
OBJEKTNO ORIJENTIRANA ANALIZA I DIZAJN
primjenom UML notacije
Dženana Đonko
Samir Omanović

Objektno orijentirana analiza i dizajn primjenom UML notacije

Izdavač:
Elektrotehnički fakultet u Sarajevu
Za izdavača:
Red. prof.dr.sci. Kemo Sokolija, dipl. ing. el.

Recezenti:
Red. prof. dr. sci. Zlatko Lagumdžija, dipl. ing. el.
Doc. dr. sci. Fahrudin Oručevič, dip. ing. el.

Lektor:
Behija Vranić, lektor

Štamparija, GrafArt

Sarajevo, juni 2009

CIP-Katagolizacija u publikaciji
Nacionalna i univerzitetska biblioteka
Bosne i Hercegovine, Sarajevo

ISBN 978-9958-629-28-0
COBISS.BH-ID 17448966

Odlukom Senata Univerziteta u Sarajevu broj 01-38-1507/09 od 17.06.2009 data je saglasnost za izdavanje univerzitetske knjige Objektno orijentirana analiza i dizajn primjenom UML notacije autora doc. dr. sci. Dženana Đonko, dipl. ing. el, mr. Samir Omanović, dipl. el. ing.

UNIVERZITET U SARAJEVU

ELEKTROTEHNIČKI FAKULTET

Dženana Đonko

Samir Omanović

OBJEKTNO ORIJENTIRANA ANALIZA I DIZAJN

primjenom UML notacije

Sarajevo, 2009

SADRŽAJ

POGLAVLJE 1
OBJEKTNO ORIJENTIRANA ANALIZA I DIZAJN 1

1.1 Objekt 1

1.2 Objektno orijentirana strategija i programiranje 2

1.3 Objektno orijentirana analiza i dizajn 2

1.4 Osnovne karakteristike objektno orijentiranog pristupa 4

1.5 Objektno dizajnirani sistem 5

POGLAVLJE 2
UVOD U UML I RAZVOJNI PROCES 8

2.1 Historija UML-a 9

2.2 Osnovni UML model elementi 10

2.3 UML dijagrami 11

2.4 Proširenje UML-a 15

2.5 Priroda i namjena modela 17

2.6 UML i razvojni proces softvera 21

2.7 UML i metode razvoja softvera 23

POGLAVLJE 3
OPIS SISTEMA ZA MODELIRANJE 27

POGLAVLJE 4
DIJAGRAM SLUČAJEVA UPOTREBE 31

4.1 Namjena dijagrama slučajeva upotrebe 32

4.2 Elementi dijagrama slučajeva upotrebe 32

4.3 Akter 33

4.4 Slučaj upotrebe 35

4.5 Granice sistema 36

4.6 Relacije 38

4.7 Kreiranje dijagrama slučajeva upotrebe 41

4.8 Opis slučajeva upotrebe i scenarija 43

POGLAVLJE 5 49
DIJAGRAM KLASA: KLASE

5.1 Namjena dijagrama klasa 49

5.2 Klase u UML-u 50

5.3 Atributi 50

5.4 Kardinalnost 53

5.5 Izvedeni atributi 54

5.6 Operacije 55

5.7 Stereotip oznake operacija 57

5.8 Statičke operacije i atributi 57

5.9 Osobine 59

5.10 Model klase u fazi analize i dizajna 60

5.11 Terminologija za imenovanje klasa, atributa i operacija 64

POGLAVLJE 6 66
DIJAGRAM KLASA:VEZE IZMEĐU KLASA

6.1 Asocijacija 66

6.2 Ime asocijacije 66

6.3 Navigabilnost 67

6.4 Dvosmjerne asocijacije 68

6.5 Uloga asocijacije 68

6.6 Kardinalnost asocijacije 69

6.7 Glagolski oblik obilježavanja asocijacije 71

6.8 Atributi i asocijacija 72

6.9 Asocijacije opisane kvalifikatorom 74

6.10 Klasa asocijacija 75

6.11 Agregacija i kompozicija 78

6.12 Generalizacija 80

6.13 Višestruka generalizacija 82

6.14 Ovisnost 84

POGLAVLJE 7
DIJAGRAM KLASA: NAPREDNI KONCEPTI 87

7.1 Apstraktne klase 87

7.2 Interfejsi i apstraktne klase 90

7.3. Parametrizirane klase 93

7.4 Dodatna notacija na dijagramu klase 95

7.5 Dijagram klase za `e-restoran` 99

POGLAVLJE 8
DIJAGRAM OBJEKATA 102

8.1.Instance objekata 102

8.2 Anonimni objekt 103

8.3 Atributi objekta 103

8.4 Link 104

8.5 Linkovi i ograničenja 105

8.6. Kreiranje dijagrama objekata 107

POGLAVLJE 9
PRINCIPI I METRIKA OBJEKTNO ORIJENTIRANOG DIZAJNA 111

9.1 Kvalitet dizajna 111

9.2 Karakteristike lošeg dizajna 112

9.3 Principi dobrog dizajna 113

9.4 Metrika objektno orijentiranog dizajna 119

POGLAVLJE 10
DIJAGRAM AKTIVNOSTI 124

10.1. Namjena dijagrama aktivnosti 125

10.2 Aktivnost i akcija 125

10.3 Početni, krajnji čvor aktivnosti 126

10.4 Tokovi i ivice 127

10.5 Čvor odluke i čvor stapanja 128

10.6 Grananje i spajanje 130

10.7 Dijagram aktivnosti za `e-restoran` 130

10.8 Objekti u aktivnosti 132
10.9 Pozivanje drugih aktivnosti 134
10.10 Particije 135
10.11 Signali 137
10.12 Prekid aktivnosti 140
10.13 Oblasti primjene 141
10.14 Specifikacije spajanja 143
POGLAVLJE 11 145
DIJAGRAMI INTERAKCIJE – DIJAGRAM SEKVENCI
11.1 Dijagrami interakcije-uvod 145
11.2 Dijagram sekvenci 146
11.3 Učesnici u sekvencijalnom dijagramu 147
11.4 Vrijeme 148
11.5 Događaji , signali i poruke 149
11.6 Trake aktivnosti 150
11.7 Označavanje poruka 150
11.8 Tipovi poruka 151
11.9 Sinhrone poruka 151
11.10 Asinhrone poruke 153
11.11 Povratne poruke 154
11.12 Kreiranje i brisanja učesnika 154
11.13 Ugniježđene poruke 156
11.14 Refleksivne poruke 157
11.15 Povezivanje slučajeva upotrebe i dijagrama sekvence 158
11.16 Upravljanje kompleksnim interakcijama sa fragmentima sekvence 162
11.17 Fragmenti na `e-restoran` dijagramima sekvence 166
POGLAVLJE 12 169
DIJAGRAM KOMUNIKACIJE
12.1 Namjena dijagrama komunikacija 169
12.2 Elementi dijagrama komunikacije 171
12.3 Učesnici 171

12.4 Linkovi 171

12.5 Poruke 172

12.6 Prikaz interakcija sa dijagramom komunikacije 175

12.7 Usporedba dijagrama komunikacije i dijagrama sekvence 181

POGLAVLJE 13
DIJAGRAM TOKA VREMENA 185

13.1 Namjena dijagrama toka vremena 185

13.2 Elementi dijagrama toka vremena 185

13.3 Učesnici 186

13.4 Stanja 186

13.5 Vrijeme 187

13.6 Linija stanja učesnika 188

13.7 Događaji i poruke 189

13.8 Vremenska ograničenja 190

13.9 Organiziranje učesnika na dijagramu toka vremena 191

13.10 Dijagrama toka vremena na osnovu dijagrama sekvence za `e-restoran` 193

13.11 Alternativna notacija 199

POGLAVLJE 14
DIJAGRAM PREGLEDA INTERAKCIJA 203

14.1 Dijelovi dijagrama pregleda interakcija 203

14.2 Dijagram pregleda interakcija za `e-restoran` 206

POGLAVLJE 15
DIJAGRAM STANJA 209

15.1 Namjena dijagrama stanja 210

15.2 Stanje i pseudostanje 211

15.3 Prijelaz 212

15.4 Klasa i stanja 214

15.5 Interno ponašanje 215

15.6 Kompozitna stanja i regioni 217

15.7 Napredna pseudostanja 218

15.8 Signali 219

15.9 Stanje historije 220

POGLAVLJE 16
DIJAGRAM SLOŽENE STRUKTURE 222

16.1 Namjena dijagrama složene strukture 222

16.2 Interna struktura 223

16.3 Osobine 224

16.4 Konektori 225

16.5 Port 225

POGLAVLJE 17
DIJAGRAM KOMPONENTI 228

17.1 Namjena dijagrama komponenti 228

17.2 Komponenta 229

17.3 Komponenta u UML-u 230

17.4 Ponuđeni i zahtijevani interfejsi komponenti 231

17.5 Prikazivanje komponenti koje rade zajedno 233

17.6 Klase koje realiziraju komponente 235

17.7 Portovi i interna struktura 237

17.8 Konektori 238

17.9 Pogled na komponente: crna kutija i bijela kutija 240

POGLAVLJE 18
DIJAGRAM PAKETA 244

18.1 Namjena dijagrama paketa 245

18.2 Paketi 245

18.3 Sadržaj paketa 246

18.4 Organizacija klasa u pakete 248

18.5 Vidljivost elemenata 250

18.6 Ovisnost paketa 251

18.7 Uvoz paketa i pristup elementima paketa 252

18.8 Implementacija ovisnosti paketa 254

18.9 Upravljanje ovisnostima paketa 255

18.10 Paketi i slučajevi upotrebe 256

POGLAVLJE 19 260
DIJAGRAM RASPOREĐIVANJA

19.1 Namjena dijagrama raspoređivanja 260

19.2 Čvor 261

19.3 Komunikacijski putevi 261

19.4 Instance čvora 261

19.5 Čvorovi hardvera i čvorovi izvršnog okruženja 262

19.6 Raspoređivanje softvera 263

19.7 Raspoređivanje artefakta na čvor 265

19.8 Specifikacije raspoređivanja 266

19.9 Dijagram raspoređivanja kroz faze projekta 268

POGLAVLJE 20 271
MEHANIZMI PROŠIRENJA UML-a

20.1 Pregled standardnih proširenja 271

20.2 Profajl 272

20.3 Stereotip 272

20.4 Označene vrijednosti 276

20.5 Ograničenja 278

20.6 Kreiranje i korištenje profile 278

20.7 OCL jezik za izražavanje ograničenja 281

POGLAVLJE 21 291
DIZAJN PATERNI

21.1 Historija dizajn paterna 292

21.2 Format opisivanja dizajn paterna 293

21.3 Proxy patern 294

21.4 Predstavljanje paterna UML notacijom 295

21.5 Prikazivanje paterna na dijagramima 302

21.6 Kako koristiti paterne 305

LITERATURA 308

PRILOG – `e-restoran` UML dijagrami 310

INDEX 319

Predgovor

Osnovna namjena ove knjige je da omogući studentima da savladaju koncepte i principe objektno orijentirane analize i dizajna primjenom UML notacije. UML je skraćenica od Unified Modelling Language i predstavlja jezik za modeliranje kompleksnih sistema. U knjizi je predstavljana osnovna UML 2.0 specifikacija. Da bi se postigao cilj da se savlada UML specifikacija ali i uloga UML-a u procesu razvoja softvera naročito u fazama analize i dizajna svi koncepti su objašnjeni na sistemu `e-restoran`. `E-restoran` je sistem koji je lako razumjeti a ipak ima dovoljno funkcionalnih zahtjeva da bi se praktično ilustrirali svi UML koncepti kroz sve faze razvoja projekta. Danas postoji jako puno alata za modeliranje pomoću UML-a. U ovoj knjizi se koristio StarUML alat otvorenog koda, koji pored podrške za većinu UML dijagrama nudi i mogućnost automatskog generiranja koda. To je omogućilo da se kroz knjigu mapiraju koncepti modela u programski kod veoma popularnih objektno orijentiranih programskih jezika Java i C++.

Struktura

Knjiga se sastoji od 21 poglavlja. Prvo poglavlje daje osnove vezane za objektno orijentirani pristup razvoju programskih rješenja. Knjiga je namijenjena čitaocima koji imaju određena predznanja vezana za objektno orijentirano programiranje pa neki koncepti objektno orijentiranog pristupa koji se obrađuju i u sklopu objektno orijentiranog programiranja nisu detaljno objašnjavani. Drugo poglavlje sadrži osnovne elemente UML-a ali i ulogu UML-a u procesu modeliranja i aktivnostima razvoja softvera. U tom poglavlju su dati i osnovni pristupi za razvoj softvera. Poslije toga slijede poglavlja koja predstavljaju namjenu i osnovnu notaciju UML dijagrama. U svakom od tih poglavlja se i praktično modelira `e-restoran`. Pored poglavlja za dijagrame važno je napomenuti da u poglavlju 9 su dati osnovni principi dobrog dizajna. S obzirom da je UML jezik za modeliranje sistema koji pripadaju različitim domenama i koji se razvijaju na različitim platformama poglavlje 20 objašnjava osnovne mehanizme proširenja UML-a. Iako nije sastavni dio UML u okviru zadnjeg 21 poglavlja je dat u uvod u dizajn paterne i uloga UML-a za opis i dokumentiranje paterna bez kojih bi proces razvoja softvera trajao puno duže.

Konvencija

S obzirom da je većina literature koja se koriste za ovu tematiku na engleskom jeziku, svi važni pojmovi pri svom prvom navođenju su dati i na engleskom jeziku. Neki od termina koji se nisu mogli adekvatno prevesti i koji su postali široko upotrjebljeni u našem jeziku su kroz knjigu pisani fonetski.

U tekstu je korišten **naglašen** font za nove termine koje imaju specijalno značenje u UML-u u tačci gdje su uvedeni, a *kosim* slovima se prikazivala sintaksa. Za model `e-restoran`a je korišten specijalni font `Courirer New`**.**

Autori

POGLAVLJE 1.

OBJEKTNO ORIJENTIRANA ANALIZA I DIZAJN

Razvoj dobrog i održivog softvera, koristeći objektne tehnologije i programske jezike, kao što su Java, C++, predstavlja vještinu u objektno orijentiranoj analizi i dizajnu. Poznavanje objektno orijentiranih jezika je neophodno, ali nije i dovoljno da bi se kreirao jedan objektni sistem. Važnu ulogu u tome ima način kako razmišljati objektno orijentirano. Naravno, kreiranje nekog softvera pored programiranja i objektno orijentirane analize i dizajna zahtijeva mnogo više vještina kao što su i kreiranja dobrog korisničkog interfejsa i kreiranje baze podataka koje nisu dio ove knjige. Ovdje je predstavljen samo jedan dio, koji igra važnu ulogu u dosta kompleksnijem problemu samog razvoja softvera, a to je objektno orijentirana analiza i dizajn.

Unified Modeling Language (UML), koji se detaljno prikazuje u ovoj knjizi, predstavlja standardnu notaciju za kreiranje dijagrama i prikazivanje raznih pogleda na sistem. Koliko god je korisno savladati samu notaciju ono što je mnogo kritičnije, a već je spomenuto, je kako razmišljati objektno i kako dizajnirati objektno orijentirani sistem. UML nije objektno orijentirana analiza i dizajn niti metoda, to je jednostavno samo notacija. Tako da nije od velike pomoći znati samu sintaksu UML-a ili možda UML kao alat, a ne biti u mogućnosti kreirati uspješan dizajn što je dosta zahtjevnija i vrednija vještina.

1.1 Objekt

Objekt je stvar, entitet, imenica, sve što se može zamisliti, a da ima svoj identitet. Primjeri iz stvarnog života su osoba, automobil, kuća, pas, drvo i slično. Svi ovi objekti imaju svoje atribute kao što na primjer automobil ima svog proizvođača, broj modela, boju, cijenu, pas

ima svoju vrstu, boju, godište i slično. Također pored atributa objekti imaju i ponašanje odnosno operacije koje mogu izvršavati kao, na primjer, automobil se kreće od jednog mjesta do drugog, psi laju.

Kod objektno orijentiranog softvera, objekti iz realnog svijeta se pretvaraju u programski kôd. Oni postaju moduli sa svojim podacima i procesima koje izvršavaju. Ako na primjer u objektno orijentiranom programiranju imamo objekt osoba, tada taj objekt bi trebao da zna svoje ime, prezime, adresu, datum rođenja, a da je također u mogućnosti i da promijeni ime, promijeni adresu, kaže koliko je star i slično. Treba naglasiti da ne treba u potpunosti simulirati realni svijet jer bi to bilo veoma teško, već objekti iz realnog svijeta trebaju uticati na softverske objekte radi lakšeg shvaćanja i lakšeg kreiranja tih objekata. Nije potrebno kreirati idealan objekt, jer previše je osobina i mogućnosti u realnom svijetu da bi se sve obuhvatile i nikad ne bi došlo na red kreiranje nekog drugog objekta u sistemu, sva bi se pažnja usmjerila ka tom „idealnom“ objektu. Tipični programi obično su usmjereni na određeno područje i problem, tako da neke od karakteristika objekata iz realnog svijeta su suvišne. Ako je u pitanju bankovni sistem, onda je vjerovatno vrednija informacija godište i plaća korisnika, a ne broj njegovih cipela ili boja kose.

Veliki dio softverskog razvoja uključuje kreiranje modela prije nego suhoparno pisanje kôda. Model predstavlja domenu problema ili prijedloga rješenja koji nam dopušta razmišljanje i shvaćanje stvarnog problema. Model nam dopušta da naučimo dosta o problemu.

1.2 Objektno orijentirana strategija i programiranje

Značajna slabost objektno orijentirane strategije u prošlosti je bila ta da se velika pažnja posvećivala klasama i objektima, a vrlo malo se radilo na prikazivanju ponašanja tih objekata i njihovih funkcionalnosti. Problem je u tome što su klase, entiteti vrlo niskog nivoa i ne opisuju u najboljoj mjeri ono što zapravo cijeli sistem radi. Međutim, moderni pristup koji je podržan UML-om omogućava da se klase i objekti ostave po strani u ranim fazama projekta i da se pažnja usmjeri ka tome što sistem treba da radi. A onda, nakon što projekt počne napredovati klase se postepeno dodaju u sistem da bi se shvatile potrebne funkcionalnosti sistema.

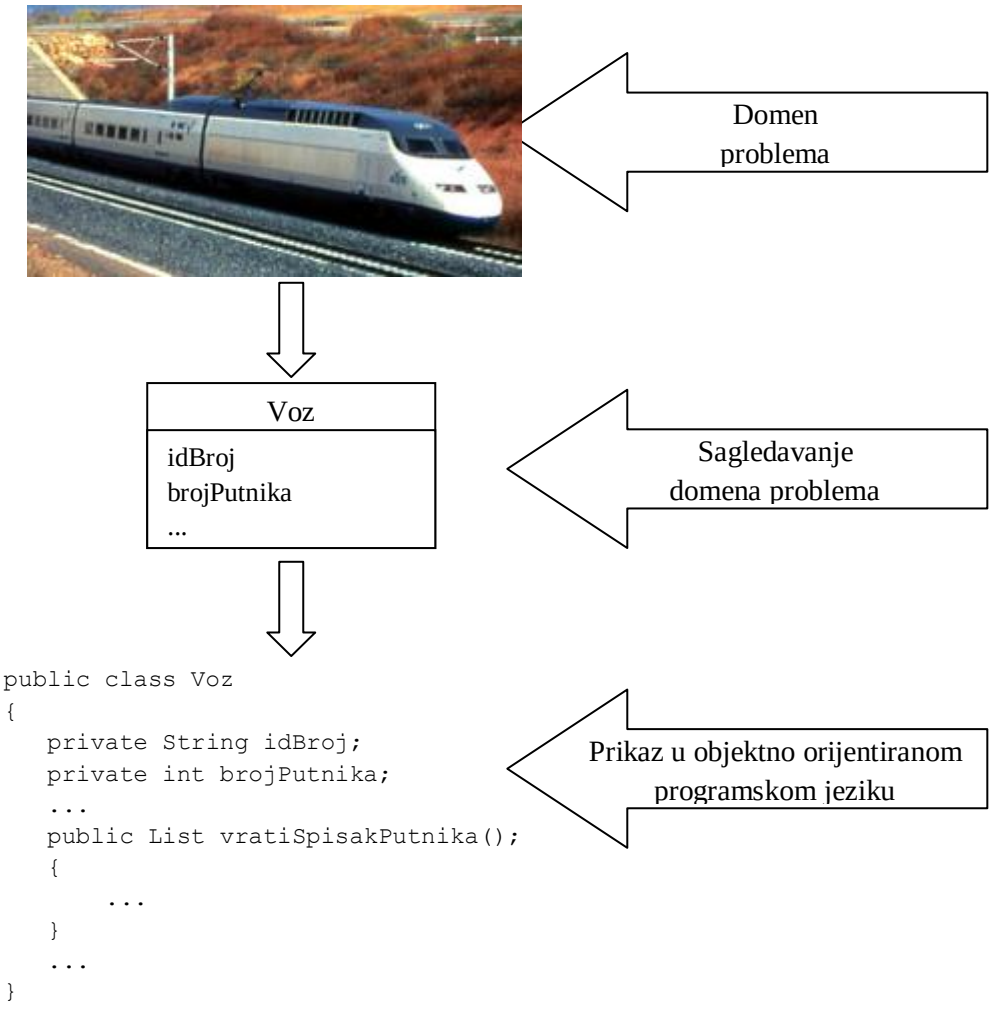
Što se tiče objektno orijentiranog programiranja, ono radi na način kreiranja objekata, njihovog povezivanja i stvaranja kolaboracije slanjem međusobnih poruka.

1.3 Objektno orijentirana analiza i dizajn

Prije samog rješavanja problema prvo treba definirati problem. Analiza se uopćeno bavi istraživanjem problema i zahtjeva, a ne bavi se samim rješenjem. Ako analiza znači definiranje problema, onda dizajn je proces definiranja rješenja problema. Dizajn se temelji na konceptualnom rješenju koje ispunjava zahtjeve, a ne na implementaciji rješenja.

U fazi objektno orijentirane analize teži se pronalaženju i opisivanju objekata odnosno koncepata problema. To su objekti iz realnog svijeta koji pripadaju domeni problema. Objektno orijentirani dizajn je proces definiranja tih komponenti, objekata, interfejsa, klasa, atributa i operacija koje će zadovoljiti zahtjevima problema. Na kraju svi ti objekti koji su definirani u fazi dizajna se implementiraju kroz neki od objektno orijentiranih programskih jezika u fazi implementacije.

Objektno orijentirana analiza ima naglasak na pronalaženju i opisivanju objekata ili koncepata u domenu problema. Na primjer u slučaju informacionog sistema željeznica, neki od koncepata su voz, vožnja, putnik. Objektno orijentirani dizajn ima naglasak na definiranju softverskih objekata i njihove kolaboracije s ciljem ispunjavanja postavljenih zahtjeva. Tako softverski objekt voz može imati atribute `idBroj` (identifikacijski broj) i `brojPutnika` i operaciju (metodu) vrati `SpisakPutnika` što je ilustrirano na slici 1.1.



Slika 1.1: Predstavljanje objekata

Na kraju, tokom implementacije ili objektno orijentiranog programiranja, dizajnirani objekti se implementiraju u programski kôd slično kao prikazana klasa voz u programskom jeziku Java.

Rezultat objektno orijentirane analize i dizajna je objektno orijentirani model koji je osnova za razvoj objektno orijentiranog sistema.

1.4 Osnovne karakteristike objektno orijentiranog pristupa

Osnovne karakteristike objektno orijentiranog pristupa su apstrakcija, enkapsulacija, modularnost i hijerarhija.

Apstrakcija

Apstrakcija je jedan od fundamentalnih principa pomoću kojeg se ljudi bore sa kompleksnošću. Suština apstrakcije je u uočavanju osnovnih karakteristika objekta i njegovom razlikovanju od ostalih objekata na temelju uočenih osobina. Ona se fokusira na vanjski izgled objekta, zanemarujući potpuno njegovu unutrašnjost odnosno implementaciju.

U objektno orijentiranim programskim jezicima, svakoj apstrakciji iz domena problema odgovara po jedna klasa. Klase su prototip iz koga se kreiraju primjerci (instance) objekata. Objekti u programskom jeziku su primjerci klasa, i do njih se dolazi instanciranjem na osnovu definirane klase.

Enkapsulacija

Enkapsulacija je koncept potpuno komplementaran apstrakciji. Apstrakcija se fokusira na zapažanje osobina i ponašanja objekata, a enkapsulacija se fokusira na implementaciju koja će dovesti do željenog ponašanja. Programski jezici omogućavaju jednostavne mehanizme za podršku enkapsulaciji. U programskim jezicima, implementacija obuhvata kreiranje identificiranih klasa u postupku apstrakcije. Svaka klasa ima svoje podatke-članove i svoje funkcije-članice, koji se jednim imenom nazivaju članovi klase. Sama enkapsulacija se realizira razdvajanjem javnih i privatnih dijelova klase. Javni dijelovi klase, obično su to funkcije članice, koje čine interfejs te klase ka okolini. Privatni dijelovi, uglavnom podaci koji izražavaju stanje objekta, obuhvaćaju detalje implementacije koji za korisnika klase nisu važni i stoga on za njih ne mora da zna.

Modularnost

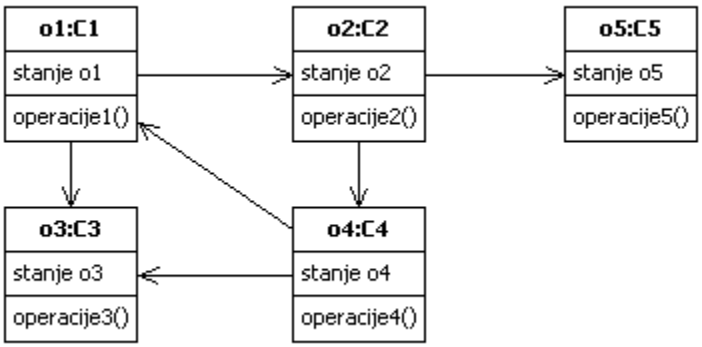
Modularnost je koncept povezan isključivo sa samim programom, domena problema više ne igra nikakvu ulogu, a sastoji se u formiranju modula koji mogu da se prevode odvojeno, ali koji ipak imaju dobro definirane veze prema drugim modulima. U tradicionalnom strukturiranom dizajnu, modularnost programa se uglavnom sastojala od grupiranja potprograma dok je u objektnom dizajnu problem malo drugačiji. Ovdje je, prvo potrebno uočiti logičke cjeline koje čine izvjestan broj klasa i onda formirati module. Razni jezici se vrlo različito odnose prema modulima. Moduli C++-a nisu ništa drugo do programski moduli koji se odvojeno prevode, a u Javi to su paketi.

Hijerarhija

I na kraju, postavlja se pitanje što raditi kada je broj apstrakcija identificiranih u domenu problema toliki da je njima teško upravljati. Za to je potrebno novo logičko grupiranje apstrakcija - hijerarhija. Hijerarhija klasa se u objektnim modelima i objektno orijentiranim programskim jezicima ostvaruje preko mehanizama nasljeđivanja uz koji vežemo i polimorfizam, a sastoji se u tome da jedna apstrakcija može biti vrsta neke druge apstrakcije sa dodatnim osobinama. Na primjer, ako smo u domenu problema identificirali apstrakcije koje ćemo nazvati "prijevozno sredstvo", "voz" i "avion", onda o druge dvije apstrakcije možemo razmišljati kao o specifičnim vrstama prve. Opravdanje za to nalazimo u činjenici da avion i voz jesu prijevozna sredstva. Klase koje odgovaraju ovim apstrakcijama povezane su na taj način što su klase koje odgovaraju avionu i vozu izvedene iz klase koja odgovara prijevoznom sredstvu, i na taj način nasljeđuju kompletnu funkcionalnost te klase i dodaju, eventualno, neke svoje karakteristike koje ih čine autonomnim entitetima.

1.5 Objektno dizajnirani sistem

Konačni rezultat objektno orijentirane analize i dizajna je objektno dizajnirani sistem. On se sastoji od objekata. Svaki objekt pripada nekoj klasi. Objekti su u međusobnoj interakciji pozivom operacija. Prilikom poziva operacija, preko kojih jedni drugim nude servise, objekti razmjenjuju podatke (parametre i rezultate). U objektno dizajniranom sistemu funkcionalnost sistema je izražena isključivo u terminima operacija vezanih uz objekte, pri čemu se više funkcionalnosti može izvršavati paralelno.



Slika 1.2: Sistem sastavljen od objekata u međusobnoj interakciji

Novi objektno dizajnirani sistemi mogu biti razvijeni uz upotrebu klasa koje su bile realizirane za neke prethodne sisteme. To smanjuje cijenu razvoja softvera i vodi ka upotrebi

standardnih objekata. Za ponovnu upotrebu pogodnije su veće cjeline od pojedinih klasa, a to su komponente.

Važno svojstvo objektno dizajniranog sistema je da on omogućuje laganu evoluciju. Naime, sistem se sastoji od objekata, dakle dijelova s jakom unutrašnjom kohezijom i labavim vezama prema okolini**.** Evolucija sistema zahtijeva promjenu unutrašnje strukture pojedinog objekta ili dodavanje novih objekata, a takvi zahvati nemaju značajnog učinka na ostatak sistema.

Jedna od glavnih prednosti objektno orijentiranih sistema je da komponente mogu biti lako prilagođene novoj namjeni. To se najčešće ostvaruje korištenjem nasljeđivanja. Mehanizam prilagođavanja se ne oslanja na mijenjanje postojećih komponenata, već se kreiraju nove komponente koje nasljeđuju atribute i operacije od originalne komponente. Mijenjaju se samo oni atributi i operacije koje stvarno trebaju promjenu.

Bez obzira na postojanje mnogih metoda i pristupa razvoju softvera često se nameće pitanje:

Kako zaista napisati dobar softver ?

Ako se gleda iz korisničkog ugla, tada bi mogli reći: Dobar softver mora zadovoljiti korisnika. Softver mora raditi ono što korisnik želi da on radi.

Međutim, razviti softver koji radi ono što treba je dobro, ali što u trenutku kada je potrebno dodati novi kôd, ili ga iskoristiti u drugoj aplikaciji? Nije dovoljno imati samo softver koji radi što klijent hoće, naš softver mora izdržati i test vremena.

Sa stanovišta razvojnog tima dobar softver je dobro dizajniran, lagan za održavanje, ponovno korištenje i proširenje. Da bi to postigli potrebno je primijeniti objektno orijentirane principe da bi se povećala fleksibilnost razvoja, eliminirao dupli kôd, da bi softver bio jednostavan za održavanje i ponovno korištenje. Primijeniti paterne i principe da bi bili sigurni da je softver spreman za korištenje i sljedećih godina.

Očito je da dizajn softvera bitno utječe na njegov kvalitet. O aspektima dizajna će biti detaljnije govoreno u većini poglavlja koji se odnose na UML dijagrame, a posvećeno je i jedno cjelokupno deveto poglavlje. U poglavljima 5, 6, 7 koji se odnose na dijagram klase i u

osmom poglavlju koji se odnosi na dijagram objekata, objasnit će se i detaljnije pojam klase kao i moguće veze između klasa, pojam objekta i veza između objekata i klasa.

Namjena ove knjige je da se svi koncepti vezani za razvoj objektno orijentiranih sistema kao i za sve faze objektno orijentiranog razvoja objasne u vezi sa jezikom za modeliranje sistema UML-om, pa će sljedeće poglavlje uvesti osnovnu namjenu i elemente UML-a, te njegovu vezu sa objektno orijentiranom analizom i dizajnom i pristupima razvoju softvera.

POGLAVLJE 2.

Uvod u UML i razvojni proces

Unified Modeling Language, skraćeno UML je grafički jezik za vizualizaciju, specificiranje, konstruiranje i dokumentiranje sistema programske podrške prema definiciji OMG (Object Managment Group) grupe.

UML je razvijen sa ciljem da pojednostavi veliki broj objektno orijentiranih razvojnih metoda. UML uključuje semantičke koncepte, notaciju i direktive. UML notacija ne definira

standardni proces ali mu je namjena da bude koristan sa iterativnim razvojnim procesima iako se u posljednje vrijeme koristi i sa naprednijim metodama razvoja softvera kao što je agilna metoda.

Postoji više ciljeva za razvoj korištenjem UML-a. Prvi i najvažniji, UML je jezik za modeliranje opće namjene kojeg koriste mnogi timove prilikom modeliranja sistema.

UML je prvenstveno namijenjen za modeliranje sistema koji se oslanjaju na programsku podršku. Veoma se efikasno koristi za poslovne i informacijske sisteme, ali UML je dovoljno izražajan i za modeliranje ostalih sistema, kao što su na primjer tok podataka i dokumenata u pravnom sistemu, struktura i ponašanje zdravstvenog sistema pacijenta, dizajn sklopova.

UML obuhvaća informacije o statičkoj strukturi i dinamičkim ponašanjem sistema. Sistem se modelira kao kolekcija diskretnih objekata koji su u interakciji. Statičke strukture definiraju vrste objekata važne za sistem i njihovu implementaciju, isto kao i veze između objekata. Dinamičko ponašanje definira historiju objekata u vremenu i komunikaciju između objekata.

Modeliranje sistema sa različitih, ali povezanih pogleda omogućava da sistem bude razumljiv za različite namjene.

Važno je uočiti da UML nije programski jezik. Neki alati omogućavaju generiranje kôda iz UML-a u aktualne objektno orijentirane jezike, isto kao i reverzni inženjering iz postojećih programa u UML dijagrame.

2.1 Historija UML-a

UML je razvijen kao pokušaj pojednostavljenja i konsolidacije velikog broja objektno orijentiranih metoda razvoja koje su našle svoju primjenu u praksi uslijed nagle popularizacije objektno orijentiranih programskih jezika. Iako se prvim objektno orijentiranim jezikom smatra Simula-67 koji je nastao 1967 godine, ekspanzija objektno orijentiranih metoda za razvoj softvera se desila u osamdesetim godinama prošlog vijeka, nakon pojave jezika kao što su Smalltalk i, nešto kasnije, C++. Neke od objektno

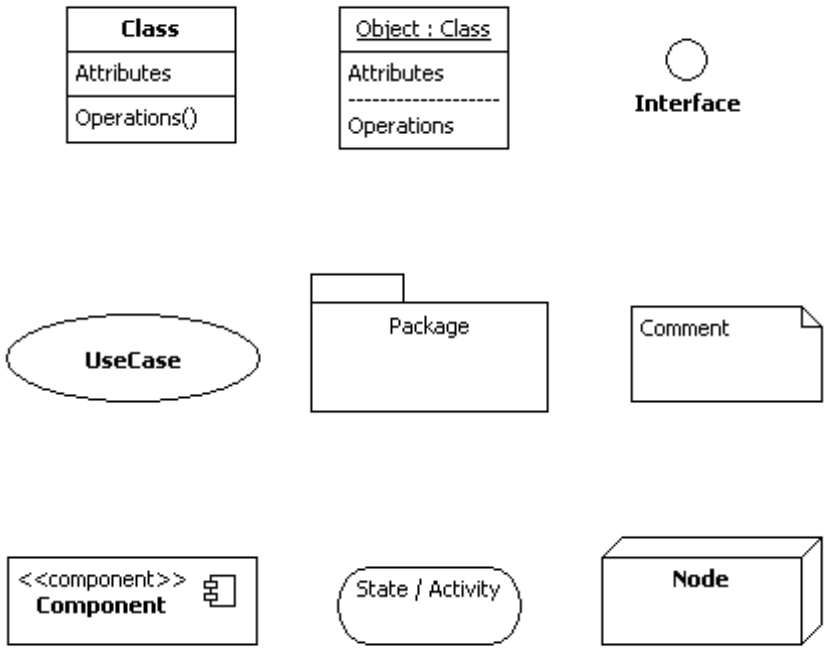
orijentiranih metoda koje su postale jako popularne u ranim devedesetim godinama bile su: Booch metoda (autor Grady Booch), OMT – Object Modeling Technique (James Rambaugh), OOSE/Objectory (Ivar Jacobson), Fusion (D. Coleman) i Coad/Yourdan. Svaki od navedenih autora je predvodio grupu praktičara kojima su se dopadale njegove ideje. Međutim, najveći problem je bio to što su metode bilo veoma slične, a ipak su obuhvaćale izvjestan broj uznemiravajućih razlika. Svaka od metoda je imala vlastitu notaciju, procese i CASE alate (CASE – Computer Aided Software Systems). Tokom tog perioda istovremeno se govorilo i o standardizaciji, ali bez nekog prevelikog uspjeha.

Ključni detalj u stvaranju UML-a se desio kada je Jim Rambaugh napustio kompaniju General Electric i pridružio se Grady Boochu u kompaniji Rational, koja je sada dio IBM-a. Grady i Jim su najavili kraj tzv. «rata metoda» i vrlo brzo pripremili prvi javni opis svog objedinjenog razvojnog postupka: to je bila verzija 0.8 dokumenta UM (Unified Method). Još značajnije je bilo to što su 1995. godine objavili da je kompanija Rational Software kupila firmu Objectory i da će se Ivar Jacobson pridružiti timu.

