTemperatura.getIzmereneVrednosti();
    if (null == temperature)
        return;
    //Pravimo podeok za svakih 10 stepeni:
    float brojPodeoka = (DnevnaTemperatura.MAX_TEMP -
        DnevnaTemperatura.MIN_TEMP) / 10 + 1;
    razmakY = getHeight()/brojPodeoka;
    razmakX = razmakY;
    int brojElemenata = temperature.size();
    if (getWidth() / brojElemenata < razmakX)
        razmakX = getWidth() / brojElemenata;
    iscrtajOse(g2);

    float temperatura = temperature.get(0);
    float x = razmakX;
    float y = racunajY(temperatura);
    float prevX = x;
    float prevY = y;
    g2.setPaint(Color.BLACK);
    Font font = new Font("Serif", Font.BOLD, 12);
    g2.setFont(font);
    //Iscrtavanje prve tačke
    iscrtajTacku(g2, x, y, temperatura);

    //Iscrtavanje preostalih tačaka spojenih linijom
    for (int i = 1; i < brojElemenata; i++) {
        x += razmakX;
        temperatura = temperature.get(i);
        y = racunajY(temperatura);
        iscrtajTacku(g2, x, y, temperatura);
        g2.drawLine(Math.round(prevX), Math.round(prevY), Math.round(x),
            Math.round(y));
        prevX = x;
        prevY = y;
    }
}

private void iscrtajOse(Graphics2D g2) {
    g2.setPaint(Color.BLUE);
    g2.drawLine(0, getHeight()/2, getWidth(), getHeight()/2);
    g2.drawLine(Math.round(razmakX), 0, Math.round(razmakX), getHeight());
}

```

Listing 4.31 (drugi deo)

```

/**
 * racunajY - Računanje y koordinate tačke u odnosu na sredinu panela
 */
private float racunajY(float temperatura) {
    return getHeight()/2 - temperatura / 10 * razmakY;
}

/**
 * iscrtajTacku - Iscrtavanje tačke na grafiku
 */
private void iscrtajTacku(Graphics2D g2, float x, float y,
    float temperatura) {
    DecimalFormat df = new DecimalFormat("0.0");
    g2.drawString(df.format(temperatura), x, y - 10);
    g2.fillRect(Math.round(x-3), Math.round(y-3), 6, 6);
}
}

```

Listing 4.31 (treći deo)

```

/**
 * Događaj koji se aktivira kada dođe do promene na modelu. Po potrebi, može da
 * sadrži atribute i metode koji opisuju promenu.
 */
public class UpdateEvent extends EventObject {
    public UpdateEvent(Object object) {
        super(object);
    }
}

```

Listing 4.32 Klasa UpdateEvent se instancira kada klasa DnevnaTemperatura treba da „javi“ da je došlo do promene podataka, u okviru metode notifyObservers

```

public class Aplikacija {
    public static void main(String[] args) {
        DnevnaTemperatura dnevnaTemperatura = new DnevnaTemperatura();
        KontrolerStanice kontroler = new KontrolerStanice(dnevnaTemperatura);
        ProzorMeteoroloskeStanice prozor = new ProzorMeteoroloskeStanice
            (dnevnaTemperatura, kontroler);
        prozor.setVisible(true);
    }
}

```

Listing 4.33 Klasa Aplikacija instancira delove MVC-a i prikazuje glavni prozor

## 4.6 Šta smo naučili

Dijagram klasa se može koristiti tokom celokupnog životnog ciklusa softverskog proizvoda, za komunikaciju ideja u okviru projektnog tima, specifikaciju implementacije, istraživanje različitih projektantskih odluka, generisanje koda i dokumentovanje rešenja. U zavisnosti od namene modela, dijagram može sadržati više ili manje detalja i biti bliži pojmovima iz problemskog domena ili pojmovima implementacione platforme.

Konceptualni, odnosno domenski modeli se fokusiraju na modelovanje konceptata domena za koji se softver implementira. Na osnovu njega se vrši implementacija domenskih klasa i strukture trajnog skladišta podataka (npr. šeme relacione baze podataka). Implementacioni modeli obično sadrže više detalja od konceptualnih i direktno se mapiraju na programski kod ciljne platforme.

Osobine klase su obeležja i metode. Obeležjima se može specificirati naziv, vidljivost, tip podataka, kardinalitet (broj instanci datog obeležja u okviru klase), kao i niz dodatnih opcija kojima se može precizno specificirati način preslikavanja na programski kod. Metodama se može specificirati ime, parametri, tip povratne vrednosti i dodatne opcije. Klase, obeležja, metode i parametre treba tako imenovati da se dobije jasna informacija o njihovoj nameni.

Kandidate za klase i obeležja u rečenicama specifikacije zahteva prepoznamo kao imenice. Možemo zaključiti da je u pitanju klasa, a ne obeležje, ako pojam koji razmatramo ima svoje osobine koje su bitne za sistem koji se implementira.

Pri određivanju šta klasa treba da sadrži od obeležja i metoda treba se voditi sledećim principima:

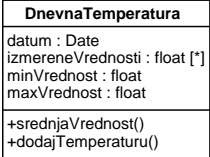

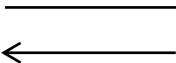
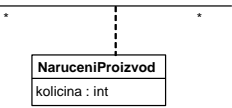

- Principom „jedne odgovornosti“ (*single responsibility principle*) – klasa treba da je zadužena za jednu, jasno definisanu namenu, radi jednostavnije saradnje sa drugim klasama, kao i lakšeg testiranja i održavanja.
- Principom kohezije – optimalno je da svaka metoda klase radi sa većinom obeležja klase (ne računajući get i set metode).
- Principom lokalizovane izmene – ako menjamo implementaciju jedne klase, ta promena ne bi trebalo da utiče na ostatak sistema.

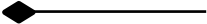


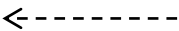
Klase korisničkog interfejsa treba projektovati tako da ostvarimo striktnu podelu nadležnosti između njih i domenskih klasa. Široko je zastupljena preporuka da se saradnja klase korisničkog interfejsa i ostatka sistema organizuje prema MVC (*Model-View-Controller*) arhitektonskom šablonu.

Klase se na dijagramu mogu povezivati sledećim vrstama veza: asocijacijom, generalizacijom, implementacijom interfejsa i vezom zavisnosti.

U tabeli 4.4 je dat pregled elemenata dijagrama klasa koji su detaljno obrađivani u prethodnim sekcijama.

Tabela 4.4 Elementi dijagrama klasa

Simbol	Naziv	Opis
 <pre> classDiagram     class DnevnaTemperatura {         datum : Date         izmereneVrednosti : float [*]         minVrednost : float         maxVrednost : float         +srednjaVrednost()         +dodajTemperaturu()     } </pre>	Klasa	Klase u UML-u se mapiraju na klase u OO programskim jezicima. U konceptualnim modelima se koriste za modelovanje učenih pojmova realnog sistema. Mogu im se specificirati obeležja i metode.
 <pre> classDiagram     class Snimanje {         +snimi()     } </pre>	Interfejs	Interfejs u UML-u se mapira na interfejs u OO programskim jezicima poput Java ili C#. Mogu imati obeležja i metode. S obzirom da klasa može implementirati više interfejsa, dobijamo efekat kao kod višestrukog nasleđivanja.
	Asocijacija	Asocijacija omogućava da se modeluje povezivanje instanci (objekata) klase putem referenci. Kardinalitet na krajevima asocijacije označava sa koliko objekata klase na suprotnom kraju asocijacije jedan objekat razmatrane klase može ili mora biti povezan. Navigabilnost asocijacije se koristi radi specificiranja kako instance klase na krajevima asocijacije treba da se „vide“ preko referenci: uzajamno ili da samo jedna instanca „vidi“ drugu.
 <pre> classDiagram     class NaruceniProizvod {         kolicina : int     } </pre>	Asocijativna klasa	Asocijativna klasa sadrži osobine koje dodatno opisuju odnos dve klase povezane asocijacijom.
	Agregacija	Agregacija je vrsta asocijacije kojom se modeluje odnos „celina-deo“ između dve klase.

	Kompozicija	Kompozicija je vrsta agregacije kod koje je životni ciklus klasa čvrsto povezan. Klasa koja modeluje celinu kreira svoje delove i njihov je jedini vlasnik. Kada se celina briše, brišu se i njeni delovi.
	Generalizacija	Veza generalizacije povezuje opštije elemente modela (predak, roditelj) sa specijalizovanim (naslednik, dete). Može povezivati klase koje su iste vrste i dele zajedničke osobine i ponašanje.
	Implementacija interfejsa	Klasa koja implementira interfejs se obavezuje da će podržati obeležja i metode koje interfejs propisuje na sebi svojstven način.
	Veza zavisnosti	Označava da između dve klase postoji zavisnost, koja se ne može iskazati vezom asocijacije, generalizacije ili implementacije interfejsa. Primer: jedna klasa kreira instance druge klase; instanca jedne klase se prenosi kao argument metode druge klase.

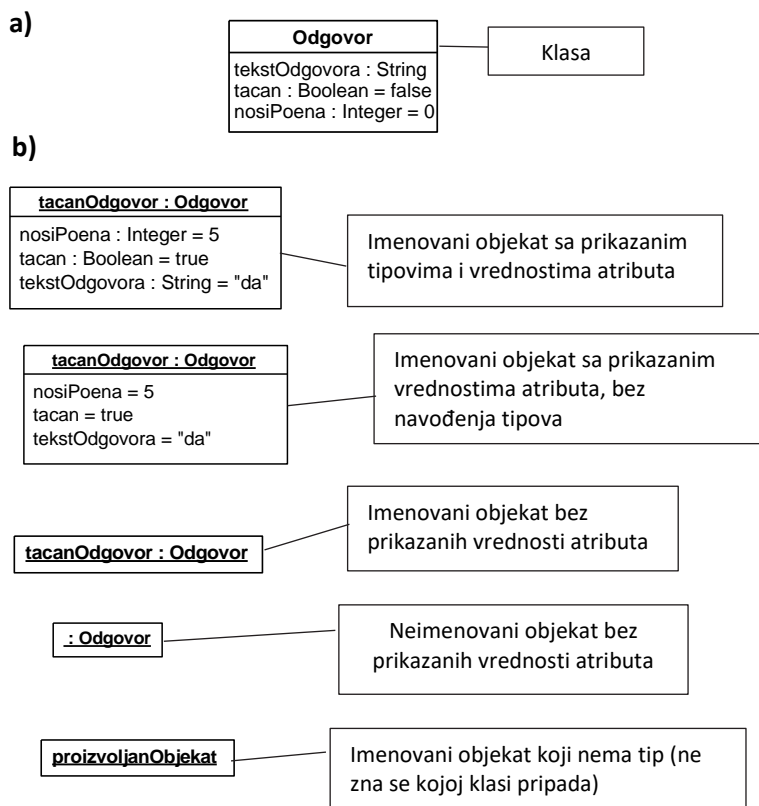




## Poglavlje 5

### Dijagram objekata

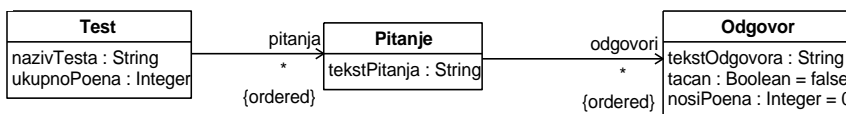
Dijagram objekata prikazuje objekte (instance klasa iz dijagrama klasa) i veze između njih (reference, odnosno pokazivače na druge objekte), preko kojih objekti mogu da komuniciraju. Namena dijagrama objekata je bolje razumevanje i dokumentovanje dijagrama klasa. Crta se za proizvoljno stanje sistema, sa željenim nivoom detalja, tako da se na odgovarajući način prikaže određeni deo dijagrama klasa koji želimo da razjasnimo ili dokumentujemo.



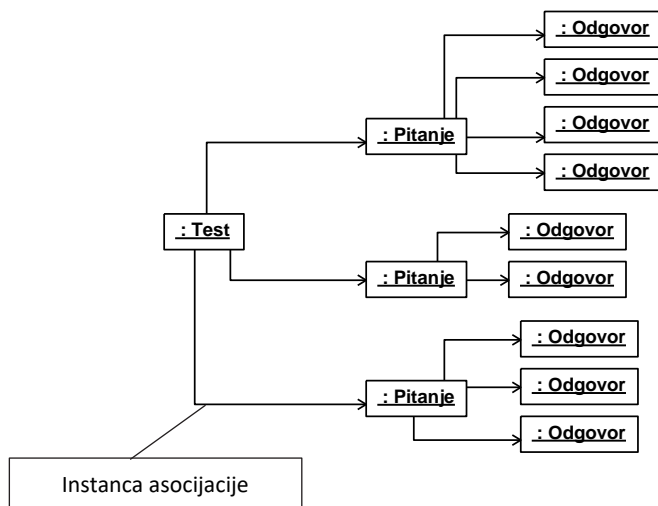
Slika 5.1 a) Klasa b) Objekti (instance) klase modelovani sa različitim nivoom detalja

Na slici 5.1 je prikazana klasa *Odgovor* iz sistema za elektronsko ocenjivanje i njeni objekti sa različitim nivoom detalja. Primećujemo da su ime objekta i ime klase kojoj objekat pripada podvučeni. Ime objekta, ako se navede, treba da počinje malim slovom.

a)



b)

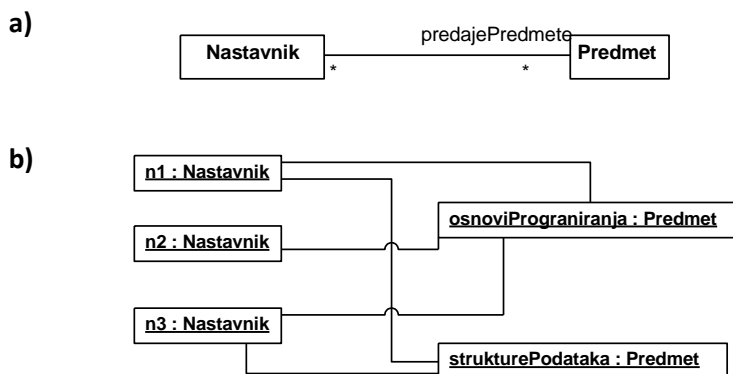


Slika 5.2 Primer modelovanja asocijacija sa kardinalitetom „1 prema više“ a) Deo dijagrama klasa iz sistema za elektronsko ocenjivanje b) Dijagram objekata.

Vrednosti obeležja ne moramo prikazivati, ako želimo da akcenat stavimo na način povezivanja objekata, radi boljeg razumevanja strukture dijagrama klasa (slika 5.2). Možemo primetiti da su asocijacije sa kardinalitetom 1 prema \* na dijagramu objekata modelovane „buketom“ veza koje sve kreću od objekta kod kojeg je kardinalitet na dijagramu klasa 1. Veza može imati sve osobine asocijacije, sem kardinaliteta - jedan objekat može biti povezan samo sa jednim drugim objektom.

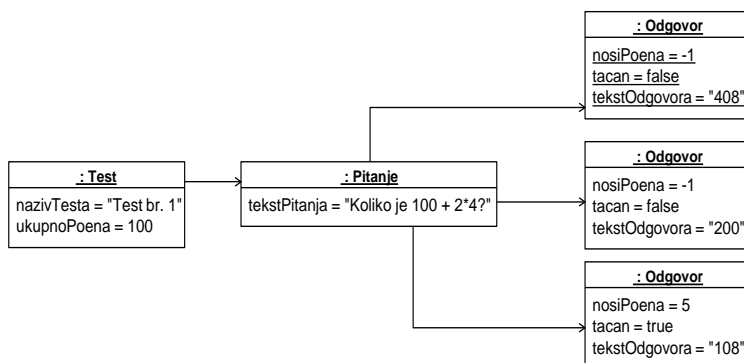
Ako strelica nije navedena, objekti se uzajamno „vide“, kao na slici 5.3, gde je prikazan dijagram objekata za bidirekcionu asocijaciju sa kardinalitetom \* na oba kraja („više na više“). Možemo nacrtati i povezati onoliko objekata koliko

mislimo da je potrebno da se na najbolji način objasni odgovarajući segment dijagrama klasa.



Slika 5.3 Primer modelovanja asocijacije sa kardinalitetom „više na više“ a) Deo dijagrama klasa iz sistema za elektronsko ocenjivanje b) Objektni dijagram koji ilustruje dati segment.

Na slici 5.4 se nalazi dijagram objekata sa prikazanim vrednostima obeležja, radi ilustracije namene i korišćenja obeležja klasa Test, Pitanje i Odgovor, sa slike 5.2a. Sa dijagrama se vidi da je predviđeno da može postojati više tačnih odgovora u okviru jednog pitanja, kao i da odgovori mogu nositi i pozitivne i negativne poene.



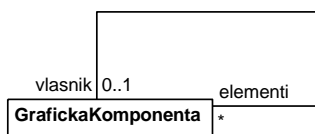
Slika 5.4 Primer dijagrama objekata kojim se ilustruje namena i korišćenje obeležja za deo dijagrama klasa sa slike 5.2a

Na slici 5.5a je prikazan deo dijagrama klasa koji modeluje odnos sadržavanja u okviru neke buduće hijerarhije grafičkih komponenti. Predak hijerarhije, klasa GrafickaKomponenta, se može nalaziti u okviru druge grafičke komponente

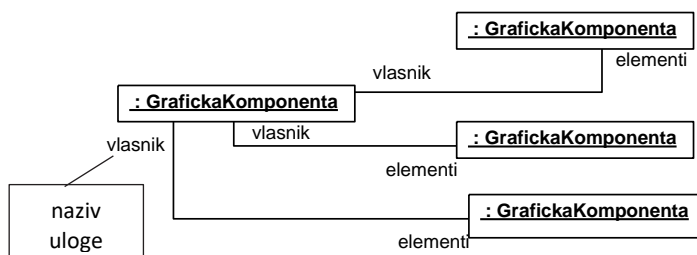
koja je njen vlasnik, odnosno, grafička komponenta može imati elemente koji su druge grafičke komponente. Pošto značenje rekurzivne asocijacije može biti nejasno, odgovarajući dijagram objekata na slici 5.5b ilustruje način uzajamnog povezivanja objekata klase *GrafickaKomponenta*. Možemo primetiti da veze na dijagramu objekata na krajevima imaju prikazane nazive uloga, što je od pomoći radi boljeg razumevanja rekurzivnih asocijacija.

Na slici 5.6a je prikazan primer kada postoji više asocijacija između dve klase. Klasa *Narudzba*, koja modeluje narudžbu robe od dobavljača u nekom poslovnom sistemu, ima tri asocijacije sa klasom *Radnik*. Jedna asocijacija modeluje radnika koji je kreirao narudžbu, druga radnika koji je odobrio narudžbu i treća radnika koji je potpisnik narudžbe. Radnik koji potpisuje i odobrava narudžbu može biti ista osoba. Navedena situacija je ilustrovana dijagramom objekata na slici 5.6b. Primećujemo da su opet navedeni nazivi uloga na krajevima veza kod objekta klase *Radnik*, da bi se bolje ilustrovale uloge koju radnik može imati u okviru narudžbe.

a)

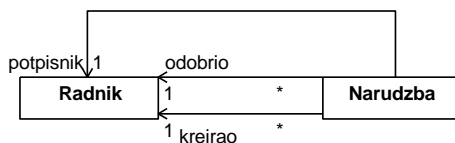


b)

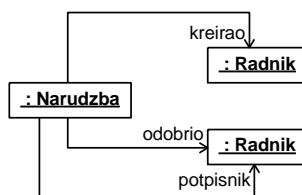


Slika 5.5 Primer dijagrama objekata za rekurzivnu asocijaciju a) Segment dijagrama klasa b) Objektni dijagram koji ilustruje dati segment.

a)

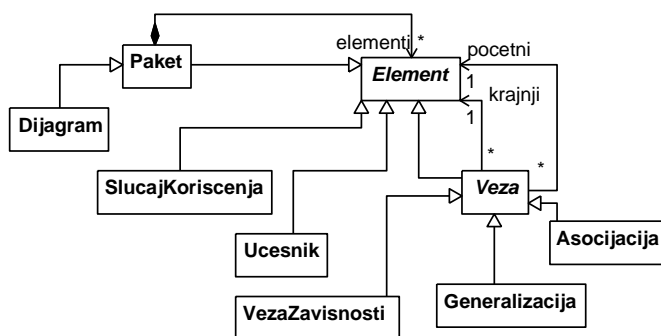


b)



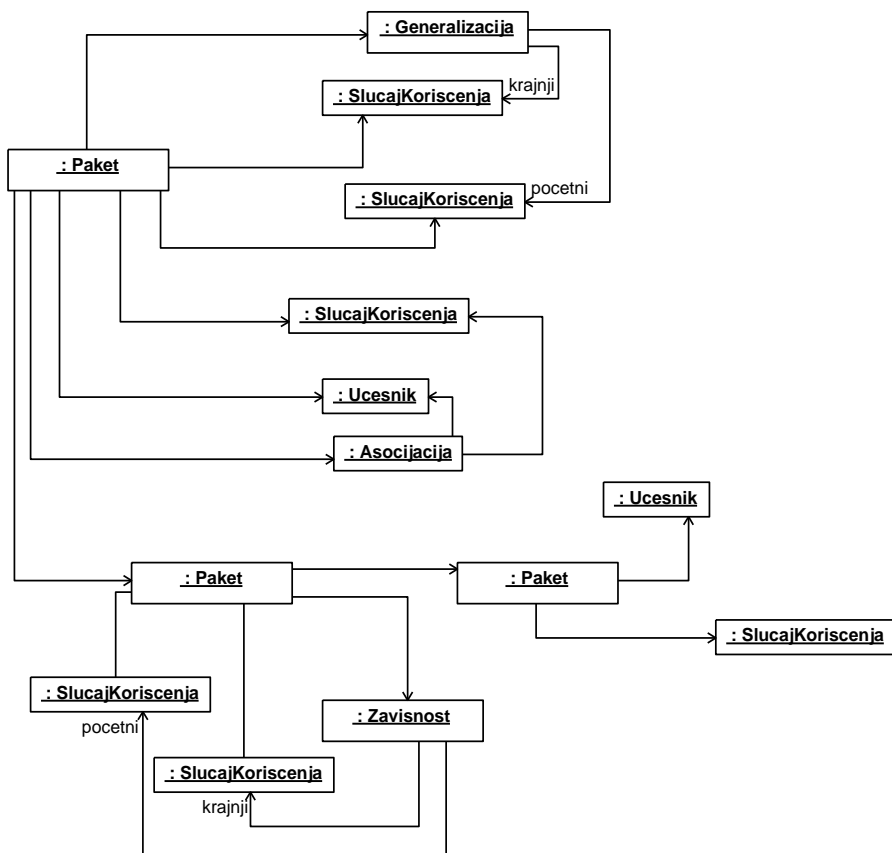
Slika 5.6 Primer dijagrama objekata za situaciju kada postoji više asocijacija između dve klase a) Segment dijagrama klase b) Objektni dijagram koji ilustruje dati segment

Na slici 5.7 je prikazana skica dijagrama klase kojom je modelovana struktura podataka za dijagram slučajeva korišćenja iz primera 4.7.



Slika 5.7 Dijagram klase koji modeluje strukturu podataka za dijagram slučajeva korišćenja (detalji u sekciji 4.4.2.3)

Dati dijagram poseduje dve konstrukcije koje bi čitaocu modela mogle biti nejasne: 1) odnos klase Paket i Element koji je baziran na kompozitnom šablonu i 2) odnos klase Veza sa klasom Element, kao i sa njenim naslednicima: Asocijacija, Generalizacija i VezaZavisnosti. Dijagram objekata na slici 5.8 pokazuje kako je zamišljeno da se formiraju paketi i na koji način se povlače veze između određenih elemenata paketa, kao i značenje obeležja pocetni i krajnji.



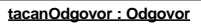
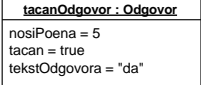
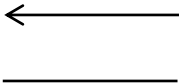
Slika 5.8 Primer dijagrama objekata za hijerarhiju klasa koja modeluje strukturu podataka za dijagram slučajeve korišćenja sa slike 5.7

## 5.1 Šta smo naučili

Dijagram objekata se crta da bismo bolje razumeli ili dokumentovali određene segmente dijagrama klasa. Možemo modelovati proizvoljan broj objekata, sa željenim nivoom detalja, da bismo na odgovarajući način ilustrovali određeni deo dijagrama klasa: način povezivanja objekata i/ili namenu i način dodele vrednosti obeležjima.

U tabeli 5.1 je dat kratak prikaz elemenata koji se koriste u dijagramu objekata.

Tabela 5.1 Elementi dijagrama objekata

Simbol	Naziv	Opis
	Objekat bez vrednosti obeležja	Objekat je instanca klase. Ime objekta i tip klase čija je instanca moraju biti podvučeni. Ime ili tip se mogu izostaviti. Primeri za vrednosti obeležja u okviru objekta se ne moraju navoditi.
	Objekat sa vrednostima obeležja	Ako je potrebno ilustrovati način korišćenja obeležja, ona se mogu navesti, sa ili bez tipova obeležja.
	Veza	Veza predstavlja instancu asocijacije iz dijagrama klasa. Može imati sve osobine asocijacije, osim kardinaliteta - jedan objekat može biti povezan samo sa jednim drugim objektom. Značenje veze je referenca, odnosno pokazivač jednog objekta na drugi u operativnoj memoriji.

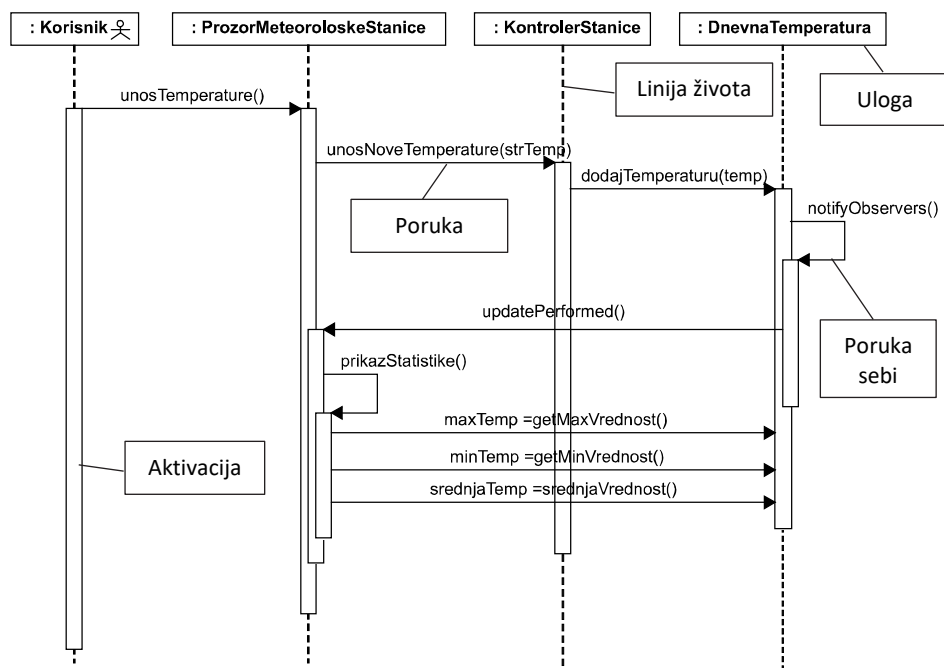




## Poglavlje 6

### Dijagram sekvenci

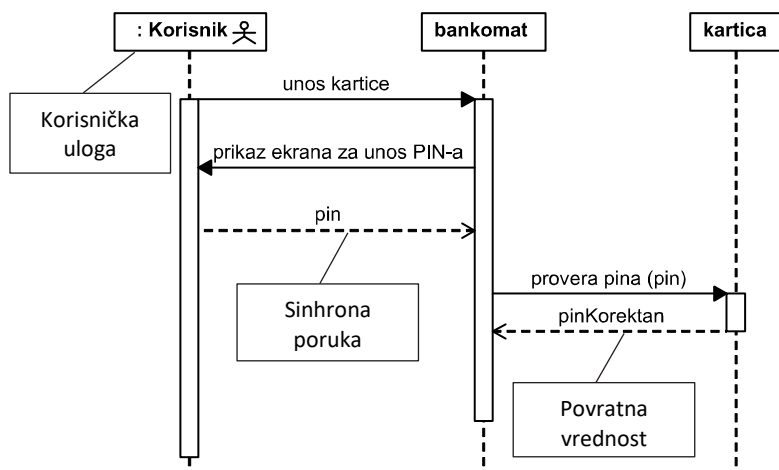
Dijagram sekvence prikazuje interakciju, odnosno komunikaciju između uloga sistema u vremenu. Uloga je zajednički naziv za učesnike sistema i elemente sistema. Učesnici su korisničke uloge koje smo izolovali tokom analize zahteva u okviru dijagrama slučajeva korišćenja (*actor*). Element sistema može biti pojedinačni objekat koji je instanca neke klase iz dijagrama klasa (slika 6.1) ili proizvoljni deo sistema koji posmatramo kao crnu kutiju (*black box*) u interakciji sa učesnicima i drugim delovima sistema (slika 6.2). Pod interakcijom se podrazumeva razmena poruka između uloga. Ako su uloge objekti (instance klase), poruke su pozivi metoda datih klase. Ako su uloge delovi sistema, porukama se



Slika 6.1 Dijagram sekvence koji dokumentuje interakciju klase ProzorMeteoroloskeStanice, KontrolerStanice i DnevnaTemperatura, kada korisnik unese novu temperaturu i pritisne dugme za unos. Klase implementiraju MVC i *Observer* šablon (videti sekciju 4.5.2)

mogu modelovati signali koji se razmenjuju između različitih uređaja, komunikacija putem servisa i sl.

Pored toga, porukama se može modelovati komunikacija između učesnika i sistema, tokom ranih faza razvoja, kada je akcenat na specifikaciji očekivanog odziva sistema na akcije korisnika (slika 6.2). U tom slučaju, obično jedna poruka odgovara jednom koraku iz dijagrama slučajeva korišćenja.



Slika 6.2 Dijagram sekvence koji ilustruje interakciju između korisnika i bankomata koja počinje unosom kartice

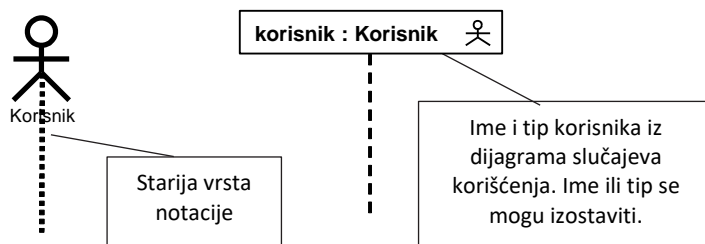
Dijagram sekvence se obično specifikira za jedan slučaj korišćenja, mada se nekad dijagram sekvence može crtati i za više povezanih slučajeva korišćenja, kao i za segment slučaja korišćenja (osnovni ili neki od alternativnih tokova).

Akcent kod dijagrama sekvenci je na otkrivanju i dokumentovanju poruka koje se razmenjuju u vremenu, a ne na algoritamskim aspektima. Mada dijagram sekvence može da prikaže i grananja i petlje, složeni algoritamski aspekti se bolje modeluju dijagramima aktivnosti.

## 6.1 Uloge

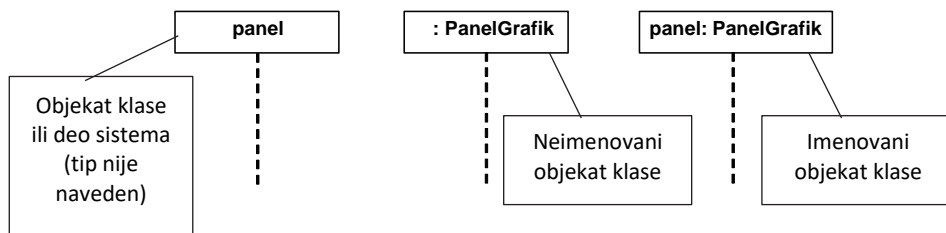
Uloge u dijagramu sekvence mogu biti korisnici (učesnici izolovani u okviru dijagrama slučajeva korišćenja), delovi sistema, kao i sistem u celini. Korisničke obično nekom svojom akcijom iniciraju početak interakcije. Vertikalna isprekidana linija koja je pridružena ulozi se zove linija života i označava vreme u kojem je data uloga aktivna (može da prima i šalje poruke). Vreme teče nadole, od uloge prema donjoj ivici dijagrama.

Na slici 6.3 su prikazani načini kako različiti alati za modelovanje prikazuju korisničke uloge u okviru dijagrama sekvence.



Slika 6.3 Načini crtanja korisničkih uloga u različitim alatima za modelovanje

Na slici 6.4 su prikazane uloge koje su delovi sistema ili instance klasa iz dijagrama klasa. Možemo primetiti da mogu imati ime i tip, kao objekti iz dijagrama objekata, ali da ime i tip nisu podvučeni. Razlog je što u UML-u 2.0 možemo modelovati i interakciju između delova sistema, ne samo između instanci klasa. U UML-u 1.0 dijagram sekvence je bio ograničen samo na objekte, pa su crtani na isti način kao u dijagramu objekata, samo sa dodatom linijom života. Neki alati su i dalje zadržali UML 1.0 notaciju, sa podvlačenjem imena i tipa.



Slika 6.4 Primer modelovanja uloga sistema

Ako se navede tip, to može biti klasa, apstraktna klasa ili interfejs. Ako je kao tip naveden interfejs, podrazumeva se da je uloga objekat neke klase koja implementira dati interfejs. Slično, ako je u pitanju apstraktna klasa, uloga je objekat nekog naslednika te klase. Ako nam je potrebno, na dijagramu sekvence možemo imati više objekata istog tipa.

Uloge bez navedenog tipa se obično koriste u ranim fazama razvoja, kada dijagram klasa još ne postoji. U tom slučaju, ime se tako daje da je čitaocu modela jasno šta je u pitanju (npr. bankomat, banka, kartica i sl), kao na slici 6.2 ili 6.5. Ako postoji dijagram klasa za koji se modeluje interakcija, preporuka je da se navede tip klase koju objekat instancira. Ime se ne mora navoditi, osim ako treba da se koristi kao argument neke poruke.

Aktivacija (pravougaonik na liniji života) označava vreme tokom kojeg primalac odgovara na primljenu poruku, odnosno pošiljalac čeka na odgovor. Prikaz aktivacija nije obavezan, ali može pomoći da se na dijagramu lakše prate ugnježdene (*nested*) poruke. Ugnježdene poruke opisuju situaciju kada jedna uloga pošalje poruku drugoj, druga uloga, radi odgovora na tu poruku, pošalje poruku sebi ili nekoj trećoj ulozi, itd. Na slici 6.1 možemo primetiti da se sve poruke izvršavaju kao ugnježdene u okviru poruke unosTemperature, koju je poslao korisnik. Poruka notifyObservers je ugnježdjena za poruku dodajTemperaturu (odnosno, aktivira se od strane poruke dodajTemperaturu) koju je poslao kontroler stanice.

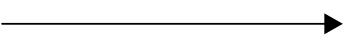



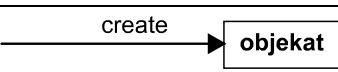
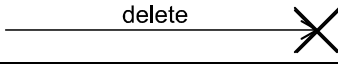
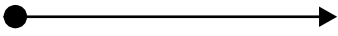
## 6.2 Poruke

Porukom se može modelovati:

- komunikacija između korisnika i sistema u ranim fazama razvoja (sistem se tada tretira kao „crna kutija“)
- slanje i prijem signala između različitih delova sistema,
- sinhroni ili asinhroni poziv metode objekta neke klase,
- vraćanje rezultata (povratna vrednost),
- kreiranje i brisanje objekata.

Vrste poruka koje se mogu koristiti u dijagramu sekvence su navedene u tabeli 6.1. U nastavku ćemo ih detaljnije obraditi i dati primere njihovog korišćenja.

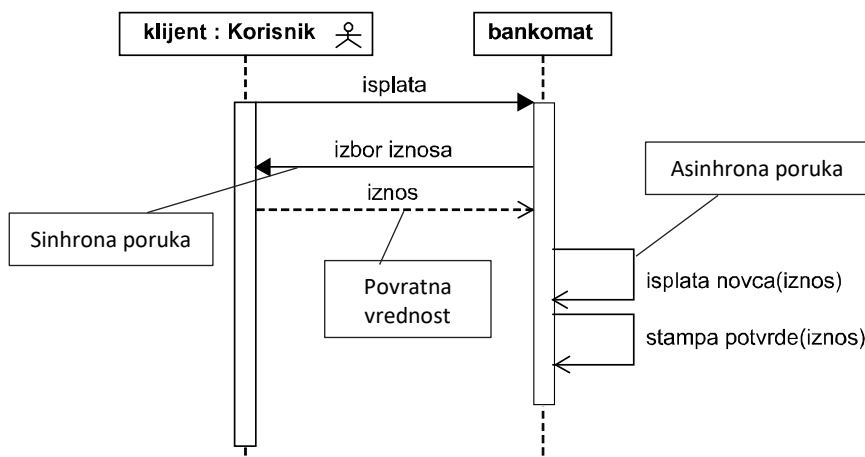
Tabela 6.1 Vrste poruka u dijagramu sekvence

Vrsta poruke	Notacija
Sinhrona poruka	
Asinhrona poruka	
Asinhrona poruka u UML 1.0 notaciji	
Poruka za vraćanje povratne vrednosti (povratna poruka)	
Poruka za kreiranje objekta	
Poruka za oslobađanje objekta	
„Nađena“ poruka	

### 6.2.1 Sinhrona i asinhrona poruke

Kod sinhronih poruka pošiljalac čeka da primalac obradi poruku da bi nastavio sa izvršavanjem. Sinhrona poruka se označavaju strelicom u obliku popunjenog trougla.

Kod asinhronih poruka pošiljalac nastavlja sa izvršavanjem odmah posle slanja poruke, dok primalac nezavisno od njega obrađuje poruku koja mu je prosleđena. Asinhrona poruka se označavaju otvorenim strelicama. U UML-u 1.0 su se označavale polovinom strelice (videti tabelu 6.1).



Slika 6.5 Dijagram sekvence koji ilustruje interakciju između korisnika i bankomata koja počinje zahtevom za isplatu

Na slici 6.5 možemo primetiti da su poruke *isplata* i *izbor iznosa* sinhrona. Kada bankomat ponudi korisniku da bira iznos za isplatu, ostaje u stanju čekanja dok korisnik ne obavi izbor. Izabrani iznos se modeluje kao povratna poruka *iznos*.

Posle izbora iznosa, bankomat šalje sebi dve asinhrona poruke, za isplatu novca i štampu potvrde o isplaćenom novcu. U okviru bankomata, brojač novca i štampač potvrda odmah kreću da izvršavaju poruku koju su primili, bez čekanja jedan na drugoga.

Format za specifikaciju poruke je sledeći:

```
promenljiva = naziv_poruke (argumenti): tip_rezultata
```

Primer:

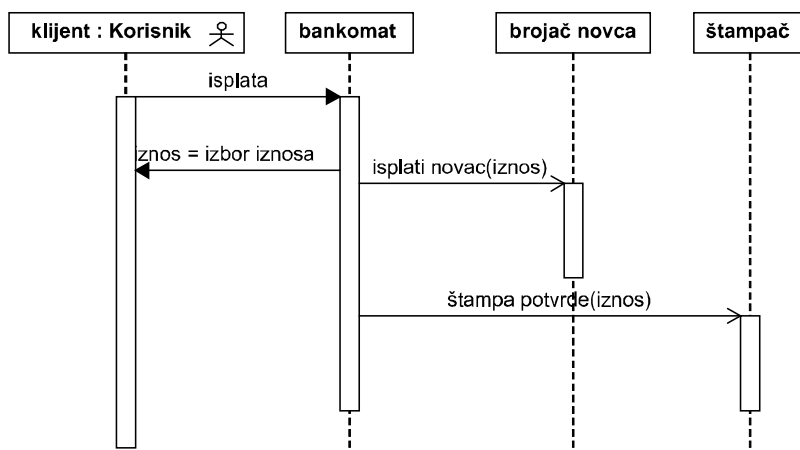
```
vrednost = suma(x, y): int  
vrednost = suma(int sabirak1 = x, int sabirak2 = y): int
```

Naziv poruke odgovara nazivu metode klase, ako su uloge objekti klasa. Promenljiva se koristi za smeštanje vrednosti rezultata, ako poruka vraća rezultat i ako nam je rezultat potreban. Argumenti poruke mogu biti promenljive kojima je prethodno dodeljena vrednost ili nazivi uloga koje učestvuju u interakciji.

Naziv poruke se mora navesti, dok su promenljiva u koju se smešta rezultat, argumenti i tip rezultata opcioni. Umesto promenljive, možemo koristiti i poruku za vraćanje povratne vrednosti. Ako uloge nisu objekti klasa, nazivi poruka koje razmenjuju mogu biti slobodan tekst.

Na slici 6.5 vidimo da poruka izbor iznosa vraća vrednost iznos koji je modelovan kao povratna poruka. Umesto povratne poruke, mogli smo iskoristiti i promenljivu za smeštanje povratne vrednosti: iznos = izbor iznosa.

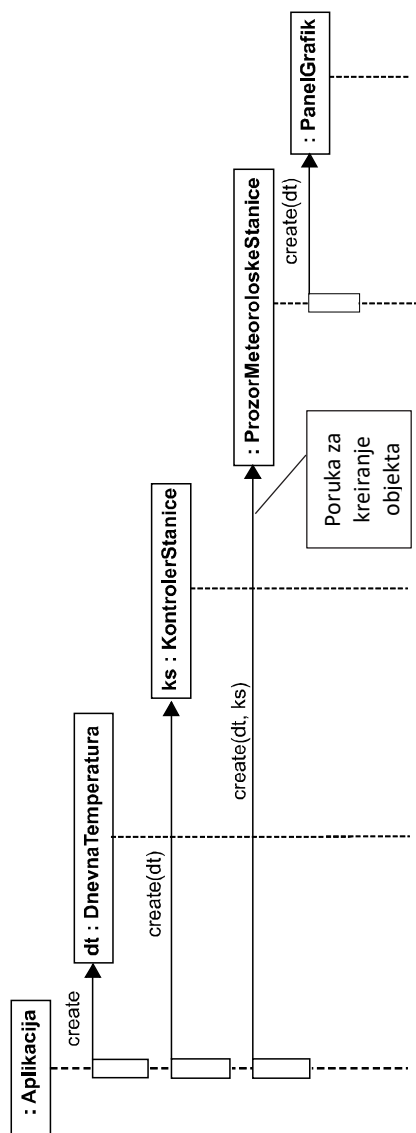
Previše povratnih poruka komplikuju dijagram i čine ga nečitkim. Kompaktniji način zapisivanja je korišćenjem promenljivih. Ako poruka nema povratnu vrednost, ne treba crtati praznu povratnu poruku, kao indikator da se poruka izvršila.



Slika 6.6 Dijagram sekvence za isplatu. Na dijagram su uvedene i uloge brojač novca i štampač, radi prikaza interne komunikacije između delova bankomata.

**Napomena:** Dijagram na slici 6.5 modeluje interakciju između korisnika i sistema, koji u grubim crtama pokazuje kakve poruke se razmenjuju tokom slučaja korišćenja za isplatu novca. Ukoliko bismo želeli da prikažemo internu komunikaciju između delova bankomata, mogli bismo da uvedemo brojač novca i štampač kao uloge, kojima bi procesor bankomata slao signale (slika 6.6).

## 6.2.2 Poruka za kreiranje objekata



Slika 6.7 Dijagram sekvence koji ilustruje inicijalizaciju aplikacije za unos izmerenih dnevnih temperatura na jednom mernom mestu. Objekti potrebni za rad aplikacije se inicijalizuju kao posledica prijema specijalne create poruke.

Na slici 6.7 je ilustrovana inicijalizacija aplikacije za unos izmerenih dnevnih temperatura na jednom mernom mestu. Instanca klase Aplikacija kreira instancu klase DnevnaTemperatura, zatim instancu klase KontrolerStanice i na kraju instancu klase ProzorMeteoroloskeStanice. ProzorMeteoroloskeStanice je zadužen za instanciranje klase PanelGrafik. Bitno je primetiti da se navedeni objekti prikazuju u nastavku create poruke koja ih je instancirala.

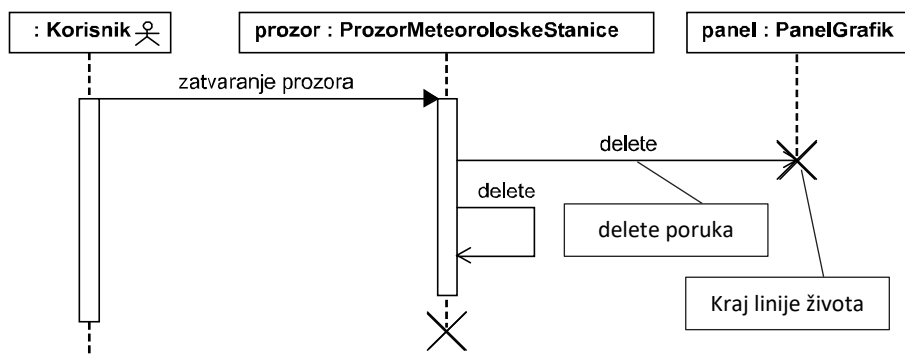
Instance klase `dt:DnevnaTemperatura` i `ks:KontrolerStanice` se prenose kao argumenti create poruka, zbog čega je bilo potrebno da im damo ime (`dt` i `ks`). Argumenti create poruke, ako postoje, mapiraju se na argumente konstruktora klasa u programskom kodu (videti listing 4.33).

Dijagrami na slici 6.1 ili 6.8 prikazuju komunikaciju već instanciranih objekata. Na slici 6.9 možemo primetiti da se, kao rezultat poruke `notifyObservers` kreira instanca klase `UpdateEvent` koji se dalje prosleđuje, radi obaveštavanja „posmatrača“ da je došlo do promene stanja klase koja se prati.

**Napomena:** Neki alati create poruku prikazuju kao `<<create>>` ili `new`. Nezavisno od toga, kreirani objekat se prikazuje u nastavku poruke, umesto uz gornju ivicu dijagrama.

### 6.2.3 Poruka za oslobađanje objekata

Pored poruke za instanciranje objekata, postoji i `delete` poruka za njihovo oslobađanje (dealokaciju). Posle prijema `delete` poruke, na liniji života se pojavljuje krstić, kao oznaka da objekat u toj tački prestaje da postoji, odnosno da više ne može da učestvuje u komunikaciji (slika 6.8).



Slika 6.8 Posle zatvaranja prozora, on prestaje da postoji, kao i sve njegove komponente. Oslobađanje objekta se modeluje `delete` porukom, posle čijeg prijema se završava linija života.



Na slici 6.8 vidimo da, kada korisnik aktivira zatvaranje prozora, oslobađa se instanca klase `PanelGrafik`, koju prozor koristi kao svoj sastavni element. Posle toga, instanca prozora takođe prestaje da postoji.

U jezicima kao što je C++, `delete` poruka se mapira na poziv destruktora koji dealocira memoriju koju objekat zauzima. U jezicima poput Jave, koji se oslanjaju na *garbage collector*, objekat neće biti uništen u trenutku prijema `delete` poruke, ali će svakako postati nedostupan za ostatak sistema.

**Napomena:** Neki alati `delete` poruku prikazuju kao `<<delete>>` ili `destroy`.

#### 6.2.4 Granica interakcije i spoljne poruke

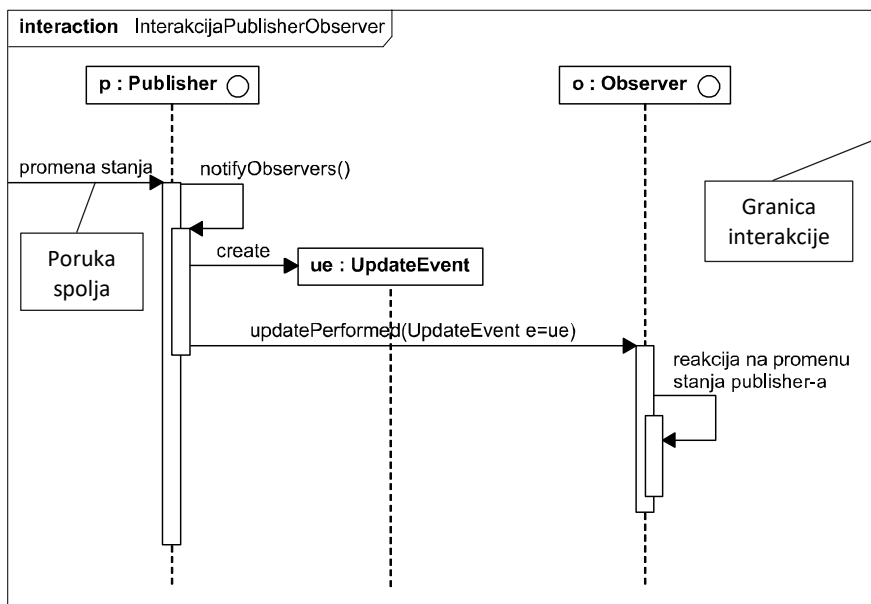
Kada želimo da se skoncentrišemo na modelovanje interakcije izabranih uloga u određenoj situaciji, a pri tome ne želimo da crtamo ceo „lanac“ poruka koji dovodi do te situacije, možemo da koristimo granicu interakcije, poruke spolja, kao i „izgubljene“ i „nađene“ poruke.

Granica interakcije nam omogućava da „ogradimo“ segment interakcije koji želimo da modelujemo. U tom slučaju, pošiljaoci i primaoci poruka mogu biti izvan granice. Poruke koje oni šalju ili primaju su tzv. spoljne poruke.

Na slici 6.9 je ilustrovan način funkcionisanja *Observer* projektnog šablona, odnosno, uzajamna interakcija klase koje implementiraju `Publisher` i `Observer` interfejs. Segment komunikacije koji želimo da prikazemo počinje u trenutku kada dođe do promene podataka klase čija se promena prati (poruka promena stanja koja dolazi izvan granice interakcije). Data poruka verovatno dolazi od strane korisnika i prosleđena je putem klase korisničkog interfejsa i njenog kontrolera, ali taj deo komunikacije nije od značaja za funkcionisanje *Observer* projektnog šablona i zato ga izostavljamo sa dijagrama.

Slično, ako odgovor na neku poruku nije bitan za situaciju koju ilustrujemo, možemo je usmeriti ka granici interakcije. To znači da će tu poruku obraditi neka uloga koja je izvan segmenta interakcije koji se trenutno modeluje.

Granica sistema se može koristiti i radi specificiranja pomoćnog dijagrama, koji se kasnije može referencirati iz drugih dijagrama sekvence, slično kao poziv potprograma, odnosno poziv pod-aktivnosti u dijagramu aktivnosti (videti referencirajući fragment u sekciji 6.3.5).



Slika 6.9 Dijagram sekvence koji prikazuje način funkcionisanja *Observer* projektnog šablona. Poruka promena stanja dolazi izvan granice interakcije (spoljna poruka). Kao odgovor na promenu stanja, kreira se događaj koji opisuje tu promenu i prosleđuje metodi `updatePerformed` klase koja implementira *Observer* interfejs. Dalje, `updatePerformed` poziva metodu svoje klase u kojoj je implementirana reakcija na promenu stanja.

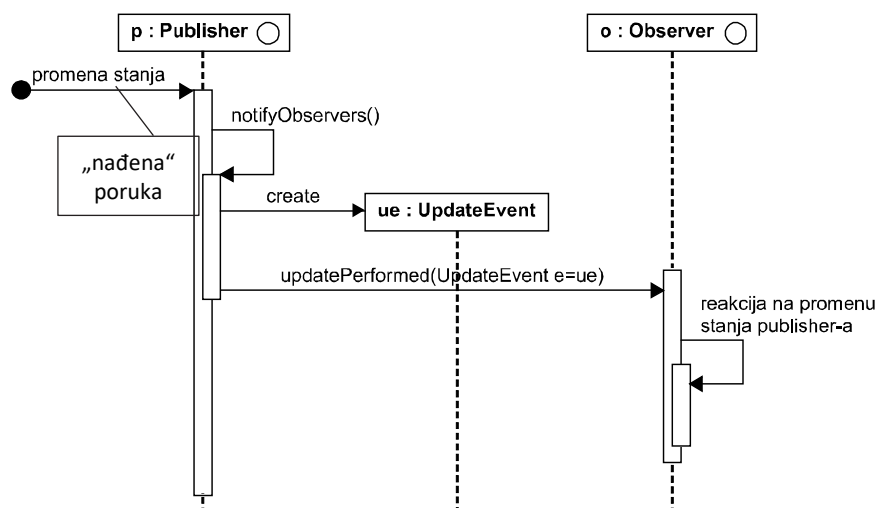
**Napomena:** Možemo birati da li ćemo koristiti poruke spolja koja dolaze van granica interakcije, kao na slici 6.9, ako želimo da se skoncentrišemo samo na projektovanje neke funkcionalnosti u sloju domenskih klasa, ili ćemo na dijagram uključiti i korisničku ulogu koja inicira interakciju, zajedno sa klasama korisničkog interfejsa.

Ako napravimo analogiju sa korišćenjem alata za otkrivanje grešaka u programskom kodu (*debugger*), dijagram sekvenci možemo crtati od određene tačke koja nas zanima (kao kada postavimo prekidnu tačku u *debugger*-u), sa ulascima u potprograme ili prelaskom preko njih. Način crtanja i nivo prikazanih detalja nije bitan – bitno je da li je dijagram od koristi za određenu namenu: pomoć pri mapiranju korisničkih zahteva na dijagram klasa, dokumentovanje komplikovanih interakcija i sl.

### 6.2.5 „Izgubljene“ i „nađene“ poruke

Poruka može imati poznatog pošiljaoca (odnosno, može biti vezane za ulogu koja šalje poruku), ali, može se i samo „pojavit“ na dijagramu, u tački od koje želimo da modelujemo interakciju (tzv. „nađena“ – *found* poruka). Slično, poruka se može završiti na liniji života uloge koja je primalac, ali se može uputiti i nepoznatom primaocu, ako ne želimo da modelujemo odziv na tu poruku (tzv. „izgubljena“ – *lost* poruka).

Na slici 6.10 je takođe ilustrovan način funkcionisanja *Observer* projektnog šablona, ali, za razliku od slike 6.10, interakciju otpočinje „nađena“ poruka.



Slika 6.10 Dijagram sekvence koji prikazuje način funkcionisanja *Observer* projektnog šablona. Poruka promena stanja je, za razliku od dijagrama na slici 6.9, modelovana kao „nađena“ poruka.

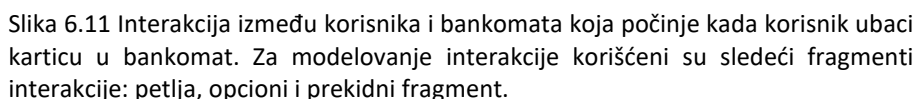
## 6.3 Fragmenti interakcije

Fragmenti interakcije u okviru dijagrama sekvence imaju ulogu iskaza za kontrolu toka u programskim jezicima. U nastavku će biti prikazani najčešće korišćeni fragmenti:

- loop – petlja,
- opt – opcioni fragment,
- break – fragment za prekid,
- alt – alternativni fragment,
- ref – referencirajući fragment,
- par – fragment za paralelno izvršavanje.

- **assert** – fragment kojim se može modelovati transakcioni režim rada (ako se bilo koja poruka uspešno ne izvrši, dejstvo svih poruka u okviru assert fragmenta se poništava),
- **region** – fragment kojim se može specificirati kritični region (*critical region*).

**Petlja – loop** je fragment interakcije kojim se može specificirati ponavljanje izvršavanja određenog niza poruka.



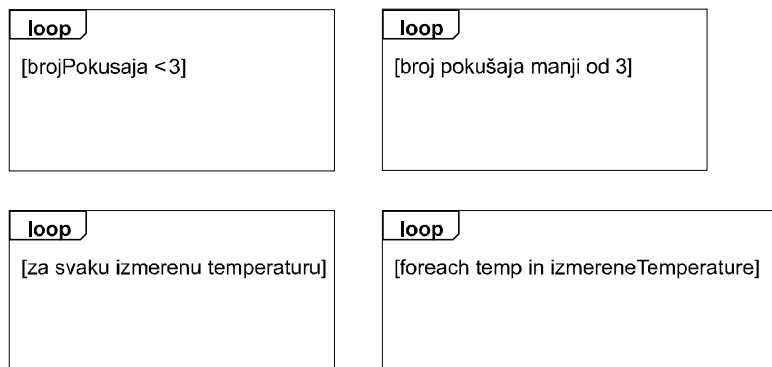
176

- navedeni broj puta, poput for petlje,
- dok je neki uslov zadovoljen, po ugledu na while petlju, ili
- za svaki element neke kolekcije, poput foreach petlje na nekom programskom jeziku.

Na slici 6.11 je modelovana petlja koja se izvršava tačno tri puta, da bismo omogućili klijentu banke da tri puta pokuša da unese ispravan PIN za karticu koju je ubacio u bankomat. Prekidni (break) fragment omogućava da se „iskoči“ iz petlje kada je uneti PIN ispravan.

Ako se specificira uslov, on treba da se nalazi u okviru uglastih zagrada i može se navesti kao slobodan tekst ili na nekom programskom jeziku ili pseudo-jeziku. Primeri zadavanja uslova su dati na slici 6.12.

Ukoliko je potrebno da se samo jedna poruka izvršava u petlji, to se može naznačiti znakom „\*“ ispred imena poruke – ne mora se koristiti loop fragment.



Slika 6.12 Primeri zadavanja uslova koji omogućavaju da specificiramo funkcionalnost while i foreach petlje. Primer for petlje sa fiksnim brojem ponavljanja je dat na slici 6.11.

### 6.3.2 Opcioni fragment

Opcioni fragment – opt se koristi za modelovanje uslovnog izvršavanja poruka koje se u okviru njega nalaze. Ako specificirani uslov nije zadovoljen, preskače se aktiviranje svih poruka u okviru datog fragmenta. Odgovara if iskazu na nekom programskom jeziku, kod kojeg nije specificirana else grana:

```
if (uslov) {
    //...
}
```

Slično kao kod petlje, uslov se može specificirati kao slobodan tekst ili tako da odgovara pravilima nekog jezika ili pseudo-jezika.

Na slici 6.11 je dat primer korišćenja opcionog fragmenta kojim se modeluje zadržavanje kartice od strane bankomata, pod uslovom da je klijent tri puta uneo pogrešan PIN.

Ukoliko se samo jedna poruka izvršava u zavisnosti od nekog uslova, ne mora se koristiti opcion fragment, već se uslov može direktno navesti na kraju poruke, u uglastim zagradama, na primer:

```
isplatiNovac(iznos) [iznos <= iznosNaRacunu]
```

### 6.3.3 Prekidni fragment

Prekidni fragment – break omogućava da se modeluje izlazak iz petlje ili fragmenta koji ga okružuje. Break fragment koji se ne nalazi u nekom drugom fragmentu se može iskoristiti za modelovanje funkcionalnosti regiona mogućeg prekida iz dijagrama aktivnosti.

Prekid se može specificirati:

- uslovom, kao na slici 6.11 - iz petlje se izlazi kada klijent unese tačan PIN,
- porukom, kao na slici 6.21 u sekciji 6.5 - do prekida dolazi kada student aktivira završetak testa.

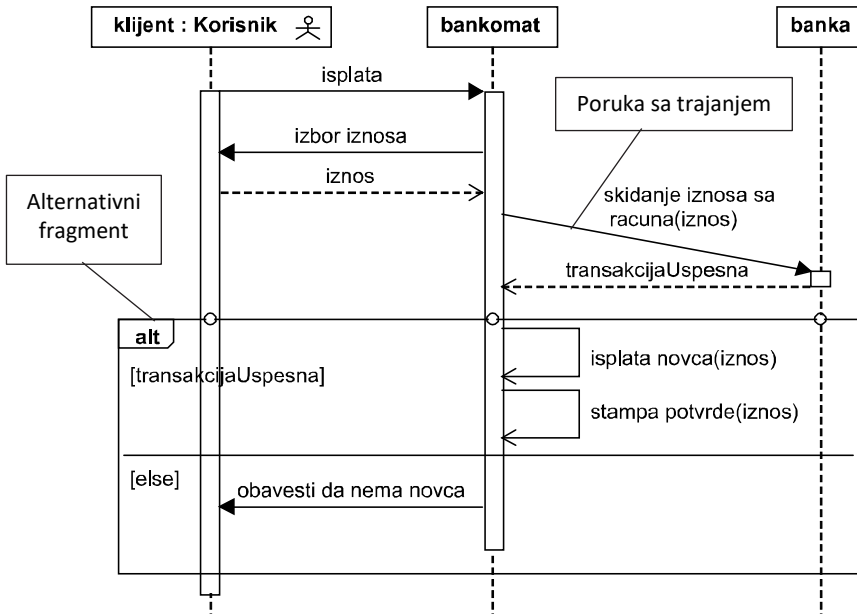
### 6.3.4 Alternativni fragment

Alternativni fragment – alt se koristi za modelovanje uslovnog izvršavanja po ugledu na if iskaz kod kojeg postoji proizvoljan broj else if grana:

```
if (uslov1) {  
    //...  
} else if (uslov2) {  
    //...  
}  
...  
} else {  
    //...  
}
```

Na slici 6.13 je dat primer korišćenja alternativnog fragmenta. Ukoliko je transakcija skidanja traženog iznosa sa računa klijenta uspešna, bankomat vrši isplatu novca i štampa potvrdu o isplaćenom iznosu. U suprotnom, klijent se obaveštava da nema dovoljno novca na računu.

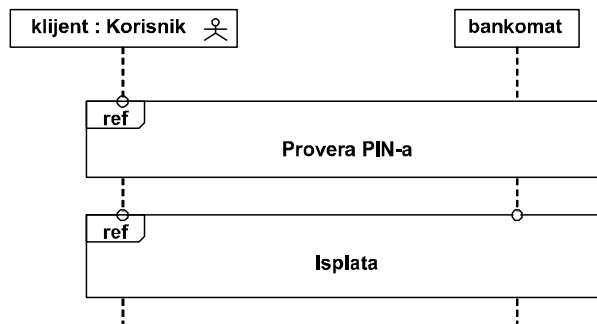
Na istoj slici možemo primetiti i tzv. poruku sa trajanjem. Ona se crta kada želimo da naglasimo da odgovor na neku poruku neće biti trenutni, već da ćemo morati da čekamo neko vreme na odgovor.



Slika 6.13 Dijagram sekvence sa slike 6.6 je proširen komunikacijom sa bankom, radi skidanja novca sa računa pre isplate.

### 6.3.5 Referencirajući fragment

Referencirajući fragment – ref nam omogućava da referenciramo (uključimo) ranije definisani dijagram sekvence, kao kada pozivamo pod-aktivnost u dijagramu aktivnosti. Na ovaj način možemo da veoma složene dijagrame sekvence učinimo čitljivijim i kompaktnijim. Pored toga, ako se neka sekvenca poruka često ponavlja, možemo je jednom modelovati i kasnije referencirati u svim situacijama kada je to potrebno.

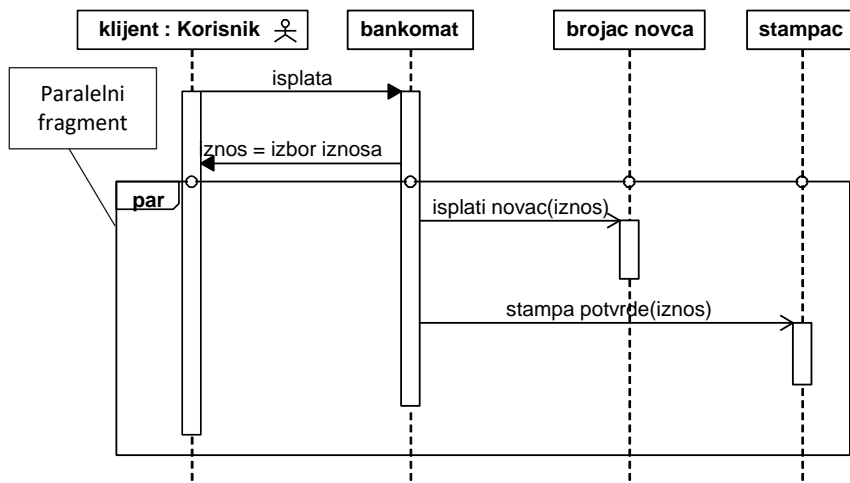


Slika 6.14 Primer korišćenja referencirajućih fragmenata

Na slici 6.14 je kao primer navedeno referenciranje dijagrama za proveru PIN-a i za isplatu. Natpis na referencirajućem fragmentu treba da odgovara nazivu korišćenog dijagrama.

### 6.3.6 Paralelni fragment

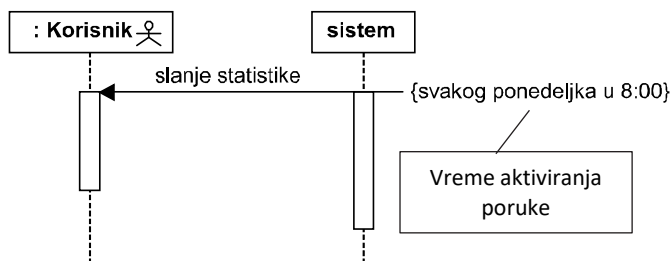
Paralelni fragment – par se koristi za specifikaciju paralelnog izvršavanja poruka koje se u njemu nalaze. Na slici 6.15 možemo primetiti da se isplata novca i štampa potvrde odvijaju paralelno.



Slika 6.15 Na dijagram sekvence sa slike 6.6 je dodat paralelni fragment, radi isticanja da se isplata novca i štampa potvrde odvijaju paralelno

### 6.4 Vremenske odrednice

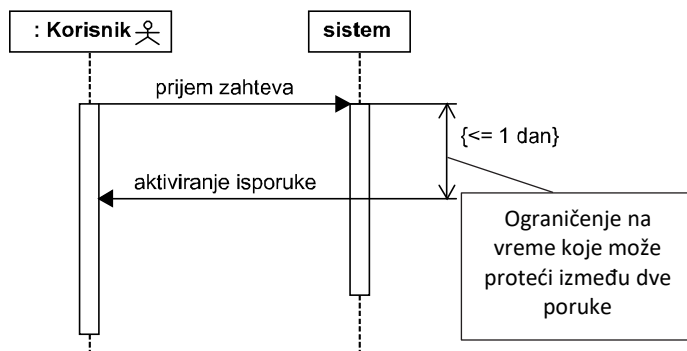
U dijagrame sekvence je moguće uključiti i vreme. To može biti vreme aktiviranja neke poruke (slika 6.16), ograničenje na vreme koje sme da protekne između dve poruke (slika 6.17), tačno vreme koje treba da protekne između dve poruke i sl.



Slika 6.16 Specifikacija vremena kada neka poruka treba da se pošalje



Vremenske odrednice se navode u vitičastim zagradama. Ako je vremenska odrednica pridružena jednoj poruci, navodi se u produžetku te poruke (slika 6.16). Ako se specificira vreme koje treba da protekne između dve poruke, navodi se na kotnoj liniji koja ih povezuje (slika 6.17).



Slika 6.17 Specifikacija ograničenja na vreme koje sme da protekne između dve poruke.

## 6.5 Dijagram sekvence kao pomoć u projektovanju dijagrama klasa

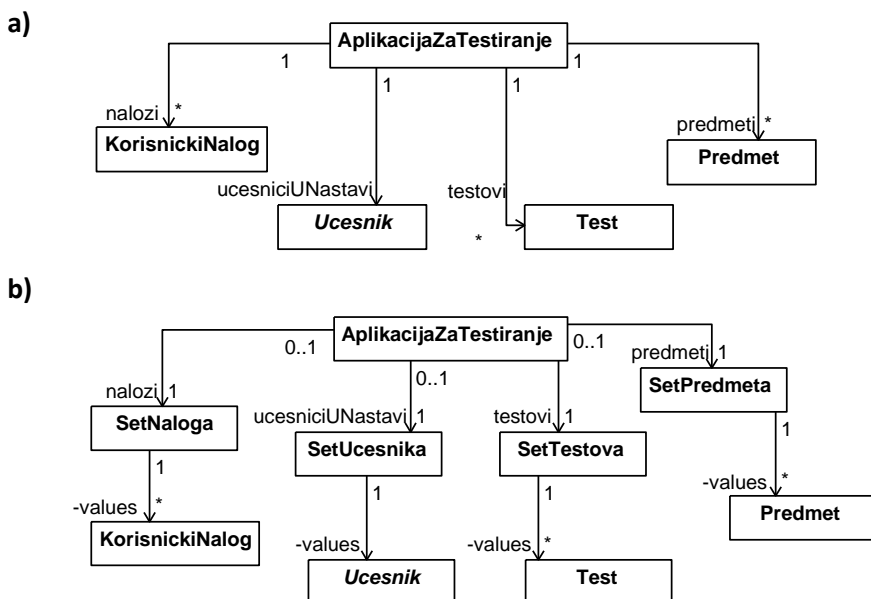
Dijagram sekvence se, tokom specifikacije dizajna, koristi kao most između dijagrama slučajeva korišćenja i dijagrama klasa. Analizom slučajeva korišćenja uočenih tokom analize zahteva i prevođenjem njihovih koraka na jednu ili više poruka koje razmenjuju objekti klasa u dijagramu sekvenci, otkrivamo metode i obeležja koji nedostaju, a često i nove klase. Dijagram sekvence nam omogućava da stavimo dijagram klasa „u pogon“ – pomaže nam da razmotrimo da li su svi scenariji korišćenja podržani od strane dijagrama klasa.

U sekciji 4.4.13 smo razmatrali slučajeve korišćenja za rešavanje testa od strane studenta, ocenjivanje rešenih testova i uvid u rešene testove, u okviru sistema za elektronsko ocenjivanje studenata. Radi podrške za njihovo izvršavanje, do tada razvijeni model je proširen novom klasom *ReseniTest* koja je povezana sa klasama *Student*, *Test* i *Odgovor*, a neke asocijacije su promenjene u bidirekzione (slike 4.26 i 4.27). Slučajevi korišćenja *Prijava na sistem*, *Zaključenje testa* i *Rešavanje testa* nisu bili razmatrani.

Da bismo mogli da podržimo navedene slučajeve korišćenja, potrebno je da uvedemo klasu *AplikacijaZaTestiranje*, koja će sadržati kolekcije instanci svih „bitnih“ klasa: *KorisnickiNalog*, *Ucesnik*, *Predmet* i *Test*. Kolekcije mogu da se implementiraju kao naslednici generičke klase *SetProperty* (listing 4.20), na način kako je pokazano u sekciji 4.4.1.5. Metode koje rade nad elementima

kolekcije treba da smeštamo u date klase, a AplikacijaZaTestiranje je zadužena da im preusmerava zahteve i podatke koje korisnik prosleđuje putem klasa korisničkog interfejsa i njihovih pridruženih kontrolera (slika 6.18).

Kada modelujemo dijagrame sekvence čije su uloge objekti klasa, možemo birati da li ćemo ih crtati za konceptualne ili implementacione dijagrame klasa, u zavisnosti od nivoa detalja koji želimo da prikazemo. Za slučajeve korišćenja koje ćemo u nastavku razmatrati, dijagrami sekvence će većinom biti modelovani za konceptualni dijagram klasa.



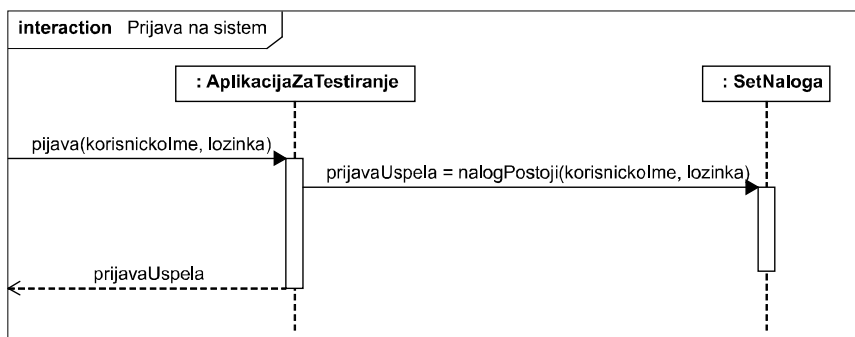
Slika 6.18 Instanca klase AplikacijaZaTestiranje sadržati kolekcije instanci klasa KorisnickiNalog, Ucesnik, Predmet i Test a) konceptualni model b) Implementacioni model

### 6.5.1 Prijava na sistem

Razmotrimo sledeće korake osnovnog toka slučaja korišćenja Prijava na sistem sa slike 2.6 iz drugog poglavlja:

1. *Korisnik unosi korisničko ime i lozinku u stranici za prijavu S1*
2. *Korisnik potvrđuje da je obavio unos klikom na „OK“ dugme*
3. *Sistem proverava da li su korisničko ime i lozinka ispravni*

Za prva dva koraka je zadužena klasa korisničkog interfejsa koja implementira stranicu S1 sa slike 2.7.



Slika 6.19 Dijagram sekvenci za prijavu na sistem. Iz dijagrama su izostavljene klase korisničkog interfejsa i korisnička uloga.

Za treći korak je zadužena klasa koja ima kolekciju korisničkih naloga – AplikacijaZaTestiranje. Trebalo bi joj dodati metodu `prijava(String korisnickoIme, String lozinka): Boolean` koja će se oslanjati na metodu `nalogPostoji(String korisnickoIme, String lozinka): Boolean` klase SetNaloga. Saradnja pomenutih klasa, tokom prijave na sistem, je prikazana na slici 6.19. Dodate metoda i obeležja se vide na dijagramu klasa na slici 6.22.

**Napomena:** Sa dijagrama na slici 6.19 smo mogli izostaviti objekat klase SetNaloga, ako je akcenat na razvoju konceptualnog modela i uslugama koje treba da obezbedi „spoljnom svetu“. Ako na nivou razvojnog tima postoji konvencija kako se konceptualni model preslikava na implementacioni, objekti kolekcija se obično ne prikazuju.

## 6.5.2 Zaključenje testa

Opis slučaja korišćenja Zaključenje testa je sledeći (slika 2.23):

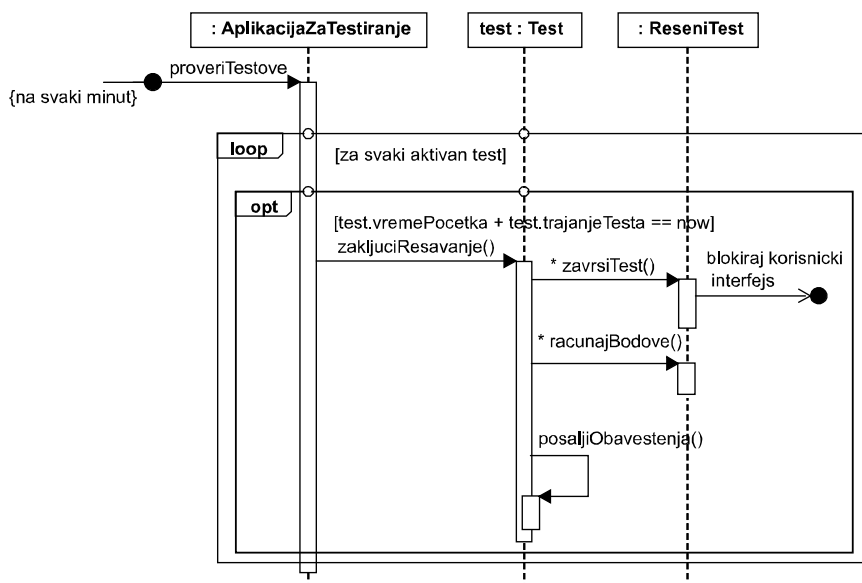
*Kada istekne vreme za rešavanje testa, treba blokirati rešavanje testova koje studenti još nisu završili, oceniti testove i poslati rezultate studentima, nastavnicima i asistentima koji su uključeni u predmet za koji se test organizuje.*

Da bi sistemski sat mogao da blokira dalje rešavanje, potrebno je znati da li je test aktiviran, kada je aktiviran i koliko je predviđeno da traje rešavanja testa. Primećujemo da nam nedostaju obeležja `aktivan: Boolean`, `trajanjeTesta: int` i `vremePocetka: DateTime` u klasi `Test`, kao i metode:

- `aktivirajTest()`, koja će postaviti aktivan na `true` i zabeležiti vreme početka.,
- `zakljuciTest()`, koja će blokirati dalji rad studentima i obaviti ocenjivanje i slanje rezultata.

S obzirom da sistem za elektronsko ocenjivanje podržava veliki broj predmeta i testova, očekuje se da u svakom trenutku postoji više testova koji su u različitim fazama rešavanja. Ako se trajanje testa unosi u minutama, potrebno je obezbediti metodu koju će sistemski sat pozivati svakog minuta da proveri da li je nekom od aktivnih testova upravo isteklo vreme (dodajemo metodu `proveriTestove()` u klasu `AplikacijaZaTestiranje`). Ako zbir vremena početka i trajanja testa daje trenutno vreme, data metoda treba da pozove `zakljuciTest()` razmatranog objekta klase `Test`.

Da bismo omogućili da studenti mogu ranije završiti polaganje, dodajemo metodu `završiTest()` u klasu `ReseniTest`, kao i obeležja `vremePocetka: DateTime` i `vremeZavrsetka: DateTime`. Vreme početka se beleži kada student počne da rešava test (ne mora biti isto kao vreme aktiviranja testa, jer možda kasni na polaganje), a vremeZavrsetka kada se pozove metoda `završiTest` (slika 6.20).



Slika 6.20 Dijagram sekvenci za zaključenje rešavanja testa, kada istekne vreme za izradu testa.

Da bismo podržali ocenjivanje, u klasu `ReseniTest` dodajemo metodu `racunajBodove()`. Klasa `ReseniTest` ima obeležje `osvojeniBodovi` i kolekciju datih odgovora, pri čemu svaki odgovor ima broj poena koji nosi, tako da je računanje osvojenih bodova jednostavno.

Ostaje još da podržimo slanje obaveštenja. Iz slučaja korišćenja sa slike 2.19 izdvajamo sledeće korake:

- 
1. *Sistem preuzima e-mail adresu osobe kojoj želi da pošalje poruku*
  2. *Sistem kreira poruku*
  3. *Sistem šalje poruku na e-mail adresu*
- 

Metoda za slanje obaveštenja o rezultatima testa `posaljiObavestjenja()` je dodata u klasu `Test`, pošto ona ima pristup svim potrebnim podacima (nastavnicima, studentima i rešenim testovima). Studentima treba poslati obaveštenje samo o njihovim bodovima, dok nastavnicima treba poslati spisak osvojenih bodova svih studenata koji su rešavali test. Metoda `posaljiObavestjenja` koristi dve pomoćne metode, za sastavljanje teksta obaveštenja studentu i nastavniku.

Metodu `tekstObavestjenja(): String` koja sastavlja tekst za studenta stavljamo u klasu `ReseniTest`. Metodu `tekstObavestjenjaZaNastavnika(): String` dodajemo u klasu `Test`, jer poseduje kolekciju instanci klase `ReseniTest`. Ako se upustimo u detalje implementacije, metoda `tekstObavestjenjaZaNastavnika` bi trebalo samo da pozove odgovarajuću metodu kolekcije `SetResenihTestova`. Radi podele nadležnosti, metode koje rade nad podacima stavljamo u onu klasu koja raspolaže datim podacima.

Primetimo da su brojna obeležja i metode izolovani tokom analize slučaja korišćenja i njihovog mapiranja na metode klase tokom modelovanja dijagrama sekvence na slici 6.20. Metoda `proveriTestove` je prikazana kao „nađena“ poruka, ali smo mogli prikazati i korisničku ulogu `Sistemska` koja je šalje. Poruka `zakljuciResavanje` u petlji šalje poruku `zavrstiTest` objektu klase `ReseniTest` (petlja je modelovana znakom „\*“ ispred imena poruke, umesto loop fragmentom). Slično važi i za poruku `racunajBodove`.

Poruka `zavrstiTest` beleži vreme završetka i šalje signal da treba da se blokira korisnički interfejs.

Odgovor na poruku `posaljiObavestjenje` nije modelovan na slici 6.20. Trebalo bi da se oslanja na poruke `tekstObavestjenja` i `tekstObavestjenjaZa-`

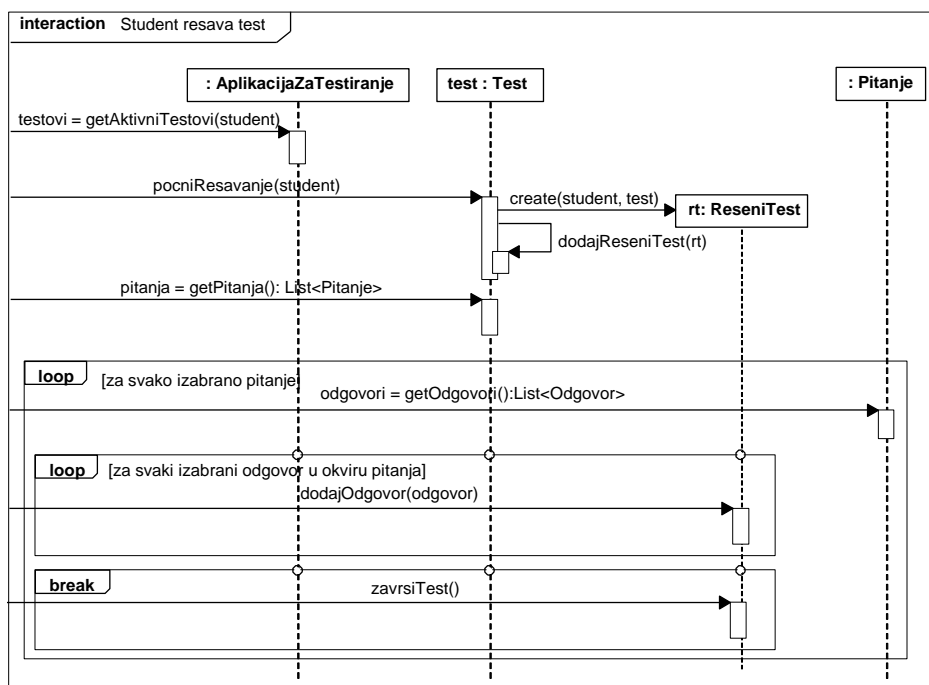
Nastavnika. Ostavljamo čitaocu da, radi vežbe, modeluje slanje obaveštenja kao poseban dijagram sekvence i da ga referencira iz dijagrama na slici 6.20.

### 6.5.3 Rešavanje testa

Slučajevi korišćenja za rešavanje testa i davanje odgovora su detaljno specificirani na slikama 2.13 i 2.14 u drugom poglavlju. U nastavku izdvajamo sledeće korake slučaja korišćenja Rešavanje testa:

1. *Student aktivira osvežavanje osnovne stranice*
2. *Sistem prikazuje sve aktivirane testove*
3. *Student bira test koji želi da rešava*
4. *Sistem prikazuje test studentu (skica S2).*
5. *[Tačka proširenja: SSC3 Davanje odgovora]*
6. *Student označava da je završio test izborom dugmeta „Test završen!“*

Kada se student prijavi, treba da izabere koji od aktivnih testova će da rešava, ako se istovremeno odvija polaganje više testova iz predmeta koje sluša. Ova situacija se može desiti ako su predmeti iz različitih godina studija. Da bismo



Slika 6.21 Dijagram sekvenci za slučaj korišćenja Rešavanje testa proširenog slučajem korišćenja Davanje odgovora

mogli da prikazemo te testove, dodajemo metodu `getAktivniTestovi` (`Student student`): `Set<Test>` u klasu `AplikacijaZaTestiranje`.

Kada student izabere test, trebalo bi kreirati instancu klase `ReseniTest`, koja povezuje datog studenta i izabrani test. U klasu `Test` dodajemo metodu `pocniResavanje()` koja će kreirati datu instancu i dodati je u kolekciju rešenih testova.

Ovo je trenutak kada se na ekranu studenta pojavljuju pitanja iz izabranog testa, tako da može da krene sa davanjem odgovora. Izdvajamo sledeće korake slučaja korišćenja `Davanje odgovora` sa slike 2.14:

- 
1. *Student aktivira link za davanje odgovora na pitanje*
  2. *Sistem prikazuje formu za davanje odgovora (skica S3)*
  3. *Student označava jedan ili više odgovora za koje misli da su tačni*
  4. *Student potvrđuje da je odgovorio na pitanje, klikom na dugme „OK“ na formi za davanje odgovora*
  5. *Sistem čuva sve odgovore*
- 

Za sve navedene korake već imamo podlogu u okviru dijagrama klasa. Ako student odluči da ranije završi test, iskoristiće metodu `zavrstiTest()` klase `ReseniTest`, koju smo već diskutovali.

Navedeni koraci slučajeva korišćenja, mapirani na poruke i fragmente interakcije dijagrama sekvence, su prikazani na slici 6.21. Fokus dijagrama je na podršci datih slučajeva korišćenja od strane domenskih klasa. Saradnja sa korisničkim interfejsom nije prikazana, tako da poruke stižu spolja.

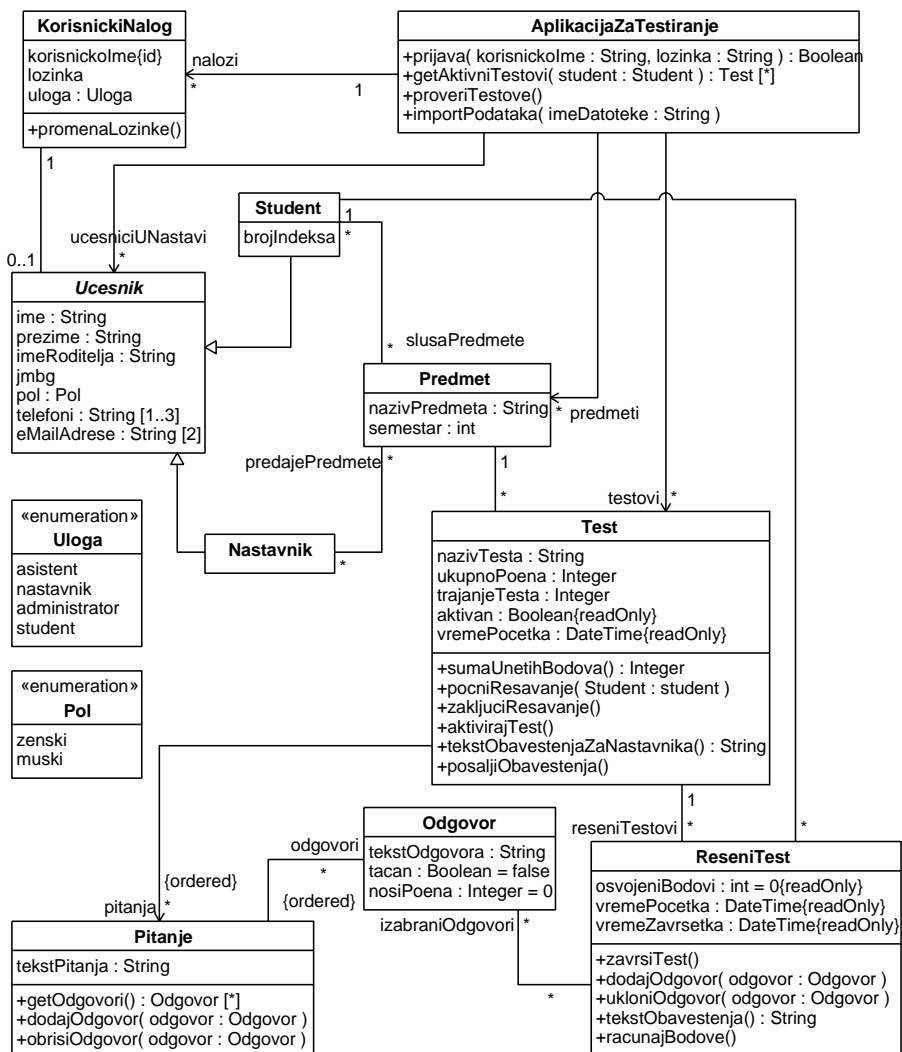
Konceptualni dijagram klasa, obogaćen do sada uočenim metodama i obeležjima, prikazan je na slici 6.22.

Preporučujemo čitaocu da, radi vežbe, modeluje i implementacioni dijagram klasa i da proširi dijagrame sekvence sa slika 6.20 i 6.21 tako da prikaže komunikaciju klasa u okviru implementacionog modela.

## 6.6 Šta smo naučili

Dijagram sekvence prikazuje interakciju, odnosno komunikaciju između uloga sistema u vremenu. Uloga je zajednički naziv za učesnike sistema i elemente sistema. Element sistema može biti pojedinačni objekat koji je instanca neke klase iz dijagrama klasa, proizvoljni deo sistema ili čak sistem u celini. Pod interakcijom se podrazumeva razmena poruka između uloga. Porukom se može modelovati:

- komunikacija između korisnika i sistema u ranim fazama razvoja (sistem se tada tretira kao „crna kutija”)
- slanje i prijem signala između različitih delova sistema,
- sinhroni ili asinhroni poziv metode objekta neke klase,
- vraćanje rezultata (povratna vrednost),
- kreiranje i brisanje objekata.



Slika 6.22 Konceptualni dijagram klasa za sistem za elektronsko ocenjivanje studenata obogaćen metodama i obeležjima uočenim tokom modelovanja dijagrama sekvence





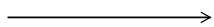
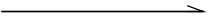
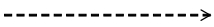
Dijagram sekvence se obično specificira za jedan slučaj korišćenja, mada se nekad može crtati i za neki njegov segment (osnovni ili neki od alternativnih tokova), kao i za više povezanih slučajeva korišćenja.

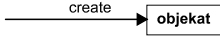

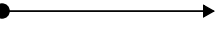
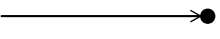
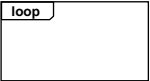
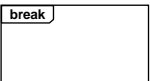
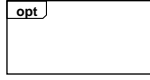
Tokom specifikacije dizajna, dijagram sekvence se koristi kao sprega između dijagrama slučajeva korišćenja i dijagrama klasa. Preslikavanjem koraka slučaja korišćenja na jednu ili više poruka koje razmenjuju objekti klasa u dijagramu sekvence, otkrivamo metode i obeležja koji nedostaju, a često i nove klase. Možemo birati nivo detalja koji želimo da prikazemo, kao i da li prikazujemo komunikaciju u okviru objekata iz konceptualnog ili implementacionog dijagrama klasa.

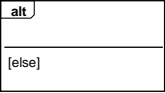


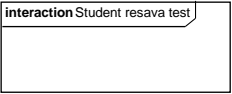
Akcent kod dijagrama sekvenci nije na algoritamskim aspektima, već na otkrivanju poruka koje se razmenjuju u sistemu.

U tabeli 6.2 su ukratko prikazani elementi koji se najčešće koriste na dijagramima sekvence.

Tabela 6.2 Elementi dijagrama sekvence

Simbol	Naziv	Opis
	Uloga	Uloge u dijagramu sekvence mogu biti korisnici, objekti klasa, delovi sistema i sistem u celini. Vertikalna isprekidana linija pridružena ulozi se zove linija života i označava vreme u kojem je uloga aktivna.
	Sinhrona poruka	Kod sinhronih poruka pošiljalac čeka da primalac obradi poruku da bi nastavio sa izvršavanjem.
	Asinhrona poruka	Kod asinhronih poruka pošiljalac nastavlja sa izvršavanjem odmah posle slanja poruke, dok primalac nezavisno od njega obrađuje poruku koja mu je prosleđena.
	Asinhrona poruka, UML 1.0	Starija UML 1.0 notacija za asinhronu poruku koju neki alati za modelovanje i dalje koriste.
	Povratna poruka	Poruka za vraćanje povratne vrednosti prethodno poslate poruke

	Poruka za kreiranje objekta	Poruka koja omogućava kreiranje objekata (instanci) klasa. Kreirani objekat se prikazuje u nastavku create poruke.
	Poruka za oslobađanje objekta	Poruka koja omogućava oslobađanje (de-alokaciju) objekta. Posle prijema delete poruke, na liniji života se pojavljuje krstić, kao oznaka da objekat u toj tački prestaje da postoji.
	„Nađena“ poruka	Poruka koja ima nepoznatog pošiljaoca. Obično se koristi radi izolovanje tačke od koje želimo da modelujemo interakciju, umesto da crtamo ceo lanac poruka koji počinje od korisničke uloge.
	„Izgubljena“ poruka	Poruka koja ima nepoznatog primaoca.
	Petlja	<p>Petlja – loop je fragment interakcije kojim se može specificirati ponavljanje izvršavanja određenog niza poruka. Poruke u petlji se mogu izvršavati:</p> <ul style="list-style-type: none"> <li>• navedeni broj puta, poput for petlje,</li> <li>• dok je neki uslov zadovoljen, po ugledu na while petlju, ili</li> <li>• za svaki element neke kolekcije, poput foreach petlje.</li> </ul>
	Prekidni fragment	Prekidni fragment – break omogućava da se modeluje izlazak iz petlje ili fragmenta koji ga okružuje. Break fragment koji se ne nalazi u nekom drugom fragmentu se može iskoristiti za modelovanje funkcionalnosti regiona mogućeg prekida iz dijagrama aktivnosti.
	Opcioni fragment	Opcioni fragment – opt se koristi za modelovanje uslovnog izvršavanja poruka koje se u okviru njega nalaze. Ako specificirani uslov nije zadovoljen, preskače se aktiviranje svih poruka u okviru datog fragmenta

	Alternativni fragment	Alternativni fragment – alt se koristi za modelovanje uslovnog izvršavanja kod kojeg postoji više grana.
	Referencirajući fragment	Referencirajući fragment – ref nam omogućava da referenciramo (uključimo) ranije definisani dijagram sekvence, kao kada pozivamo pod-aktivnost u dijagramu aktivnosti.
	Paralelni fragment	Paralelni fragment – par se koristi za specifikaciju paralelnog izvršavanja poruka koje se u njemu nalaze
	Granica interakcije	Granica interakcije nam omogućava da „ogradimo“ segment interakcije koji želimo da modelujemo. U tom slučaju, pošiljaoci i primaoci poruka mogu biti izvan granice.

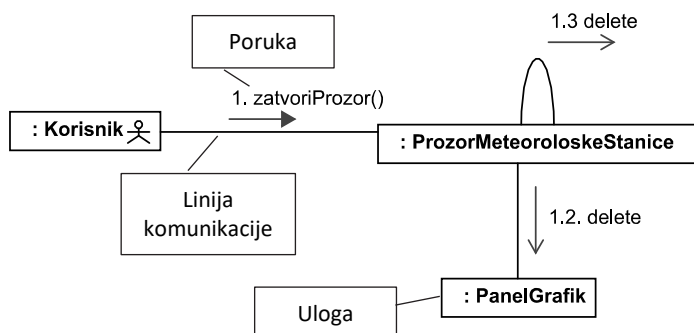


## Poglavlje 7

### Dijagram komunikacije

Slično kao dijagram sekvence, dijagram komunikacije takođe prikazuje interakciju, odnosno razmenu poruka između uloga sistema. Međutim, interakcija se ne prikazuje u odnosu na protok vremena, već se redosled slanja poruka i reagovanja na njih određuje na osnovu rednog broja poruke.

Elementi dijagrama komunikacije su: uloga, poruka i linija komunikacije (slika 7.1). Uloge imaju isto značenje i notaciju kao kod dijagrama sekvence, osim što nemaju liniju života. Poruke imaju isti format i notaciju, uz dodatak rednog broja, ali je broj vrsta manji: možemo koristiti sinhronu, asinhronu, povratnu, kao i poruke za kreiranje i oslobađanje objekata. Poruke za kreiranje i oslobađanje objekata se vizuelno ne razlikuju od ostalih poruka.