Proizvođači raznih CASE alata, uplašeni da će standard kojim upravlja Rational pružiti alatima ove firme nepravednu prednost na tržištu, pokrenuli su aktivnosti standardizacije metode, i kao rezultat toga uspostavljena je OMG grupa (*Object Management Group* – OMG). Nakon ovoga, uslijedili su brojni pregovori između velikog broja kompanija, i konačno, u novembru 1997. godine, UML je zvanično postao standard čijim daljnjim razvojem upravlja spomenuta OMG grupa. Uglavnom se smatra da su Grady Booch, Ivar Jacobson i Jim Rambaugh tvorci jezika UML. Iako se oni smatraju najzaslužnijim za razvoj ovog jezika, treba napomenuti da su zaista veliki dio posla uradili, i još uvijek rade, timovi grupe OMG. Nakon više godina iskustva u korištenju UML-a, OMG je izdao prijedlog za njegovu nadogradnju, za fiksiranje problema koji su se otkrili i za njegovim proširenjem dodatnim kapacitetima kako bi bio primjenjiv u više aplikacijskih domena. Zahtjevi po prijedlogu su se razvijali od novembra 2000. do jula 2003., sa specifikacijom UML verzije 2.0 koja je i danas aktualna.

2.2 Osnovni UML model elementi

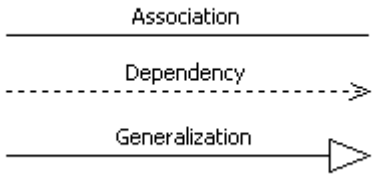
Osnovni model element u okviru UML-a je definiran sa semantikom, formalnom definicijom elementa ili egzaktnim značenjem. Model element ima odgovarajući grafički element, koji je vizualna predstava tog elementa. Primjeri nekih model elemenata su klase, objekti, stanja, paketi, komponente, komentari koji su predstavljeni na slici 2.1 sa odgovarajućim grafičkim simbolima.



Slika 2.1: Neki od simbola UML notacije

Svi ovi elementi će se detaljno objasniti u nastavku knjige.

Slika 2.2 pokazuje primjere veza, koji su također model elementi i koriste se za spajanje elemenata modela. Neke relacije su: ovisnost (eng. dependency) koja pokazuje da jedan element ovisi od drugog elementa, asocijacija (eng. association) koja spaja elemente, generalizacija koja prikazuje specijalizaciju jednog elementa na osnovu drugog. Ostale relacije će biti objašnjenje u kontekstu svog pojavljivanja.

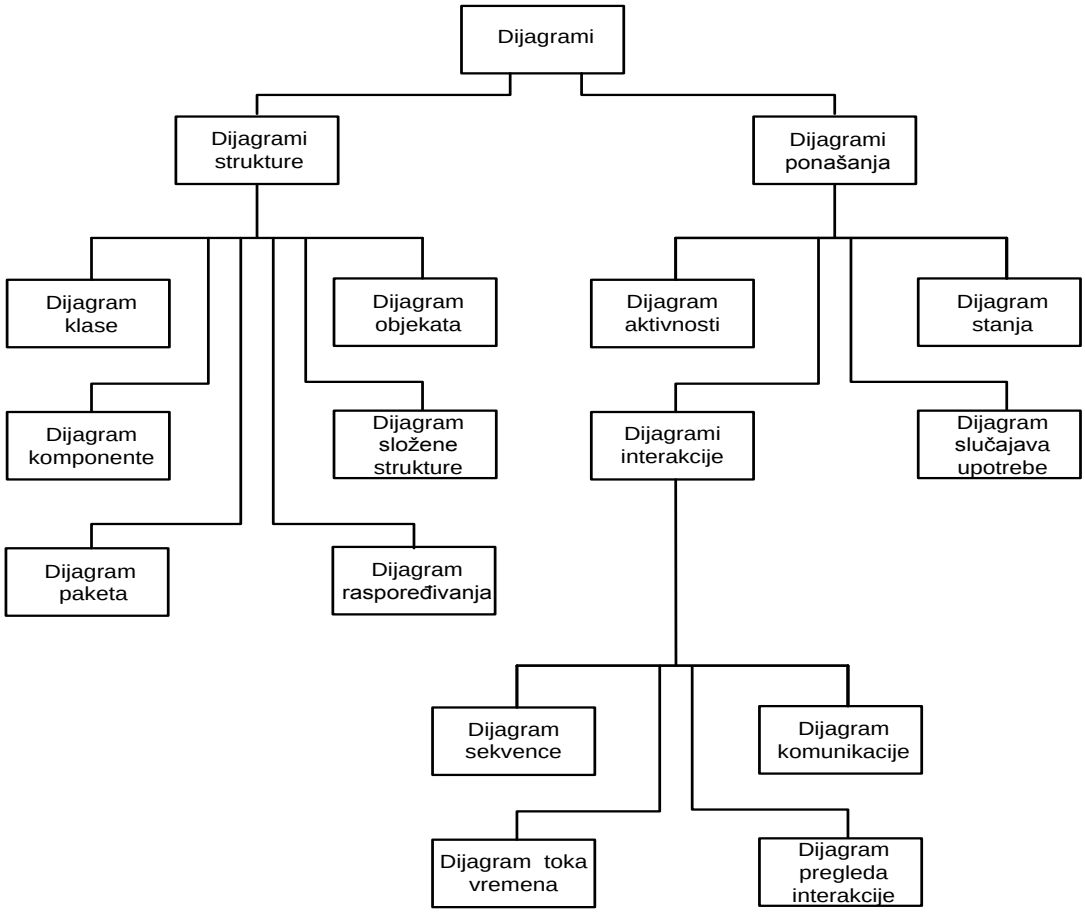


Slika 2.2: Primjeri relacija

2.3 UML dijagrami

Dijagram je grafička prezentacija skupa model elemenata. Model elementi navedeni u prethodnoj sekciji mogu postojati u više različitih tipova dijagrama, ali postoje pravila koji elementi se mogu koristiti u kojem tipu dijagrama.

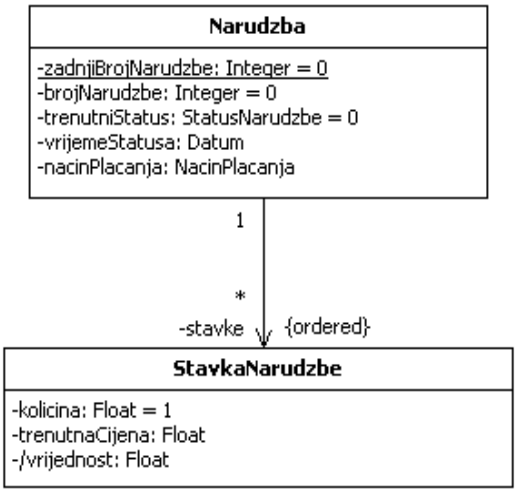
Dijagrami se crtaju kako bi se vizualizirao sistem iz različitih perspektiva odnosno pogleda. Slijedi shema koja slikovito daje pregled 13 postojećih tipova dijagrama klasificiranih u grupu dijagrama koji prikazuju strukturu sistema i grupu dijagrama koji pokazuje ponašanje sistema.



Slika 2.3: Dijagrami UML-a

Slijedi kratki uvod što predstavlja koji dijagram, neki od njih će biti prikazani sa pripadajućom slikom da bi stekli osnovni utisak što predstavlja dijagram. Svakom od dijagrama je posvećeno jedno poglavlje knjige koje između ostalog opisuje namjenu dijagrama, osnovne elemente dijagrama, primjenu dijagrama.

Dijagrami klasa (eng. class diagram) prikazuju logičku organizaciju klasa koje postoje u sistemu i njihove međusobne veze.



Slika 2.4: Dijagram klase

Dijagrami komponenti (eng. component diagram) modeliraju softverske komponente koje postoje u sistemu sa njihovim međusobnim elementima.

Dijagrami složene strukture (eng. composite structure diagram) služe za dekompoziciju složenih klasa u hijerarhijske strukture, čime se dobiva na jednostavnosti modela. Ovaj tip dijagrama je uveden sa UML 2 verzijom.

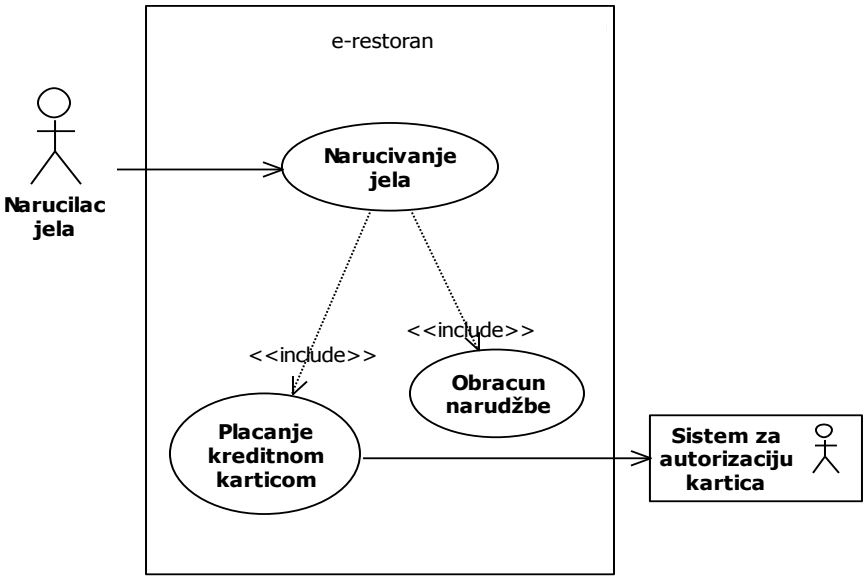
Dijagrami raspoređivanja (eng. deployment diagram) prikazuju računarsku infrastrukturu sa naznakom na kojim njenim elementima se smještaju komponente sistema.

Dijagrami objekata (eng. object diagram) su veoma slični dijagramima klasa, s tim da oni prikazuju konkretne objekte ili instance klasa i njihove relacije.

Dijagrami paketa (eng. package diagram) služe za logičko grupiranje klasa i komponenata modela i prikazivanje veza između tih grupa. Ovi dijagrami su posebno bitni za velike projekte koji mogu uključivati hiljade pa čak i desetine hiljada klasa.

Dijagrami aktivnosti (eng. activity diagram) prikazuju tok funkcionalnosti u sistemu. Ovi dijagrami definiraju gdje počinje tok procesa, gdje se završava, koje se aktivnosti u njemu odvijaju i u kom redoslijedu.

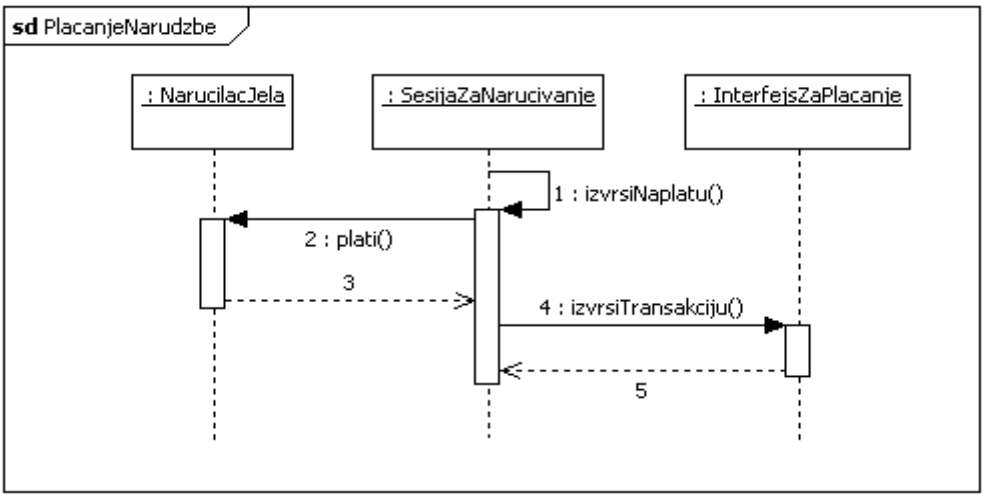
Dijagrami slučajeva upotrebe (eng. use case) predstavljaju najbolji način za prikazivanje interakcije između korisnika i sistema.



Slika 2.5: Dijagram slučaja upotrebe

Dijagrami interakcije (eng. interaction overview diagram) uključuju dijagrame sekvence, komunikacije, dijagrame toka vremena i dijagrame pregleda interakcije.

Dijagrami sekvenci (eng. sequence diagram) prikazuju interakcije između objekata u odnosu na njihov redoslijed izvršavanja.



Slika 2.6: Dijagram sekvence

Dijagrami komunikacije (eng. communication diagram) su u prvoj, zvaničnoj verziji UML-a uvedeni pod imenom dijagrami kolaboracije. Ovi dijagrami prikazuju interakciju između objekata ali sa naglaskom na veze, odnosno poruke koje se razmjenjuju između objekata.

Dijagrami pregleda interakcije (eng. interaction overview diagram) su predstavljeni u UML 2.0 verziji standarda i predstavljaju svojevrsnu mješavinu dijagrama interakcije i dijagrama aktivnosti koja u određenim slučajevima može naći svoju praktičnu primjenu.

Dijagrami toka vremena (eng. timing diagram) su još jedan tip dijagrama uvedenih u UML 2 verziji i prikazuju vremenska ograničenja između promjena stanja različitih objekata. Dijagrami su veoma korisni za projektiranje u elektronici, konstrukciji hardvera kao i u sistemima realnog vremena.

Dijagrami stanja (eng. state machine diagram) prikazuju stanja objekta, odnosno kako događaji mijenjaju objekte tokom vremena. Za razliku od dijagrama klasa koji prikazuju statičku sliku sistema, klasa i relacija, dijagrami stanja se koriste za modeliranje dinamike sistema.

Pored osnovnih model elemenata na dijagramima se mogu naći i razna proširenja, što predstavlja izuzetno važnu osobinu za primjenu UML u raznim domenskim problemima. Više o mehanizmima proširenja UML-a bit će u poglavlju 20., a u ovom poglavlju su samo osnove vezane za elemente i principe proširenja standardne UML notacije.

2.4 Proširenje UML-a

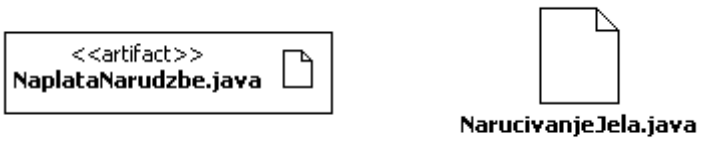
UML može biti proširen ili adaptiran za specifičnu namjenu. Postoje tri tipa mehanizama: stereotipi, označene vrijednosti (eng. tagged values) i ograničenja (eng. constraint).

Stereotipi

Stereotip je mehanizam proširenja koji definira novu vrstu model elementa bazirajući se na postojeći element. Stereotip elementa se može koristiti u istim situacijama u kojima se koristi i originalni element. Stereotip se može bazirati na svim tipovima elemenata kao što su klase, čvorovi, komponente, paketi, ali i na relacijama kao što su asocijacija, generalizacija i ovisnost. Postoji već predefinirani stereotipi u UML-u i oni se trebaju koristiti i prilagođavati

kada je to moguće bez uvođenja novih stereotip elemenata. Ova strategija čini osnovni UML jednostavnijim. Stereotip ili skup stereotipa na elementu se opisuje naziv stereotipa unutar << >> kao <<StereotipIme>> ili ako je više stereotipa tada <<StereotipIme1, StereotipIme2>>.

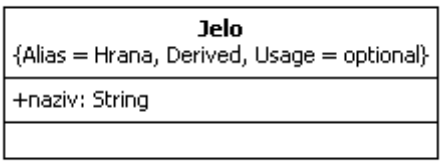
Stereotip može imati i svoju grafičku predstavu u obliku ikone. Element specifičnog stereotipa može biti prikazan jednom od varijanti ili njihovom kombinacijom što je vidljivo na slici 2.7.



Slika 2.7: Stereotipi

Označene vrijednosti

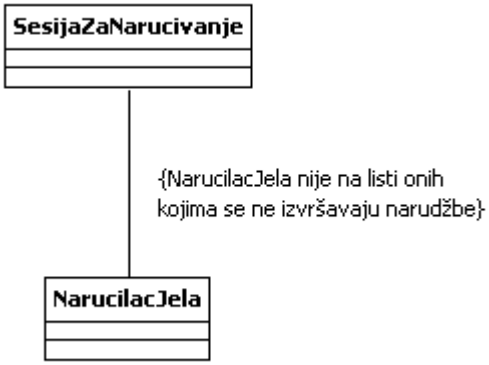
Element može imati osobine koje sadrže par ime-vrijednost. Ove osobine se nazivaju i označene vrijednosti. Jedan broj osobina postoji predefiniran u UML-u, ali korisnici mogu definirati dodatne osobine koje mogu predstavljati između mnogih ostalih informacija, informacije o nekim specifičnostima metode, administrativne informacije, informacije potrebne alatima za generiranje kôda. Obilježavaju se unutar vitičastih zagrada što je vidljivo na slici 2.8 unutar klase.



Slika 2.8: Označene vrijednosti

Ograničenja

Ograničenje je restrikcija na element koja limitira upotrebu elementa ili daju dodatnu semantiku (značenje) elementu. Jedan primjer ograničenja je dat na slici 2.9 na asocijativnoj vezi u okviru vitičastih zagrada.



Slika 2.9: Ograničenja

Za UML možemo reći da je pomalo kompleksan i proširiv. Ali ne mora se znati ili koristiti svaka njegova karakteristika da bi se modelirao neki sistem. Postoji mali skup osnovnih koncepata koji su široko rasprostranjeni. Ostale karakteristike se mogu učiti postepeno i koristiti kada je potrebno a najviše u ovisnosti od domena problema i složenosti samog sistema.

Svi gore navedeni dijagrami sa osnovnim elementima i proširenjima koriste se da bi se dobio model dobro dizajniranog objektno orijentiranog sistema, pa u nastavku slijedi diskusija o prirodi i namjeni modela.

2.5 Priroda i namjena modela

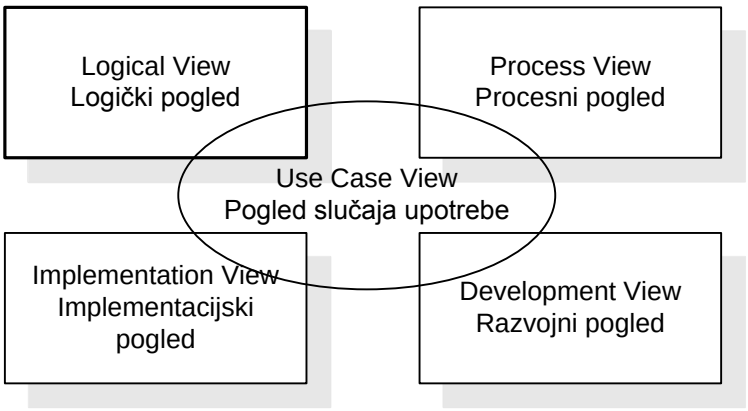
Zašto uopće modeliramo? Postoji prije svega, jedan fundamentalni razlog: gradimo modele da bismo bolje razumjeli sistem kojeg razvijamo. Modeli nam pomažu da vizualiziramo sistem kakav jeste, ili kakvog ga želimo, dozvoljavaju nam da specificiramo strukturu ili ponašanje sistema, daju nam šablone u konstrukciji sistema, i dokumentiraju odluke koje smo napravili.

U praksi, čak i u najmanjim projektima, vrši se neka vrsta modeliranja, uglavnom vrlo neformalno. Međutim, ovi neformalni modeli uglavnom ne osiguravaju zajednički jezik koji se jednostavno može dijeliti sa drugima. Kako postoje zajednički jezici za inženjere elektronike, matematsko modeliranje, građevinsko projektiranje. Jasno je da je jedan takav zajednički jezik i za modeliranje softverskih sistema bio neophodan. Modeliranje ima bogatu historiju u svim inženjerskim disciplinama. Iz ovih iskustava izdvajaju se četiri bazna principa modeliranja:

- Izbor modela ima veliki utjecaj na rješavanje problema. Iz ovoga proističe da modele trebamo birati jako pažljivo.
- Svaki model može se predstaviti na različitim nivoima preciznosti.
- Najbolji modeli su povezani sa stvarnošću. Svi modeli pojednostavljuju realnost; treba biti siguran da to pojednostavljenje ne sakriva značajne detalje.
- Ne postoji model ili pogled koji je dovoljan. Da bi opisali strukturu i ponašanje sistema, trebamo prikazati čitav skup modela i pogleda na sistem sa različitih perspektiva.

Potrebno je dosta truda da se modelira složen sistem. Idealan slučaj bi bio da možemo predstaviti čitav sistem jasno sa jednom slikom da ga svi mogu razumjeti. Međutim ovaj idealan slučaj je nemoguć, samo trivijalni sistemi mogu postići takav cilj. Niko ne može generirati jedan dijagram koji definira nedvosmisleno čitav sistem i koji je jasan svima koji ga gledaju. Jedan graf ne može obuhvatiti sve informacije potrebne da se opiše sistem. Sistem ima mnogo različitih aspekata, ima svoje statičko i dinamičko ponašanje. Opis sistema zahtijeva više pogleda, gdje svaki pogled predstavlja projekciju kompletnog sistema koji prikazuje određeni aspekt. Svaki pogled zahtijeva nekoliko dijagrama koji sadrže informaciju ističući određeni aspekt sistema. Određeno preklapanje postoji, tako da jedan dijagram može biti dio više različitih pogleda. Analizom sistema sa različitih gledišta, moguće se u određenom vremenu koncentrirati na jedan aspekt sistema. Dijagram u određenom pogledu treba da bude dovoljno jednostavan da može jasno prenijeti informaciju, i još uvijek usklađen sa ostalim dijagramima i pogledima tako da je kompletna slika sistema opisana sa svim pogledima zajedno. Svaki dijagram sadrži grafičke simbole koji predstavljaju elemente modela sistema.

Jedan od pogleda na sistem je i Krutchten 4+1 model koji se dosta koristi u praksi. Shema pogleda je data na slici i sastoji se od pogleda slučaja upotrebe, logičkog pogleda, implementacijskog pogleda, procesnog pogleda i razvojnog pogleda. Svi ostali pogledi koriste pogled na slučajeve upotrebe. Zbog toga se ovaj cjelokupni model i naziva 4+1 model.



Slika 2.10: Philippe Krutchten 4+1 pogled na sistem

Pogled slučaja upotrebe

Pogled slučaja upotrebe opisuje funkcionalnost koji bi trebao dati sistem, kako je uočeno od eksternih aktera. Akter koji je u interakciji sa sistemom može biti korisnik ili drugi sistem. Pogled slučaja upotrebe se koristi od strane klijenta, dizajnera, razvojnih inženjera, testera. On je opisan dijagramima slučajeva upotrebe, često sa podrškom od strane dijagrama aktivnosti. Željena upotreba sistema je opisana sa više slučajeva upotrebe, gdje je pojedinačan slučaj upotrebe generalni opis zahtijevane funkcije.

Pogled slučaja upotrebe je centralan, jer njegov sadržaj rukovodi razvoj ostalih pogleda. Konačan cilj sistema je da osigura funkcionalnost opisanu u ovom pogledu. Zbog toga, ovaj pogled utiče na sve ostale poglede. Ovaj pogled se također koristi da se potvrdi sistem i konačno da se verificira funkcionalnost sistema testiranjem pogleda slučaja upotrebe sa klijentom pitajući ga „Da li je to što želite?“, i „Da li sistem radi ono što je specificirano?“.

Logički pogled

Logički pogled opisuje kako je omogućena funkcionalnost sistema. Prvenstveno je namijenjen dizajnerima i razvojnim inženjerima. Za razliku od pogleda slučaja upotrebe, logički pogled gleda unutar sistema. On opisuje obadvoje, statičku strukturu (klase, objekte i relacije) i dinamičku kolaboraciju koja se dešava kad objekti pošalju poruku jedni drugima da bi ispunili neku funkcionalnost. Karakteristike kao što su dosljednost i konkurentnost su također definirane, kao i interfejsi i interne strukture klase. Statička struktura je prvenstveno opisana sa dijagramom klase i dijagramom objekata. Dinamičko modeliranje je opisano najčešće dijagramima stanja, dijagramima interakcije i dijagramima aktivnosti.

Implementacijski pogled

Implementacijski model opisuje glavne modele i njihove ovisnosti. Namijenjen je prvenstveno razvojnim inženjerima i sastoji se od glavnih softverskih artefakta. Artefakti uključuju različite tipove modula kôda prikazanih sa njihovom strukturom i ovisnostima. Dodatne informacije o komponentama kao što su alokacija resursa ili druge administrativne informacije, kao što su izvještaj o progresu razvoja, također mogu biti dodani. Implementacijski pogled najčešće zahtijeva prošireni prikaz na odgovarajuće izvršno okruženje.

Procesni pogled

Procesni pogled se bavi dijeljenjem sistema u razne procese. Ovaj aspekt, koji je nefunkcionalna karakteristika sistema, dozvoljava efikasnu upotrebu resursa, paralelno izvršavanje, i rukovanje asinhronim događajima iz okruženja. Pored dijeljenja sistema u konkurentne izvršne niti, ovaj pogled se mora također baviti komunikacijom i sinhronizacijom ovih niti.

Naglasak na pogledu koji pokazuje konkurentnost osigurava kritične informacije članovima razvojnog tima i integratorima sistema. Pogled se uglavnom sastoji od dijagrama dinamičke strukture (stanja, interakcije, aktivnosti). Dijagrami toka vremena također nude specijalnu notaciju za procesni pogled.

Razvojni pogled

Razvojni pogled opisuje kako su dijelovi sistema organizirani u module i komponente. Veoma su korisni za upravljanje nivoima unutar arhitekture sistema. Ovaj pogled tipično sadrži dijagrame paketa i komponenti.

Napomenimo da i drugi pogledi mogu biti korišteni, kao što su statičko-dinamički, logičko-fizički. Moderni UML alati uključuju i mogućnost definiranja vlastitih pogleda i vezanih različitih modela unutar istog sistema ili projekta. Sa evolucijom UML-a i njegovom rastućom upotrebom duž svih vrsta procesa, fokus se pomjerio od pokušaja da se obuhvati čitav sistem unutar samo jednog pogleda modela. MDA (Model Driven Arichtecture) unapređuje notaciju različitih nivoa modela koji mogu koristiti odgovarajuće ekstenzije paketa, ili profajl (eng. profile), koji odgovaraju namjeri modela. Na višem nivou sistem može imati generalni model koji reflektira domenske ili poslovne koncepte klijenta. Ovaj neovisni model koji povezuje jedan ili više specijalnih modela je bolji pri implementaciji UML modela u odgovarajuće ciljno okruženje. Međutim, da zadržimo ove modele da ne izađu izvan kontrole važno je da se modelima upravlja tako da oni mogu da implementiraju glavnu funkcionalnost sistema.

2.6 UML i razvojni proces softvera

Iako postoji veliki broj pristupa razvoja softverskih sistema, o njima ćemo u nastavku, svi ti pristupi imaju isti skup aktivnosti koje se odnose na razvoj softver a to su:

1. Određivanje zahtjeva
2. Analiza
3. Dizajn sistema
4. Implementacija
5. Testiranje

Određivanje zahtjeva

Inženjering zahtjeva ili određivanje softverskih zahtjeva je aktivnost određivanja što se treba proizvesti u softverskom sistemu, koja je svrha i namjena sistema. Ova aktivnost se smatra

jednom od najvažnijih u izgradnji softverskog sistema. U ovoj fazi se koristi UML dijagram slučajeva upotrebe da bi se analizirali i prikupili zahtjevi korisnika. Dodatne detalje o sistemu mogu se opisati koristeći dijagrame ponašanja kao što su dijagrami stanja, dijagrami aktivnosti u kombinaciji sa dijagramima strukture kao što je dijagram klase na visokom nivou.

Važno je zapamtiti da u ovoj fazi treba koristiti što jednostavnije dijagrame i ne treba uvoditi ništa što je specifično za realizaciju softvera. UML dijagrami trebaju biti dovoljno jasni korisnicima tako da ne treba koristiti detalje i specifičnosti jezika koji su im nejasni.

Analiza

Analiza se bavi primarnim apstrakcijama (klasama i objektima) i mehanizmima koji su aktualni u domenu problema. Model identificiranih klasa za domenu problema, zajedno sa međusobnim relacijama, opisuje se sa UML dijagramima klase. Kolaboracija između klasa koja je neophodna za izvršavanje slučajeva upotrebe također se opisuje u ovoj fazi, korištenjem nekog od UML dijagrama dinamičke strukture. U analizi jedino klase koje su domeni problema se modeliraju, ne modeliraju se tehničke klase koje definiraju detalje o softverskom sistemu kao što su klase za korisnički interfejs, baze podataka, komunikacija, konkurentnost.

Dizajn

U fazi dizajna rezultat analize se proširuje na tehničko rješenje. Dodaju se nove klase da bi se opisala i tehnička infrastruktura: korisnički interfejs, baze podataka, komunikacija sa drugim sistemima i slično. Klase domena problema iz analize su ugrađene u tehničku infrastrukturu čineći mogućim da se mijenja obadvoje i domena problema i infrastruktura. Dizajn rezultira detaljnom specifikacijom koja je potrebna za fazu implementacije.

Implementacija

U implementaciji klase iz dizajn faze se konvertiraju u objektno orijentirani programski jezik. Kada se kreira model analize i dizajna pomoću UML-a, bolje je izbjegavati brzi direktni prijevod modela u kôd. Modeli su namijenjeni razumijevanju strukture sistema tako da rani

zaključci o kôdu mogu biti kontraproduktivni za kreiranje jednostavnog i korektnog modela. Programiranje je odvojena aktivnost, za vrijeme koje se modeli konvertiraju u kôd.

Testiranje

Sistem se normalno izvršava sa testovima programskih modula, integracijskim testovima, sistem testovima i testovima prihvatljivosti.

Različiti timovi za testiranje koriste različite UML dijagrame kao osnovu za svoj rad: testovi programskih modula zasnivaju se na dijagramima klase i specifikacijama, integracijski testovi koriste dijagrame komponenti i dijagrame komunikacije, sistem testovi koriste implementaciju za dijagrame slučajeva upotrebe i dijagrame aktivnosti da bi verificirali ponašanje koje je inicijalno definirano sa ovim dijagramima.

Svi navedeni dijagrami se mogu koristiti i da bi se dokumentirao napisani softver.

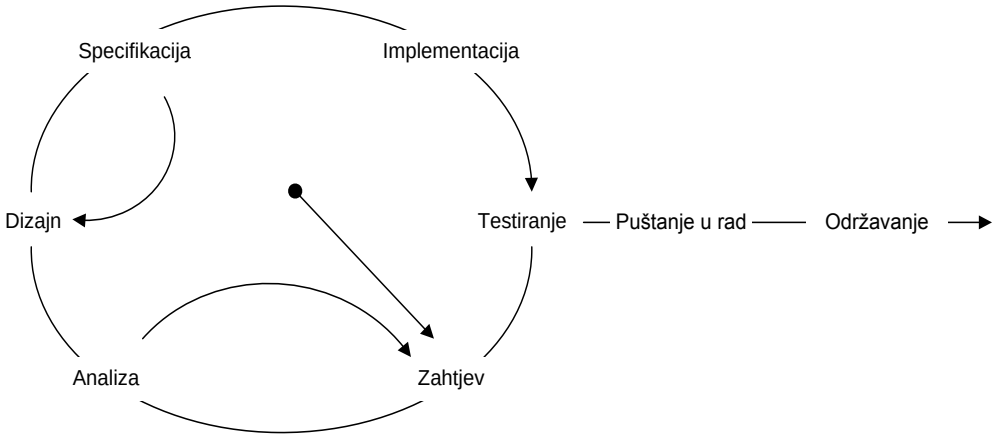
2.7 UML i metode razvoja softvera

UML je nastao iz mnoštva metoda objektno orijentirane analize i projektiranja. Sve one su, u izvjesnom obimu, obuhvaćale korištenje jezika za grafičko modeliranje i propisivale postupak razvoja softvera. Zanimljivo je da su različite firme, kada je nastao UML, otkrile da su se mogle složiti oko jezika za modeliranje, ali uglavnom nisu mogle da usuglase proces razvoja softvera. Na kraju je postignut sporazum o ograničavanju standarda samo na jezik za modeliranja.
Ipak, UML se ne može razmatrati izvan razvojnog procesa. Jedan od popularnijih modela sa kojima se UML koristi je Rational Unified Process – RUP, koja će i biti obrađena u nastavku, ali UML se može koristiti u bilo kojem drugom razvojnom procesu, među kojima su tradicionalni vodopadni model, spiralni model, iterativni model, a u posljednje vrijeme sve više i agilni model razvoja softvera.

Unified Process (UP)

Najpoznatiji primjer iterativnog modela razvoja softvera je *Rational Unified Process* (RUP). RUP je također razvijen od istih autora koji su razvili i UML, tako da je RUP veoma

komplementaran sa UML-om. U suštini, *Rational* se odnosi na to da je svaki projekt različit i sa različitim potrebama. Jedinstveni proces je zasnovan na komponentama. Softverski sistem je izgrađen od softverskih komponenti koje su između sebe povezane odgovarajućim interfejsom. Njegova glavna karakteristika je iterativni razvoj u smislu da je organiziran u serije kratkih i vremenski ograničenih projekata (iteracija), tako da je rezultat svake iteracije testirani, integrirani i izvršivi dio sistema. RUP je dosta prilagodljiv i dopušta da se svaka faza projekta može mijenjati i prilagođavati daljim potrebama sistema. Iterativan ciklus je zasnovan na stalnom uvećavanju i usavršavanju sistema kroz svaku iteraciju, te se zato ovaj pristup zove iterativno-inkrementalan razvoj softvera.



Slika 2.11: Iterativni model razvoja

Pored iterativnog razvoja, centralna ideja UP-a je korištenje objektno orijentiranih tehnologija, uključujući objektno orijentiranu analizu i dizajn i objektno orijentirano programiranje.

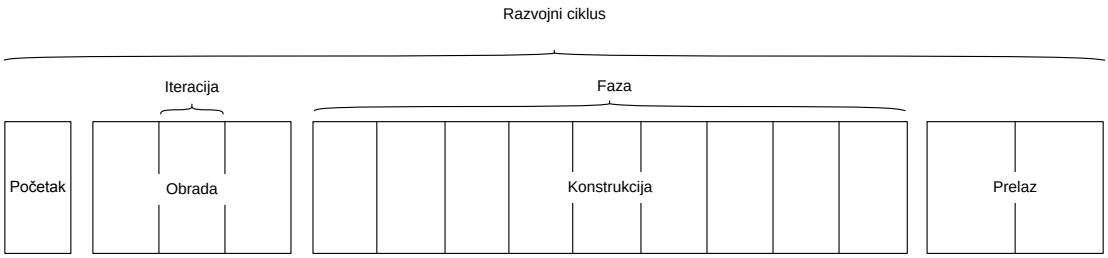
Neke od dodatnih ideja i koncepata UP-a su:

- Najvažnije probleme kao probleme sa visokim rizikom rješavati u ranim iteracijama.
- Konstantno uključivati samog korisnika u razvoj sistema.
- Često testirati i provjeravati kvaliteta dijelova sistema.
- Primjenjivati slučajeve upotrebe.
- Grafički predstaviti model softvera pomoću UML-a.
- Po potrebi izvršavati promjene zahtjeva i konfiguracije.

Sa aspekta vremenske dimenzije UP je organiziran kroz četiri glavne faze:

1. Početak (*Inception*) – definira se opseg i vizija projekta, kao i studija koja je slična studiji izvodljivosti projekta. Navode se osnovni slučajevi upotrebe.
2. Obrada (*Elaboration*) – obuhvata poboljšanje vizije, razvoj plana projekta, razradu slučajeva upotrebe, nacrt arhitekture sistema kao i prve iteracije i njene implementacije.
3. Konstrukcija (*Construction*) – stvara produkte.
4. Tranzicija (*Transition*) – Prenosi produkte u korisničko okruženje. U ovoj fazi se također vrši i obuka korisnika.

Ovo nije klasični vodopadni pristup ili sekvencijalni životni ciklus gdje se prvo definiraju svi zahtjevi, pa tek nakon toga se vrši dizajn. Početna faza ne predstavlja samo fazu zahtjeva nego je to više faza izvodljivosti, gdje se nakon dovoljno istraživanja može vidjeti da li se neke odluke trebaju podržati. To je faza gdje se definira opseg projekta. Tako isto faza obrade nije samo faza dizajna već faza u kojoj se planira projekt, specificiraju osobine i koja daje osnove računarske arhitekture. Na slici 2.12 je prikazan jedan razvojni ciklus koji je sastavljen od većeg broja iteracija.



Slika 2.12: Razvojni ciklus projekta

U ovom poglavlju smo uveli osnovne pojmove vezane za UML, osnove softver inženjeringa u domeni discipline razvoja i samih metodologija razvoja. Mi želimo sve to detaljnije primijeniti, odnosno želimo detaljnije upoznati UML notaciju, njegovu primjenu u analizi i

dizajnu, želimo kreirati model koji će omogućiti razvoj nekog softverskog sistema. Da bi ostvarili taj cilj bazirat ćemo se na jednom primjeru (case study) koji će nam pomoći da na konkretnom slučaju izučimo sve teorijske aspekte. Taj primjer se prikazuje u sljedećem poglavlju i koristi se u okviru svih poglavlja ove knjige.

Pitanja za ponavljanje

1. Koja je namjena UML-a?
2. Ko su autori UML-a?
3. Koja organizacija je odgovorna za UML standard?
4. Navedite osnovne UML model elemente.
5. Navedite osnovne relacije između UML elemenata.
6. Što je UML dijagram?
7. Navedite jednu od klasifikacija UML dijagrama.
8. Koji UML dijagrami opisuju strukturu sistema?
9. Koji UML dijagrami opisuju ponašanje sistema?
10. Navedite tri mehanizma proširenja UML-a za specijalne namjene.
11. Navedite osnovnu namjenu modeliranja.
12. Koji pogledi su bitni prilikom modeliranja sistema?
13. Što se opisuje sa pogledom na slučajeve upotrebe?
14. Što se opisuje sa logičkim pogledom na sistem?
15. Što se opisuje sa procesnim pogledom na sistem?
16. Što se opisuje sa implementacijskim pogledom na sistem?
17. Što se opisuje sa razvojnim pogledom na sistem?
18. Objasniti vezu pogleda slučajeva upotrebe i ostalih pogleda?
19. Koje su osnovne aktivnosti prilikom razvoja softvera?
20. Koja je uloga UML-a u okviru faza razvojnog procesa softvera?
21. Koja je veza između UML-a i metoda razvoja softvera?
22. Sa kojom metodom razvoja softvera se preporučuje korištenje UML-a?

POGLAVLJE 3.

Opis sistema za modeliranje

Primjer sistema, koji će biti upotrijebljen za praktično pokazivanje objektno orijentirane analize i dizajna primjenom UML notacije, naziva se `e-restoran`. Kao i kod svakog softverskog projekta, i u primjeru `e-restorana` prvo je neophodno sagledati sistem i prikupiti relevantne informacije o tome kako taj sistem treba da izgleda poslije implementacije. Ovaj korak obično nazivamo specifikacija zahtjeva. Specifikacija zahtjeva obuhvaća informacije o postojećem stanju, ali i informacije o njegovim nedostacima iz perspektive korisnika sistema, odnosno menadžera. Sistem se gradi ili unapređuje s ciljem da se putem toga realizira neki poslovni cilj, odnosno da sistem koji se gradi bude poslovno koristan.

Zahtjeve možemo tekstualno opisati, a kasnije će biti riječi o tome kako to formalizirati kroz primjenu UML notacije.

`E-restoran`

U ovom našem primjeru, vlasnik restorana – naručilac, želi da modernizira poslovanje uvođenjem informacionih tehnologija, te da na taj način učini usluge restorana jednostavnijim i interesantnijim, a time i poboljša poslovanje restorana. Cilj je da restoran, prije svega, omogući naručivanje i plaćanje u elektronskom obliku, ali da se osim toga izvrši informatizacija i ostalih ključnih poslovnih procesa u restoranu. Zbog toga ciljni sistem nazivamo **`e-restoran`**.

Naručivanje

Proces naručivanja se treba odvijati putem interfejsa. Dakle, gost `e-restorana`, treba da ima na raspolaganju interfejs za naručivanje i da može da ga koristiti putem ekrana osjetljivog na

dodir. Formirana narudžba se nakon toga provjerava da se potvrdi da je istu moguće realizirati, odnosno da postoji dovoljno raspoloživih supstanci za realizaciju iste. Zatim se vrši obračun narudžbe te obračun popusta, ukoliko naručilac jela ima potrošačku karticu. Nakon toga se gost `e-restorana`, odnosno naručilac jela, vodi u proces plaćanja. Ako naručilac jela izvrši plaćanje naručenog, onda se narudžba automatski prenosi do kuhinje na realizaciju.

Osim mogućnosti da gost e-restorana putem interfejsa na stolu izvrši naručivanje, treba da postoji i mogućnost naručivanja putem Interneta. Takve narudžbe bi se dostavljale na adresu naručioca jela.

Svakoj narudžbi se dodjeljuje jedinstveni identifikacijski broj koji se gostima restorana štampa na tiketu kao dokaz da su izvršili i platili narudžbu, a naručiocima jela putem Interneta se nudi ispis potvrde sa brojem narudžbe (tiketa).

Plaćanje

Pretpostavka je da će većina gostiju `e-restorana` koristiti plaćanje kreditnom karticom, a da će manji broj njih željeti izvršiti plaćanje gotovinom. Poslije obračuna narudžbe, naručiocu jela se pokazuje račun koji treba platiti sa izborom načina plaćanja. Ukoliko naručilac jela odabere plaćanje kreditnom karticom, onda se prikazuje interfejs za unos podataka vezanih za izvršenje plaćanja. Poslije unošenja podataka slijedi provjera podataka za plaćanje. Ako su podaci validni onda se šalju eksternom sistemu za autorizaciju kartica koji treba da odobri ili odbije transakciju plaćanja. U slučaju odabira gotovinskog plaćanja, blagajnik dolazi do naručioca jela gdje vrši gotovinsku naplatu, a zatim u sistem potvrđuje da je narudžba plaćena i da ista može preći na realizaciju, te izdaje tiket sa brojem narudžbe naručiocu jela.

Upravljanje zalihama

Sistem treba da automatizira upravljanje zalihama tako što bi se za svaku supstancu koja se koristi prilikom pripremanja jela definirala minimalna zaliha pa kad se desi da stanje zaliha padne ispod minimalnog da se iste supstance stavljaju na listu za naručivanje.

Validiranje narudžbe

Za sva jela postoje recepture koje određuju količinu pojedinih supstanci potrebnih za pripremanje istog. Kada se validira narudžba tada se provjerava da li za sva jela navedena na narudžbi postoji dovoljna količina supstanci za pripremu. Ako ne postoji dovoljna količina supstanci za pripremu nekog jela onda se naručilac jela vraća na prethodni korak, odnosno na proces formiranja narudžbe.

Realizacija narudžbe

Kada se potvrdi da je narudžba plaćena, onda ista dolazi u kuhinju na realizaciju. Kuharima se za svaku pristiglu narudžbu, na osnovu receptura jela, generira dokument za izuzimanje supstanci iz skladišta (izdatnicu). Kuhar može modificirati izdatnicu ukoliko nešto od navedenih supstanci već postoji u kuhinji. Nakon potvrđivanja izdatnice kuhar uzima navedene supstance, a stanje u sistemu će biti usklađeno sa stanjem u skladištu. Kuhar koji je preuzeo narudžbu na realizaciju, po završetku pripreme svih jela sa narudžbe, prosljeđuje istu na preuzimanje ili na pakovanje, ukoliko se vrši dostava iste na adresu naručioca jela. Narudžbe koje su spremne za preuzimanje nalaze se prikazane na velikom ekranu `u e-restoranu` i poslužilac uzima redom narudžbe i servira naručiocima jela, te pri tome uzima tiket sa brojem narudžbe od naručioca jela.

U slučaju dostave narudžbe, ista se pakuje u kutiju i dostavljač je odnosi na adresu naručioca jela.

Nabavka

Proces nabavke supstanci treba da se odvija svaki dan, tako što kuhar pregleda listu za nabavku (supstance koje imaju stanje manje od minimalno dozvoljene količine) i eventualno je modificira. Nakon toga se lista za nabavku proslijedi vlasniku restorana koji vrši nabavku. Po prijemu nabavljenih supstanci u skladište formira se dokument za prijem (primka), na osnovu liste za nabavku. Primka se može modificirati ukoliko nabavljene količine odstupaju od planiranih za nabavku. Nakon potvrđivanja, stanje zaliha se ažurira da se uskladi sa stvarnim stanjem skladišta.

Održavanje jelovnika

Jelovnik mijenja vlasnik restorana u dogovoru sa kuharom. Osim toga, kada količina raspoloživih supstanci za pripremu nekog jela padne ispod potrebnih za pripremu tog jela, onda se to jelo automatski skida sa jelovnika.

Održavanje receptura

Kuhar definira nove recepture za nova jela i specificira supstance i njihove količine potrebne za pripremu jela. Isto tako kod promjene recepture za jelo, kuhar mijenja supstance i/ili količine potrebne za pripremu jela.

NAPOMENA: Postoje različite vrste specifikacija zahtjeva. Najčešće su one vezane za poslovne sisteme, ali mogu biti vezane i za automatizaciju manualnih procesa u fabrici ili za procese u realnom vremenu.

U ovom primjeru je riječ o poslovnom sistemu i specifikacija zahtjeva treba da sagleda: poslovne procese, ograničenja, pravila i performanse.

Sistem svoje ponašanje realizira kroz realizaciju procesa, kao što su naručivanje, plaćanje, isporučivanja i slično. Da bi neko koristio sistem potrebno je da poznaje način interakcije sa njim. Poslovni proces opisuje interakciju sa sistemom.

Ograničenja predstavljaju granice do kojih se može mijenjati stanje sistema.

Pravila opisuju „dogovorene“ principe ponašanja, odnosno funkcioniranje sistema.

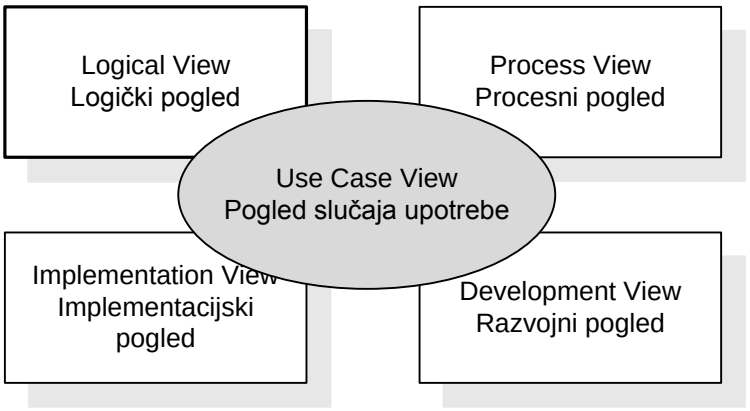
Performanse opisuju karakteristike sistema prilikom njegove upotrebe.

POGLAVLJE 4.

DIJAGRAM SLUČAJEVA UPOTREBE

Prvi korak pri razvoju softverskih rješenja je specifikacija zahtjeva naručioca, te predstavljanje sistema na jednostavan i razumljiv način, iz kojeg se jasno vide postavljeni ciljevi i zahtjevi. U okviru UML-a se za ove svrhe koriste dijagrami slučajeva upotrebe (eng. use-case diagram) i tekstualni opisi u obliku scenarija.

Dijagrami slučajeva upotrebe upravo predstavljaju način prikupljanja funkcionalnih zahtjeva sistema. Slučajevi upotrebe su srce našeg modela i imaju utjecaj na sve ostale elemente unutar dizajna sistema što je i vidljivo na slici 4.1 koja prikazuje poglede na naš sistem.



Slika 4.1: Pogleda slučaja upotrebe

Slučajevi upotrebe (eng. use cases) predstavljaju tehniku modeliranja koja se koristi za opisivanje što bi to novi sistem trebao da radi ili što postojeći sistem radi, sa aspekta interakcije između sistema i korisnika.

Modeliranje sa slučajevima upotrebe u zvanični standard UML-a je uveo Ivar Jacobson ranih 1990. godina na osnovu scenarija (eng. scenario*)*, koji u principu još uvijek postoji u UML-u. Od tada su slučajevi upotrebe veoma bitan dio UML-a i u objektno orijentiranoj zajednici im se predaje mnogo pažnje.

4.1 Namjena dijagrama slučajeva upotrebe

Dijagrami slučajeva upotrebe opisuju funkcionalnost sistema i njegova osnovna namjena je:

- Prikaz pogleda visokog nivoa što sistem radi i ko ga koristi.
- Komunikacija između korisnika i razvojnog tima.
- Odlučivanje o funkcionalnim zahtjevima sistema i opisivanje istih.
- Davanje jasnog i konzistentnog opisa što bi sistem trebao raditi što daje osnovu za sve naredne odluke koje se tiču dizajna sistema.
- Osnova za verificiranje sistema.

Ovi dijagrami se kreiraju u ranoj fazi projekta. Korisnici sistema su zainteresirani za uvid u ove dijagrame jer isti opisuju funkcionalnost sistema i načine na koje će se sistem koristiti. Razvojnom timu su značajni kao stalni podsjetnik što bi sistem trebao raditi i predstavljaju osnovu za dalji rad na projektu. Timovima za integraciju i testiranje dijagrami slučajeva upotrebe su bitni zbog provjere i osiguravanja da sistem pruža sve dogovorene potrebne funkcionalnosti.

4.2 Elementi dijagrama slučajeva upotrebe

Elementi koji se koriste za izgradnju dijagrama slučajeva upotrebe su:

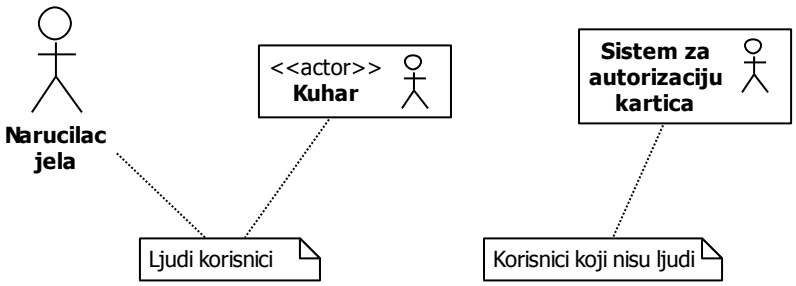
1. Učesnik - Sudionik (eng. actor)
2. Slučaj upotrebe (eng. use case)
3. Granica sistema (eng. system boundary)
4. Veze između slučajeva upotrebe: asocijacija (eng. association), proširivanja (eng. extend), uključivanja (eng. include), generalizacije (eng. generalization).

Dijagram slučajeva upotrebe može se definirati i kao kolekcija aktera, slučajeva upotrebe i njihovih veza. U nastavku ćemo detaljnije objasniti značenje i notaciju elemenata dijagrama slučajeve upotrebe.

4.3 Akter

Akter (sudionik, učesnik) je netko ili nešto tko će (što će) biti u interakciji sa sistemom koji se razvija.

Standardni stereotip za sudionika je figura "stick man" – "štapićasti čovjek" sa naznačenim imenom sudionika koje se navodi ispod figure. Njegov naziv predstavlja njegovu ulogu u modelu.



Slika 4.2: Notacija - Akter

Akter je neko ili nešto što učestvuje u izgradnji sistema. Postoje dva primarna tipa aktera: korisnici sistema -ljudi (human) i drugi sistemi.

Prvi tip aktera su fizičke ličnosti ili korisnici sistema. Oni su najčešći učesnici i mogu se naći gotovo u svakom sistemu. Za naš sistem `e-restorana`, akteri su ljudi koji će direktno koristiti sistem i koji su u ulozi gosta. Kuhar je također akter koji na osnovu primljene narudžbe priprema jelo. Mi također znamo da se i preko Interneta može naručiti jelo, u tom slučaju imamo i novog aktera dostavljača.
Prilikom uočavanja aktera često imamo i podjelu na aktere radnike koji su u sklopu poslovnog procesa same organizacije i aktere koji iniciraju izvana neke poslovne procese, a ne utiču direktno na njihovu organizaciju.
Akter, radnik u poslovnom procesu je određena uloga u organizaciji. Radnike u poslovnim procesima modeliramo da bi razumjeli pojedine uloge u poslovnim procesima i njihovu

interakciju. Na taj način možemo precizirati odgovornosti vezane za pojedinačne uloge, uočiti zahtijevane vještine za određene uloge i druge detalje. Tipični primjeri radnika poslovnog procesa za jedan `e-restoran` su kuhar, dostavljač, menadžer restorana. Akter poslovnog procesa ne mora biti pojedinac, naprotiv, akteri su često grupe ljudi ili kompanija.

Nadalje, akteri koji nisu u poslovnom procesu organizacije mogu biti gosti restorana, kupci, kreditori, investitori ili dobavljači. Svaki od ovih aktera je od interesa za aktivnosti koje se obavljaju u kompaniji ali ih ne organiziraju.

Kako se vrši identifikacija aktera? Sistem funkcionira realizacijom svojih poslovnih procesa. U tim poslovnim procesima učestvuju akteri, bilo interni, bilo eksterni. Osim toga, sistem ima poslovne ciljeve koje želi realizirati kroz uvođenje novog softverskog sistema. Taj cilj je osnova za odabir procesā koji se analiziraju, a analiza tih procesā nam daje aktere koji su značajni za sagledavanje željenog funkcioniranja sistema. U primjeru `e-restorana` jasno je da je cilj poboljšanje poslovanja kroz uvođenje informacijskih tehnologija, prije svega u kontaktu sa klijentima restorana, a onda i u okviru internih procesa realizacije usluga. Iz toga može se zaključiti da je naručilac jela (klijent restorana) akter od koga treba krenuti, a da je on uključen u proces naručivanja jela. Analizom procesa naručivanja jela, uzimajući u obzir željena poboljšanja u okviru tog procesa pronalazimo ostale aktere.

Drugi tip aktera je neki drugi sistem. Na primjer, `e-restoran` može koristiti komunikaciju sa nekom eksternom aplikacijom za realizaciju transakcija plaćanja putem kreditnih kartica. Takva aplikacija predstavlja još jednog aktera. To je sistem koji nećemo mijenjati, tako da je izvan domena tekućeg projekta, ali ipak treba da komunicira sa našim novim sistemom. Prilikom kreiranja dijagrama slučajeva upotrebe postoji dilema da li i do kojeg nivoa predstavljati ovaj drugi tip aktera. Obično se ne prikazuje detaljno kao prvi tip aktera.

Svi akteri nisu ljudi ili eksterni sistem. Akter može biti i vrijeme, ali ne u istom smislu kao navedena dva tipa aktera. Vrijeme dolazi do izražaja kada želimo da uključimo neko ponašanje u okviru sistema u nekom vremenskom trenutku.

Proces identifikacije aktera u sistemu bi se mogao zasnovati na pitanju da li je neko ili je nešto u interakciji sa sistemom. Ako jeste, onda je naš odabir kandidat za aktera, koji postaje akter; ako potencijalnog kandidata ne možemo mijenjati sa nekim već identificiranim.

4.4 Slučaj upotrebe

Slučaj upotrebe (eng. use case) jest uzorak ponašanja sistema, podsistema ili klase što se ogleda u razmjeni poruka između sistema i jednog ili više vanjskih sudionika (sudionik ili neki drugi sistem). Obično je slučaj upotrebe specifičan način upotrebe sistema iz perspektive aktera.

Slučaj upotrebe se označava elipsom koja sadrži ime – naziv slučaja upotrebe . Ime bi trebalo odražavati slijed događaja sudionika i početi sa glagolom.

Slika 4.3: Notacija - Slučaj upotrebe

Slučajevi upotrebe poslovnih procesa predstavljaju grupu međusobno povezanih tokova poslovnih procesa u organizaciji koji su značajni za aktere poslovnih procesa. Slučajevi upotrebe govore što organizacija radi.
Za `e-restoran` neki od slučajeva upotrebe koje možemo uočiti su: `Naručivanje jela u restoranu`, `Naručivanje jela putem Interneta`, `plaćanje kreditnom karticom`.

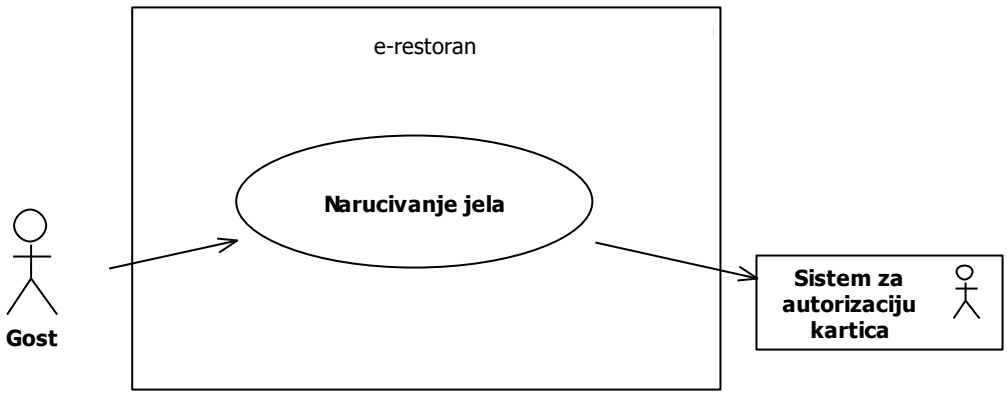
Kao što vidimo, nazivi slučajeva upotrebe se tipično formiraju po formatu "<*glagol*> <*imenica(e)*>". Ovaj standard je dobro poštovati iz nekoliko razloga. Na taj način imamo suglasnost slučajeva upotrebe i poslovnih procesa, čak i ako su definirani na osnovu višestruke analize. Također, krajnjem korisniku je lakše da ih ovako razumije. Na kraju, možda i najvažniji razlog za standardno pisanje naziva je to što tako imamo fokus na to što je predmet poslovnog procesa i što se postiže, a ne samo koji se entiteti koriste.
Jedan od načina na koji možemo prikupiti slučajeve upotrebe poslovnog procesa je pregled procesa i procedura koje se javljaju u organizaciji, intervjui sa korisnicima usluga ili naše vlastito znanje o poslovnom procesu.

Slučajevi upotrebe su visok nivo funkcionalnosti koji nam sistem pruža. Drugim riječima, slučajevi upotrebe nam pokazuju kako neko može koristiti sistem.
Slučaj upotrebe predstavlja drugačiji pristup od tradicionalnih metoda. Dijeleći projekt na slučajeve upotrebe dobivamo procesno orijentiran pogled na sistem, a ne izvršno orijentiran pogled na sistem. Slučaj upotrebe se usmjerava na to što sistem treba uraditi, a ne kako će to sistem uraditi. Slučaj upotrebe nam daje pregled visokog nivoa na sistem. Taj pregled treba da dopusti klijentima da lako vide, na vrlo visokom nivou, naš cijeli sistem. Svaki slučaj upotrebe bi trebalo da pokaže kompletnu transakciju između korisnika i sistema, koja rezultira u nečemu što je od važnosti za korisnika.

4.5 Granice sistema

Granice sistema (eng. system boundary) definiraju funkcionalnost sistema, a funkcionalnost je predstavljena određenim brojem slučajeva upotrebe. Granica sistema ima ulogu da odvoji učesnike od slučajeva upotrebe, pri čemu se učesnici nalaze izvan granica sistema. Predstavlja se sa pravougaonikom (okvirom) sa nazivom koji odgovara nazivu sistema. Veoma je korisna upotreba granica sistema jer se na takav način interakcija sistema više naglašava.

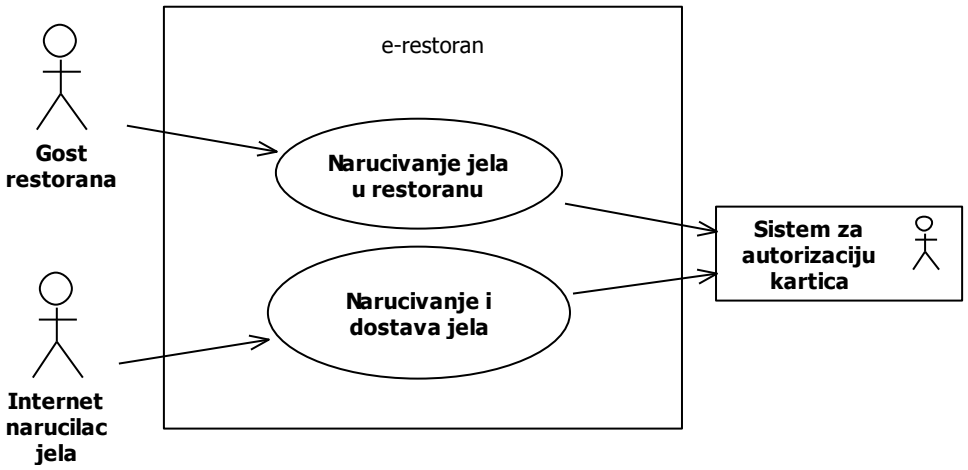
Za `e-restoran` na osnovu postavljenih zahtjeva naručioca, prvi problem koji novi sistem mora riješiti je da omogući naručivanje i plaćanje u elektronskom obliku. Iz toga proističe prva varijanta dijagrama slučaja upotrebe prikazan na slici 4.4 koja ujedno ilustrira novo uvedene elemente i notaciju aktera, slučaja upotrebe i granica sistema.



Slika 4.4: Dijagram slučaja upotrebe sa granicama sistema

Svaki slučaj upotrebe u okviru granica sistema mora uvijek dostaviti odgovarajuću vrijednost učesniku, i ta vrijednost predstavlja nešto što učesnik želi od sistema, što se može pokazati i sa našim drugim zahtjevom na sistem.

Drugi zahtjev koji se postavlja pred sistem je da omogući naručivanje putem Interneta i dostavu naručene hrane, pri čemu je dostavljač zaposlenik restorana pa se za početak, dok se na sistem gleda sa najvišeg nivoa apstrakcije, i ne predstavlja na dijagramu (slika 4.5).



Slika 4.5: Dijagram slučajeva upotrebe sa više slučajeva upotrebe

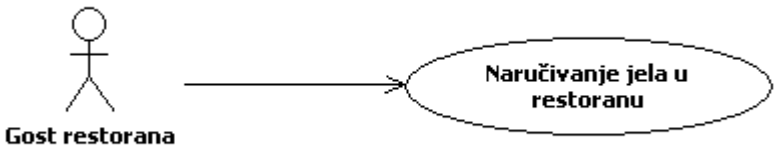
U modeliranju sistema putem slučajeva upotrebe isti se predstavlja kao crna kutija koja prikazuje slučajeve upotrebe. Kako to sistem radi, kako se slučajevi implementiraju i kako

rade interno, uopće nije bitno. U suštini, pošto se ovo modeliranje često radi u najranijim fazama projekta, razvojni inženjeri nemaju jasnu ideju kako će se ti slučajevi upotrebe uopće i implementirati.

4.6 Relacije

Postoji više tipova relacija koji se mogu javiti u dijagramima slučajeva upotrebe: asocijativne, sadržajne, proširene i relacije generalizacije. Ovdje trebamo uočiti relacije između aktera i slučaja upotrebe, relacije između slučajeva upotrebe, kao i relacije između aktera.

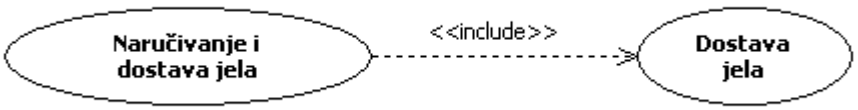
Veze između aktera i slučajeva upotrebe su asocijativne veze. Predstavljaju se strelicom i upoznali smo ih u prethodnom primjeru. Mnogi odnosi se mogu prikazati asocijacijom.



Slika 4.6: Asocijativna veza

Ostale veze se odnose na veze između slučajeva upotrebe.

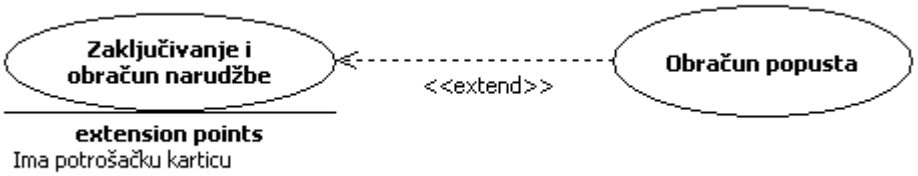
Sadržajne veze (eng. includes) označavaju relaciju u kojoj želimo istaći da jedan slučaj upotrebe predstavlja korak ili dio složenijeg slučaja upotrebe. Tipičan način označavanja sadržajne veze je sa stereotipom <<include>> iznad isprekidane strelice koja povezuje slučajeve upotrebe. Za stereotip <<include>> koristi se i stereotip <<uses>>.



Slika 4.7: Sadržajna veza

Prethodna veza naglašava da slučaj upotrebe `Naručivanje i dostava jela` uključuje i `Dostavu jela` naručiocu.

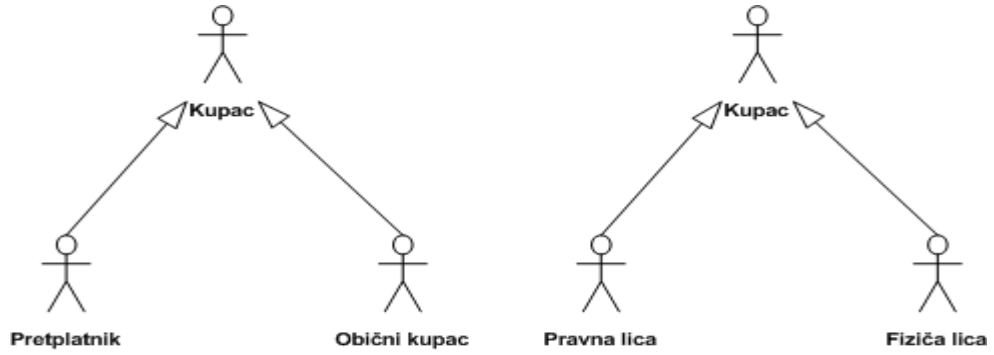
Proširene veze (eng. extend) omogućavaju jednom slučaju upotrebe da proširi funkcionalnost koju pruža neki drugi slučaj upotrebe što ćemo ilustrirati i za naš sistem.
Sistem `e-restorana` uključuje naručivanje jela i plaćanje, ali se od sistema zahtijeva da za imaoce potrošačkih kartica obračuna popust od 5% na ukupnu vrijednost narudžbe. Jasno je da treba prvo izvršiti obračun narudžbe, pa tek onda primijeniti popust, i to tek ako naručilac jela ima potrošačku karticu. Nazovimo ovaj novi slučaj upotrebe `Obračun popusta`. Mogla bi se napraviti sadržajna veza između `Zaključivanje i obračun narudžbe` i `Obračun popusta`, ali bi to značilo da se pri svakom obračunu narudžbe vrši i obračun popusta, što nam ne odgovara. U ovakvim situacijama se koristi proširena veza, koja postoji ako je udovoljen uvjet proširenja (u ovom primjeru je to da naručilac jela koji ima potrošačku karticu):



Slika 4.8: Proširena veza

Može se uočiti da se veza proširivanja označava kao i sadržajna veza ali sa stereotipom <<extend>>.

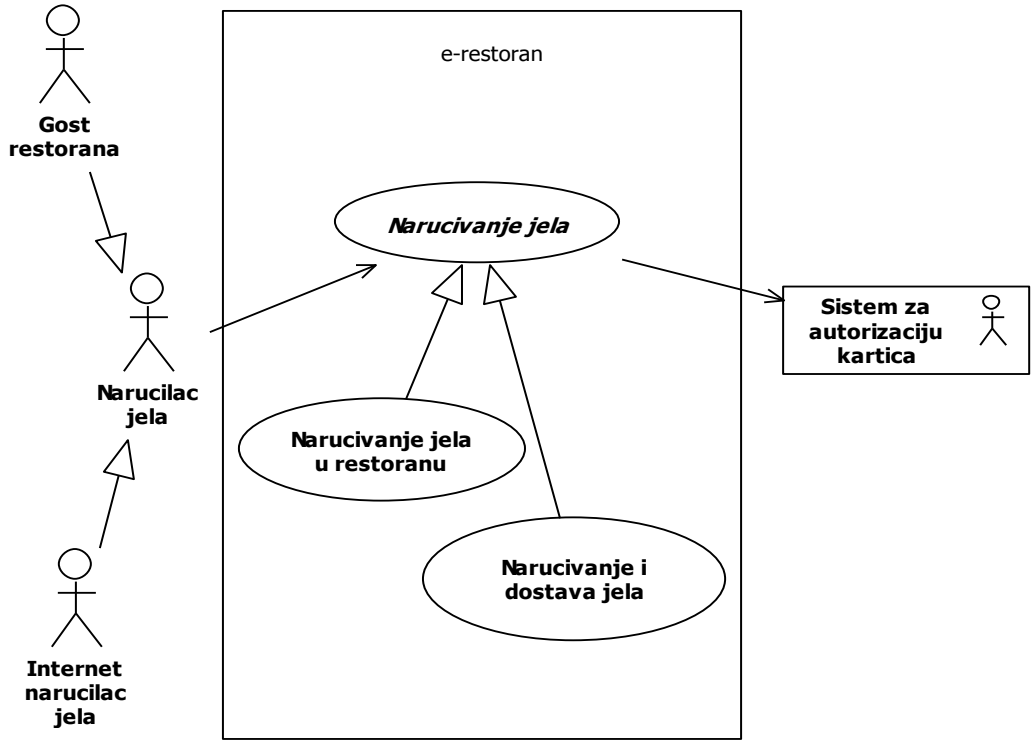
Veza generalizacije predstavlja relaciju između aktera u sistemu. Ove veze je obično potrebno prikazivati samo onda kada se jedan tip aktera ponaša značajno drugačije od drugog tipa.



Slika 4.9: Relacija generalizacije

Isti koncept generalizacije može se primijeniti i na slučajeve upotrebe.

Vezano za `e-restoran` i prikazane slučajeve upotrebe na slikama 4.4 i 4.5 možemo uočiti da su `Naručivanje jela u restoranu` i `Naručivanje i dostava jela` slični procesi te da se može napraviti generalizacija. Analogno tome, može se uočiti i generalizacija vezana za aktere: `Gost restorana` i `Internet naručilac jela`. Tako dolazimo do treće varijante dijagrama koji je prikazan na slici 4.10.



Slika 4.10: Dijagram slučaja upotrebe `e-restoran` na kojem se uočava generalizacija

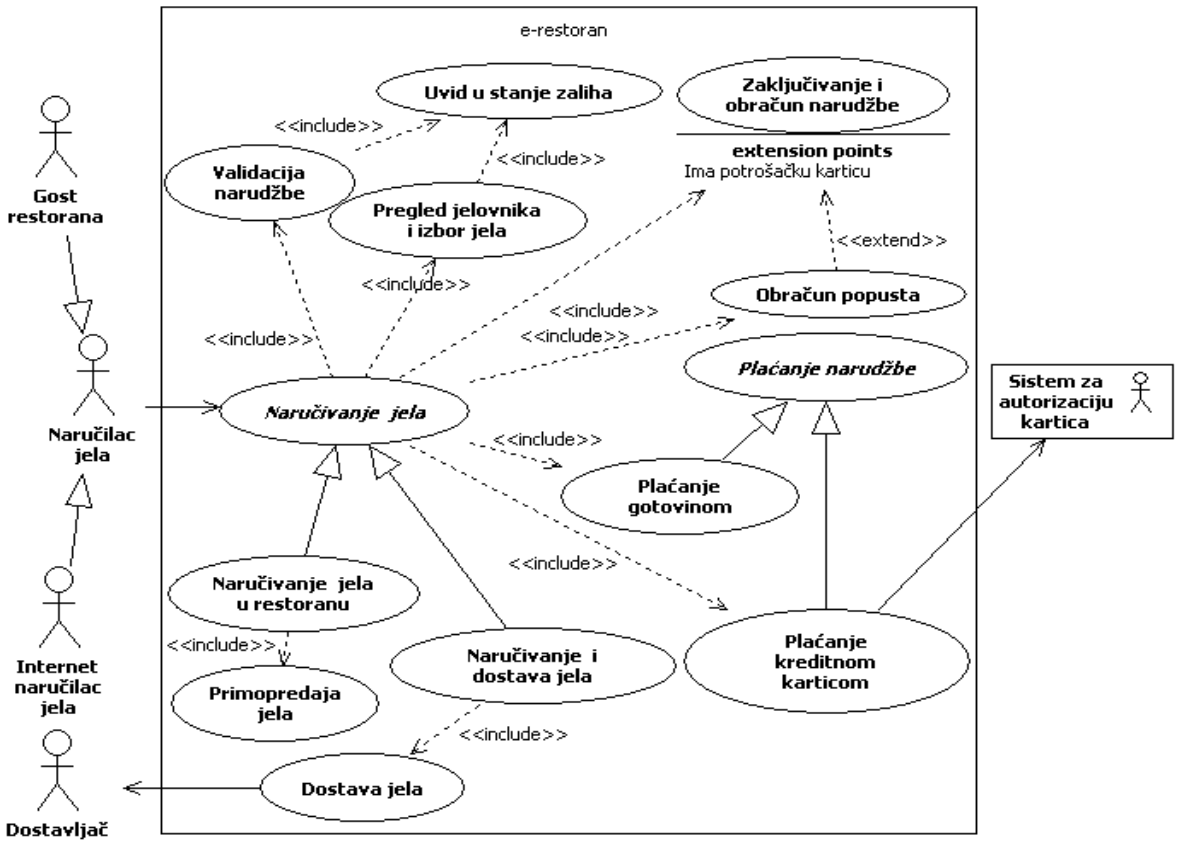
4.7 Kreiranje dijagrama slučajeva upotrebe

Konačno, ako smo definirali sve aktere, slučajeve upotrebe i relacije između njih, možemo konstruirati i složeniji dijagram slučajeva upotrebe. U tu svrhu nastavljamo sa analizom `e-restorana`.