Slika 7.1 Dijagram komunikacije koji ilustruje oslobađanje objekata kada korisnik zatvori prozor meteorološke stanice. Ekvivalentan je dijagramu sekvence sa slike 6.8.

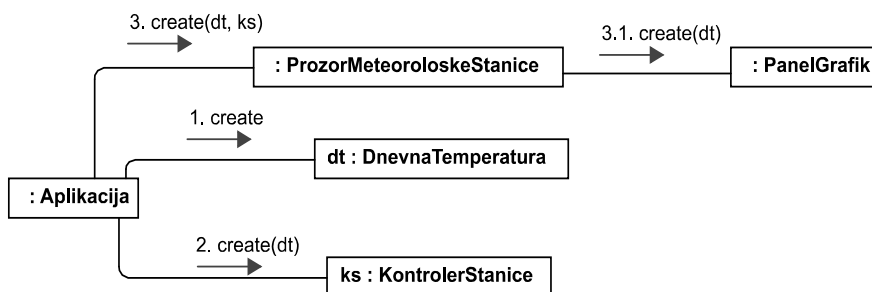
Linija komunikacije spaja uloge koje razmenjuju poruke, tako da je dijagram komunikacije pogodniji od dijagrama sekvenci za uočavanje zavisnosti između elemenata sistema. Ovo je posebno važno ako su elementi koje prikazujemo na dijagramu objekti klase. Ukoliko uloga šalje sebi poruku, linija komunikacije počinje i završava se na istom mestu (videti objekat klase **ProzorMeteoroloskeStanice** na slici 7.1).

Dijagram komunikacije na „kompaktniji“ način prikazuje razmenu poruka između uloga, što može biti pogodnije u situacijama kada u nekom scenariju korišćenja imamo puno uloga koje učestvuju u interakciji. Međutim, ako postoji veliki broj poruka koji se razmenjuje između manjeg broja uloga, dijagram komunika-

cije može postati težak za praćenje, pa je u takvim situacijama dijagram sekvence obično pogodniji. Nezavisno od navedenih razloga, izbor između ove dve vrste dijagrama zavisi pre svega od navika projektnog tima.

## 7.1 Određivanje rednog broja poruke

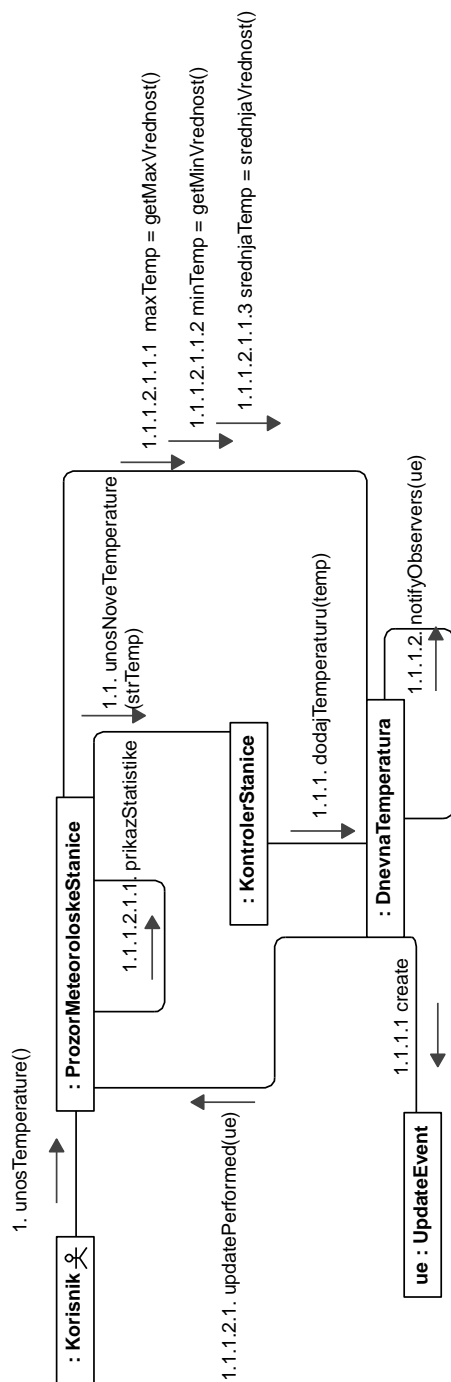
Redosled izvršavanja specificiramo rednim brojem poruke. Na slici 7.2 možemo primetiti da aplikacija prvo kreira objekat klase DnevnaTemperatura (redni broj poruke je 1), zatim objekat klase KontrolerStanice kojem prethodno kreirani objekat prosleđuje kao argument konstruktora (redni broj 2) i na kraju objekat klase ProzorMeteoroloskeStanice (redni broj 3). Poruka za kreiranje objekta klase PanelGrafik ima redni broj 3.1, s obzirom da se ona izvršava u kontekstu poruke sa rednim brojem 3 (odnosno, ugnježdjena je u poziv poruke za kreiranje objekta klase ProzorMeteoroloskeStanice).



Slika 7.2 Dijagram komunikacije koji ilustruje inicijalizaciju aplikacije za unos izmerenih dnevnih temperatura na jednom mernom mestu. Odgovara dijagramu sekvence sa slike 6.7.

Na slici 7.3 je prikazan malo složeniji primer, koji pokazuje razmenu poruka koju inicira korisnik, kada pritisne dugme za unos temperature u prozoru meteorološke stanice. Dijagram na slici 7.3 prikazuje komunikaciju koja je već modelovana dijagramom sekvence sa slike 6.1.

S obzirom da su sve poruke ugnježdene u okviru poruke unosTemperature koju je poslao korisnik, njihovi redni brojevi počinju cifrom 1. Svaki put kada jedna poruka aktivira drugu, dodaje se jedan nivo u numeraciji. Na primer, poruke za preuzimanje minimalne, maksimalne i srednje temperature, upućene objektu klase DnevnaTemperatura, imaju brojeve 1.1.1.2.1.1.1, 1.1.1.2.1.1.2 i 1.1.1.2.1.1.3, iz razloga što se tim redom aktiviraju od strane poruke prikaziStatistiku, koja ima redni broj 1.1.1.2.1.1. Poruka prikaziStatistiku je aktivirana od strane poruke notifyObservers koja ima redni broj 1.1.1.2.1, itd.

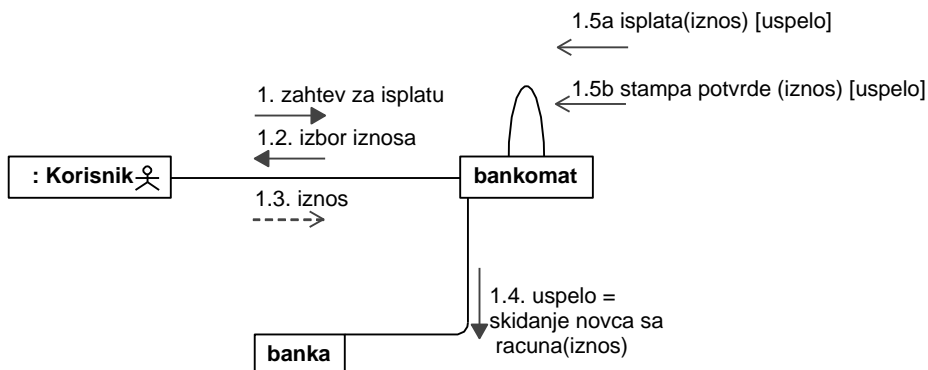


Slika 7.3 Dijagram komunikacije ilustruje interakciju klasa *ProzorMeteoroloskeStanice*, *KontrolerStanice* i *DnevnaTemperatura*, koje implementiraju MVC i *Observer* šablon. Ekvivalentan dijagram sekvence je modelovan na slici 6.1.

**Napomena:** Primećujemo da na slici 7.3 nije lako shvatiti šta se tačno dešava, za razliku od dijagrama sekvence na slici 6.1. Ovo je primer kada je bolje iskoristiti dijagram sekvence za specifikaciju interakcije.

## 7.2 Specifikacija kontrole toka

Dijagrami komunikacije nemaju fragmente interakcije kojima se za niz poruka može specificirati izvršavanje u petlji ili uslovno izvršavanje. Razlog za to je što akcenat treba da bude na otkrivanju poruka i zavisnosti između uloga u sistemu, koja se vidi preko linija komunikacije, ne toliko na algoritamskim aspektima izvršavanja.



Slika 7.4 Dijagram komunikacije koji pokazuje saradnju korisnika, bankomata i banke, tokom isplate novca. Ova komunikacija je ranije modelovana dijagramom sekvence sa slike 6.13.

Ipak, može se specificirati uslovno izvršavanje i izvršavanje u petlji na nivou jedne poruke, na isti način kao kod dijagrama sekvence, kada ne želimo da koristimo opt i loop fragmente interakcije (videti sekcije 6.3.1 i 6.3.2).

Da ponovimo, uslovno izvršavanje na nivou jedne poruke se može specificirati dodavanjem uslova u uglastim zagradama na kraju poruke. Izvršavanje u petlji se postiže dodavanjem znaka „\*“ ispred poruke, na primer:

\* zavrstiTest()

Na slici 7.4 je modelovano uslovno izvršavanje poruka isplata i stampa. Posle zahteva za isplatu i izbora iznosa, pokušava se skidanje novca sa računa u banci (poruka 1.4). Ako je skidanje uspešno, poruke isplata i stampa se izvršavaju (uslov se nalazi u uglastim zagradama na kraju poruke).



Paralelno izvršavanje se može specificirati tako što se porukama dodeli isti broj, na način kako je objašnjeno u sekciji 7.1, sa dodatkom slova koje mora biti različito za svaku poruku koja se paralelno izvršava.

Na slici 7.4 asinhrona poruka isplata i stampa se izvršavaju paralelno, što se vidi na osnovu njihovog broja: 1.5a i 1.5b. Kada se ne bi izvršavale paralelno, njihovi brojevi bi bili 1.5 i 1.6.

### 7.3 Šta smo naučili

Dijagram komunikacije prikazuje interakciju, odnosno razmenu poruka između uloga sistema, ali se redosled izvršavanja poruka ne vidi na osnovu položaja na vremenskoj osi, već putem rednog broja poruke.

Elementi dijagrama komunikacije su: uloga, poruka i linija komunikacije.

Uloga ima isto značenje i notaciju kao u dijagramu sekvence, ali bez linije života.

Poruke imaju isti format i notaciju, uz dodatak rednog broja, ali je broj vrsta manji: možemo koristiti sinhrona, asinhrona, povratne, poruke za kreiranje i poruke za oslobađanje objekata.

Linija komunikacije spaja uloge koje razmenjuju poruke, tako da je dijagram komunikacije pogodniji od dijagrama sekvenci za uočavanje zavisnosti između elemenata sistema.

Dijagrami komunikacije nemaju fragmente kojima se za niz poruka može specificirati izvršavanje u petlji, odnosno uslovno izvršavanje. Navedeno se može specificirati za jednu poruku, na isti način kao kod dijagrama sekvence, kada ne koristimo loop i opt fragmente interakcije: dodavanjem znaka „\*“ ispred poruke, odnosno, uključivanjem uslova u uglastim zagradama na kraj poruke.

Paralelno izvršavanje se može specificirati tako što se porukama dodeli isti broj, sa dodatkom slova koje mora biti različito za svaku poruku koja se paralelno izvršava.

Dijagram komunikacije na „kompaktniji“ način prikazuje razmenu poruka između uloga, što može biti pogodnije u situacijama kada u nekom scenariju korišćenja imamo puno uloga koje učestvuju u interakciji. Međutim, ako postoji veliki broj poruka koji se razmenjuje između manjeg broja uloga, dijagram komunikacije može postati težak za praćenje, pa je u takvim situacijama dijagram sekvence obično pogodniji.



## Poglavlje 8

### Dijagram prelaza stanja

Dijagram prelaza stanja, odnosno, dijagram konačnih automata, koristi se za projektovanje softverskih ili hardverskih sistema za čije ponašanje je karakteristično da se mogu nalaziti u konačnom skupu stanja i da je prelazak iz jednog stanja u drugo uzrokovan događajima.

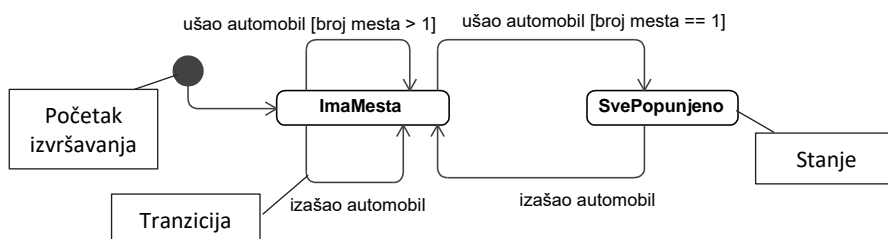
Na slici 8.1 je prikazan jednostavan dijagram prelaza stanja za industrijski kontroler koji upravlja parkingom. Specifikacija kontrolera je data u nastavku.

---

**Primer 8.1** Zadatak kontrolera je:

- *da prima signale za ulazak i izlazak automobila i ažurira broj slobodnih mesta koji prikazuje na ekranu postavljenim kod ulaza na parking,*
  - *da na semaforu koji se nalazi ispred ulaza na parking drži upaljeno zeleno svetlo ako ima slobodnih mesta, odnosno da uključi crveno svetlo kada se zauzme poslednje slobodno mesto.*
- 

Možemo primetiti da se kontroler parkinga može naći u stanju kada ima mesta i kada je sve popunjeno (slika 8.1). Događaji koji izazivaju prelazak iz jednog stanja u drugo su ulazak i izlazak automobila. Oni se na dijagramu prelaza stanja crtaju kao *tranzicije*.

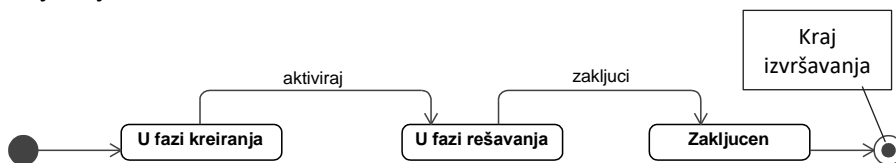


Slika 8.1 Početni dijagram prelaza stanja za kontroler parkinga

Ukoliko postoji početno stanje u kojem se sistem uvek nalazi posle pokretanja, to možemo specificirati dodavanjem simbola za početak izvršavanja i povezivanjem sa tim stanjem. Na slici 8.1 početno stanje je ImaMesta.

Slično, ako postoji završno stanje u kojem sistem ostaje (ne može se više vratiti u neko drugo stanje), spajamo ga tranzicijom sa simbolom za kraj izvršavanja.

Na slici 8.2 je prikazan dijagram prelaza stanja za test iz sistema za elektronsko ocenjivanje.



Slika 8.2 Dijagram prelaza stanja za test iz sistema elektronskog ocenjivanja

## 8.1 Tranzicija

Tranzicija modeluje reakciju na događaj, koja može izazvati prelazak iz jednog stanja u drugo ili povratak u isto stanje, uz izvršenje pridruženih akcija. Na dijagramu se tranzicija crta kao strelica sa natpisom koji se može sastojati od sledećih elemenata:

događaj [uslov] / akcija

Pored naziva događaja, koje smo imali priliku da vidimo na prethodnim dijagramima, mogu se navesti i vrednosti koje opisuju taj događaj, ako postoje. Na primer, kod događaja `pritisnutaCifra`, argument može biti cifra koju je korisnik izabrao na tastaturi: `pritisnutaCifra(cifra)`.

Uslovom se specificira da do tranzicije ne dolazi uvek, već samo ako je dati uslov zadovoljen. Uslov je obavezno navesti ukoliko postoji više tranzicija koje izaziva isti događaj i koje polaze iz istog stanja. Na slici 8.1 vidimo da:

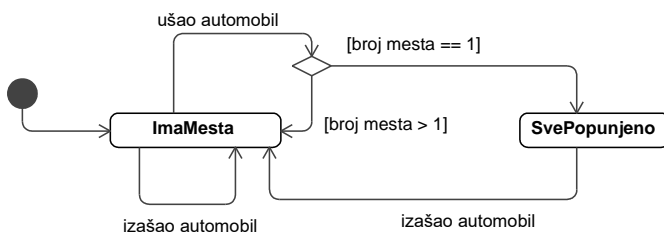
- ako uđe automobil, a broj mesta je veći od jedan, kontroler parkinga ostaje u stanju `ImaMesta`,
- ako uđe automobil, a broj mesta je 1, kontroler parkinga prelazi u stanje `SvePopunjeno` (automobil koji ulazi će zauzeti poslednje slobodno mesto).

Pregledniji način za modelovanje ovakvih situacija je korišćenjem simbola za uslovno izvršavanje, kao na slici 8.3. Vidimo da je odluka o prelasku na sledeće stanje ili povratku na isto specificirana uslovima na granama iza navedenog simbola.

**Napomena:** U dijagramima prelaza stanja se mogu koristiti simboli za početak, kraj, uslovno izvršavanje, razdelnik i spoj, koje smo ranije koristili u dijagramima aktivnosti. U kontekstu dijagrama prelaza stanja, oni se tretiraju kao *pseudo-stanja* – nisu stanja, ali se mogu povezivati tranzicijama sa drugim stanjima.

Primer paralelnog izvršavanja korišćenjem razdelnika i spoja će biti prikazan u sekciji 8.3.

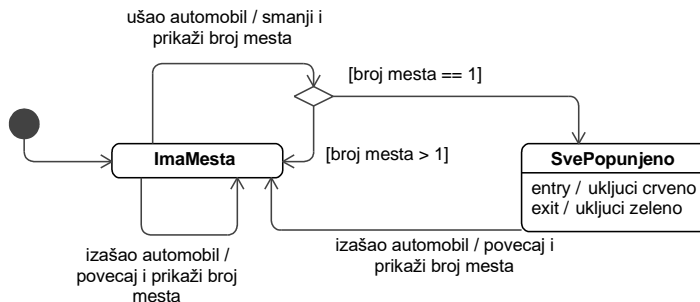
Akcija koja se može navesti iza kose crte u okviru specifikacije tranzicije, označava šta sistem treba da preduzme tokom tranzicije.



Slika 8.3 Dijagram prelaza stanja za kontroler parkinga, preglednije nacrtan korišćenjem simbola za uslovno izvršavanje

Na slici 8.4 je prikazan završen dijagram prelaza stanja za kontroler parkinga, u koji su uključene i akcije na tranzicijama, kao i entry i exit akcije u okviru stanja SvePopunjeno.

Možemo primetiti da je reakcija kontrolera na ulazak ili izlazak automobila ažuriranje broja mesta i prikaz tog broja na ekranu.



Slika 8.4 Završen dijagram prelaza stanja za kontroler parkinga, sa specificiranim akcijama u okviru tranzicija i stanja SvePopunjeno

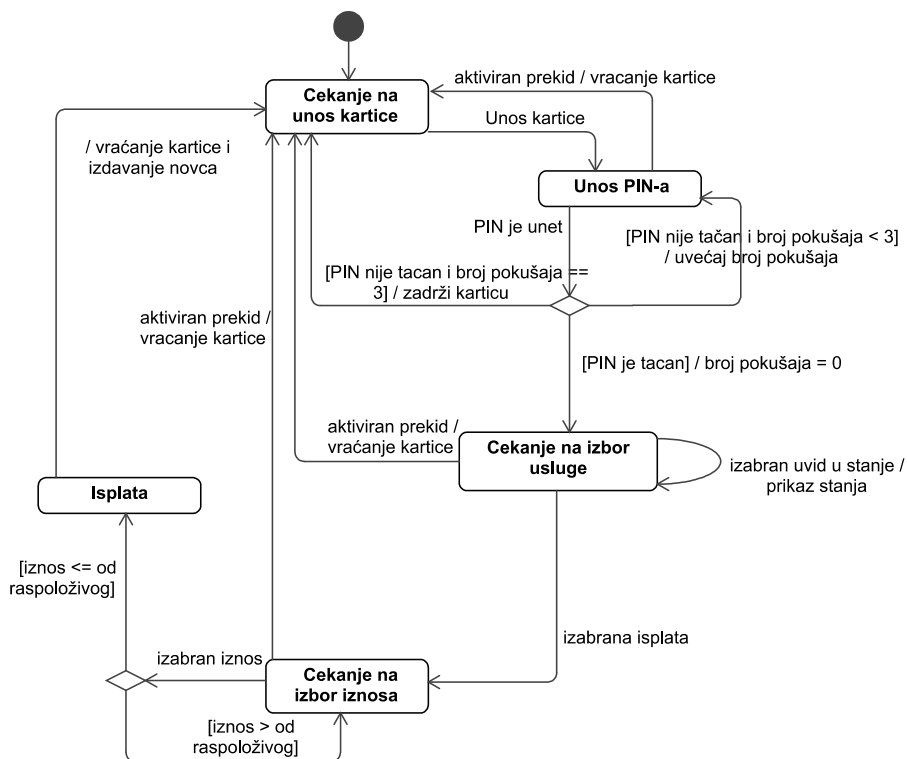
Dijagram prelaza stanja za bankomat je dat na slici 8.5. Bankomat nema krajnje stanje. Pored toga, iz svakog stanja postoji izlazna tranzicija, što je znak da je sistem dobro projektovan: automati koji treba da pružaju usluge, ne bi smeli da se nađu u stanju iz kojeg ne mogu da izađu. Dijagram prelaza stanja omogućava da se detaljno razmotre stanja i tranzicije između njih kod projektovanja složenih sistema.

## 8.2 Stanje

Prvi korak u projektovanju ponašanja sistema je izolovanje stanja i događaja koji izazivaju tranzicije. Sledeći korak je analiza akcija koje dati sistem treba da preduzme kao reakciju na događaje. Akcije se mogu specificirati u okviru tranzicija i u okviru stanja.

Načini na koje stanje može da izvrši akciju su:

- u trenutku ulaska u stanje (entry),
- u trenutku izlaska iz stanja (exit),
- tokom boravka u stanju (do),
- u okviru internih tranzicija.



Slika 8.5 Dijagram prelaza stanja za bankomat

### 8.2.1 Entry i exit akcije

Na slici 8.4 vidimo da se, prilikom ulaska u stanje SvePopunjeno, uključuje crveno svetlo na semaforu (entry/ ukljuci crveno). Prilikom izlaska iz datog stanja se uključuje zeleno svetlo (exit/ ukljuci zeleno).

Akcije pridružene entry i do događajima se izvršavaju pri svakoj tranziciji koja omogućava ulazak u stanje. Akcije pridružene exit događaju se izvršavaju pri svakoj tranziciji koja obezbeđuje izlazak iz datog stanja. Ovo važi i za tranzicije koje izlaze i vraćaju se u isto stanje, kao npr. kod izlaska automobila sa parkinga u okviru stanja ImaMesta.

Akcija koja se izvršava tokom boravka u stanju (do) se aktivira posle izvršenja entry akcije.

**Napomena:** Potrebno je pažljivo analizirati gde ćemo smestiti akcije. Ako sve tranzicije koje ulaze u neko stanje izvršavaju istu akciju, možemo tu akciju vezati za entry događaj datog stanja. Ali, ako samo jedna tranzicija koja ulazi u stanje ne treba da izvrši datu akciju, ne bi trebalo koristiti entry, već ostaviti akciju u tranzicijama kojima je to potrebno.

Slično važi i za exit događaj: ako sve tranzicije koje izlaze iz nekog stanja obavljaju istu akciju, treba je vezati za exit događaj datog stanja. U suprotnom, akcija treba da ostane u tranzicijama.

Na slici 8.4 smo akcije ukljuci crveno i ukljuci zeleno mogli staviti i u tranzicije, ali su, radi ilustrovanja korišćenja ulaznih i izlaznih akcija stanja, modelovane na pokazani način.

Navođenje akcija nije obavezno – mogu postojati tranzicije i stanja bez akcija.

Ako želimo da reagujemo na događaje, ali da ne aktiviramo entry, exit i do akcije, možemo koristiti tzv. interne tranzicije (videti sekciju 8.2.3)

### 8.2.2 Do akcija

Razmotrimo sledeću specifikaciju jednostavnog kontrolera koji treba da rukuje štampačem.

---

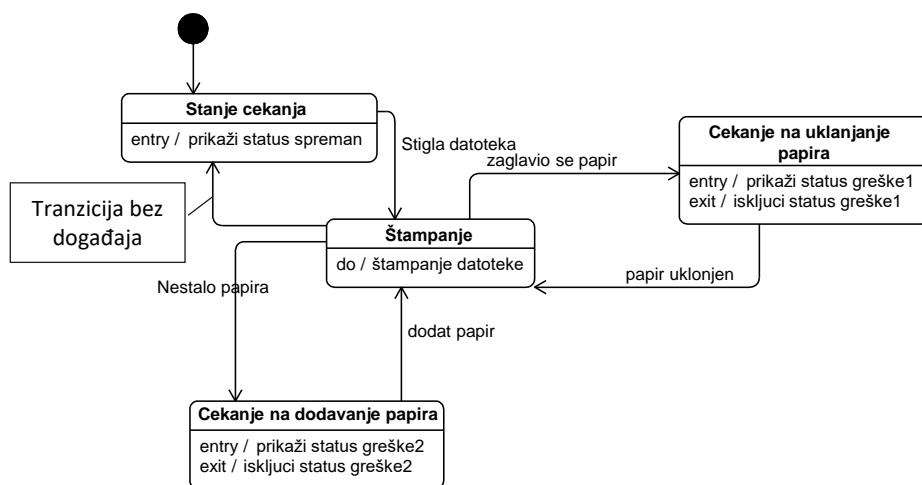
**Primer 8.2** *Kada se uključi, štampač se nalazi u stanju čekanja na datoteku koju treba da štampa. U tom trenutku štampač treba da prikaže status koji označava da je spreman. Kada datoteka stigne, počinje sa štampom, a status se menja u „zauzet“. Po završetku štampe, štampač se vraća u stanje čekanja na novi dokument i ponovo prikazuje status „spreman“.*

Ako se prilikom štampe zaglavi papir, štampač treba da prikaže odgovarajući status greške i da čeka da neko ukloni papir. Po uklanjanju papira, nastavlja sa štamptom, a status menja na „zauzet“.

Ako prilikom štampe nestane papira, štampač prikazuje status greške i čeka na dopunu papira. Kada se dopuna obavi, nastavlja sa štamptom i prikazuje status „zauzet“.

Radi jednostavnosti, pretpostavimo da štampač može da prima nove datoteke samo dok je u stanju čekanja na datoteku.

Dijagram prelaza stanja za primer 8.2 je prikazan na slici 8.6. Sa dijagrama je izostavljena akcija postavljanja statusa štampača na „zauzet“ tokom boravka u stanju Štampanje, što ostavljamo čitaocu da završi radi vežbe.



Slika 8.6 Dijagram prelaza stanja za primer 8.2

Primećujemo da stanje Štampanje ima do akciju koja izvršava štampanje proleđene datoteke. Ukoliko se desi neki problem, ova akcija se prekida i štampač prelazi u odgovarajuće stanje čekanja dok se problem ne otkloni. Posle toga, vraća se u prethodno stanje i do akcija nastavlja svoj posao. Po završetku posla, odmah se prelazi u Stanje čekanja, bez navođenja događaja koji aktivira tranziciju (obratiti pažnju na „praznu“ tranziciju na slici 8.6).

**Napomena:** Tranzicija ne mora da ima događaj koji je aktivira *jedino* ako izlazi iz stanja koje ima do akciju, što znači da se stanje napušta u trenutku kada je posao koji obavlja završen. U svim ostalim slučajevima, događaj koji aktivira tranziciju mora da postoji, inače model nije korektno specificiran.

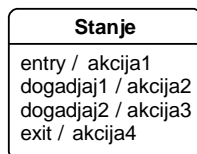


Ako ne možemo da otkrijemo događaj koji je okidač za tranziciju u neko stanje ili iz njega, moguće je da nije u pitanju stanje, već akcija u okviru tranzicije nekog drugog stanja. Način da prepoznamo da li je u pitanju stanje je da se zapitamo da li sistem provodi neko vreme u tom stanju? Ako ne provodi, verovatno bi trebalo da to stanje uklonimo sa dijagrama.

Akcije koje se navode u okviru tranzicija, kao i entry i exit akcije stanja, treba da traju kratko i ne smeju se prekidati. Do akcija može da traje dugo i dozvoljeno ju je prekidati, u slučaju potrebe.

### 8.2.3 Interne tranzicije

Kada nam je potrebna tranzicija koja izvršava akciju kao odgovor na neki događaj i zatim se vraća u isto stanje, a pri tome ne želimo da se aktiviraju entry, exit i do akcije datog stanja, možemo koristiti interne tranzicije. Na slici 8.7 vidimo dve takve tranzicije koje se aktiviraju na događaj1 i događaj2.

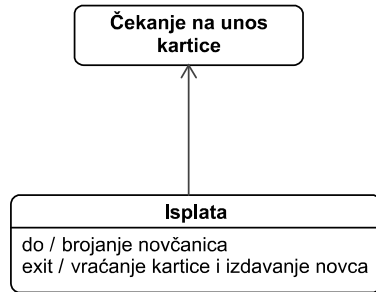


Slika 8.7 događaj1 i događaj2 su primer specifikacije internih tranzicija u okviru stanja

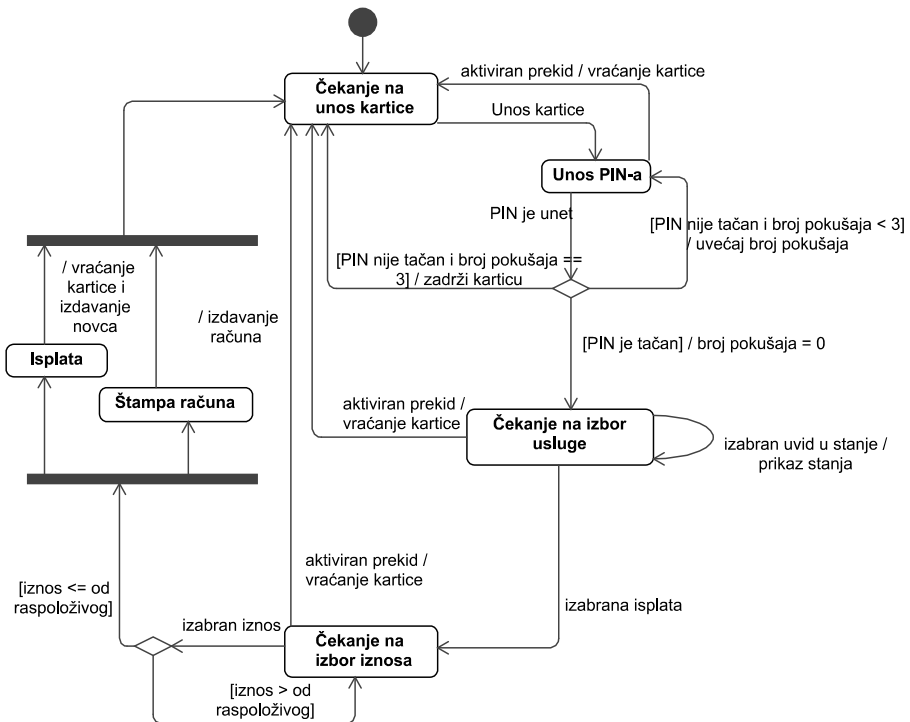
Interne tranzicije mogu imati iste elemente kao i „obične”: događaj, uslov i akciju. Na slici 8.7 nije korišćen uslov.

### 8.3 Paralelno izvršavanje

Na slici 8.9 vidimo primer paralelnog izvršavanja stanja Isplata i Štampa računa. Isplata i Štampa su stanja koja imaju do metodu, tako da njihova izlazna tranzicija nema događaj – stanja se napuštaju u trenutku kada je posao koji obavljaju završen. Na slici 8.8 je prikazan ekvivalentan način modelovanja stanja za isplatu, gde se akcije vraćanja kartice i izdavanja novca obavljaju pri izlasku iz stanja (exit), umesto na izlaznoj tranziciji, kao na slici 8.9.



Slika 8.8 Modelovanje stanja za isplatu, gde se akcije vraćanja kartice i izdavanja novca obavljaju pri izlasku iz stanja

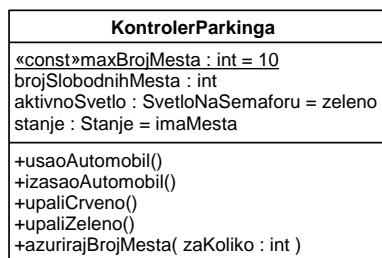


Slika 8.9 Dijagram prelaza stanja za bankomat. Stanja Isplata i Štampa računa se paralelno izvršavaju.

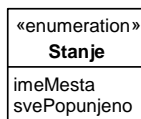
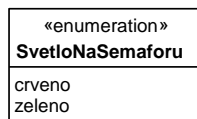
## 8.4 Prevođenje dijagrama prelaza stanja na programski kod

Preostaje da vidimo kako se dijagram prelaza stanja, koji se modeluje tokom projektovanja ponašanja budućeg sistema, može iskoristiti za njegovu implementaciju u programskom kodu.

Naizgled, najjednostavniji način jeste da modelujemo klasu KontrolerParkinga kao na slici 8.10. Obeležja klase KontrolerParkinga su: broj trenutno slobodnih mesta na parkingu, aktivno svetlo na semaforu (crveno ili zeleno) i stanje u kojem se kontroler nalazi, koje je nabrojanog tipa. Klasa KontrolerParkinga ima metode koje obezbeđuju odgovor na signale koje kontroler prima od svojih senzora da je ušao ili izašao automobil (usaoAutomobil() i izasaoAutomobil()), kao i metode kojima se obraća semaforu i ekranu za prikaz slobodnih mesta (upaliCrveno(), upaliZeleno() i azurirajBrojMesta(zaKoliko: int)).



Ne preporučuje se!



Slika 8.10 Implementacija konačnog automata korišćenjem nabrojanog tipa Stanje. Ovakav način se ne preporučuje!

Za model sa slike 8.10, programski kod za obradu događaja bi izgledao kao na listingu 8.1. Za kompleksnije sisteme sa velikim brojem stanja i događaja, poput bankomata, za svaki događaj koji stiže iz spoljnog sveta bi trebalo napraviti jednu metodu u kojoj bi bio switch sa puno grana.