Sad se može dublje ući u analizu što je to `Naručivanje jela`, što znači preći na sljedeći nivo hijerarhije, drugi nivo apstrakcije, odnosno primijeniti dekompoziciju problema. Time se može utvrditi da se `Naručivanje jela` sastoji od:

- `Pregled jelovnika i izbor jela`
- `Zaključivanje i obračun narudžbe`
- `Plaćanje narudžbe` (koje može biti kreditnom karticom ili gotovinom).

Osim toga, već je spomenuto da poslovni proces predviđa da se za imaoce potrošačkih kartica predviđa obračunavanje popusta, što predstavlja proširenje `Zaključivanja i obračuna narudžbe`. Ova analiza nas je dovele do sljedećeg složenijeg dijagrama slučajeva upotrebe.



Slika 4.11: Složenija varijanta `e-restoran` dijagrama slučajeva upotrebe

Preporuke prilikom postavljanja aktera na dijagrame slučajeva upotrebe:

1. Postaviti primarne aktere u gornji lijevi ugao na dijagramu.
2. Crtati aktere izvan dijagrama.
3. Imenovati aktere sa imenicom u jednini, koja je poslovno relevantna.
4. Povezati svakog aktera sa jednim ili više slučajeva upotrebe.
5. Modelirati uloge aktera ne njegovu poziciju.
6. Koristiti <<system>> za identificiranje sistemskog aktera.
7. Aktere na dijagramima slučajeva upotrebe ne postavljati da su u interakciji jedan sa drugim.
8. Uvesti aktera "Time" za inicijalizaciju nekih vremenskih rasporeda.

Preporuke prilikom postavljanja slučaja upotrebe na dijagram slučajeva upotrebe:

1. Slučaj upotrebe treba da počinje sa glagolom.
2. Imenovati slučaj upotrebe na osnovu domenske terminologije.
3. Postaviti primarne slučajeve upotrebe na lijevi ugao dijagrama.
4. Primijeniti vremensko dešavanje prilikom postavljanja dijagrama (`Naručivanje jela putem Interneta`, `Dostava jela`).

Korisni savjeti prilikom konstrukcije dijagrama slučaja upotrebe bi se mogli sumirati sa sljedećim:

- Ponekad se na dijagramu ne prikazuju manje bitne veze ili one veze koje se "podrazumijevaju" da se ne pretrpa dijagram. S druge strane se ponekad nacrta neka veza, da bi se naglasila povezanost, iako ta veza postoji nacrtana indirektno, preko drugih simbola.
- Interne komponente sistema treba što više izbjegavati prikazivati jer ne daju nove informacije. Isto se odnosi na "podrazumijevane" interne aktere u sistemu. Ali, ukoliko želimo da naglasimo neku internu komponentu, ili aktera ili vezu, onda to možemo i predstaviti. Sistem treba što više posmatrati kao "crnu kutiju" da se ne izgube iz vida poslovni ciljevi.

U ovoj se fazi treba što više fokusirati na to "što" i "kada" sistem treba raditi, a manje na to "kako" jer se tome više pažnje posvećuje u fazi dizajna.

4.8 Opis slučajeva upotrebe i scenarija

Scenarij je sekvenca koraka koja opisuje interakciju između korisnika i sistema, a zasniva se na slučajevima upotrebe koje smo već objasnili.

Tako, u slučaju `e-restoran` sistema možemo imati scenarij `Naručivanje jela u restoranu i plaćanje kreditnom karticom`, koji može opisati sljedeće:
Naručilac jela putem Web-interfejsa, dostupnog u restoranu, pregleda jelovnik, zaključuje narudžbu te plaća obračunati iznos kreditnom karticom.

Scenarij je jedna stvar koja se može desiti. Na primjer, autorizacija kartice može biti neuspješna i to može biti odvojeni scenarij. Možemo imati i slučaj da imamo poznate klijente kojima ne moramo tražiti podatke o načinu isporuke i informacijama o karticama i to je neki drugi scenarij. Svi ovi scenariji su slični. Ustvari, slučaj upotrebe je skup scenarija povezanih sa zajedničkim korisničkim ciljem koji ne mora uvijek biti ispunjen.
Procesi u sistemu koji su prikazani na dijagramu slučajeva upotrebe se prezentiraju i putem scenarija. Pri tome se tokovi događaja mogu tabelarno prikazati i mogu se koristiti razgraničenja između aktera ili podsistema da bi tokovi događaja bili puno jasniji. Bitno je da se opisani scenarij može vidjeti i na dijagramu kao slučaj upotrebe.

Tabela za opis scenarija ili tekstualni opis slučajeva upotrebe prikazuje detalje koji se mogu opisati za scenarij i slučaj upotrebe i daje njihovo objašnjenje.

Tabela 4.1: Opis scenarija

Naziv	**Ime slučaja upotrebe**
Opis	Opis slučaja upotrebe.
Vezani zahtjevi	Indikacija koje zahtjeve slučaj upotrebe ili scenarij djelomično ili potpuno ispunjavaju.
Preduvjeti	Što se treba ispuniti prije nego što se slučaj upotrebe ispuni.
Posljedice - uspješan završetak	Stanje sistema ako je slučaj upotrebe

Naziv	**Ime slučaja upotrebe**
	uspješno izvršen.
Posljedice - neuspješan završetak	Stanje sistema ako je slučaj upotrebe neuspješno izvršen.
Primarni akteri	Glavni akteri koji učestvuju u slučaju upotrebe. Najčešće se uključuju akteri koji uzrokuju ili direktno dobivaju informacije od slučaja upotreba.
Ostali akteri	Akteri koji učestvuju ali nisu glavni akteri prilikom izvršavanja slučaja upotrebe.
Glavni tok	Opisuje sve važne korake izvršavanja slučaja upotrebe.
Proširenja/Alternative	Opisuje alternativne korake za neke korake glavnog toka.

Prilikom opisa scenarija ili slučaja upotrebe ne moramo sve gore navedene detalje ubaciti, a možemo dodati i neke koji nisu navedeni, ukoliko smatramo da će dati odgovarajuću vrijednost opisu slučaja upotrebe ili scenarija.

Zasnovano na gornjoj tabeli za `e-restoran` slijedi opis za:

Scenarij 1. Naručivanje jela u restoranu i plaćanje kreditnom karticom.

Naziv: `Naručivanje jela u restoranu i plaćanje kreditnom karticom`

Opis: `Naručilac jela` putem Web-interfejsa, dostupnog u restoranu, pregleda jelovnik, zaključuje narudžbu te plaća obračunati iznos kreditnom karticom.

Glavni tok: Završava uspješno izvršenim plaćanjem narudžbe.

Preduvjeti: `Naručilac jela` ima dostupan Web-interfejs za naručivanje.

Posljedice: `Naručilac jela` ima potvrdu da je narudžba prihvaćena.

Tok događaja:

Naručilac jela	**Sistem `e-restoran`**	**Sistem za autorizaciju kartica**
1. Pristupanje interfejsu za naručivanje.	2. Prikaz trenutne ponude na jelovniku i omogućavanje izbora.	
3. Izbor jela i zaključivanje narudžbe.	4. Validacija narudžbe.	
	5. Rezerviranje sastojaka za jela koja treba napraviti.	
	6. Obračun narudžbe.	
	7. Očitanje potrošačke kartice, ako je ubačena i obračun popusta na narudžbu.	
	8. Prikaz računa (iznosa za plaćanje) i prikaz obrasca za unos informacija za plaćanje karticom.	
9. Unošenje imena i prezimena, adrese, broja kartice i datuma isteka.	10. Validacija unesenih podataka.	
	11. Prosljeđivanje transakcije Sistemu za autorizaciju kartica.	12. Izvještavanje da je transakcija prihvaćena.
	13. Označavanje narudžbe prihvaćenom.	
	14. Generiranje jedinstvenog broja narudžbe.	
	15. Štampa tiketa s brojem narudžbe i obavještava Naručioca jela da je narudžba prihvaćena.	
16. Uzimanje tiketa i završavanje interakcije sa sistemom, te odlazak da se sačeka završetak pripremanja jela.		

Za korak 4 – Validacija narudžbe može se vezati poslovno pravilo 1: Za sva naručena jela moraju postojati dovoljne količine supstanci na zalihi da bi se mogla napraviti jela. Ukoliko ovo pravilo nije ispunjeno, potrebno je imati alternativni tok za izvršavanje slučaja upotrebe, odnosno cjelokupnog ovog scenarija.

Alternativni tok 1: Neuspješna validacija narudžbe.

Preduvjeti: Na koraku 4. glavnog toka nije zadovoljeno poslovno pravilo 1.

Tok događaja:

Naručilac jela	**Sistem `e-restoran`**	**Sistem za autorizaciju kartica**
	1. Narudžba ne prolazi validaciju zbog poslovnog pravila 1.	
	2. Upozoravanje korisnika u čemu je problem.	
	3. Ažuriranje trenutne ponude na jelovniku i omogućavanje izbora.	
4. Nastavak na koraku 3. glavnog toka događaja.		

Možemo uočiti i postojanje još alternativnih tokova kao što je alternativni tok 3 – `Transakcija karticom odbijena` ili alternativni tok 4 – `Odustajanje od naručivanja` koje nećemo detaljno prikazivati.

Nakon što je napravljena osnovna verzija dijagrama slučajeva upotrebe koja pokriva najvažniji segment cijelog sistema, u ovom slučaju naručivanje jela, prelazi se na sljedeću fazu analize, koja u ovisnosti od metodologije koja se primjenjuje na projektu, te procjene tima, može krenuti nekim od sljedećih pravaca:

1. Produbljivanje analize naručivanja jela da bi detaljnije specificirali zahtjeve i olakšali dizajniranje.
2. Proširivanje analize na druge segmente: realizacija narudžbe, nabavka.

Na dijagramu slučajeva upotrebe, kao i na svim ostalim dijagramima, mogu se dodavati komentari koji se mogu iskoristi na ovim dijagramima da bi se iskazala bitna poslovna pravila, koja mogu poboljšati dizajn sistema i smanjiti broj pogrešnih odluka. Za naš sistem jedno od pravila bi moglo biti: Naručioci jela koji imaju potrošačku karticu imaju 5% popusta na ukupnu narudžbu, a ostali nemaju popusta.

Scenarij treba da održava poslovni proces na način kako ga vidi naručilac sistema, pa je shodno tome dobro u njemu koristiti i terminologiju naručioca sistema. Scenariji su najbolji način da se naručiocu sistema pokaže kako će se odvijati poslovni proces, te da se dobije prihvaćanje istog, radi prelaska u fazu dizajniranja.

U okviru scenarija mogu se koristiti i elementi pseudokôda koji bi olakšali predstavljanje više scenarija na jednom, ali to otežava čitljivost istog pa nije poželjno.

Završno razmatranje

Sada pošto imamo pogled sa visokog nivoa na sistem, i pošto smo se detaljno upoznali sa sistemom `e-restoran`, već imamo ideje koje klase bi mogli koristiti pa prelazimo na novo poglavlje koje uvodi osnovne pojmove vezane za dijagrame klase.

Pitanja za ponavljanje

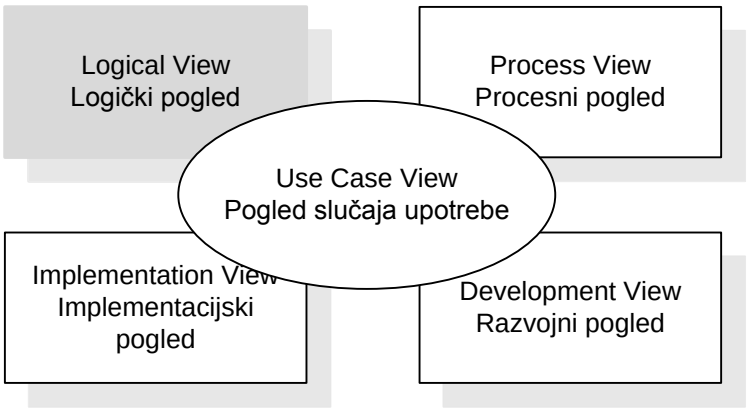
1. Koja je osnovna namjena dijagrama slučajeva upotrebe?
2. Kojem pogledu pripada dijagram slučajeva upotrebe?
3. Nabrojati osnovne elemente dijagrama slučajeva upotrebe.
4. Objasniti što je slučaj upotrebe.
5. Koja je notacija slučaja upotrebe?
6. Objasniti što je akter.
7. Koja je notacija aktera?
8. Koji tipovi aktera postoje u okviru dijagrama slučaja upotrebe?
9. Koja vrsta veze je dozvoljena između aktera?
10. Koja vrsta veze je dozvoljena između aktera i slučajeva upotrebe?

11. Koje vrste veza postoje između slučajeva upotrebe?
12. Koja je razlika između sadržajne i proširene veze?
13. Koja je notacija za relaciju uključivanja-sadržajnu relaciju?
14. Koja je notacija za relaciju proširivanja?
15. Koja je notacija za vezu generalizacije?
16. Što je granica sistema i koja se notacija koristi za njeno prikazivanje?
17. Navedite preporuke za postavljenje i imenovanje aktera u okviru dijagrama slučaja upotrebe.
18. Navedite preporuke za postavljanje i imenovanje slučajeva upotrebe u okviru dijagrama slučaja upotrebe.
19. Što je scenarij?
20. Koji su elementi tabele za opis scenarija ili slučaja upotrebe?
21. Koja je veza dijagrama slučaja upotrebe i ostalih dijagrama?
22. U kojoj fazi razvojnog procesa se koriste dijagrami slučajeva upotrebe?

POGLAVLJE 5.

DIJAGRAM KLASA: KLASE

Dijagram klasa (eng. class diagram) opisuje tipove objekata u sistemu i različite vrste statičkih veza koje postoje među njima. Ovi dijagrami, također, prikazuju svojstva i operacije klasa, kao i razne načine povezivanja objekata, i predstavljaju logički pogled na sistem.



Slika 5.1: Dijagram klase pripada logičkom pogledu na sistem

Dijagram klasa prikazuje samo klase, iako postoji varijanta ovog dijagrama koja prikazuje aktualne objekte.

5.1 Namjena dijagrama klasa

Glavna namjena i upotreba dijagrama klasa je kako slijedi:

- Koriste se za dokumentiranje klasa koje čine jedan sistem ili podsistem.
- Koriste se da opišu veze između klasa kao što su sve vrste asocijacije, generalizacija, ovisnost.

- Koriste se da pokažu karakteristike klasa, prije svega pripadajućih atributa i operacija.
- Koriste se za vrijeme razvoja softvera, počevši od specifikacija klasa u domenu problema do implementacijskog modela za predloženi sistem, da pokažu strukturu klasa sistema.
- Koriste se za dokumentiranje kako klase pojedinog sistema surađuju sa postojećim bibliotekama klasa.
- Koriste se da pokažu interfejs koji se podržava sa datom klasom.

5.2 Klase u UML-u

Klase opisuju različite tipove objekata koje sistem može imati.

U UML-u klasa se prikazuje kao pravougaonik podijeljen u 3 dijela. Gornji dio podijeljenog pravouganika sadrži ime klase, srednji dio sadrži atribute klase, a donji dio operacije klase. Dio UML klase namijenjene za atribute i operacije je opcionalan. U skladu s tim neki od mogućih načina prikazivanja klasa data su na slici ispod.

Slika 5.2: Načini prikazivanja UML klase

Ukoliko ne postoje operacije ili atributi klase dio koji je namijenjen za njihovo navođenje može biti prikazan kao prazan dio u okviru UML simbola klase.

5.3 Atributi

Atributi klase, koji se prikazuju u srednjem dijelu ikone za klasu, opisuju podatke sadržane u objektu klase.

Potpuni oblik zapisa atributa je:

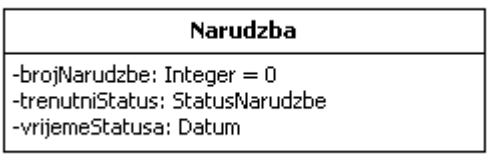
vidljivost ime:tip kardinalnost = inicijalna_vrijednost {opis_osobine}

Primjer za atribut je: ime:String [1] = "Neko ime" {readOnly}

Obavezno je samo ime. Pojedini dijelovi zapisa atributa imaju slijedeće značenje:

- **Vidljivosti** (eng. visibility) označava da li je atribut javni (+), privatni (-), zaštićeni (#) ili paketni (~). Privatna (eng. private) vidljivost znači da je atribut raspoloživ samo unutar klase unutar koje je i definiran. Javna (eng. public) vidljivost znači da je atribut raspoloživ za sve klase koje su asocijaciji sa klasom koja je vlasnik atributa. Zaštićena (eng. protected) vidljivost označava da je atribut raspoloživ unutar klase koja posjeduje atribut i svakog podtipa klase i vezana je za koncept generalizacije koji će se uvesti u okviru sljedećeg poglavlja. Paketna (eng. package) vidljivost znači da je atribut raspoloživ samo drugim klasama u istom paketu u kojem je i klasa unutar koje je definiran atribut. Atributi najčešće imaju privatnu vidljivost.
- **Ime atributa** određuje ime atributa u klasi i približno odgovara imenu polja u programskom jeziku.
- **Tip atributa** ukazuje na to da postoji ograničenje vrste objekata koji se mogu smjestiti u atribut. O tipu atributa možemo razmišljati kao o tipu polja u programskom jeziku. Tipovi podataka atributa mogu biti primitivni tipovi podataka, ali isto i klase.
- **Kardinalnost** pokazuje na koliko objekata se odnosi osobina i u nastavku će biti detaljnije objašnjeno njeno značenje.
- **Inicijalna vrijednost** definira inicijalno dodijeljenu vrijednost atributu objekta instanciranog na osnovu definirane klase.
- **Opis osobine** omogućava da definiramo dodatne osobine atributa i detaljnije će u nastavku biti objašnjeno njeno značenje.

Na slici 5.3 je prikazan UML model konkretne klase `Narudžba` koja sadrži više atributa.



Slika 5.3: Klasa `Narudzba`

Na primjeru klase `Narudžba` možemo vidjeti da svaki objekt klase `Narudžba` ima atribute:

- `BrojNarudžbe` koji je cjelobrojni tip podataka (Integer) i inicijalizira se na nulu, prilikom instanciranja objekta;
- `trenutniStatus` koji je tipa `StatusNarudzbe`, što predstavlja pobrojani tip;
- `vrijemeStatusa` koji je tipa `Datum` (složeni tip podataka realiziran kroz klasu `Datum`).

Na osnovu UML modela klase `Narudžba` bi se generirao sljedeći kôd u programskim jezicima Java i C++:

Java :
```
public class Narudzba
{
    private int brojNarudzbe;
    private StatusNarudzbe trenutniStatus;
    private Datum vrijemeStatusa;
    ...
}
```

C++ :
```
#if !defined(_NARUDZBA_H)
#define _NARUDZBA_H

#include "StatusNarudzbe.h"
#include "Datum.h"
...

class Narudzba {
       ...
private:
       int brojNarudzbe;
       StatusNarudzbe trenutniStatus;
       Datum vrijemeStatusa;
       ...
};

#endif  // _NARUDZBA_H
```

Generirani kod za klasu `Narudzba`

Kod generiranja kôda pomoću alata za UML modeliranje, kao što je StarUML™ treba imati u vidu da generirani kôd ovisi od tipova podataka korištenih pri definiranju klase, pa ako se upotrijebe tipovi karakteristični za određeni programski jezik onda će isti biti i upotrijebljeni pri generiranju (npr. int umjesto Integer – UML standard).

5.4 Kardinalnost

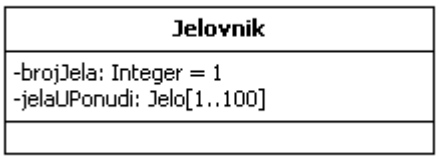
Kardinalnost (eng. multiplicity) atributa pokazuje na koliko objekata se odnosi neka osobina. Kardinalnost se piše odmah iza imena atributa i sastoji se od donje i gornje granice u okviru uglatih zagrada [ ] sa formom prikaza [m..n]. Ovdje m prikazuje vrijednost za donju granicu a n vrijednost za gornju granicu broja objekata na koji se primjenjuje osobina. Donja granica može biti bilo koji pozitivan broj ili nula. Gornja granica je bilo koji pozitivan broj ili zvjezdica koja znači da nema ograničenja.

Najčešće se susreće sljedeća konvencija skraćenog označavanja kardinalnosti:
1..1 označava se skraćeno kao 1
0..* označava se skraćeno sa *
Ako kardinalnost atributa nije označena podrazumijeva se da je 1.

Neka validna pojavljivanja kardinalnosti atributa:

1. `telefonBroj[1..3]` – ova definicija znači da je potrebna najmanje jedna vrijednost za atribut telefonBroj, a moguće je da se dodijele do tri vrijednosti;
2. `telefonBroj[0..1]` – ova definicija znači da atribut telefonBroj može sadržavati jednu ili nijednu vrijednost;
3. `telefonBroj[1..*]` – ova definicija znači najmanje jedna vrijednost je potrebna za atribut `telefonBroj`, a moguće je specificirati neograničen broj vrijednosti.

Slijedi deklaracija klase `Jelovnik` iz domena našeg problema koja sadrži atribute različite kardinalnosti.



Slika 5.4: Kardinalnost atributa

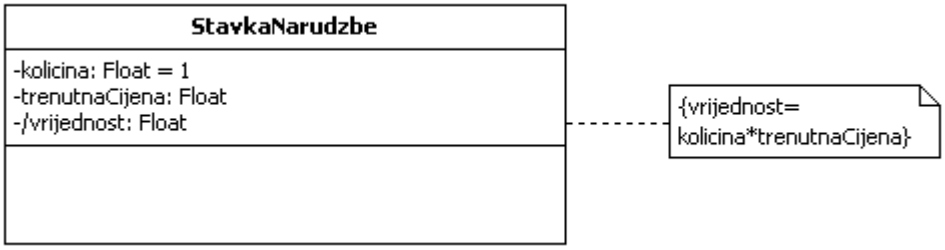
Na slici je prikazana klasa `Jelovnik` i njena dva atributa, pri čemu atribut `jelaUPonudi` ima kardinalnost 100, odnosno predstavlja niz od maksimalno 100 jela, a svako jelo je klase `Jelo`. Kôd kojim se to realizira i koji se generira na osnovu UML modela klase `Jelovnik` izgleda:

```
Java :  public class Jelovnik
        {
            ...
            private Jelo jelaUPonudi[];
            ...
            jelaUPonudi = new Jelo[100];
            ...
        }
C++ :   class Jelovnik {
                ...
        private:
                Jelo jelaUPonudi[100];
                ...
        };
```

Generirani kod za klasu `Jelovnik`

5.5 Izvedeni atributi

Vrijednosti za izvedene atribute može se odrediti na osnovu vrijednosti drugih atributa klase ili atributa drugih klasa. UML notacija za izvedeni atribut je "/" ispred imena atributa.
Klasa `StavkaNarudzbe` na slici 5.5 ima izvedeni atribut vrijednost. Komentarom je naznačeno da se njegova vrijednost izračunava preko druga dva atributa: `kolicina` i `trenutnaCijena`. U kôdu to bi se realiziralo tako da se pri svakoj promjeni jednog od atributa: `kolicina` ili `trenutnaCijena` ponovno računa i atribut vrijednost.



Slika 5.5: Klasa `StavkaNarudzbe` – izvedeni atribut

5.6 Operacije

Operacije su akcije koje klasa može obaviti. Operacijama se realizira ponašanje klasa. Sasvim je očigledno da operacije odgovaraju metodama klase. Najčešće se ne prikazuju jednostavne operacije nad osobinama.

Potpuna sintaksa operacija u okviru UML-a je:

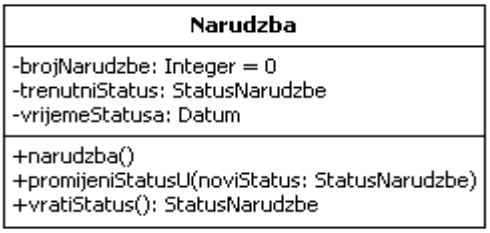
vidljivost ime (parametri) : tip_rezultata {opis_osobine}

- **Vidljivost** može biti javna (+), privatna (-), zaštićena (#) ili paketna (~) sa istim značenjem opisanim pri objašnjenju atributa. Operacije su najčešće javne (public) vidljivosti.
- **Ime** – niz znakova koji obilježava operaciju.
- **Parametri** – lista parametara operacije. Parametri se označavaju na sličan način kao i atributi: *smjer ime : tip = default_vrijednost*

Smjer ukazuje na to da li je parametar ulazni (in), izlazni (out) ili ulazno-izlazni (inout). Ako smjer nije označen, podrazumijeva se in.

Ime, tip i default vrijednost parametra imaju isto značenje i način označavanja naznačen za atribute klase.

- **Tip rezultata** je tip povratne vrijednosti operacije, ukoliko takva postoji. Može biti primitivni tip podataka ili klasa.
- **Opis osobine** ukazuje na posebna svojstva operacije i detaljnije će biti objašnjena u nastavku.



Slika 5.6: Klasa `Narudzba` – operacije

Na slici 5.6 je prikazana klasa `Narudžba` sa svojim operacijama. Operacije su:

- `+narudzba()` – konstruktor klase,

- `+promijeniStatusU(noviStatus:StatusNarudzbe)` – operacija koja mijenja atribut `statusNarudzbe` u vrijednost proslijeđenu prilikom preko parametra `noviStatus` prilikom poziva operacije,
- `+vratiStatus():StatusNarudzbe` – operacija koja vraća vrijednost atributa `statusNarudzbe` (tip rezultata: pobrojani tip),

Ovako definirane operacije bi se u kôdu realizirale na sljedeći način:

Java :
```
public class Narudzba
{
    ...
    public void narudzba()
    {
        brojNarudzbe = sljedeciBrojNarudzbe();
        ...
    }

    public void
        promijeniStatusU(StatusNarudzbe noviStatus)
    {
        trenutniStatus = noviStatus;
    }

    public StatusNarudzbe vratiStatus()
    {
        return trenutniStatus;
    }
    ...
}
```

C++ :
```
//---------------------- Narudzba.h ----------------------//

#if !defined(_NARUDZBA_H)
#define _NARUDZBA_H

#include "StatusNarudzbe.h"
...

class Narudzba {
public:
       narudzba();
       void promijeniStatusU(StatusNarudzbe noviStatus);
       StatusNarudzbe vratiStatus();
       ...
};

#endif  //_NARUDZBA_H

//---------------------- Narudzba.cpp ----------------------//

#include "Narudzba.h"
#include "StatusNarudzbe.h"
...

Narudzba::narudzba() {
        brojNarudzbe = sljedeciBrojNarudzbe();
        ...
}

void Narudzba::promijeniStatusU(StatusNarudzbe noviStatus) {
        trenutniStatus = noviStatus;
}

StatusNarudzbe Narudzba::vratiStatus() {
        return trenutniStatus;
}
...
```

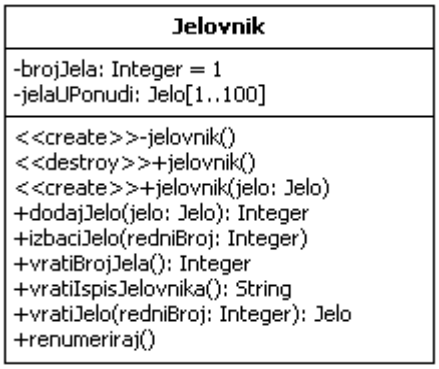
Klasa `Narudzba` sa operacijama

5.7 Stereotip oznake operacija

Nekada je korisno napraviti razliku između operacija koje mijenjaju stanje sistema i operacija koje ga ne mijenjaju. Notacija UML jezika omogućava da se operacije u klasi obilježavaju korištenjem stereotipa <<create>>, <<destroy>>, <<query>>, <<update>> i slično.

Sa stereotipom <<create>> se mogu označiti sve operacije koje mijenjaju stanje, a služe za inicijalizaciju objekata prilikom kreiranja, odnosno sve operacije koje nazivamo konstruktorima. Sa stereotipom <<destroy>> se mogu označiti sve operacije koje mijenjaju stanje, a služe za čišćenje objekata prije njihovog uklanjanja, odnosno sve operacije koje nazivamo destruktorima.

Na slici 5.7 je UML model klase sa stereotipima <<create>> i <<destroy>>.



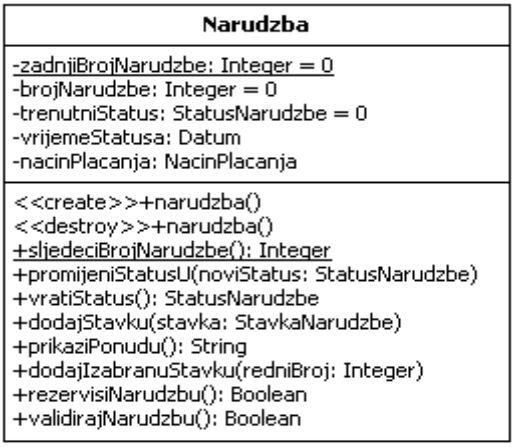
Slika 5.7: Stereotipi operacije

Većinom oznake stereotipa ovise od alata za modeliranje i od dodataka instaliranih za određene programske jezike. Ako npr. u alatu StarUML™ dodamo 'add-in' za C++ ili C# onda dobijemo mogućnost da operacije obilježimo stereotipima specifičnim za te programske jezike.

5.8 Statičke operacije i atributi

U UML-u, operacije i atributi mogu biti deklarirani kao statički elementi. Većina atributa i operacija klase nije statička, i u tom slučaju oni su povezani sa instancom odnosno objektom klase. To znači da svaki objekt neke klase (npr. `Narudzba`) ima svoje kopije atributa i operacija. Ponekad je potrebno da svi objekti pojedine klase dijele istu kopiju atributa ili

operacije. Npr. svaka instanca klase `Narudzba` treba da ima jedinstveni broj narudžbe koji će se generirati tako da svaka nova narudžba dobija broj uvećan za jedan u odnosu na prethodnu narudžbu , što je zajednička potreba svih instanci klase `Narudzba`. U tom slučaju takvi atributi i operacije se odnose na cijelu klasu, a ne na jednu instancu te klase, i zovu se statički. Statički elementi su ekvivalentni statičkim članovima u programskim jezicima. Statičke karakteristike su podvučene na ikoni klasa (slika 5.8). U klasi `Narudzba` je definiran statički atribut `zadnjiBrojNarudzbe` i statička operacija `+sljedeciBrojNarudzbe()`.



Slika 5.8: Označavanje statičkih karakteristika

Programski kôd koji odgovara ovoj klase u Javi, C++ je:

Java :
```
public class Narudzba
{
    ...
    private static int zadnjiBrojNarudzbe = 0;
    ...
    public static int sljedeciBrojNarudzbe()
    {
    }
    ...
}
```

C++ :
```
#if !defined(_NARUDZBA_H)
#define _NARUDZBA_H
...

class Narudzba {
public:
       ...
       static int sljedeciBrojNarudzbe();
       ...
private:
       ...
       static int zadnjiBrojNarudzbe = 0;
       ...
};

#endif  //_NARUDZBA_H
```

Klasa `Narudzba` sa statičkim karakteristikama

5.9 Osobine

Osobine (eng. properties) se koristi za dodavanje deskriptivnih informacija o atributima, operacijama i parametrima prikazanim modelom klasa. To su ustvari riječi sa predefiniranim značenjem i uvijek su zadnji element u definiciji atributa, operacija ili parametara. Osobine se uvijek pišu u okviru vitičastih zagrada. Nazivaju se i označene (eng. tagged) vrijednosti i bit će detaljnije objašnjene u poglavlju o UML mehanizmima proširenja.

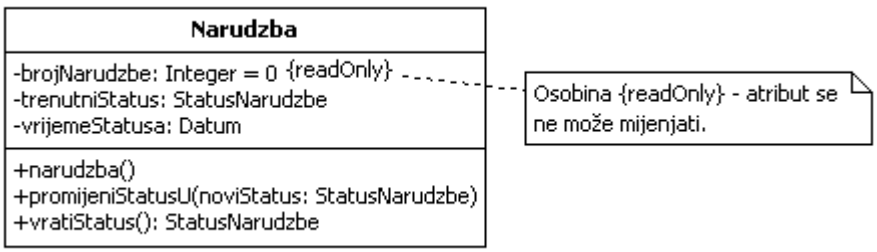
Npr: -`/starost`:Integer {readOnly}

`brojTelefona`[0..3]:String{ordered}

Postoji više predefiniranih značenja za osobine u okviru UML. Neke od njih su:

- {readOnly} - indicira da se vrijednost atributa ne može mijenjati nakon inicijalizacije.
- {union} – indicira da atribut predstavlja uniju vrijednosti drugih atributa. Npr.,/`kontaktTelefonBrojevi` {union} može se definirati kao unija `kucniTelefonBroj`, `posaoTelefonBroj` i `mobilniTelefonBroj` atributa. Ova specifična definicija može se dodati kao dio generalnog opisa atributa.
- {subsets property} - indicira da je definirani atribut limitiran na podskup vrijednosti drugog atributa.
- {ordered} – ova osobina se primjenjuje na atribute čija je mogućnost da sadrže skup podataka. Korištenje ove osobine indicira da se vrijednosti skupa podataka prikazuju u određenom redoslijedu.
- {composite} - ova osobina indicira da se atribut predstavlja kao sastavljena vrijednost na osnovu drugih vrijednosti.

UML 2.0 specifikacija ne definira eksplicitno validne osobine za operacije i parametre. Međutim, većina osobina identificiranih iznad može se primijeniti i na parametre i operacije.



Slika 5.9: Označavanje osobina

5.10 Model klase u fazi analize i dizajna

U procesu modeliranja, kreiranje dijagrama klase zahtijeva iterativno modeliranje zasnovano na seriji aktivnosti. Svaka od njih pridonosi cjelokupnoj slici i specifikaciji dijagrama klasa.

Aktivnosti se sastoje od pronalaženja i identifikacije svih klasa i njihovih veza u domeni problema, identifikaciji atributa i operacija za klase, a nakon toga i raznih struktura generalizacije.

Identificiranje objekata i klasa

Klase i njihove međusobne veze se identificiraju na osnovu informacija koje imamo prikupljene na osnovu inženjeringa korisničkih zahtjeva i specifikacije sistema. Ukoliko u procesu razvoja softverskog sistema koristimo UML u svim fazama, identifikaciju klasa, objekata i njihovih veza ćemo uraditi na osnovu dijagrama slučajeva upotrebe i opisa pripadajućih scenarija.

Jedna od tehnika na osnovu koje identificiramo klase i poslije njihove međusobne veze je identificiranje imenica. Bennett (2002) sugerira neke kategorije klasa koje treba identificirati i predlaže uočavanje sljedećih imenica u domeni problema koje će pomoći samom procesu identifikacije klasa.

Specifični slučajevi generalnog tipa kao što su ljudi ('Aldin Traljic'), organizacije ('XOsiguranje') i organizacijske jedinice ('Prodajni tim')

Strukture, stvari koje su navedene u domenu problema, kao što je 'kontrolori', 'volonterski tim'.

Apstrakcije stvari kao što su:

Ljudi i uloge: 'prodavač', 'volonter', 'student'.
Fizičke stvari: 'automobil', 'polica', 'knjiga'
Koncepti: 'prodaja', 'vještina', 'zahtjev'.

Trajne relacije između identificiranih klasa kao što je 'sporazum', 'registracije'

Glagoli i fraze izražene glagolima indiciraju asocijacije između klasa, koje se opisuju u sljedećem poglavlju.

Najčešće identifikaciju klasa radimo u dvije faze:

1. Identificiramo potencijalne kandidate za klase skupljajući sve imenice u jednini i fraze iskazane u zahtjevima sistema.
2. Odbacimo klase koje su neodgovarajuće zbog nekog razloga, kao one što predstavljaju imenice koja ne pripadaju i nisu relevantne u domeni problema, ili je u pitanju vremenska jedinica (sedmica, mjesec), ili nije dovoljno jasno samo značenje imenice.

Primjer:

Restoran nudi **jela** koja su prikazana na **jelovniku**. Kada neko naručuje, onda se za to formira **narudžba**, na kojoj se nalaze **stavke**. Stavke narudžbe su jela, s tim da je svakom jelu pridružena **količina**. Na jelovniku je uz svako jelo prikazana njegova **trenutna cijena**. Kada se narudžba zaprimi, onda ista treba da se plati u **roku** od 10 **minuta**, da bi nakon toga mogla da se proslijedi na realizaciju koja traje u ovisnosti od **vrste jela**, od 15 minuta do 45 minuta.

U primjeru su označene imenice koji su kandidati za klase. Sljedeće je **odbacivanje klasa** koje nisu dobri kandidati za klase.

- restoran, jer je izvan vidokruga sistema (to je ustvari glavni objekt sistema);
- količina, jer je samo osobina stavke narudžbe;

- trenutna cijena, jer je samo osobina stavke narudžbe;
- minuta, jer je to jedinica vremenskog perioda, nije stvar:
- rok, jer se odnosi na vrijeme koje nije jasno definirano, nije stvar;
- vrsta jela, jer nije jasno definirano, niti ima značajnog utjecaja na sistem.

Ostale su imenice koje predstavljaju klase:

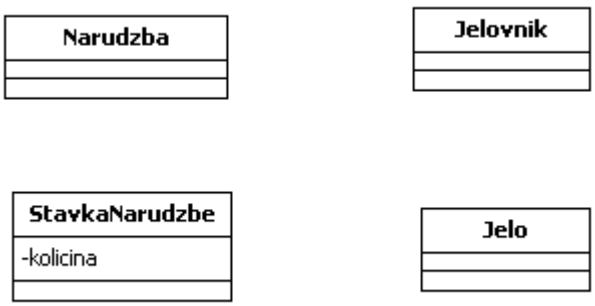
- jelo,
- jelovnik,
- narudžba,
- stavka narudžbe

Identificiranje atributa i operacija

Kao i klase, atributi i operacije se identificiraju iz raspoloživih informacijskih izvora vezanih za opis i funkcionalnost sistema. Prilikom identificiranja operacija, biraju se one koje omogućavaju klasi dodatno obavljanje akcija. Prilikom konceptualnog modeliranja često nije moguće identificirati sve operacije za klase dok se i dijagrami interakcije ne kompletiraju. Atributi koji se alociraju za klasu opisuju strukturu klase i moraju imati odgovornost za pojedinačnu stavku podatka kojeg predstavljaju.

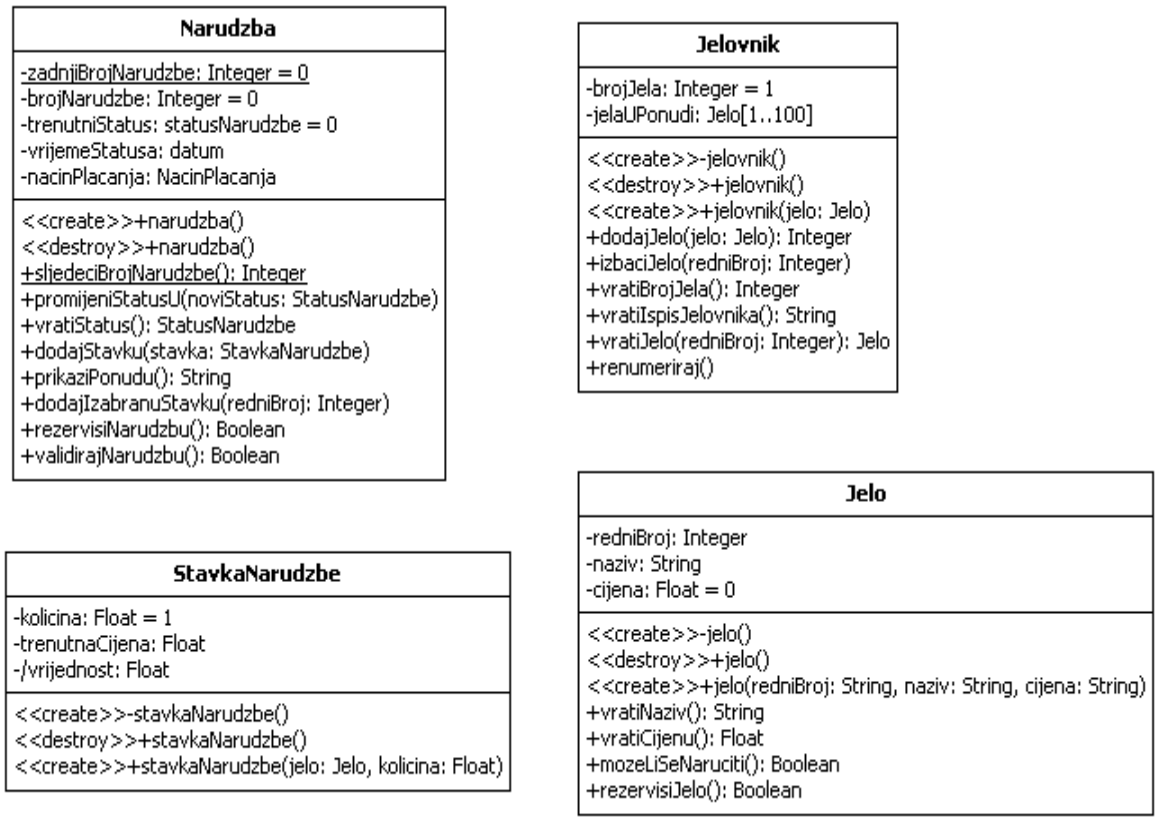
Nivo detaljnosti prikazivanja atributa i operacija varira od faze razvojnog ciklusa u kojima se dijagram klasa koristi. U fazi analize samo se generalne informacije prikazuju, jer u ovoj fazi mnogi detalji i nisu poznati. Parametri operacija još uvijek nisu jasno uspostavljeni, a i vidljivost atributa i operacija u ovoj fazi je nevažna. U fazi dizajna, nivoi detaljnosti neophodni za uspješnu implementaciju se uspostavljaju. Parametri za operacije su poznati, isto kao i tipovi parametara, vidljivost atributa i operacija je poznata i važna. Sljedeće dvije slike prikazuju neke od uočenih klasa `e-restorana` u fazi analize i u fazi dizajna.

U fazi analize klase izgledaju:



Slika 5.10: Klase u fazi analize

U fazi dizajna dodaju se detalji vezani za atribute i operacije, pa klase nakon višestrukog unapređivanja izgledaju:



Slika 5.11: Klase u fazi dizajna

5.11 Terminologija za imenovanje klasa, atributa i operacija

Klase se trebaju imenovati na zajednički prihvatljivoj terminologiji, da bi ih lakše razumjeli oni koji ih koriste.

- Za klase iz domena problema uzimaju se imena koja su bazirana na terminologiji domene. Na primjer, `Musterija`, `Narudzba`, `Isporuka`.
- Za tehničke klase imena se baziraju na tehničkoj terminologiji kao: `MessageQueue`, `ErrorLogger`.

Preporučuje se korištenje kompletne imenice za klasu. Na primjer, `Narudzba`, a ne `NarMust`, jer su one mnogo više opisne i lakše su za razumjeti. Isto tako za klase se prakticira korištenje imenice u jednini `Narudzba`, a ne `Narudzbe`.

Imenovanje operacija sa glagolima

Operacije implementiraju funkcionalnost objekta i zbog toga se trebaju imenovati glagolom koji jasno opisuje samu tu funkcionalnost. Pri tome voditi računa o sljedećim preporukama:

- koristiti punu riječ za imenovanje operacija, jer je puna riječ čitljivija i ima jasnije značenje.

- glagol pomjerati na početak riječi da bi se više naglasila operacija.

- ne koristiti specijalne znakove za razdvajanje ukoliko je operacija sastavljena od više riječi, prvu riječ početi sa malim slovima, a sve ostale sa velikim.

Primjeri dobro nazvanih operacija u fazi dizajna su `vratiStatus()`, `rezervisiJelo()`, itd., dok se u fazi analize mogu nazvati Vrati Status, Rezervisi Jelo..

Imenovanje atributa

Atributi implementiraju strukturu objekta i trebaju se imenovati sa imenicom koja je adekvatno opisuje. Pri tome treba uzeti u obzir i sljedeće preporuke, koje su slične preporukama i za imenovanja operacija:

- ne koristiti razne specijalne znakove,
- održavati ime kraće i jednostavnije za čitanje,

- ukoliko je atribut sastavljen od više riječi, prvu riječ početi sa malim slovom, a svaku drugu sa velikim slovom, jer to dovodi do lakše čitljivosti,
- konzistentno imenovati srodne atribute.

Primjeri dobro nazvanih atributa u fazi dizajna su na primjer `redniBroj`, `trenutnaCijena`, dok se u fazi analize mogu imenovati kao `Redni Broj`, `Trenutna Cijena`.

O vrstama veza između klasa, načinu uočavanja i imenovanja istih u sljedećem poglavlju.

Pitanja za ponavljanje

1.Koja je osnovna namjena dijagrama klasa?
2. Kojem pogledu na sistem pripada dijagram klasa?
3.Koji je osnovni model element dijagrama klasa?
4.Kako se definira klasa?
5.Koja je osnovna notacija za klasu?
6.Što su atributi klase?
7.Koja je osnovna notacija atributa klase?
8.Objasniti pojam kardinalnost atributa?
9. Koja je notacija za prikazivanje kardinalnosti atributa?
9.Objasniti vidljivost i notaciju za vidljivost atributa?
10. Što su izvedeni atributi?
11.Što su operacije klase?
12. Koja je osnovna notacija operacija klase?
13.Kako se može specificirati inicijalna vrijednost za atribute?
14.Što se postiže sa označavanjem operacija sa stereotipima?
15. Koja je osnovna uloga statičkih atributa i operacija?
16. Što se postiže sa {readOnly}osobinom?
17.Definirajte i ilustrirajte kako faze projekta utiču na prikazivanje nivoa detaljnosti klasa.
18.Objasniti jedan od načina identificiranja klasa.
19. Navesti osnovne preporuke za terminologiju imenovanja klasa.
20. Navesti osnovne preporuke za terminologiju imenovanja atributa i operacija klase.

POGLAVLJE 6.

DIJAGRAM KLASE:VEZE IZMEĐU KLASA

U okviru ovog poglavlja objašnjavaju se veze između klasa koje se odnose na asocijaciju, agregaciju, kompoziciju, ovisnost i generalizaciju.

6.1 Asocijacija

Asocijacije izražavaju veze između klasa. Postoje instance asocijacije, upravo kao i instance klase. Instanca asocijacije specificira da su objekti jedne vrste spojeni sa objektima druge vrste.

Klasa A i klasa B su u relaciji asocijacije ako:

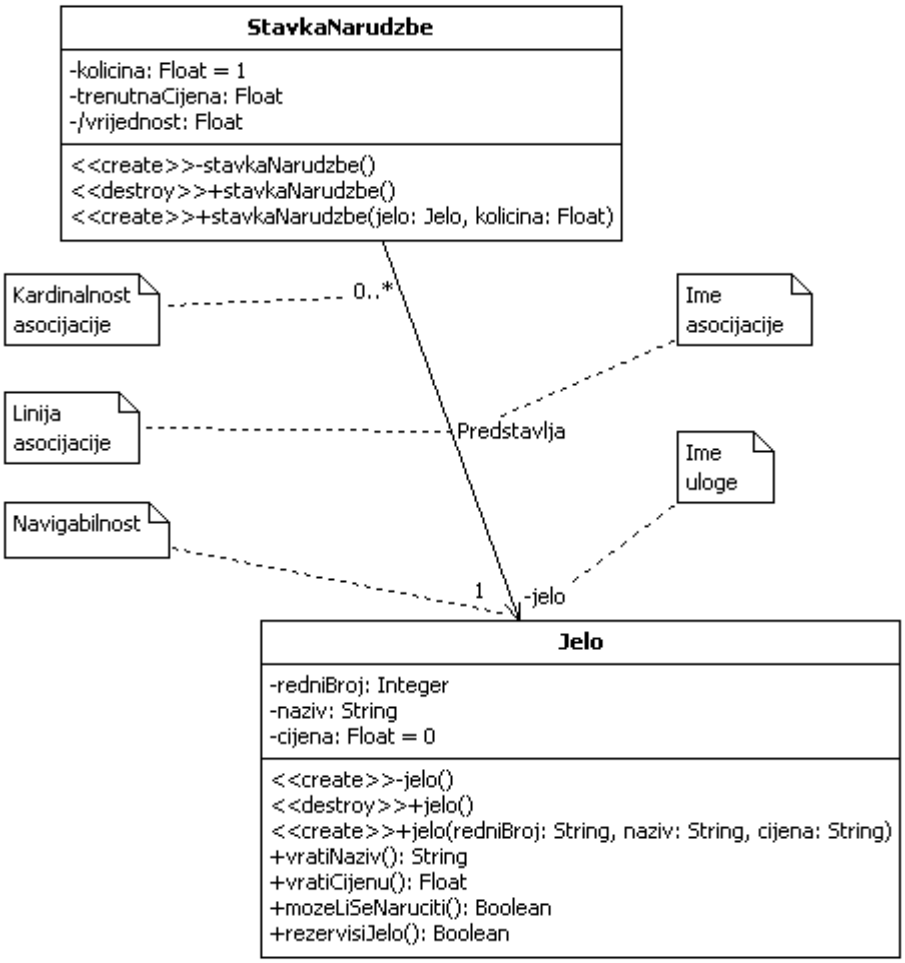
- objekt klase A šalje poruku objektu klase B,
- objekt klase A kreira objekt klase B,
- objekt klase A ima atribute čije vrijednosti su objekti klase B ili kolekcije objekata klase B,
- objekt klase A dobiva poruke sa argumentom koji je objekt klase B.

Ukratko relacija asocijacije se pojavljuje, ako neki objekt klase A zna nešto o objektu klase B.

Asocijacija između dvije klase se prikazuje linijom koja spaja dvije klase što je i vidljivo na slici 6.1.

6.2 Ime asocijacije

Dobro je dati asocijaciji ime koje označava prirodu asocijacije. Ime asocijacije se prikazuje kao labela na liniji asocijacije.



Slika 6.1: Primjer asocijacije između klasa

Između klasa `StavkeNarudžbe` i `Jelo` postoji asocijacija jer svaka stavka na narudžbi predstavlja, zapravo, jelo koje je prikazano sa klasom `Jelo`.

Mnoge informacije o asocijaciji dodaju se i na njene krajeve. Krajevi asocijacije mogu označavati navigabilnost, imati naziv i vidljivost uloge asocijacije, kao i kardinalnost.

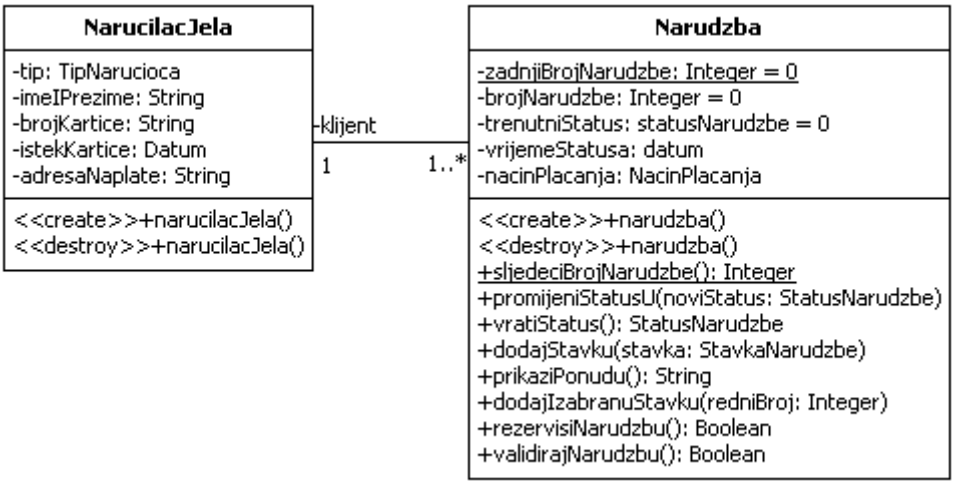
6.3 Navigabilnost

Ako imamo relaciju asocijacije kao na slici 6.1 tada je moguća navigacija od objekta jedne vrste prema objektu druge vrste. Međutim, postoje situacije u kojima želimo da limitiramo navigaciju samo u jednom smjeru. U primjeru na slici 6.1 postoji samo potreba da

`StavkaNarudzbe` koristi `Jelo`, ali ne i obratno. U tom slučaju se eksplicitno predstavlja direkcija navigacije sa strelicom koja pokazuje na smjer navigacije. Ako ne postoji eksplicitno označen smjer navigacije tada je asocijacija u oba smjera moguća.

6.4 Dvosmjerne asocijacije

Dvosmjerna asocijacija (eng. bidirectional) se može prikazati i sa strelicama na oba kraja asocijacija i bez strelica (slika 6.2). Alati za modeliranje obično pri uključivanju navigabilnosti na oba kraja ne prikazuju strelicu ni na jednom kraju.



Slika 6.2: Dvosmjerna asocijacija

6.5 Uloga asocijacije

Ponekad je teško odrediti ime asocijacije da bi se dovoljno jasno opisala njena namjena. UML daje i alternativu koja se može koristiti umjesto imena asocijacije ili sa njim da bi se dodatno objasnio razlog postojanja asocijacije. Ova alternativa se naziva uloga (eng. role) jer opisuje kako objekt učestvuje u asocijaciji. Svaka uloga je postavljena na kraju asocijacije u blizini objekta za kojeg se i veže.

Postoji još jedna značajna stvar vezana za uloge, a to je da generatori kôda na osnovu njihovih naziva daju nazive atributu koji se generira u klasi nasuprot kraja te uloge. Na UML dijagramu klase na slici 6.1, postoji naveden naziv uloge klase `Jelo` u klasi

`StavkaNarudzbe`. Na osnovu toga će u klasi `StavkaNarudzbe` biti generiran privatni atribut naziva `jelo`, klase `Jelo`. Generatori dozvoljavaju i da se generiraju atributi za uloge, čak i ako nema specificiranog naziva, ali je bolja praksa navoditi naziv uloge za koju želimo definirati atribut.

Generirani kôd za sliku 6.1 gdje je uočena asocijacija čija je uloga obilježena sa `jelo`.

```
Java :  public class StavkaNarudzbe
        {
            ...
            private Jelo jelo;
            ...
        }
C++ :   class StavkaNarudzbe {
                ...
        private:
                Jelo jelo;
                ...
        };
```

Generirani kod za asocijaciju klasa `Narudzba` i `Jelo`

U slučaju dvosmjerne asocijacije, kao što je to u primjeru na slici 6.2, za obadva kraja asocijacije, koja mogu, ali i ne moraju imati ime uloge, generira se atribut u klasi koja je na drugom kraju asocijacije. Ako se odlučimo da dozvolimo da se generira kôd i za kraj na kojem nije naveden naziv uloge, taj atribut će imati neki generički naziv kao što je npr. "`Unnamed1`".

```
Java :  public class Narudzba
        {
            ...
            private NarucilacJela narucilac;
            ...
        }
        ...
        public class NarucilacJela
        {
            ...
            public Narudzba Unnamed1;
            ...
        }
C++ :   class Narudzba {
                ...
        private:
                NarucilacJela narucilac;
                ...
        };

        class NarucilacJela {
                ...
        public:
                Narudzba Unnamed1;
                ...
        };
```

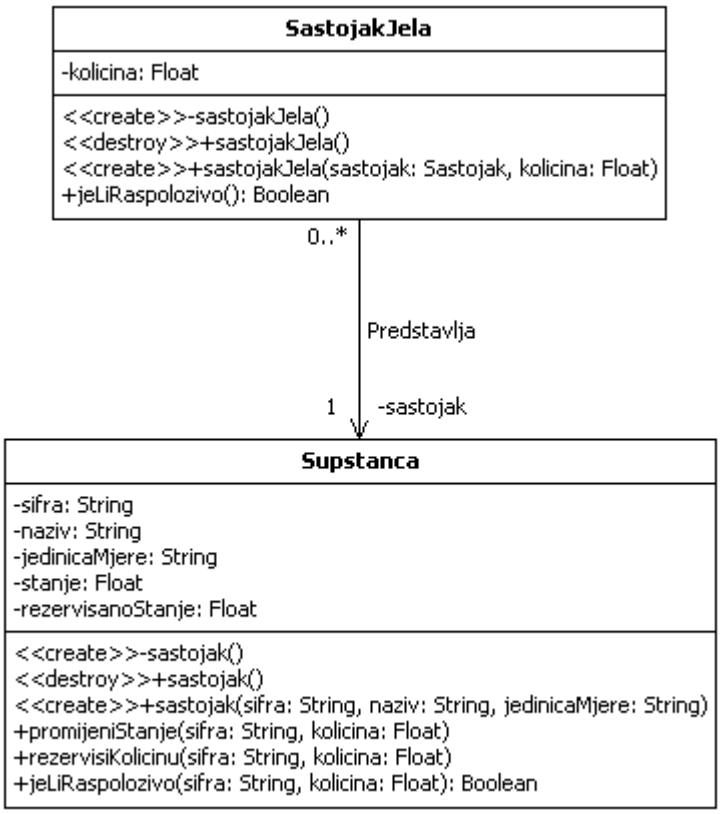
Generirani kod za dvosmjernu asocijaciju klasa `Narudzba` i `NarucilacJela`

Ista klasa može imati neku drugu ulogu u asocijaciji sa nekom drugom klasom.

Ime uloge može imati prefiks indikator vidljivosti sa nama već poznatim simbolima -, +, # i ~ za privatnu, javnu, zaštićenu i paketnu vidljivost. Indikator vidljivosti je vodič dizajneru kako da implementira asocijaciju. Na slici 6.1 za uloga je označena sa `–jelo`, pa je generirani atribut u C++ klasi `Jelovnik private: Jelo jelo`.

6.6 Kardinalnost

Sljedeće što možemo dodati na asocijaciju je kardinalnost. Kardinalnost koja se primjenjuje na atribute već je diskutirana u prethodnom poglavlju. Označavanje kardinalnosti asocijacije je na isti način kao označavanje kardinalnosti atributa sa donjom i gornjom granicom. Vezano za asocijaciju, kardinalnost indicira broj instanci objekata klase na jednom kraju asocijacije koji su povezani za instancu klase na drugom kraju asocijacije.



Slika 6.3: Kardinalnost asocijacije

Na slici 6.3 možemo vidjeti primjer asocijacije između klasa `Supstanca` i `SastojakJela`. Kada razmišljamo o kardinalnosti pojavljuje se problem kako je ispravno odrediti. Ako se stavimo u poziciju klase `Supstanca` i pitamo:

- Mora li se `Supstanca` naći u bar jednom jelu? (Ne → kardinalnost 0)
- Može li se `Supstanca` naći u više od jednog jela? (Da → kardinalnost *)

Tako dođemo do odgovora koja je kardinalnost, gledano iz perspektive klase `Supstanca` i istu pišemo na kraj koji je na drugom kraju asocijacije u odnosu na klasu `Supstanca`.

Zatim ponovimo postupak postavljajući se u poziciju klase `SastojakJela`:

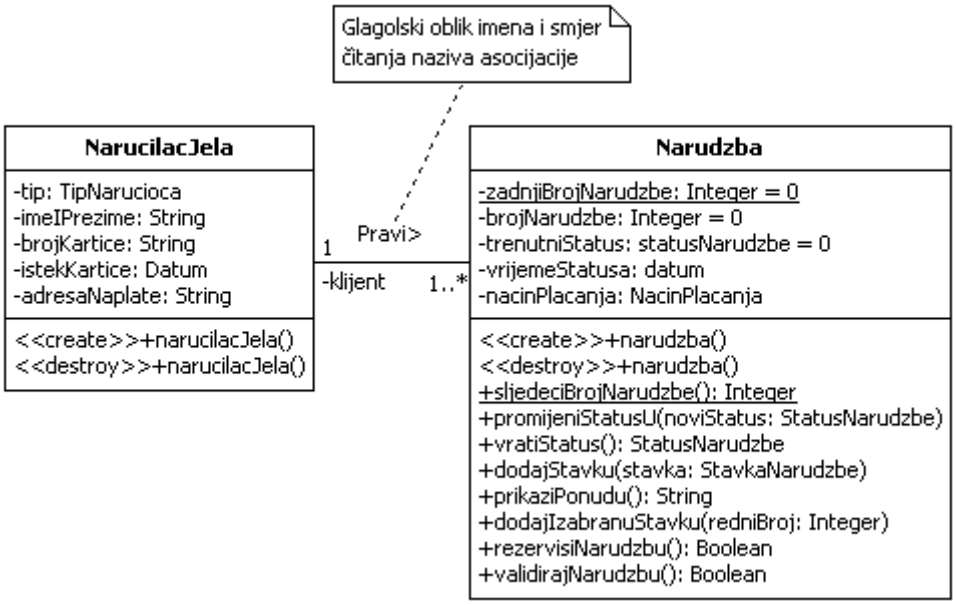
- Mora li `SastojakJela` biti jedna supstanca? (Da → kardinalnost 1)
- Može li `SastojakJela` biti sastavljen od više supstanci? (Ne → kardinalnost 1)

Tako dođemo do odgovora da je kardinalnost, gledano iz perspektive `SastojakJela`, jednaka 1 i istu pišemo na kraj koji je na drugom kraju asocijacije u odnosu na klasu `SastojakJela`.

Ako je na dijagramu klase izostavljena kardinalnost, to znači da ista nije poznata. To je različito od kardinalnosti atributa, gdje ako je kardinalnost izostavljena pretpostavlja se da je jedan. Ukoliko je od bilo kakve važnosti kardinalnost asocijacije, ona se treba i izričito naglasiti.

6.7 Glagolski oblik obilježavanja asocijacije

Da bi se prilikom tumačenja raznih vrsta asocijacije izbjegla dvosmislenost, neki projektanti obilježavaju asocijaciju glagolskim oblikom (slika 6.4), tako da se ime veze može koristiti u rečenici koja opisuje samu asocijaciju. Uz ime asocijaciji možemo dodijeliti i strelicu da bi tumačenje asocijacije bilo još jasnije. Naime, slijed čitanja je obično s desna na lijevo i odozgo prema dolje, pa ako postoji mogućnost da čitalac dođe u zabunu kako da pročita asocijaciju, preporučuje se stavljanje strelice u nazivu asocijacije. Puno praktičnije je nastojati da asocijacije ne budu dvosmjerne, pa da smjer asocijacije određuje i način čitanja. U primjeru na slici 6.4 bi asocijaciju pročitali na način: Objekt koji je tipa klase `NarucilacJela` **pravi** objekt narudžbu koji je tipa klase `Narudzba`.

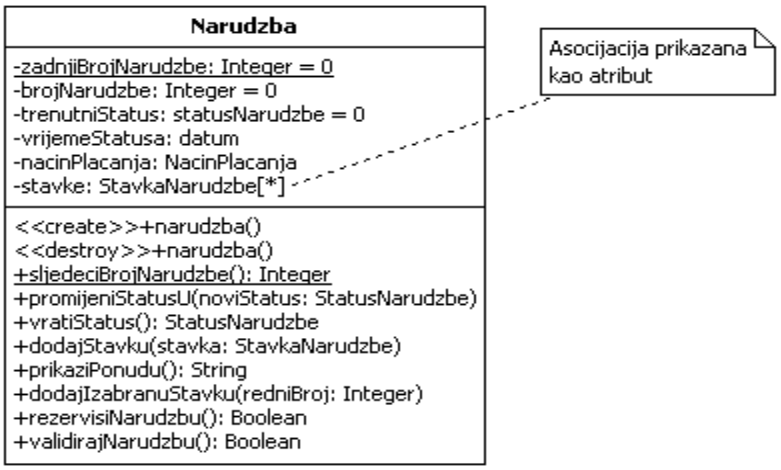


Slika 6.4: Upotreba glagolskog oblika za imenovanje asocijacije

6.8 Atributi i asocijacija

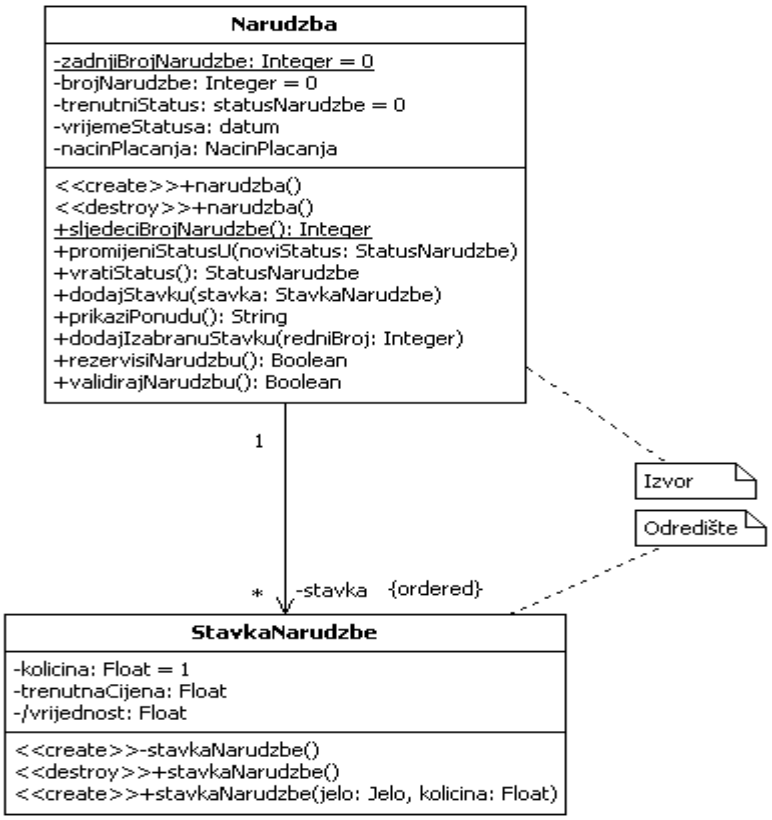
Drugi način označavanja asocijacije je prikaz asocijacije pomoću atributa klase.

Na slici 6.5 prikazane je asocijacija pomoću atributa. Uz atribut se navodi tip atributa i kardinalnost u okviru zagradama [ ].



Slika 6.5: Asocijacija prikazana kao atribut

Ekvivalentan način prikaza asocijacije sa slike 6.5 je na slici 6.5. Na toj slici se može uočiti nama već poznata notacija asocijacije te da je asocijacija usmjerena od izvorne ka odredišnoj klasi, na kojoj je ime osobine i njena kardinalnost. Odredišna klasa određuje tip osobine.



Slika 6.6: Osobina klase prikazana kao asocijacija

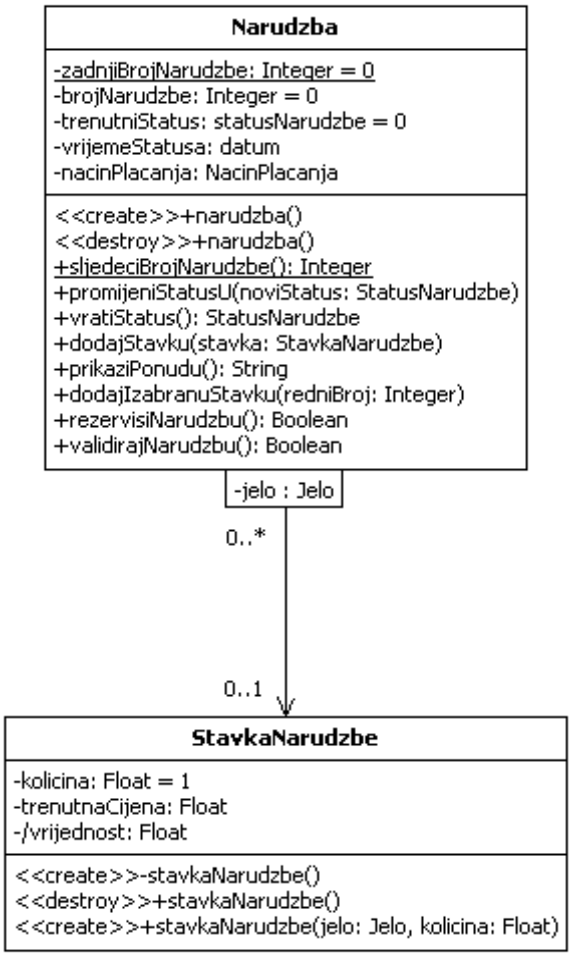
Asocijacija prikazana na gornje dvije slike označava asocijaciju između klasa `Narudzba` i `StavkaNarudzbe` sa značenjem da jedna narudžba može ima više vezanih stavki.

Iako se većina informacija može prikazati na obadva načina, neki elementi se razlikuju. Konkretno, kardinalnost asocijacije može biti definirana na oba kraja linije.

Pošto postoje dva načina označavanja iste stvari, postavlja se pitanje izbora. Uobičajeni pristup je prikazivati osobine kao atribute ukoliko su njihovi tipovi elementarni tipovi podataka, a asocijacije ukoliko je tip osobine klasa.

6.9 Asocijacije opisane kvalifikatorom

Asocijacija opisana kvalifikatorom (eng. qualified association) UML jezika ekvivalentna je raznim pojmovima u programskim jezicima kao što su: asocijativni nizovi, mape, heš tabele. Na slici 6.7 je prikazana upotreba kvalifikatora da bi se opisala asocijacija između klasa `Narudzba` i `StavkaNarudzbe`. Taj kvalifikator označava da jedna instanca klase `Narudzba` može biti povezana sa jednom instancom klase `StavkeNarudzbe` za svaki objekt tipa `Jelo`.



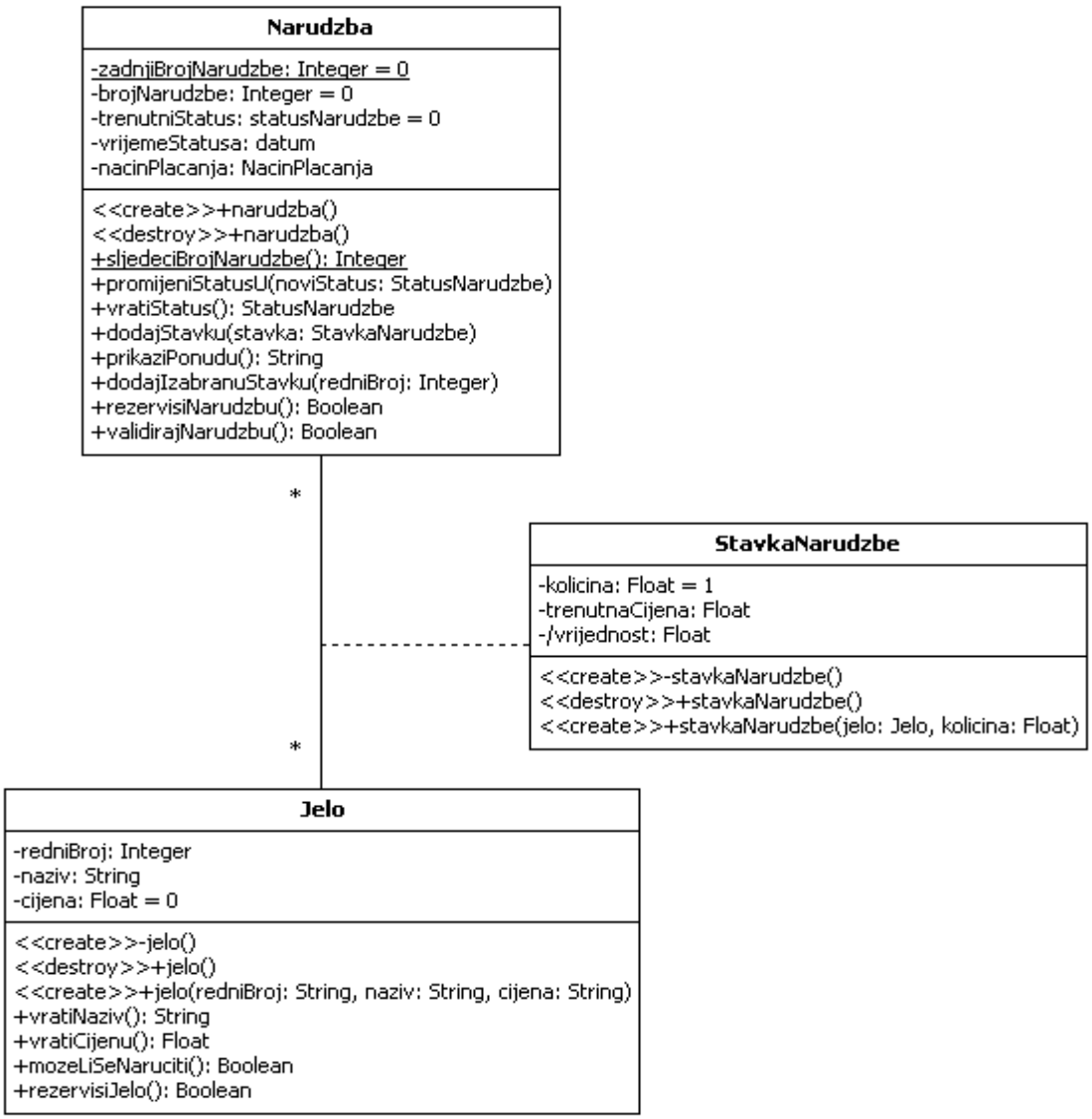
Slika 6.7: Asocijacija opisana kvalifikatorom

Treba obratiti pažnju da se kardinalnost asocijacije posmatra u odnosu na kvalifikator. Na slici 6.7 , može se vidjeti da narudžba ima ili jednu ili nijednu (0..1) stavku, ali po jelu koje je prikazano klasom `Jelo`.

6.10 Klasa asocijacija

Ponekad asocijacija između dvije klase uključuje i nove klase koje se nazivaju klase asocijacije. Klase asocijacije su korisne kada želimo pokazati da je neka klasa povezana sa dvije klase koje imaju relaciju asocijacije jedna sa drugom.

Klasa asocijacija omogućava da asocijacijama između klasa, dodajemo atribute, operacije i druge karakteristike, kao što je to prikazano na slici 6.8. Na osnovu prikazanog dijagrama možemo zaključiti da jedna narudžba može sadržavati više jela, te da svako jelo može biti na više narudžbi. Za svaku vezu između narudžbe i jela moguće je definirati i neke dodatne atribute kao što je recimo količina jela na narudžbi (npr. sezonska salata – 2 komada). Ova informacija se ne može smjestiti niti u klasu `Narudzba`, niti u klasu `Jelo`.