```

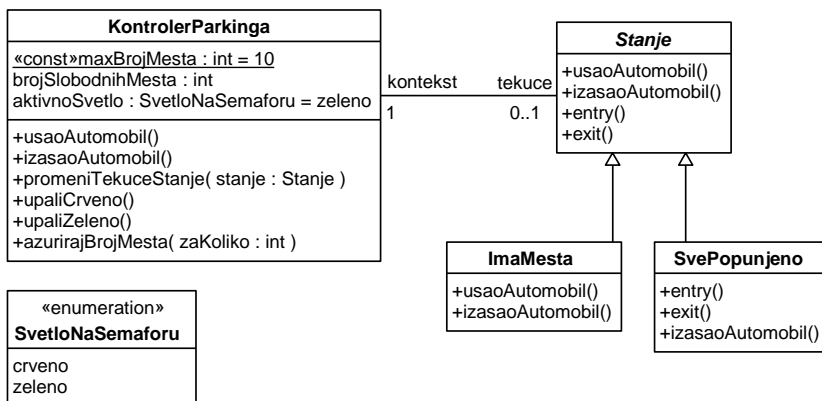
public Boolean izasaoAutomobil() {
    switch(stanje) {
        case IMA_MESTA:
            // ...
            break;
        case SVE_POPUNJENO:
            // ...
            break;
    }
    ...
}

```

Listing 8.1 Ilustracija implementacije metode za obradu događaja za model sa slike 8.10

Kao što je rečeno u sekciji 4.4.2.2, indikator da nam nedostaje nasleđivanje je kada u programskom kodu primetimo postojanje glomaznih switch ili if... else if... else if... iskaza, koji se izvršavaju u zavisnosti od vrednosti obeležja nekog nabrojanog tipa.

Preporučeni način za implementaciju dijagrama prelaza stanja je korišćenjem *State* projektnog šablona [Gamma94] (slika 8.11). *State* šablon podrazumeva uvođenje apstraktnog stanja koje nasleđuju klase koje odgovaraju uočenim stanjima konačnog automata. Za svaki *događaj* koji izaziva tranziciju (internu ili eksternu) treba dodati po jednu metodu u apstraktno stanje. Ako stanja imaju entry, exit ili do akcije, za njih se takođe kreiraju metode u pretku (videti klasu Stanje na slici 8.11).



Slika 8.11 Preporučena implementacija konačnog automata korišćenjem *State* projektnog šablona

Apstraktna klasa Stanje može implementirati metode sa praznim telima, tako da naslednici redefinišu samo one metode koje za njih imaju smisla (videti listinge 8.3, 8.4 i 8.5).

Drugi način je da predak ima apstraktne metode, ili da se koristi interfejs, tako da svaki naslednik mora da implementira sve metode. Oba pristupa se ravno-pravno sreću u literaturi.

Klasa koja komunicira sa spoljnim svetom i ima stanje se naziva kontekst izvršavanja konačnog automata. Za primer 8.1, kontekst izvršavanje je klasa KontrolerParkinga. Ona ima sva obeležja koja su potrebna za posao koji kontroler obavlja i po jednu metodu:

- za svaku *akciju* koja se pročita sa dijagrama prelaza stanja (sa tranzicija i u okviru stanja),
- za svaki *događaj* koji stiže iz spoljnog sveta.

Kontekst izvršavanja treba da ima i metodu `promeniTekuceStanje(stanje: Stanje)`, koje će koristiti stanja kada treba da aktiviraju tranziciju na drugo stanje, usled događaja koji izaziva tranziciju. Klasa KontrolerParkinga je implementirana na listingu 8.7.

Bitno je primetiti da se kontekst izvršavanja i stanja uzajamno „vide“. Stanja se koriste kao „skretničari“ – kada dobiju događaj, u okviru reakcije na njega pozivaju odgovarajuću metodu kontrolera, a ako događaj izaziva i tranziciju, tada:

- kreiraju instancu stanja na koje treba preći,
- pozivaju metodu `promeniTekuceStanje` sa novim stanjem kao argumentom.

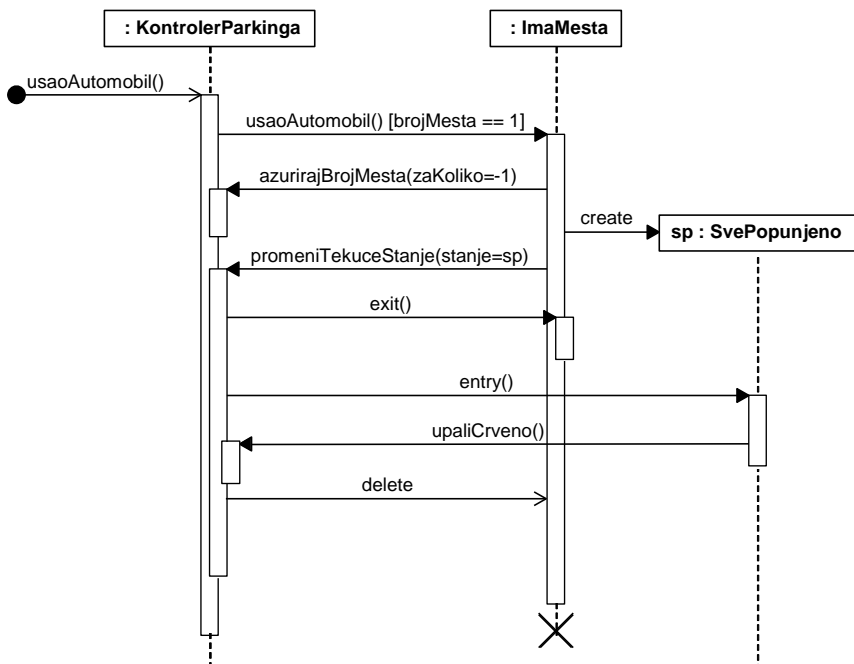
Metoda za promenu stanja u okviru klase koja predstavlja kontekst izvršavanja radi sledeće:

- poziva `exit` metodu tekućeg stanja, ako postoji, radi izlaska iz stanja
- poziva `entry` metodu novog stanja, ako postoji, radi ulaska u stanje
- postavlja novo stanje za tekuće,
- dealocira prethodno stanje (ili to u nekom trenutku obavi *garbage collector*),
- poziva do metodu novog stanja, ako postoji.

Na ovaj način, svako radi samo ono što je njegov posao i postiže se jasna raspodela nadležnosti između stanja i konteksta izvršavanja, što za posledicu ima

značajno jednostavnije metode koje reaguju na događaje (više nemamo switch iskaze).

Pomenuta interakcija između stanja i instance klase KontrolerParkinga je ilustrovana dijagramima sekvence na slikama 8.12 i 8.13.



Slika 8.12 Dijagram sekvence za tranziciju u stanje SvePopunjeno, kada automobil zauzima poslednje slobodno mesto

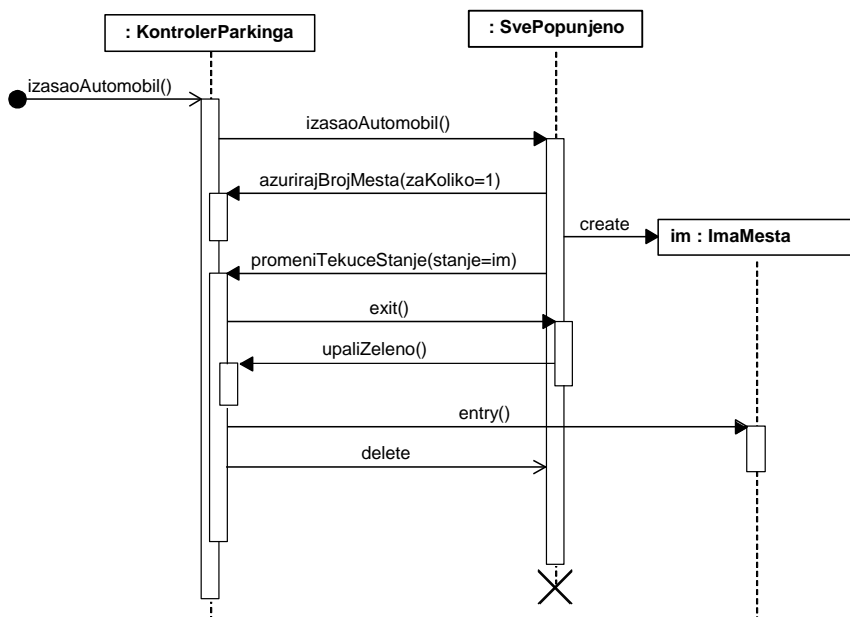
Na slici 8.12 je prikazan dijagram sekvence za situaciju kada automobil ulazi i zauzima poslednje slobodno mesto. Tada dolazi do prelaza u stanje SvePopunjeno.

Na slici 8.13 je prikazan dijagram sekvence koju inicira automobil koji izlazi sa popunjenog parkinga. Izlazak automobila aktivira tranziciju u stanje ImaMesta.

Radi testiranja rada kontrolera parkinga, implementiran je prozor za testiranje (klasa `TestProzor`, listing 8.8) koji se sastoji od dva panela, za prikaz slobodnih mesta (`BrojMestaPanel`, listing 8.9) i semafora (`SemaforPanel`, listing 8.10).

Radi komunikacije bez uvođenja uzajamnih zavisnosti, klasa `KontrolerParkinga` implementira `Publisher` interfejs, a paneli `Observer` interfejs.

Izgled testnog prozora je prikazan na slici 8.14. Dugmići se koriste za kreiranje događaja ulaska i izlaska automobila, što bi u realnom sistemu bio posao senzora.



Slika 8.13 Dijagram sekvence za tranziciju u stanje ImaMesta, kada automobil izađe sa popunjenog parkinga

**Napomena:** U situacijama kada je kreiranje novog objekta stanja i oslobađanje postojećeg pri svakoj tranziciji „skupo“ sa stanovišta performansi, ili ako stanja imaju obeležja čije je vrednosti potrebno čuvati, implementacija se sprovodi na drugačiji način. U tom slučaju, klasa koja je kontekst izvršavanja na početku rada kreira instance svih stanja. Stanje, prilikom tranzicije, treba da izabere neku od postojećih instanci koju kontekst čuva i prosledi je metodi za promenu tekućeg stanja.

U nastavku su prikazani listinzi<sup>20</sup> za situaciju kada se, pri tranziciji, svaki put kreira nova instance stanja i oslobađa postojeća.

<sup>20</sup> Kompletan programski kod se može naći na <https://github.com/milosavljevicg/SIMS-primeri>.



Slika 8.14 Prozor za testiranje kontrolera za upravljanje parkingom (klasa TestProzor)