Slika 6.8 : Klasa asocijacije

Klase prikazane u Java i C++ koje bi odgovarale slici 6.8 su:

```
Java :  public class Narudzba
        {
            ...
            private Jelo jela[];
            private StavkaNarudzbe stavke[];
            ...
        }
        ...
        public class StavkaNarudzbe
        {
            ...
        }
        ...
        public class Jelo
        {
            ...
            private Narudzba narudzbe[];
            ...
        }
```

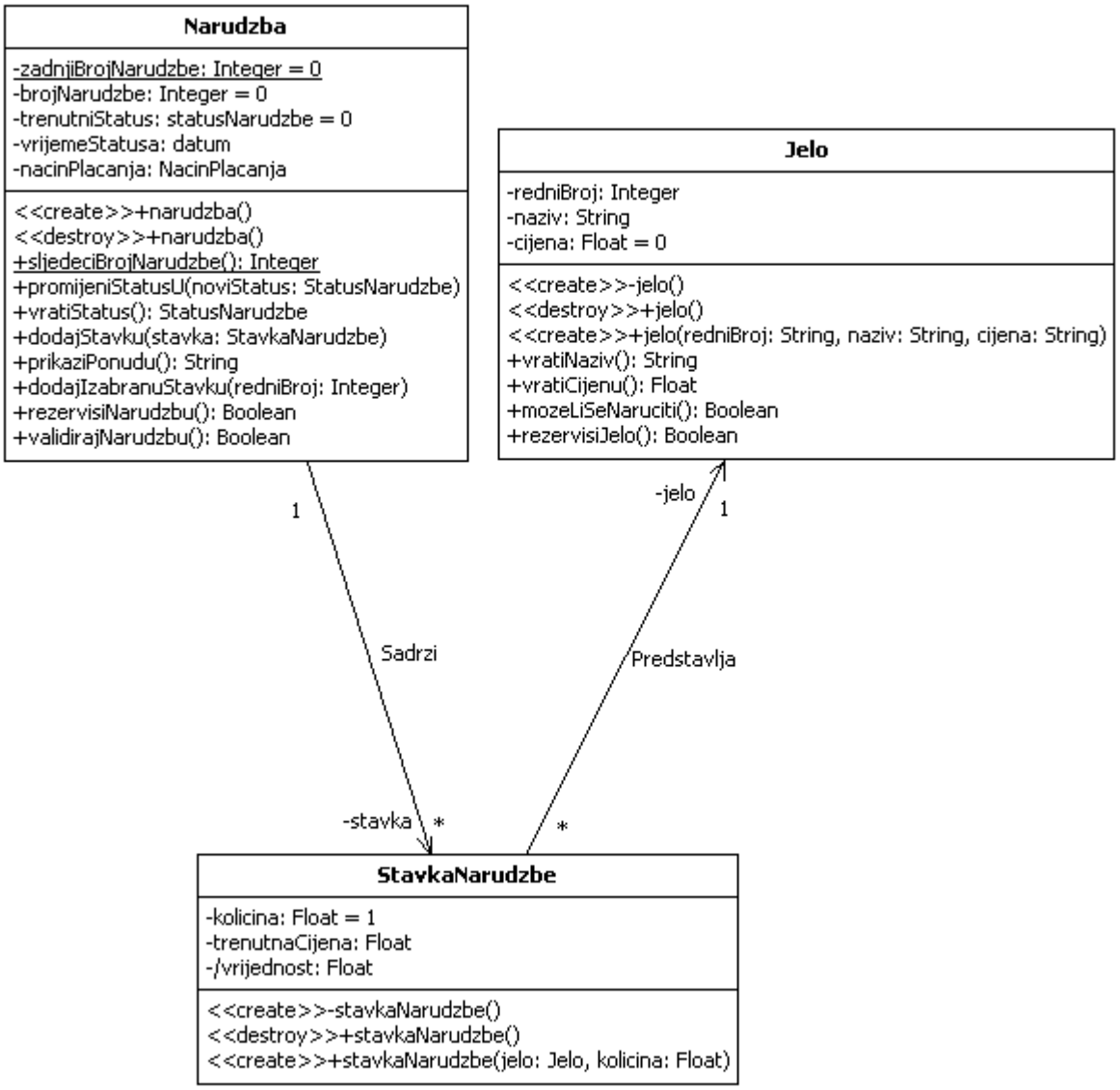
```
C++ :   class Narudzba {
                ...
        private:
               Jelo jela[];
               StavkaNarudzbe stavke[];
               ...
        };
        ...
        class StavkaNarudzbe {
               ...
        };
        ...
        class Jelo {
               ...
        private:
               Narudzba narudzbe[];
               ...
        };
```

Generirani kod za klase `Narudzba` i `Jelo` povezane sa klasom `StavkaNarudzbe`

U navedenom primjeru kôda treba zapaziti da je u klasi `Narudzba` deklariran atribut jela, ali i da je deklariran i atribut stavke koji je tipa klase `StavkaNarudzbe` i koji je generiran na osnovu klase asocijacije `StavkaNarudzbe`, koja je povezana sa klasom `Narudzba`. Klasa `Jelo` može imati atribut klase `StavkaNarudzbe`, ali za tim nema praktične potrebe.

U samoj klasi asocijacije `StavkaNarudzbe` nema potrebe da se referenciraju klase između kojih je asocijacija jer je smisao da klase `Narudzba` i `Jelo` koriste klasu `StavkaNarudzbe`, a ne obratno.

Na slici 6.9, ova informacija je predstavljena na drugi način: klasa asocijacije `StavkaNarudzbe` je sada postala samostalna, potpuna klasa. Treba obratiti pažnju na promjene vezane uz kardinalnost.



Slika 6.9: Unapređivanje klase asocijacije u klasu

Generatori kôda bolje prihvaćaju potpune klase i generiraju potpuniji kôd za njih.

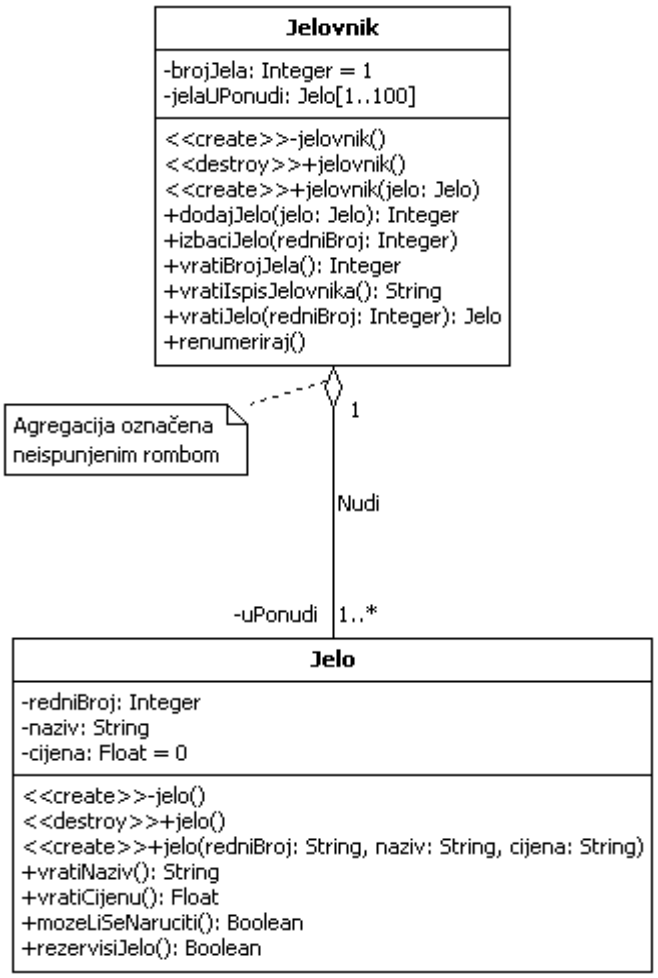
6.11 Agregacija i kompozicija

Agregacija i kompozicija su vrste asocijacije. I agregacija i kompozicija su načini predstavljanja da je objekt jedne klase dio objekta druge klase i obadvije predstavljaju odnos između cjeline i njenih dijelova i predstavljaju jače verzije asocijacije.

Agregacija

Agregacija kao vrsta asocijacije se koristi da se indicira da klasa pored vlastitih atributa, može uključivati, u ovisnosti od kardinalnosti, određen broj instanci drugih klasa.

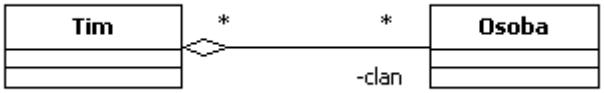
Termin cjelina-dio nekad se koristi kada se misli na agregaciju. Notacija agregacije je neispunjeni romb koji ide na kraj cjeline a ne dijela. Na slici 6.10 a je prikazan primjer agregacije.



Slika 6.10 a: Agregacija

Na slici 6.10 a je prikazana relacija agregacije koja prikazuje da jedno jelo može biti član više jelovnika. Osim toga klasu `Jelo` koriste i druge klase, tako da ona nije isključivo vezana za klasu `Jelovnik`.

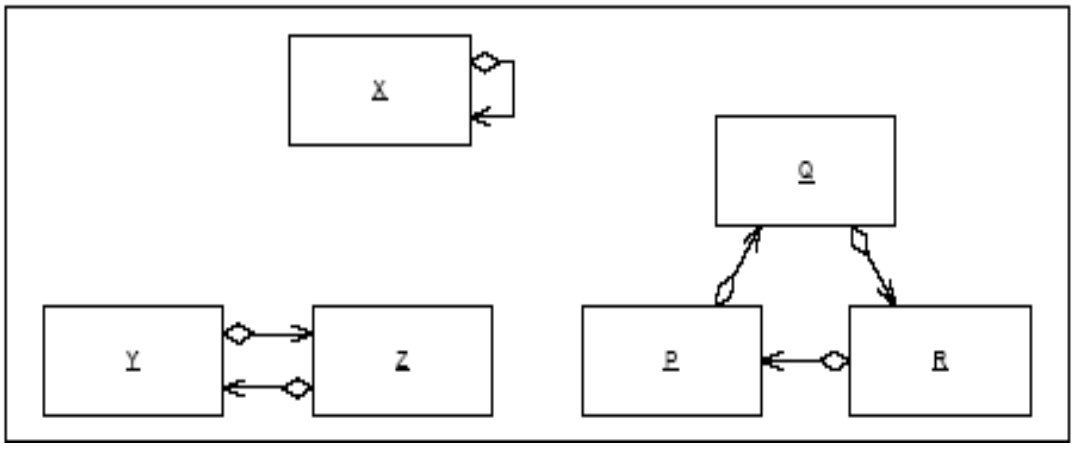
Dobar primjer agregacije je i između tima i osoba, pri čemu je tim cjelina i može se sastojati od više osoba. Jedna osoba može biti član više timova. Broj osoba može se smanjivati ili povećavati, ali će cjelina tim i dalje postojati.



Slika 6.10 b:Agregacija

Agregacija je u biti konceptualna notacija: agregacija u modelu klasa nam pomaže da razumijemo veze između klasa na neformalnom nivou, ali nam ne daje precizne formalne informacije kako se treba implementirati. Ovaj tip veze između klasa je ponekad teško razumjeti, i ispravno primijeniti u radu.

Klasa ne može imati agregaciju na sebe samu. Također nisu dozvoljene niti ciklične agregacijske forme među instancama klasa poput ovih sa naredne slike:

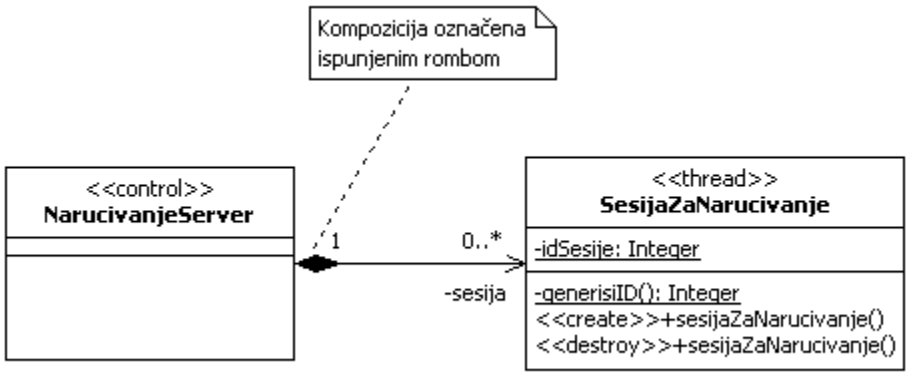


Slika 6.11: Nekorektna upotrebe agregacije

Kompozicija

Osim agregacije, UML obuhvaća preciznije definirani pojam kompozicije, koji uključuje neke dodatne restrikcije. U asocijaciji kompozicije, cjelina strogo posjeduje svoje dijelove, ako se cjelina kopira ili briše, njeni dijelovi se kopiraju ili brišu sa njima što nije slučaj kod agregacije. Da bi se ovo implementiralo, asocijacija mora biti navigabilna od cjeline prema dijelovima. Kardinalnost na kraju cjeline kompozicije mora biti 1 ili 0..1 – a dio može biti dio samo jedne cjeline. Kompozicija se isto prikazuje kao i agregacija osim što je romb ispunjen.

Primjer na slici 6.12 prikazuje klasu `NarucivanjeServer` koja tokom svog rada kreira sesije - instance klase `SesijaZaNarucivanje`, koja predstavlja nit izvršenja programa (stereotip <<thread>>). Dok postoji aplikacija, odnosno `NarucivanjeServer` klasa mogu postojati i njene niti.



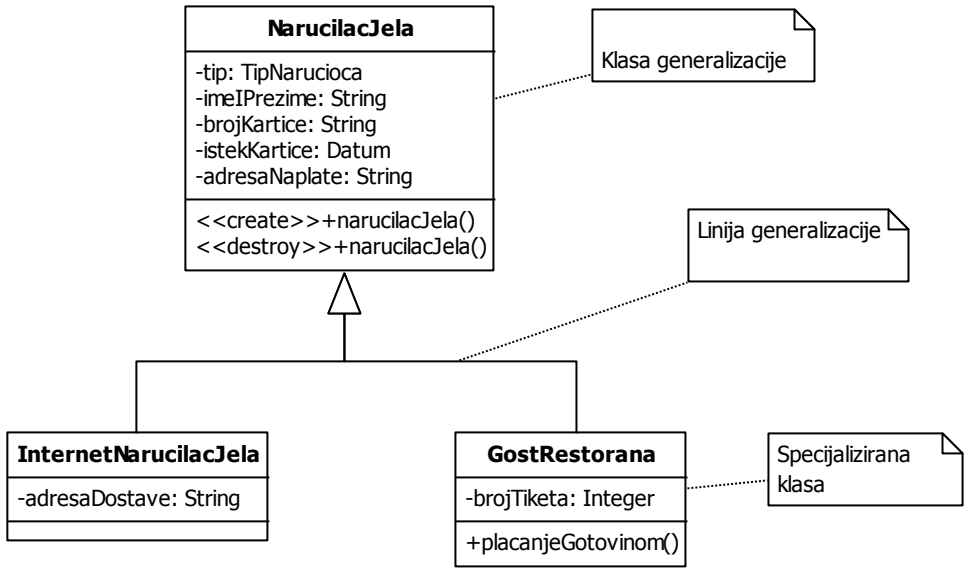
Slika 6.12: Kompozicija

Osnovno pravilo kompozicije je da nema dijeljenja. Agregacija i kompozicija spadaju u najčešće izvore nejasnoća na UML dijagramima.

6.12 Generalizacija

Druga važna veza koja može postojati između klasa je generalizacija. Generalizacija je poznata i kao nasljeđivanje u objektno orijentiranim programskim jezicima. Generalizacija se koristi da opišu da je jedna klasa tip neke druge klase. U UML-u strelica generalizacije, puna neispunjena strelica, koristi se da pokaže da je klasa tip druge klase kao na slici 6.13.

Za `e-restoran` možemo uočiti dvije različite vrste naručioca jela, a to su gosti restorana i naručioca jela preko Interneta. Oni imaju većinu zajedničkih atributa, ali ipak i nekih razlika. Na primjer gost restorana može platiti gotovinom, za Internet naručioca jela je bitna adresa dostave. U tu svrhu je najbolje da se kreira klasa `NarucilacJela` na osnovu koje se mogu izvesti specijalizirane klase `GostRestorana` i `InternetNarucilacJela`.



Slika 6.13: Generalizacija između klasa

Implementacija generalizacije u programskim jezicima:

Java :
```
public class NarucilacJela
{
    private TipNarucioca tip;
    private String imeIPrezime;
    private String brojKartice;
    private Datum istekKartice;
    private String adresaNaplate;
    public void narucilacJela()
    {

    }
}
...
public class GostRestorana extends NarucilacJela
{
    private Integer brojTiketa;
}
...
public class InternetNarucilacJela extends NarucilacJela
{
    private String adresaDostave;
}
```

C++ :
```
...
class NarucilacJela
{
public:
    narucilacJela();
    ~narucilacJela();
private:
    TipNarucioca tip;
    char imeIPrezime[];
    char brojKartice[];
    Datum istekKartice;
    char adresaNaplate[];
};
...
class GostRestorana : public NarucilacJela
{
private:
    int brojTiketa;
};
...
class InternetNarucilacJela : public NarucilacJela
{
private:
    char adresaDostave[];
};
```

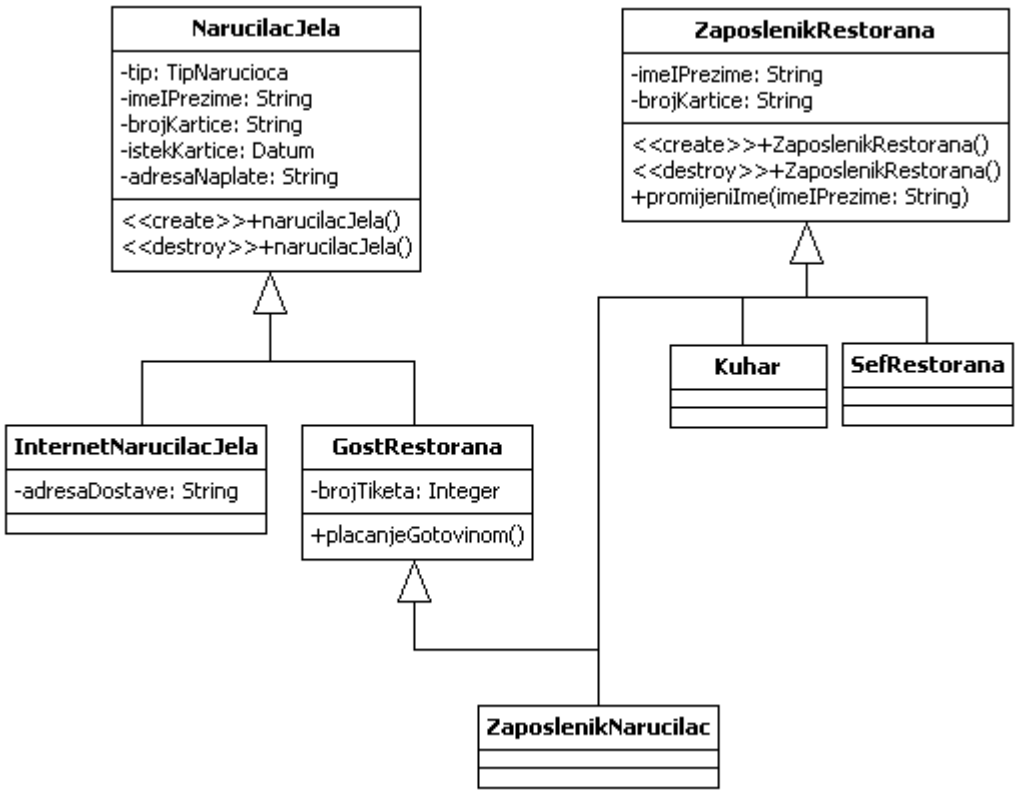
Generirani kod za klase `NarucilacJela`, `GostRestorana` i `InternetNarucilacJela` povezane generalizacijom

6.13 Višestruka generalizacija

Višestruka generalizacija ili višestruko nasljeđivanje u UML terminologiji se dešava kada jedna klasa nasljeđuje osobine od dvije ili više roditelj-klasa.

Zamislimo situaciju da trebamo u okviru sistema `e-restoran` riješiti i proces ishrane zaposlenika, odnosno omogućiti da zaposlenik restorana, u vrijeme pauze za jelo, naruči

hranu na način kako to radi gost restorana, a da ipak ne mora platiti jer je on zaposlenik. U tom slučaju trebamo sve ono što može jedan gost restorana, ali i ono što može jedan zaposlenik. Rješenje za to je višestruko nasljeđivanje kao na slici ispod.



Slika 6.14: Višestruko nasljeđivanje

Iako je višestruko nasljeđivanje podržano u UML-u, u većini slučajeva se pokazuje da nije najbolja praksa koristiti ga. Razlog za to je kompliciranje modela koji se uvodi na ovaj način. Na slici 6.14 vidimo da klasa `ZaposlenikNarucilac` nasljeđuje sve atribute i ponašanja od klasa `GostRestorana` i `ZaposlenikRestorana`, što nije problem ako su to različiti atributi i ponašanja, ali često se dešava da ima istih. Npr. atribut `imeIPrezime` je i u klasi `GostRestorana`, ali i u `ZaposlenikRestorana`. Slično je i sa operacijom `promijeniIme(imeIPrezime: String)`. Postavlja se pitanje: Iz koje od klasa naslijediti atribut ili operaciju, jer je ovo konfliktna situacija. Odgovor na ovo pitanje je skriven u detaljima implementacije. Programski jezik C++ podržava višestruko nasljeđivanje i ima

skup pravila kojima rješava ovakve konflikte. S druge strane Java i C# ne podržavaju koncept višestrukog nasljeđivanja.

Implementacija UML višestruke generalizacije u programskim jezicima je:

```
Java :
        nema podršku za višestruko nasljeđivanje
```

```
C++ :   #if !defined(_ZAPOSLENIKNARUCILAC_H)
        #define _ZAPOSLENIKNARUCILAC_H

        #include "GostRestorana.h"
        #include "ZaposlenikRestorana.h"

        class ZaposlenikNarucilac :
                    public GostRestorana, public ZaposlenikRestorana
        {
        };

        #endif  // _ZAPOSLENIKNARUCILAC_H
```

Implementacija višestruke generalizacije

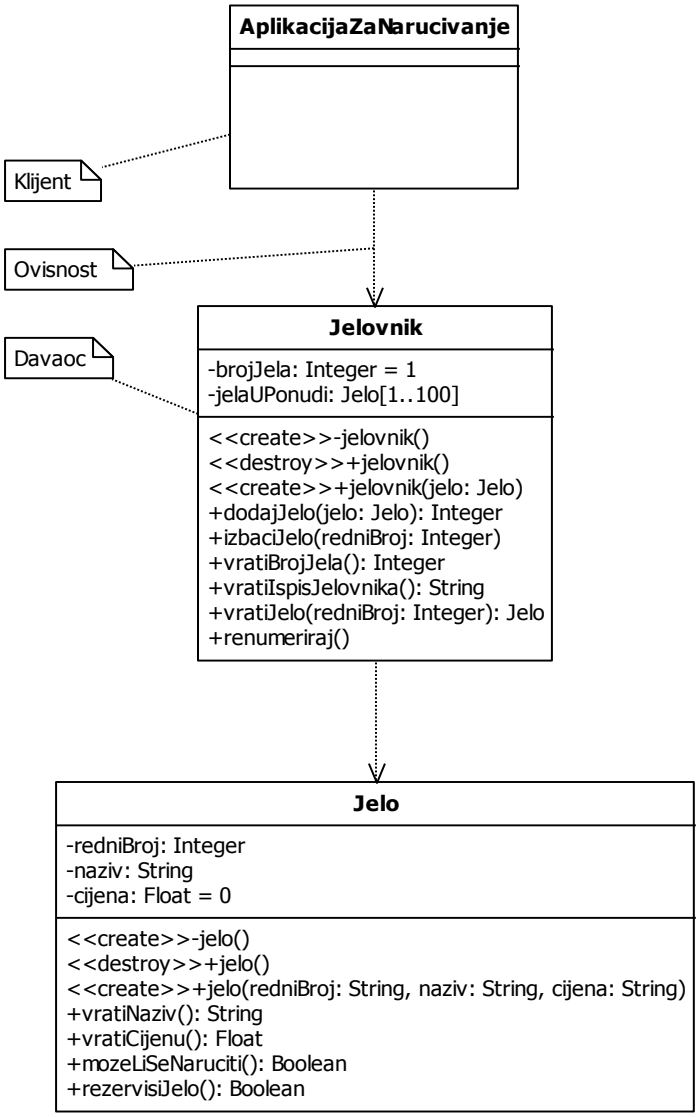
6.14 Ovisnost

Između dva elementa postoji ovisnost (eng. dependency) ako promjene u definiciji jednog elementa, davaoca, mogu izazvati promjene drugog elementa, klijenta. Ovisnost između klasa se označava isprekidanom linijom. Postoje različiti razlozi ovisnosti među klasama: jedna klasa šalje poruku drugoj klasi; jedna klasa sadrži drugu klasu; objekt jedne klase prosljeđuje objekt druge klase kao parametar neke operacije. Ako se promijeni interfejs klase, može se dogoditi da poruke poslane toj klasi ne budu više ispravne. Ako ovisnosti izmaknu kontroli, svaka promjena se prenosi na ostale elemente modele koji se također moraju mijenjati. Što više elemenata treba promijeniti, to je promjena teža.

Slika 6.15 prikazuje neke ovisnosti koje se mogu nalaziti u višeslojnoj aplikaciji. Klasa `AplikacijaZaNarucivanje` jeste klasa korisničkog okruženja i ovisi od klase `Jelovnik` koja je klasa domene problema. To znači da bi klasu `AplikacijaZaNarucivanje` trebalo promijeniti ako se promjeni interfejs klase `Jelovnik`, odnosno klase `Jelo`.

Razmotrimo sljedeću situaciju. Nakon određenog vremena upotrebe sistema `e-restoran`, pojavi se potreba da se u prikazu ponude pored naziva jela prikaže i slika sa izgledom jela.

Zbog ovisnosti klasa ako postoji izmjena u klasi, `Jelo` će se reflektirati na klasu `Jelovnik`, te dalje na sve ovisne klase, uključujući i klasu `AplikacijaZaNarucivanje`.



Slika 6.15: Ovisnost između klasa

Na slici 6.15 ovisnost je jednosmjerna i usmjerena od klase korisničkog okruženja ka klasi domena problema. Tako znamo da promjena klase `AplikacijaZaNarucivanje` neće uticati na klasu `Jelovnik`, niti na druge klase domena problema.

UML može prikazati mnogo vrsta ovisnosti i svaka od njih se opisuje na drugačiji način pomoću unaprijed definiranih stereotipa, kao što je <<create>> sa značenjem izvor pravi instance odredišta ili <<use>> sa značenjem izvor u realizaciji koristi odredište.

Treba koristiti što manje ovisnosti, naročito onih koje se prostiru preko velikih dijelova sistema. Posebno treba obratiti pažnju na ciklične ovisnosti, pošto one dovode i do cikličnih promjena.

Prikazivanje svih ovisnosti na dijagramu klasa je beskorisno, pošto ih ima puno i često se mijenjaju. Ovisnosti se najčešće prikazuju samo ako su neposredno značajne za temu koju opisujemo.

Pitanja za ponavljanje

1.Objasniti značenje asocijacije između klasa.
2.Objasniti navigabilnost asocijacije.
3. Koje informacije se mogu dodati na krajeve asocijacije?
4.Objasniti kardinalnost i njeno označavanje.
5.Objasniti značenje dvosmjerne asocijacije?
6.Koja je notacija za indiciranje uloge klase u asocijaciji?
7.Koja su dva načina predstavljanja asocijacije?
8. Objasniti ulogu i način imenovanja asocijacije glagolskim oblikom.
9.Objasniti ulogu kvalifikatora prilikom izražavanja asocijacije između klasa.
10.Diskutirati način predstavljanja asocijacije u programskim jezicima.
11. Što je agregacija?
12. Koja je razlika između agregacije i 'obične' asocijacije?
13.Što je kompozicija?
14. Koja je razlika između agregacije i kompozicije?
15. Koja je UML notacija za agregaciju i kompoziciju?
16.Koja je UML notacija za agregaciju?
17.Objasniti značenje generalizacije?
18. Što je višestruka generalizacija?
19. Kako se generalizacija predstavlja u programskim jezicima?
20. Definirati vezu ovisnosti među klasama?
22. Koju opasnost može donijeti veza ovisnosti među klasama?

POGLAVLJE 7.

DIJAGRAM KLASE: NAPREDNI KONCEPTI

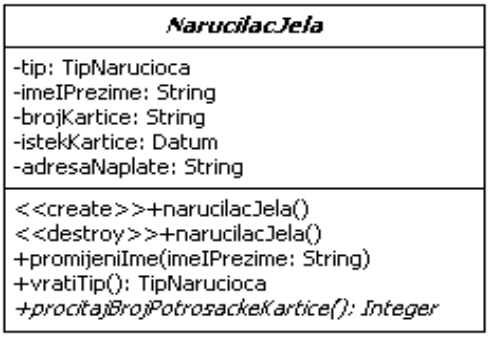
U okviru ovog poglavlja će se opisati i još neki dodatni aspekti vezani za dijagrame klase, kao što apstraktne klase, interfejsi i primjena ostalih UML model elemenata i proširenja na dijagramima klase.

7.1 Apstraktne klase

Apstraktna klasa je klasa koja nema nijedan objekt ili preciznije na osnove nje nije dozvoljeno instanciranje objekata. Apstraktna klasa opisuje zajedničke atribute i ponašanje za druge klase. Apstraktna klasa obično ima apstraktne operacije. Apstraktna operacija nema implementacijsku metodu gdje je specificirana. Klasa koja ima najmanje jednu apstraktnu operaciju mora biti definirana kao apstraktna klasa. Apstraktne operacije su definirane u apstraktnim klasama da specificiraju ponašanje koje moraju imate sve njihove potklase. Klasa koja se nasljeđuje iz klase koja ima jednu ili više apstraktnih operacija može implementirati ove operacije ili može i sama postati apstraktna klasa.

Uobičajeni način da se ukaže na apstraktnu klasu ili operaciju u UML-u jeste da se ime klase ili operacije piše kosim (italik) slovima. Drugi način jeste koristiti rezerviranu riječ {abstract}.

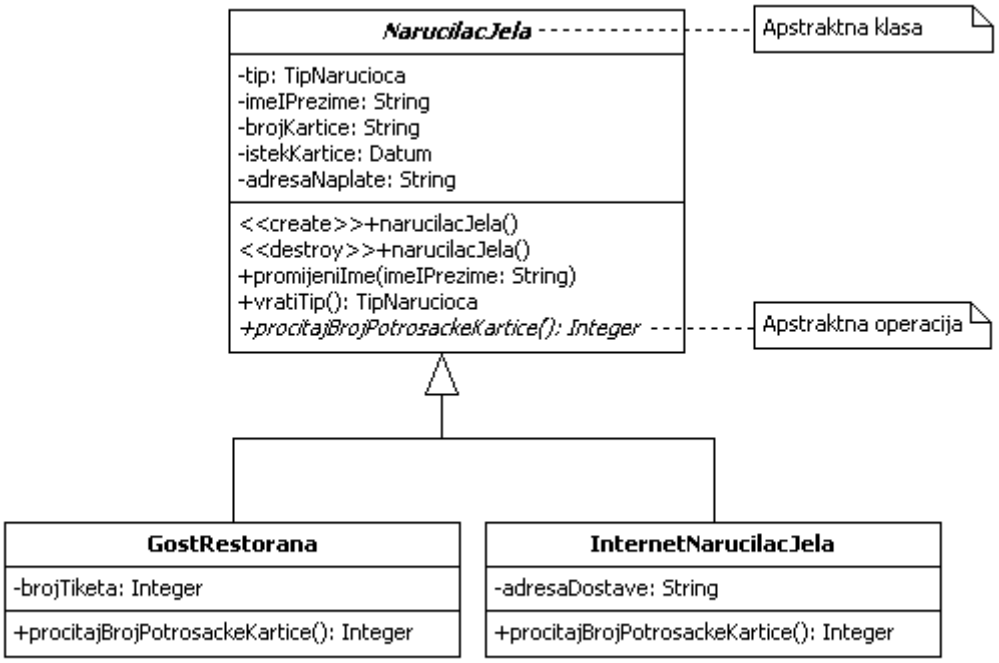
Na slici 7.1 UML je prikazana apstraktna klasa `NarucilacJela` sa apstraktnom operacijom `procitajBrojpotrosackeKartice`.



Slika 7.1: Primjer UML notacije apstraktne klase

Sa generalizacijom implementiraju se apstrakcije. Osim apstraktnih klasa postoje i konkretne klase koje smo većinom do sada i koristili. Na osnovu konkretne klase, moguće je kreirati objekte, jer konkretne klase imaju implementaciju za sve operacije.

Sljedeća slika pokazuje apstraktnu klasu i dvije konkretne klase koje su naslijeđene na osnovu apstraktne klase, i na osnovu kojih se mogu instancirati objekti.



Slika 7.2: Apstraktna klasa i konkretne klase

U primjeru na slici 7.2 klasa `NarucilacJela` je apstraktna klasa jer sadrži apstraktnu operaciju *`procitajBrojPotrosackeKartice`*, koja može biti implementirana na različite načine: očitavanjem sa čitača kartica ili unosom broja kartice i PIN-a putem interfejsa. Klase `GostRestorana` i `InternetNaručilacJela` nasljeđuju apstraktnu klasu `NarucilacJela` naslijediti, a time i sve njene operacije, s tim da apstraktne operacije mogu implementirati na način kako to njima odgovara. Kako vidimo, upotreba apstraktnih klasa podrazumijeva i nasljeđivanje kao vezu među klasama. Budući da se radi o jakoj vezi među klasama, o tome treba posebno voditi računa. Međutim, apstraktne klase su veoma efikasno sredstvo za objektno orijentirani pristup u programiranju, jer ne moramo nužno implementirati u potpunosti atribute i ponašanje klase, već to ostavljamo njenoj potklasi.

Implementacija u programskim jezicima Java i C++ navedenih koncepata je:

Java :
```
public abstract class NarucilacJela
{
    ...
    public TipNarucioca vratiTip()
    {
        ...
    }
    // Apstraktna operacija
    public abstract Integer procitajBrojPotrosackeKartice();
}
...
public class GostRestorana extends NarucilacJela
{
    ...
    public Integer procitajBrojPotrosackeKartice()
    {
        // Implementacija čitanja sa čitača kartica
    }
}
...
public class InternetNarucilacJela extends NarucilacJela
{
    ...
    public Integer procitajBrojPotrosackeKartice()
    {
        // Implementacija čitanja putem GUI
    }
}
```

C++ :
```
// NarucilacJela.h
class NarucilacJela
{
public:
    ...
    TipNarucioca vratiTip();
    virtual int procitajBrojPotrosackeKartice();
    ...
};
...
// GostRestorana.h
class GostRestorana : public NarucilacJela
{
public:
    ...
    int procitajBrojPotrosackeKartice();
    ...
};
...
// GostRestorana.cpp
...
int GostRestorana::procitajBrojPotrosackeKartice()
{
    // Implementacija čitanja sa čitača kartica
}
...
// InternetNarucilacJela.h
class InternetNarucilacJela : public NarucilacJela
{
public:
    ...
    int procitajBrojPotrosackeKartice();
    ...
```

Implementacija apstraktnih i konkretnih klasa

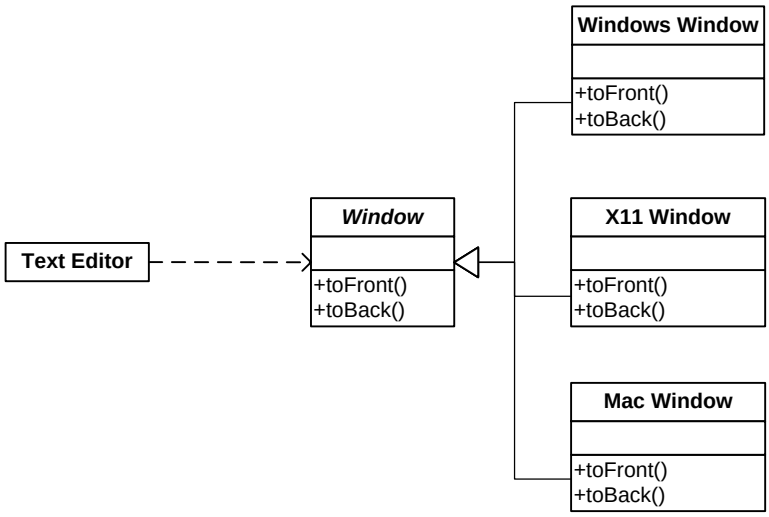
7.2 Interfejsi i apstraktne klase

Jedna od prednosti objektno orijentiranog razvoja je i da se može odvojeno posmatrati interfejs klase od implementacije. U biti klasa ima dvije namjene: definira interfejs koji objekti predstavljaju za ostatak sistema i definira implementaciju interfejsa. Ponekad je veoma važno za dizajn odvojiti ova dva koncepta, posebno da bi se označili različiti nivoi ovisnosti između elemenata.

Interfejs-apstraktna klasa

U nekim programskim jezicima, kao što je C++, interfejs se implementira kao apstraktne klase koje ne sadrže implementirane operacije. Takve klase, kako smo vidjeli, mogu imati i neku implementaciju, ali se one često primarno koriste za deklariranje interfejsa.

Tekst editor prikazan na slici 7.3 je tipičan primjer. Da bi se osiguralo da editor bude neovisan od platforme, definiramo neovisnu od platforme apstraktnu klasu `Window`. Ova klasa nema operacija, ona samo definira interfejs za tekst editor. Potklase koje su ovisne od platforme se mogu po želji koristiti.



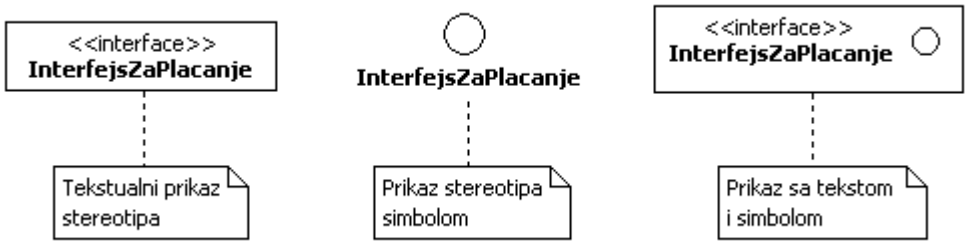
Slika 7.3: `Window` kao apstraktna klasa

Interfejs je ustvari klasa koja nije realizirana i sve njene karakteristike su apstraktne.

Interfejs

Ponekada, kada imamo samo jednostruko nasljeđivanje (govorimo o Java programskom jeziku) umjesto apstraktne klase, bolje je da koristimo klasu *interfejs*. Java i C# imaju posebnu konstrukciju interfejsa. Interfejs, također, sadrži samo deklaraciju apstraktnih metoda, bez njihove implementacije. Iz razloga što se kod interfejsa radi o jednostrukoj relaciji nasljeđivanja, interfejs je mnogo sigurniji za upotrebu od apstraktne klase, koje omogućavaju višestruko nasljeđivanje. Interfejs dakle sadrži apstraktne metode, koji su implementirani u klasi koja ga realizira odnosno nasljeđuje. Također može sadržavati i atribute, koji su obično statičke i konstantne vrijednosti.

Na UML dijagramu klasu interfejsa možemo označiti sa stereotipom <<interface>> ili sa krug (loptica) notacijom ili sa njihovom kombinacijom što je vidljivo sa slike 7.4.



Slika 7.4: Označavanje interfejsa stereotipom i lopticom

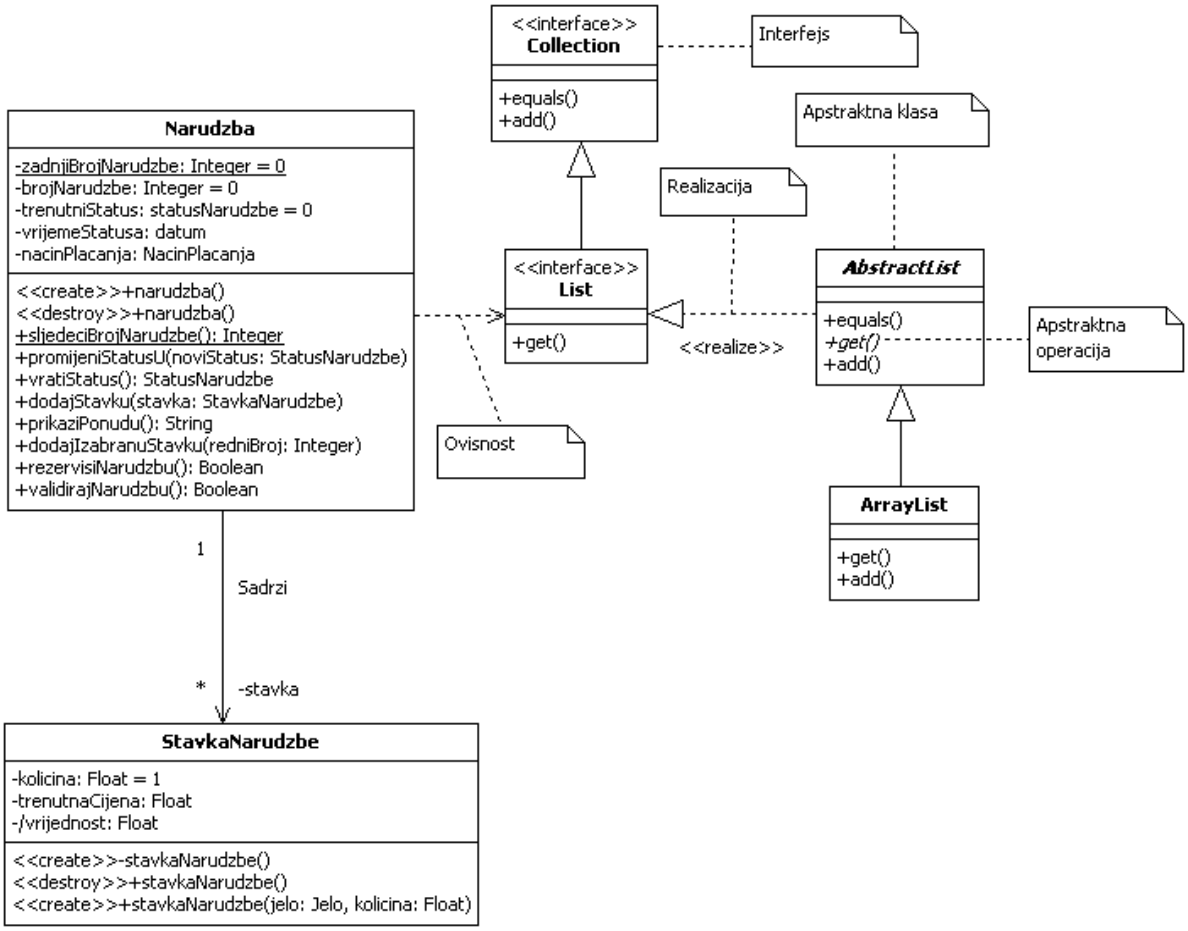
Klase uspostavljaju dvije vrste veza sa interfejsima: klasa može realizirati interfejs i može zahtijevati interfejs. Klasa realizira interfejs ako implementira operacije interfejsa, pri čemu ne mora implementirati sve operacije interfejsa. U tom slučaju klasa realizacije je apstraktna klasa. Klasa zahtijeva interfejs ako joj za rad treba instanca interfejsa.

Realizacija interfejsa se obilježava isprekidanom linijom sa ispunjenom strelicom, a zahtijevani interfejs se obilježava relacijom ovisnosti.

Slika 7.5 prikazuje kako funkcioniraju ove veze na osnovu nekoliko klasa kolekcija jezika Java. Klasa `Narudzba` ovisi od interfejsa `List` i koristi metode `equals`, `add` i `get`. Kada se

objekti povežu, klasa `Narudzba` će zapravo koristiti instancu klase `ArrayList`, ali to ne mora da zna da bi upotrijebila navedene tri metode, pošto se sve one nalaze u interfejsu `List`.

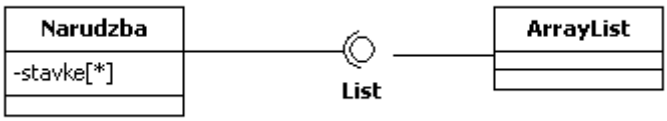
Klasa `ArrayList` je potklasa od `AbstractList`. Klasa `AbstractList` realizira neke, ali ne i sve, operacije klase `List`. Metoda `get` je i dalje apstraktna. Zato klasa `ArrayList` realizira metodu `get`, ali i redefinira metodu `add`, i nasljeđuje metodu `equals`.



Slika 7.5: Interfejsi i apstraktne klase

Postavlja se pitanje, zašto se sve ovo ne bi jednostavno izbjeglo, odnosno zašto klasa `Narudzba` direktno ne koristi klasu `ArrayList`? Upotrebom interfejsa olakšava se naknadna promjena realizacije. Neki drugi tip liste može raditi bolje, čitati podatke iz baze ili biti drugačije poboljšan. Upotrebom interfejsa umjesto konkretne realizacije izbjegavaju se obimne promjene kôda u slučaju potrebe za drugačijom realizacijom klase `List`.

Potpuna notacija sa slike 7.5 je jedan način za grafički prikaz interfejsa. Na slici 7.6 je prikazan sažetiji, drugi način označavanja za primjer iznad. Činjenica da klasa `Narudzba` realizira interfejse `List` i `Collection` je prikazana ikonom sa sastavljenim polukrugom i krugom (eng. socket).

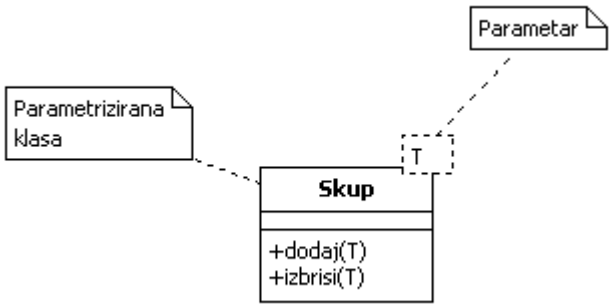


Slika 7.6: Drugi način označavanja interfejsa

Upotreba interfejsa je dobra i korisna zbog toga što nam odvaja definirano ponašanje klase od njene implementacije i time ostvarujemo „slabljenje“ veza među klasama, jer klasu u tom slučaju referenciramo na interfejs a ne na konkretnu klasu. Takav sistem je pouzdaniji i lakši za održavanje.

7.3. Parametrizirane klase

Parametrizirana klasa je u principu klasa koja prima parametar. Neki jezici imaju notaciju parametrizirane klase poznatu i kao templejt (eng. template). Templejt klasa je korisna kada želimo odgoditi u nekoj tački dizajniranja informaciju o konkretnoj vrsti objekta sa kojim će klasa raditi.

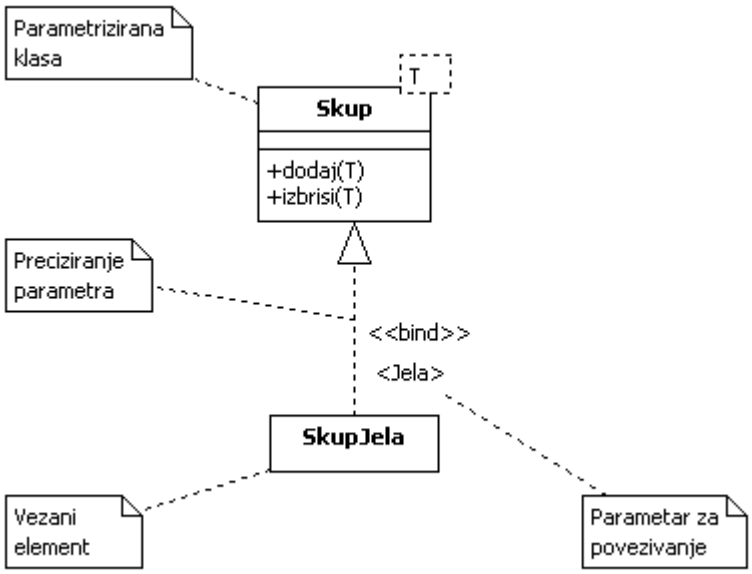


Slika 7.7: Parametrizirana klasa

Slika iznad predstavlja klasu Skup sa parametrom označenim kao T. T nije klasa u našem modelu, uloga parametra T je da u kasnijoj fazi informira klasu Skup o tipu objekta koji treba smjestiti.

Element koji se koristi u parametariziranoj klasi prikazuje se u varijanti 1 kao na slici 7.8 i varijanti 2 kao na slici 7.9.

Slika 7.8: Element parametrizirane klase



Slika 7.9: Element parametrizirane klase – verzija 2

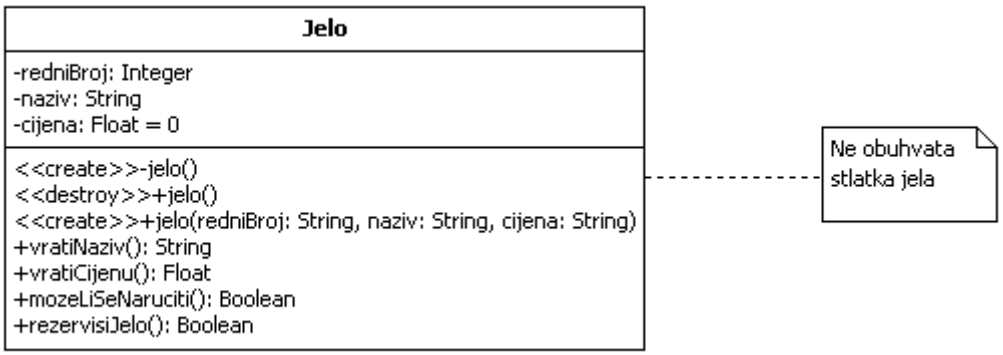
Stvarna snaga parametriziranih klasa je kada se koristi drugi pristup formiranja parametra za vrijeme izvršavanja koji se odnosi na objekte, a ne na klase, za što nam je potreban drugi tip dijagrama – dijagram objekata koji uvodimo u sljedećem poglavlju.

7.4 Dodatna notacija na dijagramu klase

U okviru uvodnog razmatranja UML-a uveli smo i dodatne elemente i simbole koji se mogu koristiti na dijagramima. To su bili komentari, ograničenja i stereotipi. Sada ćemo vidjeti i primjenu istih u sklopu dijagrama klase.

Komentari

Komentari su dodatna objašnjena odnosno napomene vezane za klasu, atribut, operaciju ili neku vrstu veze. Na dijagramima klase mogu biti neovisni od drugih elemenata dijagrama ili spojeni isprekidanom linijom sa elementima koje objašnjavaju (slika 7.10). Isto tako, mogu se pojaviti na svakoj vrsti dijagrama.



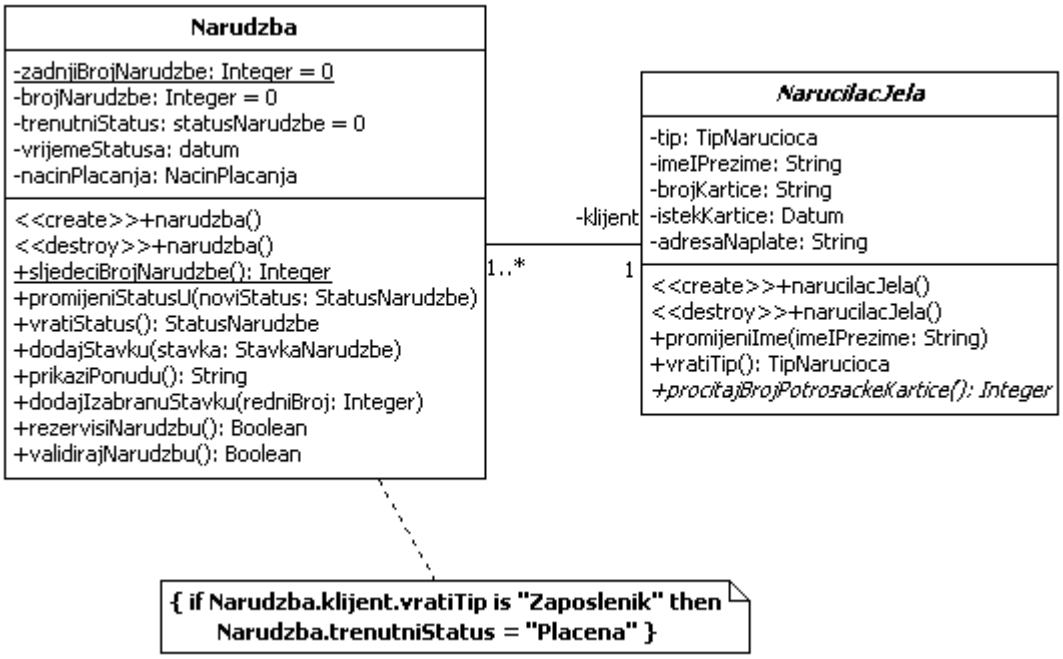
Slika 7.10: Komentar za element dijagrama klase

U navedenom primjeru komentar daje dodatno objašnjenje vezano za upotrebu klase `Jelo`, odnosno pojašnjava da se klasa `Jelo` odnosi na sva jela, osim na slatka jela.

Ograničenja

Dijagrami klasa u velikoj mjeri prikazuju ograničenja sistema. Osnovni elementi kao što su asocijacija, atributi i generalizacija definiraju većinu važnih ograničenja, ali ne mogu ukazati na sva ograničenja. Preostala ograničenja ipak negdje moraju biti prikazana, a dobro mjesto za to je dijagram klasa.

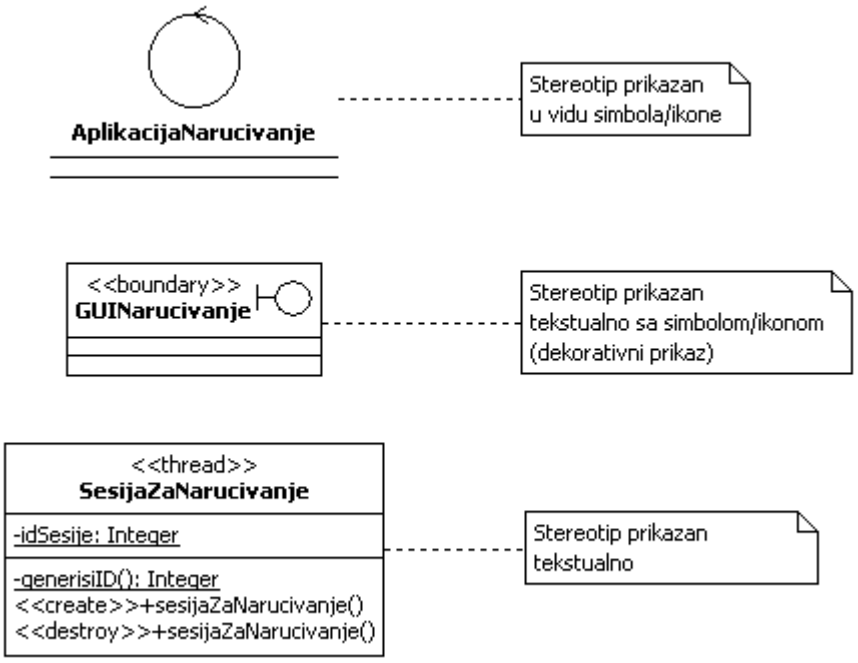
Jezik UML dozvoljava da opišemo ograničenja kad god želimo. Jedino pravilo je da opis moramo smjestiti unutar vitičastih zagrada - {}. Pri tome možemo koristiti neki prirodan jezik, programski jezik ili UML-ov formalni jezik ograničenja objekata (eng. Object Constraint Language, *OCL*) koji je zasnovan na predikatskom računu i koji je detaljnije objašnjen u poglavlju 20. Upotreba formalne notacije otklanja opasnost od pogrešnih tumačenja, koja mogu nastati zbog nepreciznosti prirodnog jezika. Međutim, ona unosi novu opasnost od pogrešnog tumačenja, ako autori i čitaoci ne razumiju dovoljno OCL.



Slika 7.11: Ograničenje na dijagramu klase

Stereotipovi

Stereotipovi je UML način da se doda ekstra klasifikacija na dio modela. Stereotip opisuje element modela i postavlja se na element dijagrama. Osnovna UML notacija za specificiranje stereotipa je dodavanje imena stereotipa u okviru << >> iznad imena klase. Uz tekstualnu notaciju može se dodati i ikona i to je sve ilustrirano na slici ispod.

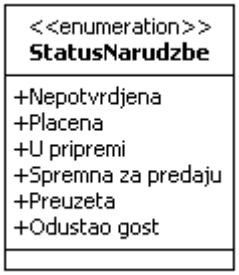


Slika 7.12: Različite forme označavanja stereotipa

Neki stereotipovi su u UML-u predefinirani, oni su automatski raspoloživi i ne mogu se redefinirati. Neki od njih su <<enumeration>>, <<interfaces>>, <<type>> i <<implementation>>.

Prebrojiva lista

Prebrojiva lista (eng. enumeration) koristi se radi prikazivanja nepromjenjivog skupa simboličkih vrijednosti (slika 7.13). Prikazuju se kao klase označene sa stereotipom «enumeration».



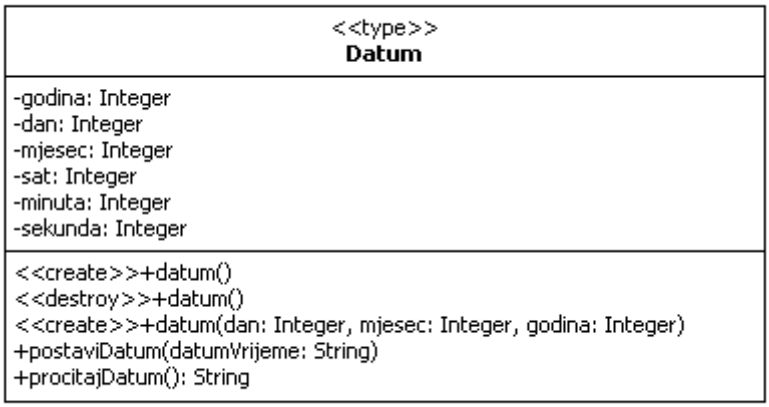
Slika 7.13: Prebrojiva lista <<enumeration>> notacija

Ova prebrojiva lista može imati i operacije i prikazuju se u donjem dijelu ikone na isti način kao i za klase.

Tip podataka

Ponekad je poželjno eksplicitno modelirati tip podataka na dijagramu klase. Klasa koje se obilježava kao tip podataka koristi stereotip <<datatype>> ili <<type>>. Sasvim je moguće za <<datatype>> da ima i atribute i operacije.

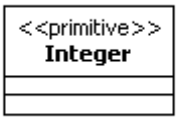
Klasa `Datum` na slici 7.14 je tip podataka koji ima atribute: `godina`, `dan`, `mjesec`, `sat`, `minuta` i `sekunda`, te operacije: `datum`, `postaviDatum` i `procitajDatum`.



Slika 7.14: Tip podataka <<type>> notacija

Osnovni tipovi

UML podržava predstavljanje tipova podataka, koje mogu imati operacije modelirane unutar ili izvan UML-a, ali koje nemaju internu strukturu. Primjeri takvih tipova su: String, Integer i Boolean. Klase koje predstavljaju ove tipove podataka obilježavaju se sa stereotipom <<primitive>>.



Slika 7.15: Osnovni tip podataka <<primitive>> notacija

Uslužne klase

UML omogućava notaciju za modeliranje kolekcije globalnih atributa i operacija u okviru uslužnih klasa koje se obilježavaju sa <<utility>> stereotipom.

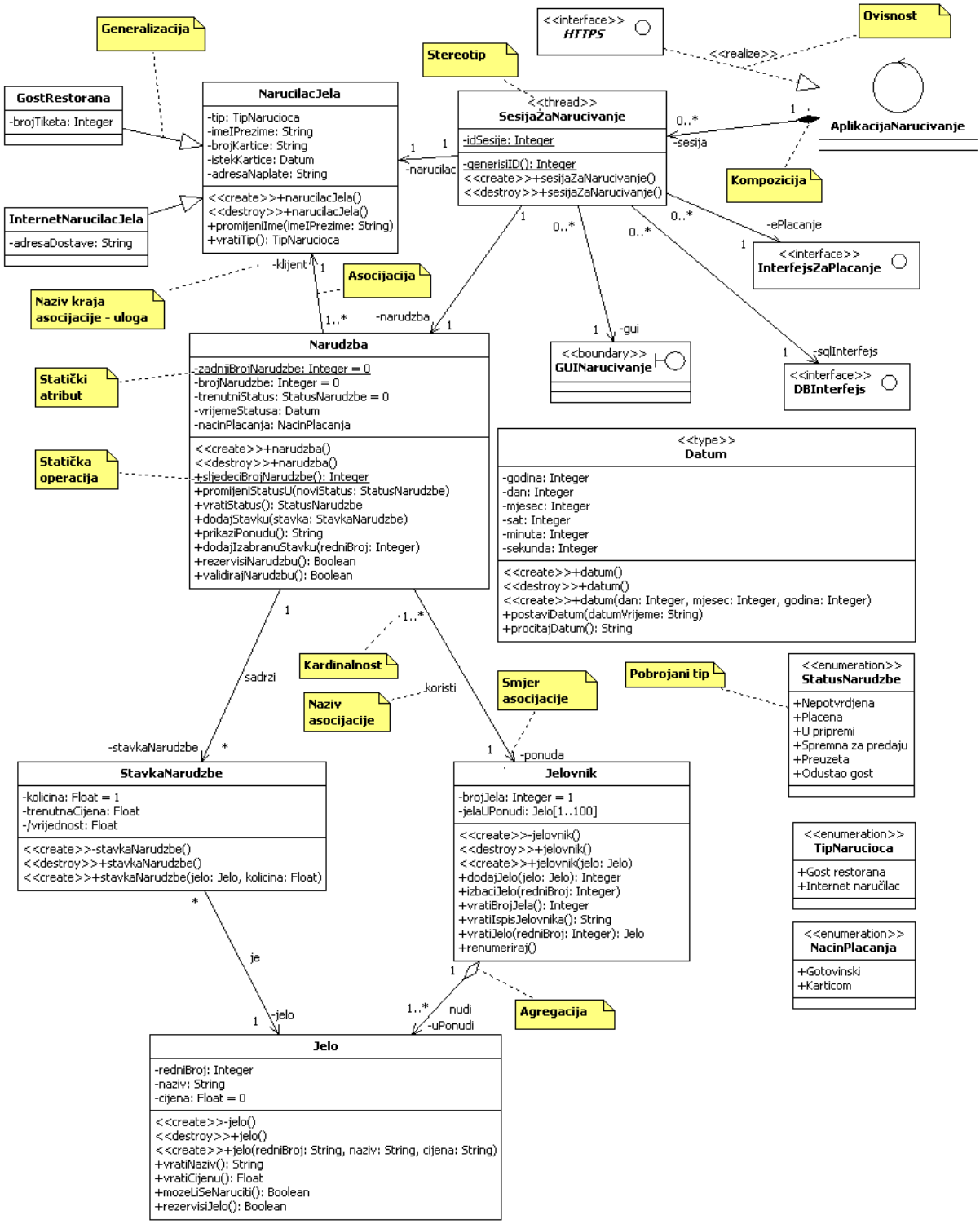
Slika 7.16: Uslužne klase <<utility>> stereotip

Nije neophodno povezivati ove uslužne klase sa klasama koje ih koriste, pošto su atributi i operacije ovih klasa globalno raspoložive.

7.5 Dijagram klase za `e-restoran`

Na slici 7.17 je dijagram klasa na kome su prikazane najvažnije klase u procesu naručivanja, u okviru sistema `e-restoran`. Na slici, također, možemo uočiti i skoro sve elemente UML notacije koji su spomenuti u dosadašnjim poglavljima. Sa slike se može uočiti sljedeće: `AplikacijaNarucivanje` je kontrola (stereotip prikazan simbolom) koja realizira HTTPS interfejs za komunikaciju i apsolutni je vlasnik (kompozicija) svih instanci `SesijaZaNarucivanje`. `SesijaZaNarucivanje` je programska nit (stereotip <<thread>>) koja u stvarnosti komunicira sa naručiocem jela (klasa `NarucilacJela`). Za tu komunikaciju koristi instancu klase `GUINarucivanje`, koja je granična klasa sistema pa je označena stereotipom <<boundary>>. U toj interakciji sa naručiocem jela se formira instanca klase `Narudzba` i ista se zapisuje u bazu podataka preko instance interfejsa klase `DBInterfejs`. Osim toga za plaćanje karticom vrši se komunikacija sa sistemom `za autorizaciju kartica` preko instance interfejsa klase `InterfejsZaPlacanje`. Narudžba se formira na osnovu jelovnika (klasa `Jelovnik`), a izabrana jela (klasa `Jelo`) se spremaju kao stavke narudžbe (klasa `StavkaNarudzbe`). Spomenute klase imaju parametre od kojih su neki složenog tipa, odnosno klase: `Datum`, `StatusNarudzbe`, `TipNarucioca`, `NacinPlacanja`. S obzirom na to da ovakve klase mogu biti upotrijebljene u mnogim drugim klasama, onda se iste ne povezuju asocijacijama, jer bi to nepotrebno opteretilo dijagram i učinilo ga

nečitljivim. Klasa `NarucilacJela` je generalizacija za klase `GostRestorana` i `InternetNarucilacJela`.



Slika 7.17: Sistem `e-restoran` – dio dijagrama klase

Završno razmatranje

Dijagram klasa pokazuje statičku strukturu klasa u sistemu. Dijagram klasa se koristi da opiše model klasa iz tri perspektive; konceptualne, specifikacijske i implementacijske. Pristup modeliranju može ovisiti od perspektive koja se modelira. U objektno orijentiranim sistemima, operacije definirane u dijagramima klase će biti u interakciji jedna sa drugom da bi ispunile funkcionalne zahtjeve sistema. Funkcionalni zahtjevi su opisani sa dijagramom slučajeva upotrebe u poglavlju 4. Dinamičko korištenje operacija i klasa se dokumentira sa dijagramom komunikacije i dijagramom sekvence, koji će biti objašnjeni u poglavljima 11-14.

Pitanja za ponavljanje

1. Koja je razlika između apstraktne i konkretne klase?
2. Koja je UML notacija apstraktne klase?
3. Koja je uloga apstraktne klase pri formiranju interfejsa?
4. Objasniti ulogu interfejsa pri dizajnu sistema.
5. Što je ustvari interfejs?
6. Koja se stereotip notacija koristi da bi se prikazala interfejs klasa?
7. Diskutirati način implementacije interfejsa u različitim programskim jezicima.
8. Koje vrste veza uspostavlja klasa sa interfejsima?
9. Koja je uloga klase sa parametrom?
10. Koja je UML notacija klase sa parametrom?
11. Kako se klasa sa parametrom predstavlja u programskim jezicima?
12. Koja je uloga i notacija komentara?
13. Što su ograničenja?
14. Kako se pišu ograničenja na dijagramu klase?
15. Koja je namjena stereotipa?
16. Koja je notacija stereotipa?
17. Koja je namjena <<enumeration>> klase?
18. Koja je namjena <<utility>> klase?
19. Koja je namjena <<type>>, a koja <<primitive>> klase?

POGLAVLJE 8.

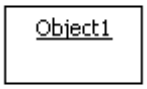
DIJAGRAM OBJEKATA

Dijagram objekata je slika objekata u sistemu u jednom vremenskom trenutku. Objekti su srce objektno orijentiranog sistema za vrijeme njegovog izvršavanja. Kako prikazuje instance a ne klase, dijagram objekata se često naziva dijagram instanci. Dijagrami objekata su korisni kada želimo da opišemo kako objekti unutar sistema mogu raditi zajedno u pojedinačnim scenarijima.
U mnogim situacijama moguće je definirati precizno strukturu sa dijagramima klase, ali je još uvijek teško razumjeti tu strukturu. U ovakvim situacijama nekoliko primjera dijagrama objekata je od velike pomoći. I dijagram objekata i dijagram klasa pomažu nam da sagledamo logički pogled na sistem. U usporedbi sa dijagramom klasa, dijagram objekata ima veoma jednostavnu notaciju.

Kada crtamo dijagram objekata, prvo što nam treba je dodavanje aktualnih objekata, odnosno instanci objekata.

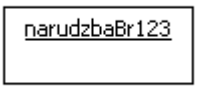
8.1.Instance objekata

Objekt se, isto kao i klasa, prikazuje kao pravougaonik, a da bi prikazali da je u pitanju instanca klase, naziv objekta se podvlači kao na slici ispod.



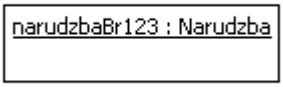
Slika 8.1: Simbol za objekt

„Object1“ sa slike se mijenja sa stvarnim nazivom objekta, npr. `narudzbaBr123`, čije je ime također podvučeno.



Slika 8.2: Konkretna instanca klase

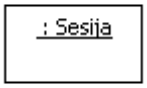
Ako želimo da označimo i klasu na osnovu koje se referencira objekt, tada dodajemo ime klase vezano za ime objekta sa notacijom: *imeobjekta:imeklase*



Slika 8.3: Ime objekta sa pripadajućom klasom

8.2 Anonimni objekt

Anonimni objekt je objekt bez imena i tipično se koriste kada naziv objekta nije važan unutar konteksta korištenja.



Slika 8.4: Anonimni objekt klase `Sesija`

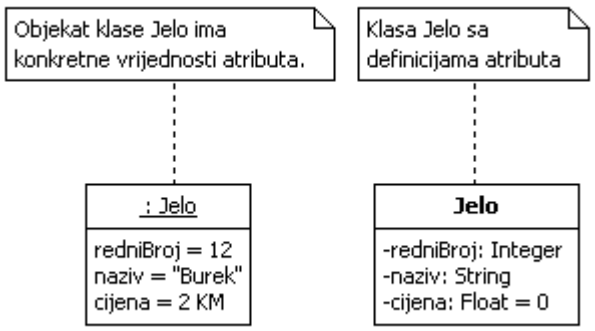
Često se može iskoristiti kada kreiramo metodu za upravljanje događajima jer tada se ne vodi računa o nazivu objekta već samo da je registriran unutar odgovarajućeg izvora događaja.

8.3 Atributi objekta

Objekt je primjerak (instanca) klase, sa stvarnim vrijednostima za neke ili sve atribute. Osnovna notacija za prikazivanje vrijednosti atributa je:

imeAtributa:tip=vrijednost

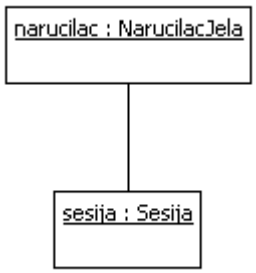
Sljedeća slika prikazuje anonimni objekt klase `Jelo` koji sadrži atribute: `redniBroj`, `naziv` i `cijena` sa konkretnim vrijednostima.



Slika 8.5: Notacija za atribute objekta

8.4 Link

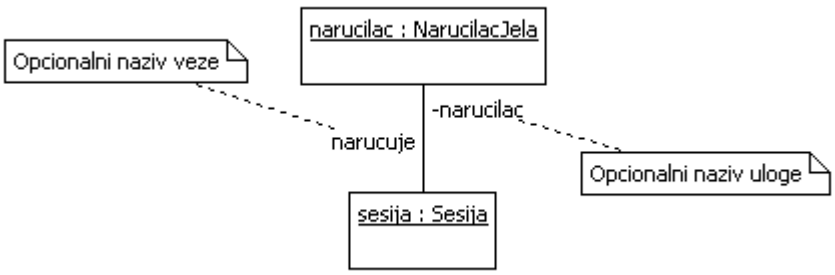
Objekti nisu interesantni ili od neke pomoći sami za sebe. Da bi se zaista pokazalo kako objekti rade zajedno u konkretnoj situaciji, potrebno je da se objekti povežu koristeći linkove. Linkovi se prikazuju kao linija između dva objekta (slika 8.6).



Slika 8.6: Link između objekata

Linkovi između objekata na dijagramu objekata prikazuju da dva objekta mogu komunicirati jedan sa drugim. Ako kreiramo link između dva objekta mora postojati i odgovarajuća asocijacija između klasa.

Na linkovima se mogu dodati i informacije za indikaciju namjene linkova što je predstavljeno na slici 8.7.

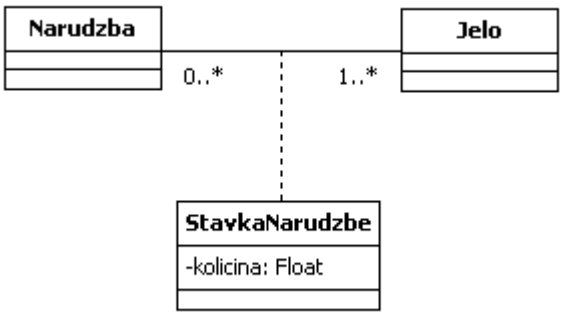


Slika 8.7: Dodavanje informacija na link

8.5 Linkovi i ograničenja

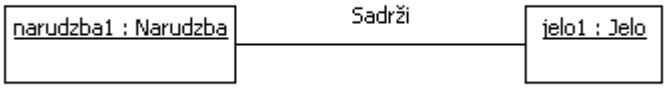
S obzirom na to da linkovi između objekata odgovaraju asocijacijama između klasa ograničenja koja se primjenjuju na asocijaciji moraju biti i na vezi – linku dijagrama objekata.

Jedan od načina prikazivanja asocijacije između klasa je pomoću asocijativne klase. Slika 8.8 prikazuje 3 klase i veze između njih. Kada se jelo koje je u našem modelu tipa klase `Jelo` dodaje na narudžbu tipa klase `Narudzba` pridružuje se količina koja je atribut asocijacije, odnosno pripada asocijativnoj klasi `StavkaNarudzbe`. `StavkaNarudzbe` je asocijativna klasa koja je povezana sa klasama `Narudzba` i `Jelo`.