```
public enum SvetloNaSemaforu {
    CRVENO, ZELENO
}
```

Listing 8.2 Nabrojani tip za svetla na semaforu

```
/**
 * Stanje - apstraktni predak hijerarhije
 */
public abstract class Stanje {
    protected KontrolerParkinga kontekst;

    public Stanje(KontrolerParkinga aKontekst) {
        kontekst = aKontekst;
    }

    public void entry() {
    }

    public void exit() {
    }

    public void usaoAutomobil() {
    }

    public void izasaoAutomobil() {
    }
}
```

Listing 8.3 Apstraktno stanje – predak hijerarhije stanja



```

/**
 * Stanje koje je aktivno kada nema mesta na parkingu
 */
public class SvePopunjeno extends Stanje {

    public SvePopunjeno(KontrolerParkinga aKontekst) {
        super(aKontekst);
    }

    public void entry() {
        kontekst.upaliCrveno();
    }

    public void exit() {
        kontekst.upaliZeleno();
    }

    public void izasaoAutomobil() {
        kontekst.azurirajBrojMesta(1);
        kontekst.promeniTekuceStanje(new ImaMesta(kontekst));
    }
}

```

Listing 8.4 Stanje koje je aktivno kada su sva mesta popunjena na parkingu

```

/**
 * Stanje koje je aktivno kada ima mesta na parkingu
 */
public class ImaMesta extends Stanje {
    public ImaMesta(KontrolerParkinga aKontekst) {
        super(aKontekst);
    }

    public void usaoAutomobil() {
        kontekst.azurirajBrojMesta(-1);
        if (kontekst.getBrojSlobodnihMesta() == 0)
            kontekst.promeniTekuceStanje(new SvePopunjeno(kontekst));
    }

    public void izasaoAutomobil() {
        if (kontekst.getBrojSlobodnihMesta() < KontrolerParkinga.MAX_BROJ_MESTA) {
            kontekst.azurirajBrojMesta(1);
        }
    }
}

```

Listing 8.5 Stanje koje je aktivno kada ima mesta na parkingu

```

/**
 * Događaj koji opisuje promene u podacima kontrolera. Kontroler parkinga
 * prosleđuje ovaj događaj svima koji su se registrovali da ga prate.
 *
 * U ozbiljnijim sistemima obično postoji hijerarhija događaja gde svaki opisuje
 * neku specifičnu promenu. Za vežbu, možete napraviti hijerarhiju događaja:
 * apstraktni UpdateEvent i više naslednika: za promenjeno svetlo na semaforu, za
 * promenjen broj slobodnih mesta, kada je izašlo poslednje vozilo...
 */
public class UpdateEvent extends EventObject {
    private int brojMesta;
    private SvetloNaSemaforu aktivnoSvetlo;

    public UpdateEvent(Object object, int aBrojMesta, SvetloNaSemaforu
        aAktivnoSvetlo) {
        super(object);
        brojMesta = aBrojMesta;
        aktivnoSvetlo = aAktivnoSvetlo;
    }

    public int getBrojMesta() {
        return brojMesta;
    }

    public SvetloNaSemaforu getAktivnoSvetlo( ) {
        return aktivnoSvetlo;
    }
}

```

Listing 8.6 Događaj koji se aktivira prilikom ulaska ili izlaska automobila

```

/**
 * Kontroler parkinga - klasa koja predstavlja kontekst za izvršavanje konačnog
 * automata
 */
public class KontrolerParkinga implements Publisher {
    public static final int MAX_BROJ_MESTA = 10;

    private int brojSlobodnihMesta;
    private SvetloNaSemaforu aktivnoSvetlo;
    private Stanje tekuceStanje;

    public KontrolerParkinga() {
        tekuceStanje = new ImaMesta(this);
        brojSlobodnihMesta = MAX_BROJ_MESTA;
        aktivnoSvetlo = SvetloNaSemaforu.ZELENO;
    }

    public SvetloNaSemaforu getAktivnoSveto() {
        return aktivnoSvetlo;
    }
}

```

Listing 8.7 (prvi deo) Klasa KontrolerParkinga predstavlja kontekst izvršavanja konačnog automata. Implementira Publisher interfejs radi komunikacije sa klasama korisničkog interfejsa.

```

public int getBrojSlobodnihMesta() {
    return brojSlobodnihMesta;
}

/**
 * U realnom sistemu, ove metode bi se direktno obraćala semaforu (fizičkom
 * uređaju)
 */
public void upaliCrveno() {
    aktivnoSvetlo = SvetloNaSemaforu.CRVENO;
    notifyObservers();
}

public void upaliZeleno() {
    aktivnoSvetlo = SvetloNaSemaforu.ZELENO;
    notifyObservers();
}

/**
 * Automat informacije o događajima koje dobija od svojih senzora prosleđuje
 * svom aktivnom stanju
 */
public void usaoAutomobil() {
    tekuceStanje.usaoAutomobil();
}

public void izasaoAutomobil() {
    tekuceStanje.izasaoAutomobil();
}

public void azurirajBrojMesta(int zaKoliko) {
    brojSlobodnihMesta = brojSlobodnihMesta + zaKoliko;
    notifyObservers();
}

public void promeniTekuceStanje(Stanje novoStanje) {
    tekuceStanje.exit();
    novoStanje.entry();
    tekuceStanje = novoStanje;
}

/**
 * Podrška za dodavanje "posmatrača"
 */
private List<UpdateObserver> observers = new ArrayList<UpdateObserver>();

public void addObserver(UpdateObserver observer) {
    observers.add(observer);
}

public void removeObserver(UpdateObserver observer) {
    observers.remove(observer);
}

```

Listing 8.7 Klasa KontrolerParkinga (drugi deo)

```

    public void notifyObservers() {
        UpdateEvent e = new UpdateEvent(this, brojSlobodnihMesta, aktivnoSvetlo);
        for (UpdateObserver observer : observers) {
            observer.updatePerformed(e);
        }
    }
}

```

Listing 8.7 Klasa KontrolerParkinga (treći deo)

```

/**
 * Testni prozor koji omogućava isprobavanje rada kontrolera parkinga
 */
public class TestProzor extends JDialog {
    private JButton btnUsaoAutomobil;
    private JButton btnIzasaoAutomobil;
    private KontrolerParkinga kontrolerParkinga;

    public TestProzor(KontrolerParkinga kontrolerParkinga) {
        this.kontrolerParkinga = kontrolerParkinga;
        setTitle("Test prozor za kontroler parkinga");
        setSize(300, 400);
        setLayout(new GridLayout(2, 2));
        kreirajPanele();
        kreirajDugmad();
    }

    /**
     * Kreiranje panela za prikaz semafora i raspoloživih mesta
     */
    private void kreirajPanele() {
        SemaforPanel pnlSemafor = new SemaforPanel();
        add(pnlSemafor);
        BrojMestaPanel pnlBrojMesta = new BrojMestaPanel();

        add(pnlBrojMesta);
        //paneli se registruju za pracenje dogadjaja o promeni podataka u
        //kontroleru parkinga
        kontrolerParkinga.addListener(pnlBrojMesta);
        kontrolerParkinga.addListener(pnlSemafor);
    }

    /**
     * Kreiranje dugmadi koja simuliraju ulaske i izlaske automobila
     */
    private void kreirajDugmad() {
        JPanel btnPanel = new JPanel();
        btnUsaoAutomobil = new JButton();
        btnUsaoAutomobil.setText("Ulaz automobila");
        btnUsaoAutomobil.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                kontrolerParkinga.usaoAutomobil();
            }
        });
    }
}

```

Listing 8.8 (prvi deo) Testni prozor za isprobavanje rada kontrolera parkinga

```

        btnPanel.add(btnUsaoAutomobil);

        btnIzasaoAutomobil = new JButton();
        btnIzasaoAutomobil.setText("Izlaz automobila");
        btnIzasaoAutomobil.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                kontrolerParkinga.izasaoAutomobil();
            }
        });
        btnPanel.add(btnIzasaoAutomobil);
        add(btnPanel);
    }
}

```

Listing 8.8 Testni prozor (drugi deo)

```

/**
 * Panel za prikaz raspoloživog broja mesta na parkingu
 */
public class BrojMestaPanel extends JPanel implements UpdateObserver {
    private JLabel prikazMesta;
    private JLabel natpis;

    public BrojMestaPanel() {
        setLayout(new GridLayout(2, 0));
        natpis = new JLabel("Broj slobodnih mesta", SwingConstants.CENTER);
        add(natpis);
        prikazMesta = new JLabel(String.valueOf(KontrolerParkinga.MAX_BROJ_MESTA),
            SwingConstants.CENTER);
        prikazMesta.setFont(new Font(prikazMesta.getFont().getName(), Font.PLAIN,
            40));
        add(prikazMesta);
    }

    /**
     * Reakcija na događaj o promeni podataka u kontroleru parkinga:
     */
    public void updatePerformed(UpdateEvent e) {
        prikazMesta.setText(String.valueOf(e.getBrojMesta()));
    }
}

```

Listing 8.9 Panel za prikaz raspoloživog broja mesta na parkingu

```

/**
 * Panel koji prikazuje svetla na semaforu
 */
public class SemaforPanel extends JPanel implements UpdateObserver {
    private JPanel crvenoSvetlo;
    private JPanel zelenoSvetlo;

    public SemaforPanel() {
        setLayout(new GridLayout(2, 0));
        crvenoSvetlo = new JPanel();
        zelenoSvetlo = new JPanel();
        add(crvenoSvetlo);
        add(zelenoSvetlo);
        ukljuciZeleno();
    }

    /**
     * Reakcija na događaj o promeni podataka u kontroleru parkinga
     */
    public void updatePerformed(UpdateEvent e) {
        if (e.getAktivnoSvetlo() == SvetloNaSemaforu.CRVENO)
            ukljuciCrveno();
        else
            ukljuciZeleno();
    }

    private void ukljuciZeleno() {
        zelenoSvetlo.setBackground(Color.green);
        crvenoSvetlo.setBackground(Color.gray);
    }

    private void ukljuciCrveno() {
        crvenoSvetlo.setBackground(Color.red);
        zelenoSvetlo.setBackground(Color.gray);
    }
}

```

Listing 8.10 Panel za stilizovani prikaz semafora

```

public class Aplikacija {

    public static void main(String[] args) {
        KontrolerParkinga kontroler = new KontrolerParkinga();
        TestProzor prozor = new TestProzor(kontroler);
        prozor.setVisible(true);
    }
}

```

Listing 8.11 Klasa Aplikacija inicijalizuje i pokreće testni prozor, sa instancom kontrolera parkinga kao argumentom

## 8.5 Šta smo naučili

Dijagram prelaza stanja se koristi za projektovanje softverskih ili hardverskih sistema za čije ponašanje je karakteristično da se mogu nalaziti u konačnom skupu stanja i da je prelazak iz jednog stanja u drugo uzrokovan događajima.

Osnovni elementi dijagrama su stanje i tranzicija. Tranzicija modeluje reakciju na događaj, koja može izazvati prelazak iz jednog stanja u drugo ili povratak u isto stanje, uz izvršenje pridruženih akcija.


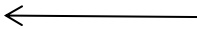
U okviru stanja se takođe mogu izvršavati akcije: prilikom ulaska u stanje (entry akcija), izlaska iz stanja (exit akcija), boravka u stanju (do akcija) i tokom internih tranzicija. Interne tranzicije se koriste kada nam je potrebna tranzicija koja izvršava akciju kao odgovor na neki događaj i zatim se vraća u isto stanje, a pri tome ne želimo da se aktiviraju entry, exit i do akcije datog stanja.





U okviru dijagrama prelaza stanja se mogu koristiti i elementi iz dijagrama aktivnosti: za početak i kraj izvršavanja, uslovno izvršavanje, razdelnik i spoj. U kontekstu dijagrama prelaza stanja oni se nazivaju *pseudo-stanja*.

Za prevođenje dijagrama prelaza stanja na programski kod preporučljivo je koristiti *State* projektni šablon.

U tabeli 8.1 je dat kratak prikaz elemenata koji se najčešće koriste na dijagramima prelaza stanja.

Tabela 8.1 Elementi dijagrama prelaza stanja

Simbol	Naziv	Opis
	Stanje	Stanje u kojem se sistem nalazi.
	Tranzicija	Tranzicija modeluje prelazak iz jednog stanja u drugo ili povratak u isto stanje, pod dejstvom nekog događaja. Za tranziciju se može specificirati događaj, uslov od kojeg zavisi da li će do tranzicije doći i akcija koja rom prilikom treba da se izvrši.

	Početak izvršavanja	Povezivanjem simbola za početak izvršavanja sa nekim stanjem, označavamo da je to stanje početno - sistem se, po uključivanju, nalazi u tom stanju.
	Kraj izvršavanja	Tranzicija koja polazi od nekog stanja i završava se u simbolu za kraj aktivnosti, označava da je to završno stanje u kojem sistem ostaje - ne može se više vratiti u neko drugo stanje.
	Uslovno izvršavanje	Ukoliko postoji više tranzicija koje polaze iz istog stanja i izaziva ih isti događaj, mogu se sve „uliti“ u simbol za uslovno izvršavanje, na čijim izlaznim granama će se navoditi uslov od kojeg zavisi gde će se tranzicija završiti.
	Razdelnik ili spoj	Omogućava specifikaciju paralelnog izvršavanja više stanja, kao i kraj paralelnog izvršavanja.



## Poglavlje 9

# Dijagram komponenti i dijagram raspoređivanja

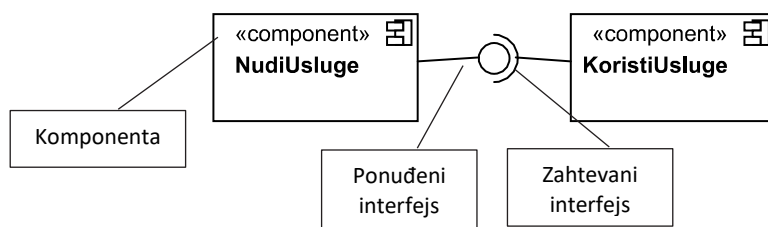
U okviru ovog poglavlja je dat kratak prikaz dijagrama komponenti i dijagrama raspoređivanja. S obzirom da studenti druge godine još nisu imali susret sa razvojem softvera baziranim na komponentama, niti sa višeslojnim aplikacijama koje se obično navode kao primer u dijagramima raspoređivanja, ograničićemo se samo na osnovne koncepte i prepoznavanje notacije. U [Rumbaugh05] su dati svi detalji koji omogućavaju samostalan rad, kada to bude potrebno na višim godinama studija.

### 9.1 Dijagram komponenti

Dijagram komponenti se koristi za prikaz komponenti sistema, njihove uzajamne saradnje i unutrašnje strukture.

Pod komponentom se podrazumeva deo softvera koji može da se nezavisno nabavi od drugih komponenti u sistemu i da, kroz interfejse koje obezbeđuje i/ili zahteva, komunicira sa ostatkom sistema. Zamena jedne komponente drugom koja nudi iste usluge, ne bi trebalo da utiče na funkcionisanje sistema.

Na sici 9.1 su prikazane dve komponente, od kojih jedna nudi uslugu, a druga je koristi.

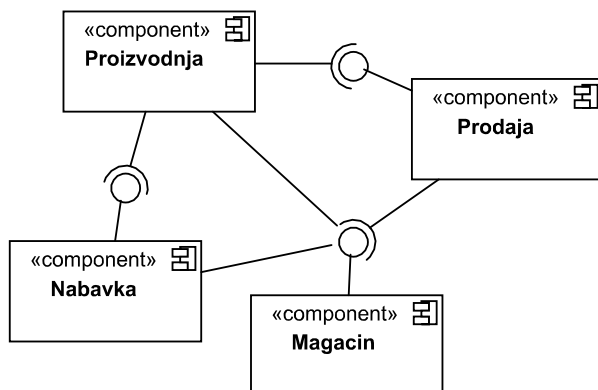


Slika 9.1 Jedna komponenta nudi uslugu kroz ponudeni interfejs, a druga koristi uslugu kroz zahtevani interfejs

Na slici 9.2 je prikazan malo realniji primer komponenti koje implementiraju različite poslovne podsisteme: nabavku, prodaju, proizvodnju i upravljanje magacinima. Date komponente uzajamno razmenjuju usluge. Magacin nudi mogućnost smeštanja proizvedenih dobara, kao i materijala koji se naručuju od strane podsistema nabavke. Podsystem prodaje koristi informacije o raspoloži-

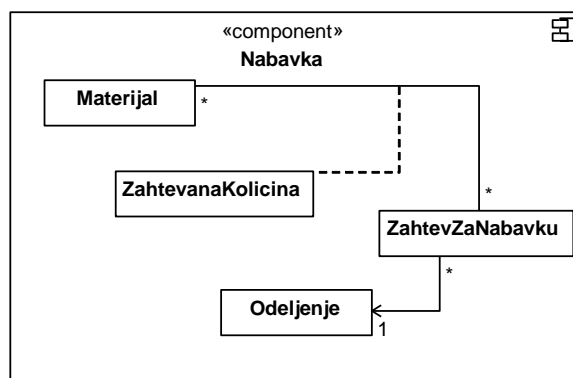
vim količinama proizvoda u magacinima tokom komunikacije sa kupcima. Pod-sistem prodaje daje informacije šta se najviše naručuje, da bi mogla da se planira proizvodnja. Proizvodnja zahteva za nabavkom materijala potrebnih za proizvodnju upućuje podsistemu nabavke.

Možemo primetiti da većina komponenti sa slike 9.2 ima i zahtevane i ponuđene interfeje.



Slika 9.2 Komponente poslovnog informacionog sistema proizvodnog preduzeća

Ako želimo da prikazemo unutrašnju strukturu komponente, možemo da uključimo i dijagram klase koje implementiraju ponašanje komponente (slika 9.3). U okviru dijagrama klase se može navesti proizvoljan nivo detalja.



Slika 9.3 Prikaz unutrašnje strukture komponente

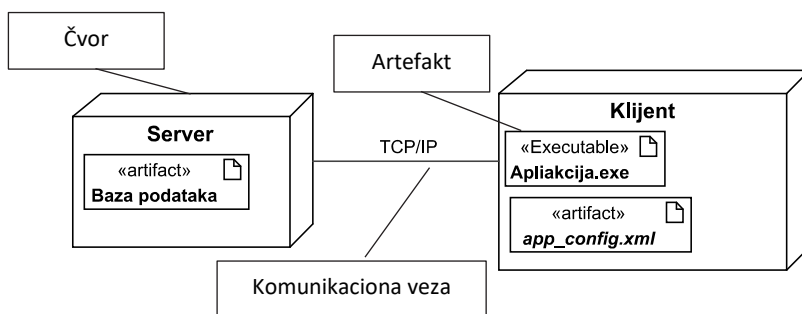
## 9.2 Dijagram raspoređivanja

Dijagram raspoređivanja (*deployment diagram*) se koristi za prikaz instalacije softverskog sistema. Na dijagramu raspoređivanja su najčešći elementi artefakti i čvorovi (*node*).

Artefakti predstavljaju datoteke različitih vrsta koje su potrebne za funkcionisanje softvera: izvršive datoteke (na primer .exe, .jar, .dll), različite konfiguracione datoteke (na primer .xml, .properties), baze podataka i sl.

Čvorovi su hardverski uređaji ili različite softverske infrastrukture na koje se artefakti raspoređuju. Čvorovi se mogu spajati komunikacionim vezama, na kojima se može navesti protokol koji se koristi za komunikaciju.

Na slici 9.4 je dat primer dijagrama raspoređivanja za dvoslojnu, klijent/server aplikaciju. Na serveru se nalazi baza podataka, na klijentima je instalirana izvršiva aplikacija i njena konfiguraciona datoteka, a komunikacija između servera i klijenata se odvija putem TCP/IP protokola.



Slika 9.4 Primer dijagrama raspoređivanja za klijent/server aplikaciju

## 9.3 Šta smo naučili

U ovom poglavlju smo se upoznali sa osnovnim elementima notacije dijagrama komponenti i dijagrama raspoređivanja, čime smo zaokružili osnove modelovanja softvera korišćenjem jezika UML 2.0.

Poznavanje i ispravno korišćenje notacije različitih vrsta dijagrama koje smo proučavali u ovom i prethodnim poglavljima je važno, da bi oni koji gledaju modele razumeli našu nameru. Međutim, još je važnije poznavanje prednosti i mana svake vrste dijagrama i razumevanje načina na koji njegovo korišćenje može doprineti razvoju softverskog projekta u celini.

Radi vežbanja izložene materije, u sledećoj sekciji su dati neki zadaci koji povezuju različite vrste dijagrama i omogućuju da se softverski sistem koji se razvija analizira iz različitih uglova.

## 9.4 Zadaci za vežbanje

**1. zadatak.** Modelovati softver za podršku sistema za slanje i prijem pošiljki. Pošiljke mogu biti: pisma, paketi i telegrami. Cena otpremanja pisma i paketa se računa kao: masa pisma ili paketa u gramima \* cena po gramu, a telegrama na osnovu broja reči (cena reči \* broj reči). Ako se prilikom slanja naznači da je u pitanju hitna pošiljka, zaračunava se dodatni iznos koji ne zavisi od vrste pošiljke.

Prilikom slanja, pošiljalac predaje referentu pošiljku, koji joj dodeljuje jedinstveni ID, beleži datum i vreme prijema, računa cenu i preuzima podatke pošiljaoca i primaoca. Podaci primaoca i pošiljaoca su: ime, prezime, poštanski broj, mesto i adresa stanovanja. Pošiljalac dobija i korisničko ime i lozinku, kao i ID dodeljen pošiljki, da bi mogao da prati njen status (primljena, u transportu, stigla u mesto isporuke, isporučena, vraćena). Pošiljka se vraća pošiljaocu ukoliko, po prispeću u mesto isporuke, ne uspeju kontaktirati primaoca u roku od 7 dana od prijema pošiljke.

Pošiljalac treba da ima mogućnost uvida u istoriju slanja svih svojih pošiljki: koja pošiljka je u pitanju, cena u trenutku slanja, datum i vreme slanja, datum i vreme prispeća. Softver treba da ima sve cene po kojima je računat ili se trenutno računa iznos pošiljki i datume kada su te cene važile.

Modelovati:

- a) dijagram slučajeva korišćenja,
- b) konceptualni dijagram klasa,
- c) dijagram koji prikazuje sekvencu događaja za prijem i slanje pošiljke. Obraditi i granični slučaj povratka pošiljke.

**2. zadatak.** Modelovati životni ciklus korisničkog naloga u okviru sajta za *online* kupovinu. Po kreiranju novog naloga, on se nalazi u stanju čekanja na verifikaciju, putem linka koji korisniku stiže na e-mail. Posle verifikacije, nalog postaje aktivan. Ukoliko se nalog ne verifikuje u vremenu od tri dana, postaje suspendovan. Nalog može postati suspendovan i ako sistem detektuje neke bezbedonosne pretnje. Iz suspendovanog režima nalog može ponovo preći u aktivni režim aktiviranjem od strane administratora.

Ako korisnik pošalje zahtev za zatvaranjem naloga, nalog postaje deaktiviran. Zatvaranje se može zahtevati u svakoj životnoj fazi naloga. Prilikom zatvaranja, neophodno je platiti sva zaostala dugovanja, ako postoje. Modelovati:

- a) dijagram prelaza stanja za životni ciklus naloga,
- b) dijagram klasa za implementaciju rešenja pod a) (koristiti *state* šablon).
- c) dva dijagrama sekvence koji pokazuje sledeće: 1) uspešnu verifikaciju naloga koji je bio na čekanju i 2) zatvaranje naloga usled zahteva korisnika.

**3. zadatak.** Jednostavan kalkulator omogućava računanje kvadrata, korena i logaritma za osnovu 2 za višecifrene brojeve. Njegova tastatura se sastoji od cifara (0..9), znakova za operacije (x2 -kvadrat, sqrt - koren i log2 - logaritam za osnovu 2), kao i tastera za poništavanje rezultata prethodnog računanja. Posle uključivanja kalkulatora ili posle poništavanja rezultata, kalkulator se nalazi u inicijalnom stanju koje postavlja rezultat na nulu. Kalkulator u tom stanju čeka sledeći pritisak tastera.

Ako se pritisne znak za operaciju (x2, sqrt ili log2), kalkulator javlja grešku (zvučni signal) i vraća se u inicijalno stanje. Ako se pritisne cifra, kalkulator ulazi u stanje formiranja operanda (višecifrenog broja). Posle svakog pritiska cifre, operand se formira na sledeći način:

operand = prethodna vrednost operanda \* 10 + uneta cifra

a na ekranu se prikazuje vrednost tog operanda. Pritiskom na taster za operaciju, kalkulator prikazuje rezultat na ekranu i prelazi u inicijalno stanje. Pritiskom na taster za poništavanje, kalkulator prelazi u inicijalno stanje i prikazuje nulu na ekranu.

Modelovati:

- a) dijagram prelaza stanja,
- b) odgovarajući dijagram klasa za implementaciju rešenja pod a) (koristiti *state* projektni šablon)
- c) dijagram sekvence koji prikazuje proces unosa operanda i računanja rezultata.

**4. zadatak.** Modelovati aplikaciju za podršku rada softverske firme koja treba da omogući podršku za ažuriranje i pretragu podataka o softverskim proizvodima koje data firma implementira, kao i podataka o njenim zaposlenima.

Softverski proizvodi mogu biti programski moduli ili gotove aplikacije. Gotove aplikacije se sastoje od programskih modula. Programski moduli se mogu sastojati od drugih programskih modula. Svaki softverski proizvod ima svoj naziv, opis, datum kada je implementiran i zaposlenog koji je vodio njegov razvoj. O zaposlenom je potrebno voditi sledeće podatke: ime, prezime, broj telefona i vrstu posla koju obavlja. Moguće vrste posla su: projektant, programer i tester. Gotove aplikacije imaju cenu po kojoj se prodaju. Cene se mogu menjati, tako da je potrebno omogućiti uvid u sve cene koje važe ili su nekada važile.

Modelovati:

- a) konceptualni dijagram klasa,
  - b) dijagram sekvence koji pokazuje unos podataka o gotovim aplikacijama.
- Obratiti pažnju da se aplikacije sastavljaju od gotovih modula i da to treba da se vidi na dijagramu sekvence!

**5. zadatak.** Modelovati ponašanje zaštitnog alarma na vratima kuće. Alarm se ponaša na sledeći način. Ako je u stanju „zaključan“, a neko želi da uđe, potrebno je ukucati šifru. Ako je šifra korektna, vrata se otključavaju, a alarm prelazi u stanje „otključan“. Ako šifra nije korektna, moguće je pokušati njen unos još dva puta. Ako je sva tri puta šifra pogrešna, alarm prelazi u stanje uzbune. Ukucavanjem korektne šifre alarm prelazi iz stanja uzbune u stanje „otključan“. Ako detektuje pokušaj nasilnog ulaska, alarm takođe prelazi u stanje uzbune. Iz stanja „otključan“ u stanje „zaključan“ alarm prelazi pritiskom na odgovarajuće dugme.

Modelovati:

- a) dijagram prelaza stanja za opisani alarm,
- b) dijagram klasa za implementaciju dijagrama prelaza stanja alarma,
- c) dijagram sekvence koji pokazuje proces otključavanja alarma.

**6. zadatak.** Projektovati programski paket za podršku upravljanja nastavnim procesom u okviru fakulteta. Fakultet poseduje departmane a departmani katedre. Nastavnici su raspoređeni u okviru katedri na radno mesto: asistent, saradnik u nastavi, docent, vanredni ili redovni profesor. U okviru departmana, nastava se izvodi u studijskim grupama. Jedan departman može imati više studijskih grupa. Studijska grupa propisuje skup predmeta koji joj pripadaju. Jedan predmet može biti u okviru više studijskih grupa. Nastavnicima se dodeljuju predmeti koje izvođe, pri čemu jedan nastavnik može izvoditi više predmeta i jedan predmet može imati više pridruženih nastavnika. Ažuriranje svih navedenih podataka obavlja administrator aplikacije.

U okviru studijske grupe, student sluša nastavu iz nastavnih predmeta i izvršava dodatne obaveze (projekti, seminarski radovi, praksa i sl), pri čemu svaki predmet i dodatna obaveza nosi određeni broj poena. Da bi student stekao diplomu određenog stepena u sklopu studijske grupe on mora osvojiti minimalno definisani broj poena za tu studijsku grupu.

Nastavni predmeti u okviru studijskih grupa mogu biti izborni ili obavezni, pri čemu neki predmeti mogu biti slušani samo ako su ispunjeni preduslovi (položeni neki drugi predmeti koji im prethode ili odrađene neke dodatne obaveze).

Nastava na izbornom predmetu se u okviru određene školske godine može izvoditi jedino ako je predmet odabrao minimalno definisani broj studenata (minimalni broj studenata se propisuje na nivou svake studijske grupe). Potrebno je da student ima uvid u predmete koje može da bira (na osnovu pripadnosti studijskoj grupi i zadovoljenih uslova) i da nastavnik može da vidi listu studenata koji su izabrali svaki njegov predmet. Ako je nastavnik u svojstvu prodekana za nastavu, on treba da ima uvid i u listu predmeta koji nemaju dovoljan broj studenata da bi se izvodili u datoj školskoj godini.

Modelovati:

- a) dijagram slučajeva korišćenja za zadati programski paket,
- b) konceptualni dijagram klasa.

**7. zadatak.** Projektovati podsistem informacionog sistema banke. Banka može imati više filijala, a svaka filijala više poslovnica. Filijale i poslovnice se mogu nalaziti u različitim mestima. Za svaku filijalu i poslovnicu potrebno je minimalno voditi sledeće podatke: naziv, mesto, adresa, kontakt osoba (radnik banke). Podatke o filijalama, poslovnicama i radnicima unosi administrator. Radnici imaju sledeće podatke: ime, prezime, adresa.

Radnici banke rade sa klijentima banke. Svaki klijent može imati otvoren jedan ili više računa u okviru date banke. Otvaranje računa obavljaju radnici banke. Za svaki račun je neophodno da se zna: broj računa, datum otvaranja, trenutni iznos na računu i podaci o klijentu. Klijent ima sledeće podatke: ime, prezime, adresa. Takođe, potrebno je da za svaki račun postoje podaci o svim uplatama i isplatama. Uplata ima sledeće podatke: datum, iznos uplate, račun sa kojeg je neko uplatio dati iznos. Isplata ima sledeće podatke: datum, iznos isplate, račun na koji je uplaćen dati iznos.

Proces otvaranja računa odvija se na sledeći način. Ako klijent prvi put dolazi u banku, prvo se unose njegovi lični podaci, a zatim mu se otvara račun. Ako klijent

već ima račun u banci i želi da otvori još jedan, tada se vrši samo otvaranje još jednog računa.

Radnici banke vrše i gašenje računa. Prilikom gašenja, neophodno je zabeležiti sledeće podatke: datum gašenja računa, račun na koji se preneti sredstva sa ugašenog računa, radnika koji je obavio gašenje računa.

Modelovati:

- a) dijagram slučajeva korišćenja,
- b) konceptualni dijagram klasa.

**8. zadatak.** Grafički editor ima podršku za iscrtavanje jednostavnih i složenih grafičkih komponenti. Jednostavne komponente su: krug, kvadrat, pravougaonik i elipsa. Složene grafičke komponente se sastoje od jednostavnih grafičkih komponenti i prethodno kreiranih složenih grafičkih komponenti. Svaka komponenta (jednostavna ili složena) treba da poseduje:

- x i y koordinate gornjeg levog i donjeg desnog ugla pravougaonika u okviru kojeg se iscrta,
- mogućnost da se iscrta i
- mogućnost da odgovori da li se zadate koordinate (x, y) nalaze u okviru nje.

Modelovati:

- dijagram klasa za podršku rada opisanog grafičkog editora,
- dijagram objekata za dijagram klasa kreiran pod a),
- dijagram aktivnosti za iscrtavanje sadržaja editora.

**9. zadatak.** Potrebno je projektovati biblioteku grafičkih komponenti za realizaciju prozora grafičkih aplikacija. U okviru prozora se mogu naći komponente, poput linija za unos, dugmića, padajućih listi (*combo-box*) i sl.

Komponente se mogu grupisati u okviru jednog ili više panela ili tab-ova u okviru prozora. Sve komponente (jednostavne komponente, paneli i tab-ovi) imaju x i y koordinatu gornjeg levog ugla, širinu i visinu. Potrebno je da svaka komponenta ume sebe da iscrta. Komponente koje služe za smeštanje drugih komponenti (paneli, tab-ovi, prozor) se iscrstavaju tako što iscrstavaju pozadinu i zatim pozovu iscrtavanje komponenti koje se na njima nalaze.

Modelovati:

- a) dijagram klasa opisanih grafičkih komponenti,



b) dijagram aktivnosti za iscrtavanje prozora.

**10. zadatak.** Modelovati aplikaciju za podršku rada zdravstvene ustanove koja treba da pacijentima olakša zakazivanje pregleda. Medicinske sestre za svaki sledeći radni dan unose slobodne termine u kojima lekari mogu da prime pacijente. Termin za zakazivanje ima sledeće podatke: datum, vreme početka, trajanje, ordinacija, lekar koji radi u tom terminu. Ordinacija ima: broj ordinacije, broj sprata na kojem se nalazi i specijalnost za koju je namenjena. Svaka specijalnost ima šifru i naziv.

Pacijenti iz liste termina koje su sestre unele biraju onaj termin koji njima odgovara. Po obavljenom pregledu, ako se pacijent pojavio, lekar označava da je termin iskorišćen.

Lekari, medicinske sestre i pacijenti imaju sledeće podatke: ime, prezime, adresa, datum rođenja, pol (muški ili ženski). Lekar ima i specijalnost.

Pored lekara, medicinskih sestara i pacijenata, korisnik ove aplikacije je i administrator, koji ima pravo da unosi i menja podatke o lekarima, ordinacijama i specijalnostima. Svi korisnici imaju pravo da pretražuju podatke o lekarima, ordinacijama i specijalnostima. Svaki korisnik ima korisničko ime i lozinku. Lekar ima pravo da pretražuje podatke o pacijentima. Unos i ažuriranje podataka o pacijentima obavljaju medicinske sestre. Svaki korisnik ima korisničko ime i lozinku.

Lekar treba da ima mogućnost da pogleda izveštaj o svojim terminima koje su pacijenti izabrali, za zadati dan. Izveštaj treba da sadrži sledeće podatke: vreme početka, trajanje, pacijent koji je izabrao termin, ordinacija u kojoj se pregled odvija.

Modelovati:

- a) dijagram slučajeva korišćenja,
- b) konceptualni dijagram klasa,
- c) dijagram sekvence ili komunikacije za izbor termina od strane pacijenta.



## Literatura

- [Ambler02] S. W. Ambler, *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*, John Wiley & Sons, Inc. New York, 2002.
- [Ambler04] S. W. Ambler, *The Object Primer: Agile Model-Driven Development with UML 2.0*, Cambridge University Press, Cambridge, 2004.
- [Asaro99] P. M. Asaro, "Transforming Society by Transforming Technology: The Science and Politics of Participatory Design", University of Illinois, 1999.
- [Avison03] D. Avison, G. Fitzgerald, *Information Systems Development: Methodologies, Techniques and Tools*, Third Edition, McGraw-Hill Publishing, 2003.
- [Boehm03] B. Boehm, R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, Boston, 2003
- [Cockburn01] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2001.
- [Cockburn02] A. Cockburn, *Agile Software Development*, Addison-Wesley, Boston, 2002.
- [Curtis02] G. Curtis, D. Cobham, *Business Information Systems: Analysis, Design and Practice*, Pearson Education, Boston, 2002
- [CWM1.1] Common Warehouse Metamodel (CWM) Specification, version 1.1, mart 2003., <https://www.omg.org/cgi-bin/doc?formal/03-03-02.zip>
- [Fowler04] M. Fowler, *UML Distilled - A Brief Guide to the Standard Object Modeling Language*, Third Edition, Addison Wesley, Boston, 2004.
- [Fowler06] M. Fowler, "GUI Architectures", jun 2016, <https://www.martinfowler.com/ eaaDev/uiArchs.html>, jun 2016, pristupljeno: april 2020.

- [Gamma94] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994
- [IFML1.0] Interaction Flow Modeling Language (IFML) februar 2015. <https://www.omg.org/spec/IFML/1.0/PDF>
- [Markus83] M. L. Markus, "Power, Politics, and MIS Implementation", *Communications of the ACM*, June 1983, (26:6), pp. 430-444
- [Martin93] J. Martin, *Principles of object-oriented analysis and design*. Prentice-Hall, Inc., 1993.
- [Miles06] R. Miles, K. Hamilton, *Learning UML 2.0*. O'Reilly Media, Inc. 2006.
- [Milosav10] G. Milosavljević, Prilog metodama brzog razvoja adaptivnih poslovnih informacionih sistema, Doktorska disertacija, Fakultet tehničkih nauka, Novi Sad, 2010.
- [Pearce] J. Pearce, Domain Modeling, <http://www.cs.sjsu.edu/~pearce/modules/lectures/ooa/domain/>, pristupljeno: mart 2020.
- [Potel96] M. Potel, "MVP: Model-View-Presenter, The Taligent Programming Model for C++ and Java", Taligent Inc, 1996, [www.wildcrest.com/Potel/Portfolio/mvp.pdf](http://www.wildcrest.com/Potel/Portfolio/mvp.pdf), pristupljeno: april 2020.
- [Rumbaugh05] J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language Reference Manual*, Second edition, Addison-Wesley, Boston, 2005.
- [Schmidt06] D. C. Schmidt, "Model Driven Engineering", *Computer*, IEEE Computer Society, Februar 2006, pp. 25-31.
- [SWEBOK] Guide to the Software Engineering Body of Knowledge (SWEBOK) Version 3.0, Editors: P. Bourque, R. E. Fairley, IEEE Computer Society, 2014., <https://cs.fit.edu/~kgallagher/Schtick/Serious/SWEBOKv3.pdf>
- [UML2.5.1] OMG Unified Modeling Language (OMG UML), Version 2.5.1, December 2017, <https://www.omg.org/spec/UML/2.5.1/PDF>, pristupljeno: januar 2020.

# Sadržaj

Predgovor .....	i
1. Uvod.....	1
1.1 Izazovi u razvoju softvera .....	3
1.2 Vrste i namena modela.....	6
1.3 Modeli u fazama razvoja softvera .....	7
1.4 Inženjerstvo softvera vođeno modelima.....	8
1.5 Modelovanje u klasičnim i agilnim metodologijama .....	9
1.5.1 Agilno modelovanje .....	11
1.6 Šta smo naučili .....	13
2. Dijagram slučajeve korišćenja .....	15
2.1 Učesnici.....	19
2.1.1 Nasleđivanje učesnika .....	23
2.2 Slučajevi korišćenja.....	24
2.2.1 Primarni i sekundarni učesnici.....	33
2.3 Veze između slučajeva korišćenja .....	33
2.3.1 Veza proširivanja (extend) .....	34
2.3.2 Veza generalizacije (nasleđivanja) .....	37
2.3.3 Veza uključivanja ( <i>include</i> ) .....	41
2.4 Modelovanje kompleksnih slučajeva korišćenja .....	44
2.4.1 Standardizacija korisničkog interfejsa .....	47
2.4.2 Korišćenje stereotipa.....	49
2.4.3 Korišćenje drugih vrsta dijagrama .....	50
2.5 Šta smo naučili .....	50
3. Dijagram aktivnosti.....	53
3.1 Dekompozicija složene aktivnosti na pod-aktivnosti .....	54
3.2 Region mogućeg prekida .....	58
3.3 Particije .....	59
3.4 Signali.....	59

3.5 Paralelno izvršavanje .....	61
3.6 Prenos podataka .....	63
3.7 Skladišta podataka .....	64
3.8 Ekspanzioni region .....	65
3.9 Šta smo naučili .....	66
4. Dijagram klasa .....	71
4.1 Modelovanje klasa .....	76
4.2 Modelovanje obeležja .....	77
4.2.1 Vidljivost .....	78
4.2.2 Tip podataka .....	78
4.2.3 Kardinalitet .....	79
4.2.4 Podrazumevana vrednost .....	82
4.2.5 Statička obeležja .....	83
4.2.6 Dodatne opcije .....	84
4.2.7 Stereotipi .....	86
4.3 Modelovanje metoda .....	88
4.4 Veze .....	89
4.4.1 Veza asocijacije .....	89
4.4.2 Veza generalizacije .....	123
4.4.3 Implementacija interfejsa .....	135
4.4.4 Veza zavisnosti .....	136
4.5 Modelovanje korisničkog interfejsa .....	138
4.5.1 <i>Model-View-Controller</i> (MVC) arhitektonski šablon .....	138
4.5.2 MVC sa <i>Observer</i> šablonom .....	141
4.5.3 Primer primene MVC i <i>Observer</i> šablona .....	143
4.6 Šta smo naučili .....	152
5. Dijagram objekata .....	157
5.1 Šta smo naučili .....	162
6. Dijagram sekvenci .....	165

6.1 Uloge.....	166
6.2 Poruke.....	168
6.2.1 Sinhrona i asinhrona poruka .....	169
6.2.2 Poruka za kreiranje objekata .....	171
6.2.3 Poruka za oslobađanje objekata.....	172
6.2.4 Granica interakcije i spoljne poruke .....	173
6.2.5 „Izgubljene“ i „nađene“ poruke .....	175
6.3 Fragmenti interakcije.....	175
6.3.1 Petlja.....	176
6.3.2 Opcioni fragment.....	177
6.3.3 Prekidni fragment .....	178
6.3.4 Alternativni fragment .....	178
6.3.5 Referencirajući fragment.....	179
6.3.6 Paralelni fragment .....	180
6.4 Vremenske odrednice.....	180
6.5 Dijagram sekvence kao pomoć u projektovanju dijagrama klasa .....	181
6.5.1 Prijava na sistem.....	182
6.5.2 Zaključenje testa.....	183
6.5.3 Rešavanje testa.....	186
6.6 Šta smo naučili .....	187
7. Dijagram komunikacije .....	193
7.1 Određivanje rednog broja poruke .....	194
7.2 Specifikacija kontrole toka.....	196
7.3 Šta smo naučili .....	197
8. Dijagram prelaza stanja .....	199
8.1 Tranzicija .....	200
8.2 Stanje .....	202
8.2.1 Entry i exit akcije .....	203
8.2.2 Do akcija.....	203

8.2.3 Interne tranzicije.....	205
8.3 Paralelno izvršavanje .....	205
8.4 Prevođenje dijagrama prelaza stanja na programski kod .....	207
8.5 Šta smo naučili.....	219
9. Dijagram komponenti i dijagram raspoređivanja.....	221
9.1 Dijagram komponenti.....	221
9.2 Dijagram raspoređivanja .....	223
9.3 Šta smo naučili.....	223
9.4 Zadaci za vežbanje .....	224
Literatura .....	231