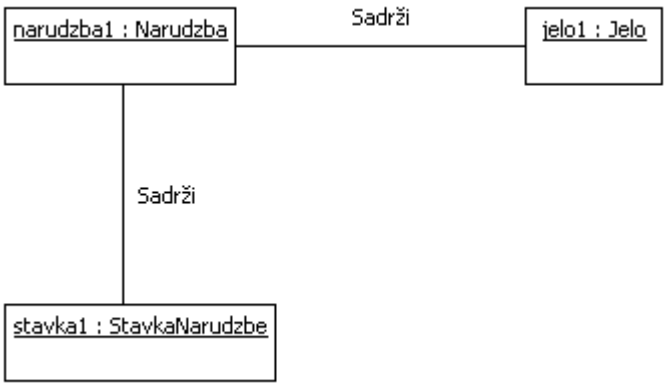


Slika 8.8: Primjer asocijativne klase

Za prikazani dijagram klasa na slici 8.8 postoje dvije validne varijante dijagrama objekata.

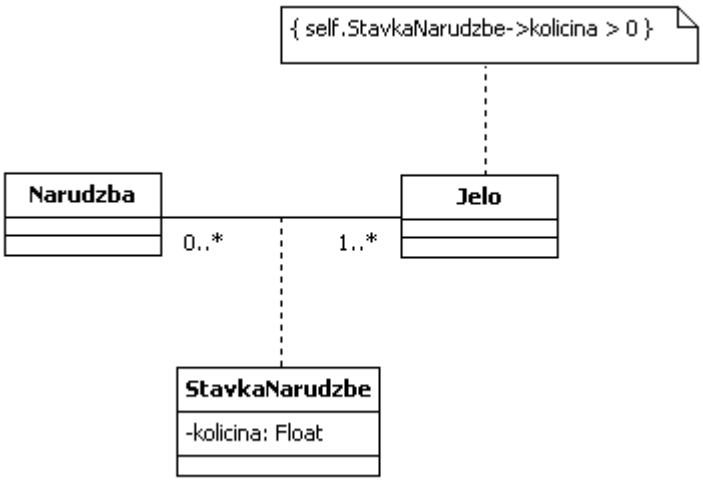


Slika 8.9: Prva varijanta dijagrama objekata



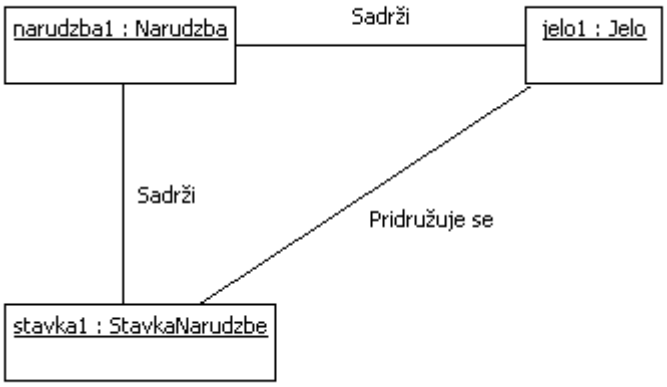
Slika 8.10: Druga varijanta dijagrama objekata

Navedene varijante dijagrama objekata ne pokazuju povezanost jela i stavke narudžbe, koja treba postojati jer nema smisla da se neko jelo nalazi na narudžbi a da za njega nema specificirana količina koja se naručuje, a koja je sadržana u stavci narudžbe. Zbog toga je neophodno naglasiti da je jelo povezano sa stavkom narudžbe kada se nalazi na narudžbi, upotrebom OCL (Object Constraint Language) notacije.



Slika 8.11: Dijagram klasa sa ograničenjem

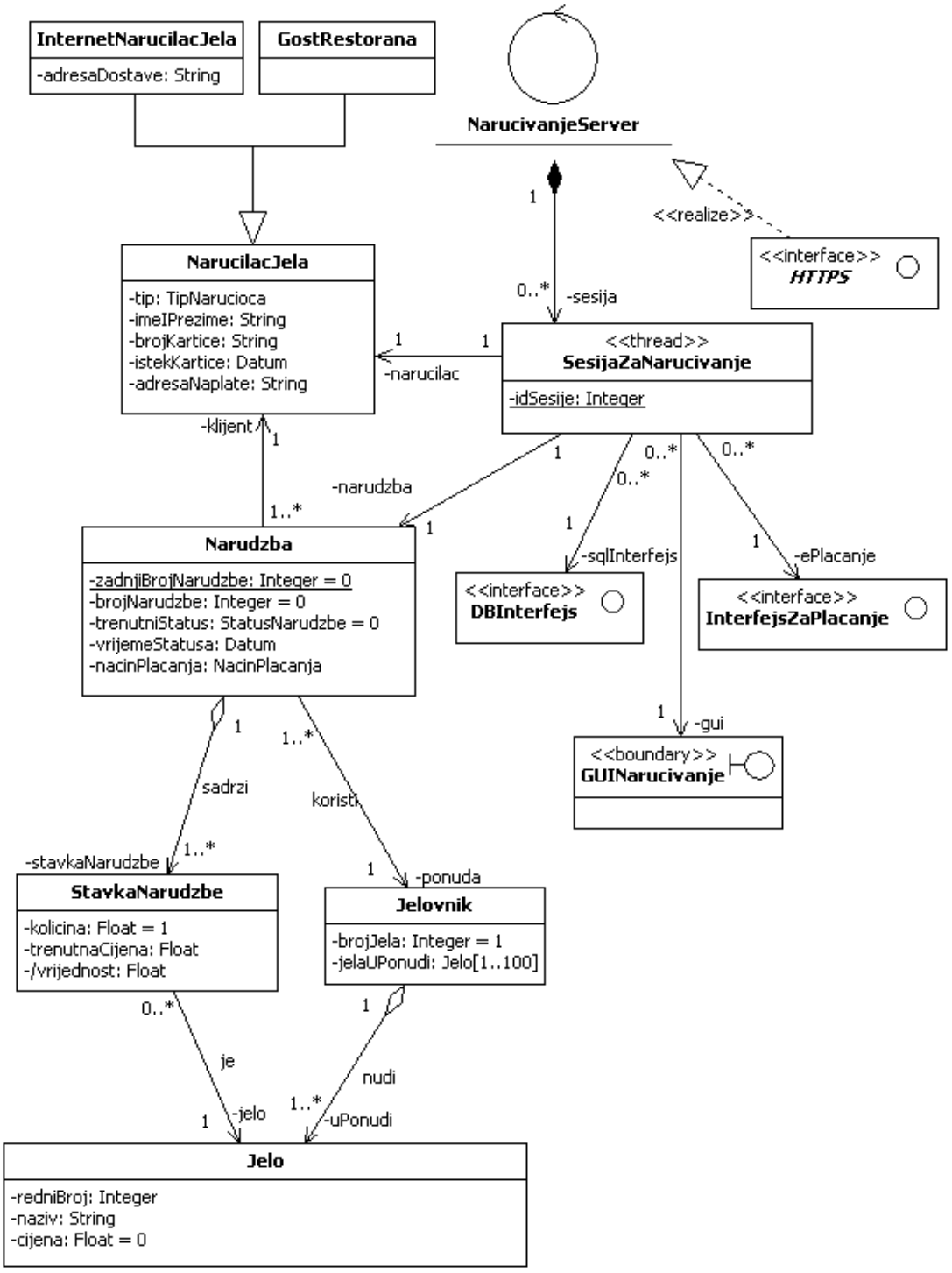
Nakon definiranja ograničenja primjenom OCL notacije na dijagramu klasa moguće opcije se reduciraju i dijagram izgleda:



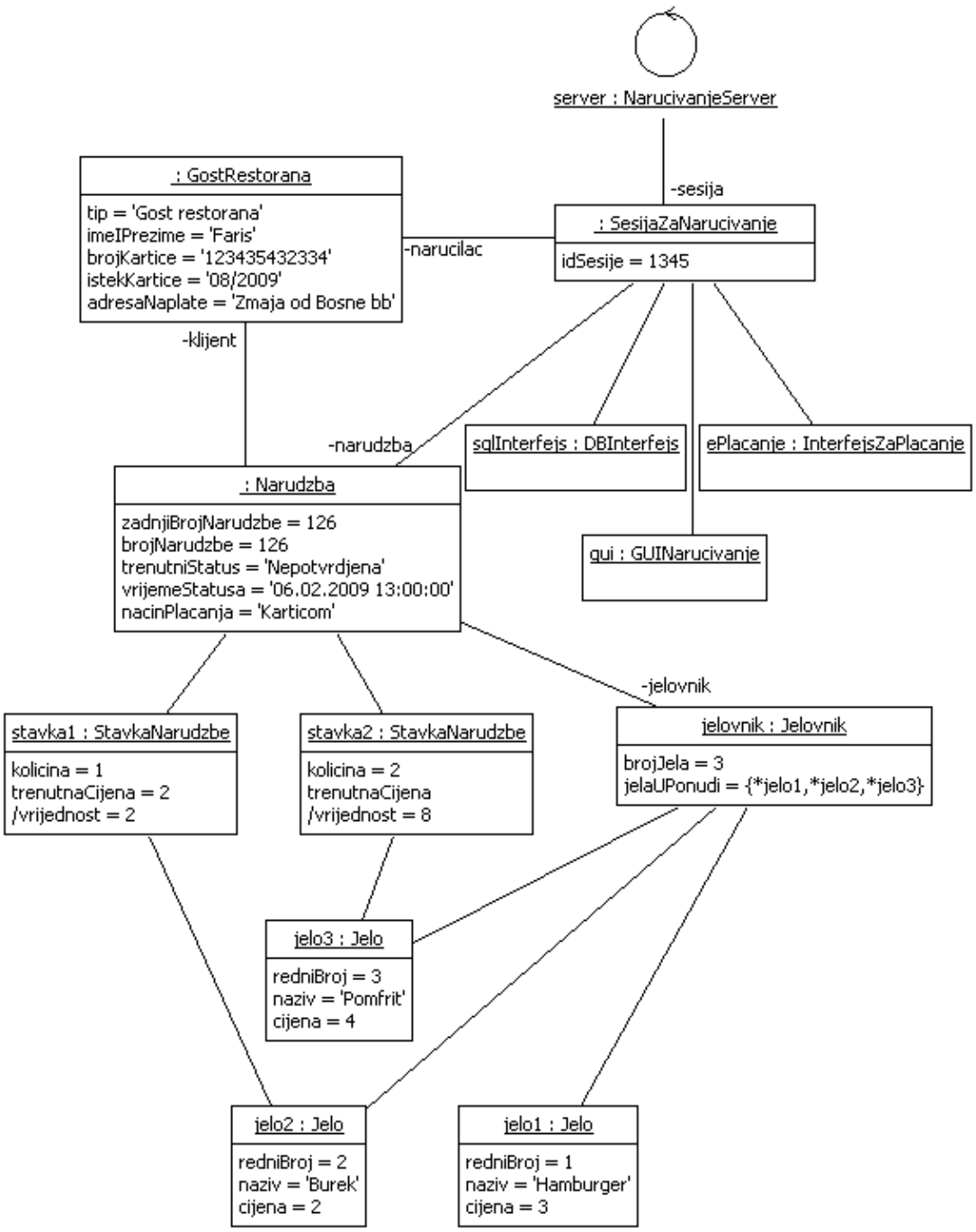
Slika 8.12: Dijagram objekata za dijagram klasa sa slike 8.11

8.6. Kreiranje dijagrama objekata

Posmatrajmo dijagram klasa `e-restoran` sistema na slici 8.13 koji sadrži klase bitne za proces naručivanja. Razmotrimo na dijagramu objekata prikazanom na slici 8.14 situaciju kada gost restorana vrši naručivanje. Kako je gost restorana klase `GostRestorana` koja predstavlja generalizaciju klase `NarucilacJela`, onda se formira jedna instanca klase `GostRestorana`, koja sadrži i sve naslijeđene atribute. Instanca klase `GostRestorana` se pojavljuje u dvije uloge, ulozi `narucilac`, u okviru instance klase `SesijaZaNarucivanje` i ulozi `klijent`, u okviru instance klase `Narudzba`, ali se u oba slučaja na njih referencira kao na instance klase `NarucilacJela`, jer klase `SesijaZaNarucivanje` i `Narudzba` imaju asocijaciju sa klasom `NarucilacJela` koja je osnovna klasa klase `GostRestorana`. Ovdje imamo situaciju gdje je primijenjen polimorfizam. Instanca klase `SesijaZaNarucivanje` se kreira pod nazivom `sesija` u okviru objekta `server`, koji je instanca klase `NarucivanjeServer`. Objekt `server` ne instancira klasu HTTPS jer je ona apstraktna, ali klasa `NarucivanjeServer` realizira interfejs HTTPS. Objekt `sesija` kreira i koristi objekte `sqlInterfejs`, `gui` i `ePlacanje`, koji su instance klasa `DBInterfejs`, `GUINarucivanje` i `InterfejsZaPlacanje`, redom. Neka su na jelovniku u ponudi tri jela. Objekt `narudzba` koristi objekt `jelovnik` koji je klase `Jelovnik`, a koji sadrži tri objekta klase `Jelo`. Gost restorana je izabrao dva jela, odnosno na objekt `narudzba` su povezani objekti `stavka1` i `stavka2`, koji povezuju objekte `jelo2` i `jelo3` sa odgovarajućim količinama i trenutnim cijenama.



Slika 8.13: Dijagram klasa za `e-restoran`



Slika 8.14 Dijagram objekata za opisanu situaciju naručivanja

Završno razmatranje

Dijagram objekata je, kako smo vidjeli, usko povezan sa dijagramom klase, sa razlikom da on opisuje instance klase u trenutku vremena. Oni su korisni u shvaćanju kompleksnih dijagrama klasa, kreiranjem različitih klasa u kojim su relacije i klase primijenjene. Veoma su korisni za

vrijeme elaboracijske faze razvojnog procesa da ispitaju stanje klasa i njihovih asocijativnih veza. Dijagrami objekata prikazuju statički dio interakcije, sadrži objekte koji kolaboriraju, ali bez poruka koje razmjenjuje, što je i razlog postojanja dijagrama interakcije koji će biti objašnjeni u narednim poglavljima.

Prije nego što nastavimo sa ostalim dijagramima slijedi jedno dodatno poglavlje o principima objektno orijentiranog dizajna.

Pitanja za ponavljanje

1.Koja je osnovna namjena dijagrama objekata?
2. U kojim situacijama su dijagrami objekata korisni?
3. Što je objekt?
4. Koja je UML notacija za objekt?
5. Što je anonimni objekt?
6. Koja je osnovna notacija zapisivanja atributa objekata?
7.Objasniti vezu između linka u okviru dijagrama objekata i asocijacije u okviru dijagrama klase.
8. Kako se predstavljaju ograničenja na linkovima između objekata?

POGLAVLJE 9.

PRINCIPI I METRIKA OBJEKTNO ORIJENTIRANOG DIZAJNA

Na kreiranim UML dijagramima, što tražimo? Kako ih procjenjujemo? Koji su principi dizajna koje bi trebali primijeniti? U ovom poglavlju ćemo diskutirati o pet takvih principa, koji će nam pomoći pri procijeni da li su UML dijagrami, ili dijelovi kôda dobro dizajnirani.

9.1 Kvalitet dizajna

Zadovoljavajući objektno orijentirani dizajn je nekada teško dostići i za inženjere sa višegodišnjim iskustvom. Zbog toga je prije početka implementacije, potrebno uložiti napor i definirati ispravan, dobar dizajn koji bi ispunio očekivanja kvalitetnog rješenja. Korektno dizajnirani objektno orijentirani sistemi veoma dostojno modeliraju stvarni svijet. Objektno orijentirane tehnike se koriste kako bismo efektivno upravljali kompleksnošću velikih softverskih sistema. Kada se kvalitetno uradi dizajn takvih sistema, objektno orijentirani razvoj rezultira softverskim sistemom koji je lakši za održavanje, nadogradnju i ponovnu iskoristivost pojedinih njegovih elemenata za druga softverska rješenja.

Kako bismo dostigli ove ciljeve, pred softver inženjere se postavlja uvjet pozornosti za odabir odgovarajućih dizajnerskih tehnika. Pažnja softver inženjera treba biti usmjerena ka principima objektno-orijentiranog dizajna kako bi kreirali kvalitetno rješenje. Dobro dizajnirano objektno orijentirano softversko rješenje odlikuje se sljedećim iskazanim karakteristikama:

- apstrakcija
- enkapsulacija
- kohezivnost
- slaba međusobna ovisnost

Apstrakcija je kako smo i ranije istaknuli osnovni princip modeliranja uopće, a samim tim i u objektno orijentiranom pristupu neizostavan koncept. Proces razvoja klase, gdje su primarni

njeni interfejs i funkcionalnost u odnosu na samu implementaciju, nazivamo apstrakcijom u programiranju. Varijable su nam interfejs ka podacima, a metode ka funkcionalnosti klase. Namjena dizajnerskog toka rada jeste definirati klase, definirati interfejse klasa i opisati interakcije među klasama.

Enkapsulacija je usko povezana sa apstrakcijom, pa je prema tome i sama veoma važna u objektno orijentiranom pristupu. Enkapsulacija je mehanizam koji se koristi za skrivanje podataka, interne strukture i detalja implementacije objekta. Sve interakcije sa objektima su preko javnog interfejsa operacija.

Kohezivnost je mjera koja iskazuje koliko su dobro podaci i operacije uklopljeni u cjelinu, kao sastavni dijelovi klase. Definicija kohezivne klase kaže da je to ona klasa koja sadrži isključivo podatke i operacije relevantne za tu klasu.

Međusobna ovisnost je ustvari stepen neovisnosti klasa. Veze između klasa koje u sebi sadrže jezičke konstrukcije poput „ima“ ili „koristi“ ukazuju na usku povezanost među klasama. Što je više takvih veza u sistemu koje govore da klasa mora komunicirati sa drugim klasama kako bi ostvarila svoju funkciju, to je teže implementirati izmjene u sistemu, nadograditi klasu ili je zamijeniti sa drugom klasom, a također i iskoristiti takvu klasu za implementaciju nekog novog softverskog rješenja. Cilj je dakle graditi klase i sistem tako da je što manja njihova međusobna ovisnost.
Uz sve gore navedeno sistem koji je dobro dizajniran, je lako razumjeti, lako mijenjati i lako ponovno upotrijebiti.

9.2 Karakteristike lošeg dizajna

Loš dizajn ima mnogo različitih komponenti, a neke od njih su:

1. **Tvrdoća**: Sistem je teško mijenjati zato što svaki put kada se mijenja neki model element, mora da se mijenja i nešto drugo u neprekidnom slijedu promjena.
2. **Krtost**: Promjena jednog dijela sistema uzrokuje padove u drugim, totalno nevezanim dijelovima.

3. **Nepokretnost**: Teško je odmotati sistem u komponente koje se mogu ponovo upotrijebiti u drugim sistemima.
4. **Bespotrebna kompleksnost**: Postoji mnogo veoma pametnih struktura kôda koje, ustvari, nisu trenutno potrebne, ali bi mogle biti korisne kasnije.
5. **Bespotrebna ponavljanja:** Kôd izgleda kao da su ga pisala dvojica programera zvani *Cut* i *Paste*.
6. **Neprozirnost:** Rasvjetljavanje namjere tvorca predstavlja posebnu poteškoću.

Naš je cilj da oslobodimo kôd ovakvih svojstava. UML dijagrami mogu često pomoći zato što se mnoga ova svojstva mogu vidjeti proučavajući ovisnosti na dijagramu.

Mnoga od ovih svojstava su rezultat pogrešnog upravljanja ovisnostima. Pogrešno upravljanje ovisnostima podrazumijeva izgled kôda koji je zamršen masom dupliranja što dovodi do kôda koji se popularno naziva "špageta kôd".

Objektno orijentirani jezici imaju programske mehanizme koji pomažu u upravljanju ovisnostima. Interfejsi se mogu kreirati da razbiju ili invertiraju smjer pojedine ovisnosti. Polimorfizam dopušta modulima da pozove funkcije bez ovisnosti od modula koji ih sadrže. Dakako, svaki objektno orijentirani programski jezik daje dosta moći za oblikovanje ovisnosti na način na koji mi želimo.

Tako, kako mi želimo da su oblikovani! To je mjesto na kojem stupaju sljedeći principi:

9.3 Principi dobrog dizajna

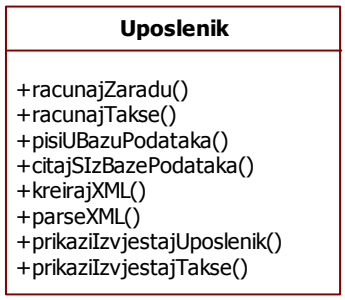
1.Princip pojedinačne odgovornosti

Princip pojedinačne odgovornosti (eng. Single Responsibility Principle-SRP) glasi:

KLASA BI TREBALA IMATI SAMO JEDAN RAZLOG ZA PROMJENU.

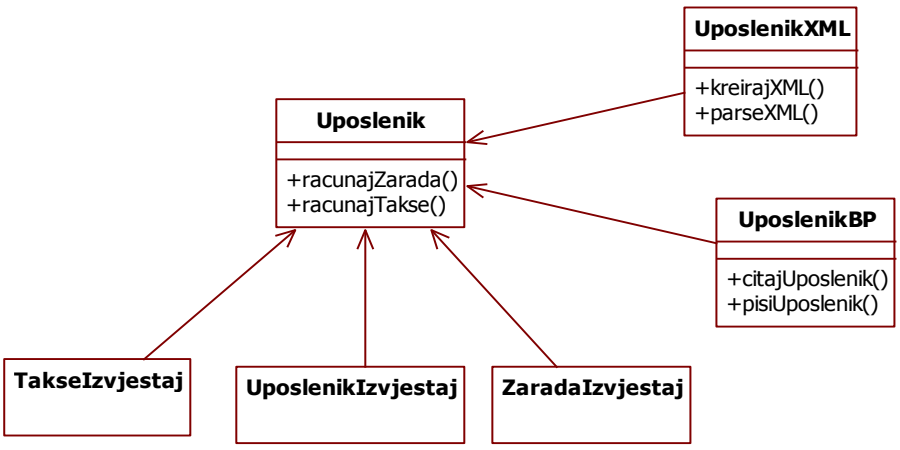
Klase treba da znaju samo o jednoj stvari. One trebaju imati pojedinačnu odgovornost. Treba postojati samo jedan razlog za promjenu klase.

Razmotrimo klasu `Uposlenik` na slici 9.1. Ova klasa zna kako da izračuna zaradu i takse, kako da učita i upiše podatke na disk, kako da konvertira podatke u XML i kako se štampaju različiti izvještaji. Ovdje se primjećuje krhkost, na primjer, ako promijenimo bazu podataka, moramo promijeniti `Uposlenik` klasu ili, ako promijenimo format izvještaja, moramo opet promijeniti `Uposlenik` klasu.



Slika 9.1: Klasa zna previše

U stvarnosti je potrebno da odvojimo sve ove koncepte u njihove vlastite klase tako da svaka klasa ima jedan i samo jedan razlog za promjenu. Dovoljno je da se `Uposlenik` klasa bavi sa plaćanjem i taksama, XML korespondentna klasa da se bavi pretvaranjem `Uposlenik` instance u XML i natrag. `UposlenikBazaPodataka` klasa da se bavi čitanjem i upisivanjem `Uposlenik` instance u i iz baze podataka, i individualne klase za svaki od različitih izvještaja. Ukratko, želimo razdvajanje poslova. Potencijalna struktura je prikazana na slici 9.2 .



Slika 9.2: Raspodjela poslova

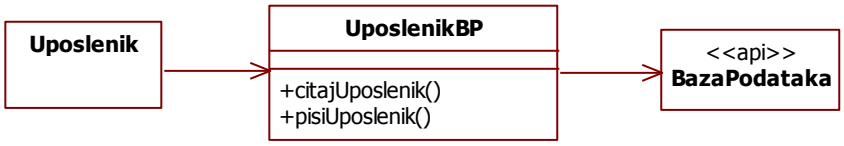
2.Otvoreno zatvoren princip

Otvoreno zatvoren princip (eng. Open Closed Principle-OCP) glasi:

> ENTITETI SOFTVERA (KLASE, MODULI, FUNKCIJE) TREBALI BI BITI OTVORENI ZA NADOGRADNJU, ALI ZATVORENI ZA MODIFIKACIJE.

Ovaj princip ima zbunjujuću definiciju, ali jednostavno značenje. Trebamo biti sposobni mijenjati okruženje oko modula bez promjene samog modula.

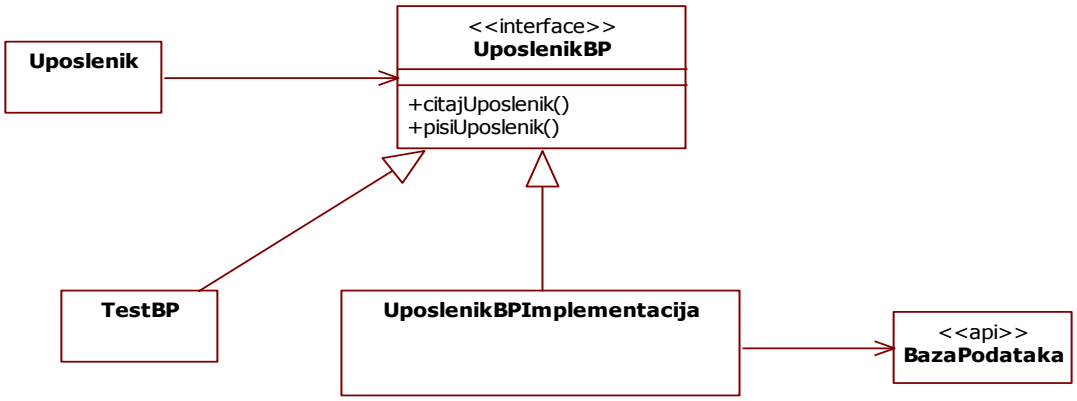
Razmotrimo primjer sa slike 9.3. Ona prikazuje jednostavnu aplikaciju koja se bavi `Uposlenik` objektom preko objekta za baze podataka zvanu `UposlenikBP`, koja se bavi direktno sa API–jem baze podataka. Ovo krši otvoreno zatvoren princip zato što promjena na implementaciji `UposlenikBP` klase može forsirati modifikaciju `Uposlenik` klase. `Uposlenik` je prijelazno vezano na API baze podataka. Bilo koji sistem koji sadrži `Uposlenik` klasu mora također sadržavati i klasu <<api>> `BazaPodataka`.
Testovi elemenata modela često su mjesto gdje želimo praviti kontrolirane promjene okruženja. Razmotrimo na primjer kako možemo testirati `Uposlenik` klasu. `Uposlenik` objekti prave promjene nad bazom. U testnom okruženju mi ne želimo promjene stvarne baze.



Slika 9.3: Narušavanje principa otvoren-zatvoren

Mi također ne želimo kreirati bazu samo u svrhu testiranja klase. Umjesto toga, želimo mijenjati okruženje tako da test obuhvaća sve pozive koje `Uposlenik` čini prema bazi, i verificira da su ti pozivi napravljeni korektno.

Ovo možemo uraditi pretvarajući `UposlenikBP` u interfejs kao na slici 9.4 Onda možemo kreirati izvedenice koje ili pozivaju stvarni API baze, ili koje podržavaju naše testove. Interfejs razdvaja uposlenika od API-ja baze i omogućava nam da mijenjamo okruženje baze bez bilo kakvog utjecaja na klasu `Uposlenik`.



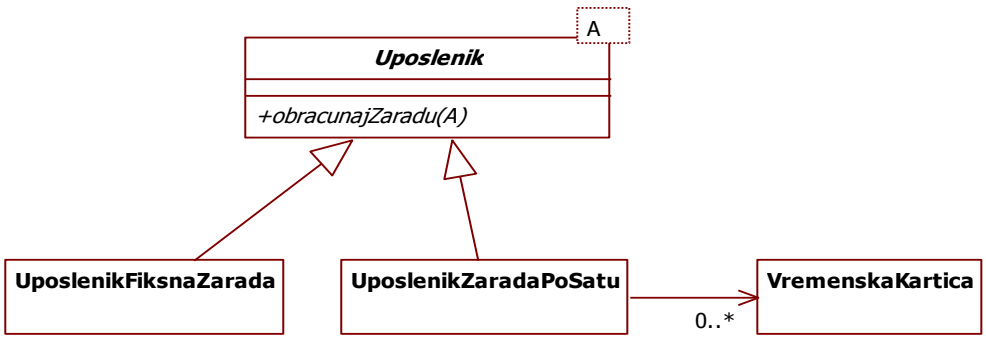
Slika 9.4: Ispunjen princip otvoren-zatvoren

3. **Liskov princip zamjene**

Liskov princip zamjene (eng. Liskov Substitution Principle-LSP) glasi:

> PODTIPOVI MORAJU BITI ZAMJENJIVI NJIHOVIM OSNOVNIM TIPOVIMA.

Da bi objasnili ovaj princip razmotrimo sliku 9.5. Klasa `Uposlenik` je apstraktna i ima apstraktnu metodu *`obracunajZaradu`*. Prilično je jasno da će je `UposlenikFiksnaZarada` implementirati da vrati zaradu (plaću) uposlenika koja se računa jednostavnim obračunom na osnovu ugovorene zarade. Također je prilično jasno da će `UposlenikZaradaPoSatu` implementirati metodu *`obracunajZaradu`* da vrati zaradu obračunatu na osnovu sati na vremenskoj kartici kojoj odgovara klasa `VremenskaKartica`.



Slika 9.5: Jednostavni primjer obračuna zarade

Što bi se desilo ako mi odlučimo da dodamo klasu `VolonterUposlenik` koja će predstavljati volontere odnosno osobe koje rade bez zarade? Kako bismo onda implementirali *`obracunajZaradu`*? Na prvi pogled ovo može izgledati očigledno. Implementirati *`obracunajZaradu`* da vraća nulu kao što je prikazano niže.

```
public class VolonterUposlenik extends Uposlenik {

   public double  obracunajZaradu() {

      return 0;

    }

}
```

Ali, da li je ovo ispravno? Da li to ima bilo kakve veze sa pozivom *`obracunajZaradu`* u `VolonterUposlenik`? Nakon svega, vraćanjem nule pretpostavljamo da je *`obracunajZaradu`* opravdana funkcija za pozivanje, i da je moguće izvršiti obračun zarade. Možemo se naći u nedozvoljenim situacijama kao štampanja i slanja platnih čekova sa ukupnom uplatom jednakom nula, ili nekoj drugoj sličnoj situaciji.

Sva ova konfuzija je došla zato što smo narušili Liskov princip zamjene. `VolonterUposlenik` klasa nije zamjenjiva sa `Uposlenik` klasom. Na korisnike `Uposlenik` klase utiče prisustvo `VolonterUposlenik`. Liskov princip zamjene se krši svaki put kada pokušamo napraviti nelegalan poziv funkcije na izvedenoj klasi.

Što je rješenje `VolonterUposlenik` problema? Volonteri nisu uposlenici. Nema smisla pozivati *`obracunajZaradu`* nad njima tako da oni ne bi trebali da se izvode iz `Uposlenik`, i da pozivaju *`obracunajZaradu`* metodu.

4.Princip inverzije ovisnosti

Princip inverzije ovisnosti (eng. Dependency Inversion Principle-DIP) glasi:

A. MODULI VISOKOG NIVOA NE BI TREBALI OVISITI OD MODULA NISKOG NIVOA. OBA BI TREBALO DA OVISE OD APSTRAKCIJA.

B. MODULI NE BI TREBALI OVISITI OD DETALJA. DETALJI BI TREBALI BITI OVISNI OD APSTRAKCIJA.

Ovaj princip možemo i jednostavnije interpretirati: Ne treba ovisiti od konkretnih klasa. Prilikom nasljeđivanja treba razmotriti slučaj da je osnovna klasa apstraktna.

Uopće, apstraktne klase i interfejsi mnogo se manje mijenjaju nego njihove konkretne izvedenice. Stoga je bolje ovisiti o apstrakcijama neko o stvarnim klasama. Slijedeći ovaj princip smanjuje se utjecaj koji promjena može imati na sistem.

Sasvim je sigurno ovisiti o konkretnim klasama koje se neće mnogo mijenjati. Vector i String klase koje, u većini objektno orijentiranih okruženja, predstavljaju tipove podataka neće se mijenjati mnogo u sljedećoj dekadi, tako da se možemo osjećati relativno sigurno upotrebljavajući ih. Za razliku od ovih klasa trebamo izbjegavati ovisnost od konkretnih klasa koje su pod aktivnim razvojem, i koje obuhvaćaju poslovna pravila koja će se vjerovatno mijenjati. Dobro je kreirati interfejse za ove klase i ovisiti od ovih interfejsa.
Sa UML tačke gledišta, ovaj princip je veoma lako provjeriti. Pratimo svaku strelicu na UML dijagramu i provjerite da li je cilj strelice interfejs ili jedna apstraktna klasa. Ako nije, i ako je to konkretna klasa, onda je princip inverzije ovisnosti narušen, i sistem će biti osjetljiv na promjene.

5.Princip izoliranja interfejsa

Princip izoliranja interfejsa (eng. Interface Segregation Principle-ISP) glasi:

KLIJENTI NE TREBA DA OVISE O METODAMA KOJE NEĆE UPOTREBLJAVATI.

Često vidimo klasu koja ima na desetine ili stotine metoda i koja se zbog toga naziva „debela“ klasa. Tipično, mi ne želimo takve klase u našim sistemima, ali ponekad su one neizbježne. Korisnici takvih klasa rijetko upotrebljavaju sve njihove metode. Tako korisnik može pozivati samo dvije ili tri metode u klasi koja ih deklarira na desetine. Nažalost, ti korisnici podliježu utjecaju promjena napravljenih nad metodama koje oni ne pozivaju. Mogu se zaštititi korisnici od metoda koje ne trebaju, dajući im interfejse prema onim metodama koje su im potrebne. Ovo čuva korisnika od promjena metoda koje ga se ne tiču. Također, to štiti korisnika od poznavanja implementacije objekta kojeg koristi.

9.4 Metrika objektno orijentiranog dizajna

Kako bi se na objektivan način okarakterizirao kvalitet softverskog rješenja razvijene su egzaktne metrike za objektno orijentiran dizajn. Dobar dizajn teško je dostići čak i iskusnim razvojnim inženjerima, bez primjene odgovarajućih principa i metodologija. Tako su 1994. godine Chidamber i Kemerer definirali šest metrika za kvalitet objektno orijentiranog pristupa. Tri od njih su kvalificirane kao metrike za dizajn, budući da postoji dovoljno informacija u toku izrade dizajnerskog rješenja, kako bi se napravile metričke kalkulacije.

Međusobna ovisnost između objekata klasa

Međusobna ovisnost između objekata klasa (eng. Coupling Between Object classes -CBO) je metrika kojom se kontrolira u kojoj mjeri su klase međusobno ovisne. CBO je suma od svih klasa koje nisu u vezi nasljeđivanja. CBO je efektivno broj svih vrsta asocijacije u kojima klasa participira. Niži brojevi indiciraju manju ovisnost i bolji su. Klase sa većim CBO su mnogo više ovisne od drugih klasa. Ove klase mogu biti mnogo teže za održavanje i za ponovno korištenje u novim softverskim sistemima. Većina klasa ima CBO između 0 i 2.

Dubina drveta nasljeđivanja

Dubina drveta nasljeđivanja (eng. Depth of Inheritance Tree - DIT) je metrika koja mjeri dubinu klasa unutar hijerarhije nasljeđivanja. Bazna klasa ima DIT 0. Njeno dijete ima DIT 1, a djeca njene djece imaju DIT 2. Klase koje imaju DIT veće od 2 su veoma kompleksne i održavanje takvih klasa je teško s obzirom na broj metoda koje nasljeđuje.

Broj djece

Broj djece (eng. Number Of Children -NOC) je metrika također povezana sa nasljeđivanjem. Ona mjeri broj potklasa koje se nasljeđuju direktno iz klase. Velike NOC vrijednosti su indikacije efektivnog ponovnog korištenja kôda, ali vrijednost veća od 10 može indicirati da su potklase neodgovarajuće naslijeđene.

Pored metrike koju su definirali Chidamber i Kemerer, i koja se više odnosni na ispitivanje individualnih klasa u sistemu, što je generalno metrika nižeg nivoa, postoje i metrike za objektno orijentirani dizajn koje kvalitet ispituju na višem nivou. Ovaj drugi set metrika ispituju osobine poput enkapsulacije i nasljeđivanja.

Faktor skrivanja metoda

Faktor skrivanja metoda (eng. Method Hiding Factor MHF) je mjera enkapsulacije i definira nevidljivost metode kao frakciju klasa za koje metoda nije vidljiva isključujući naslijeđene metode. Ako je metoda definirana kao privatna, ona je vidljiva samo za sebe i svoje potklase, i kaže se da ima nevidljivost 100% ili 1. Drugi metodi su u rangu od 0 do 1. MHF je prosječna nevidljivost za sve metode unutar sistema. Dobro dizajnirani sistemi imaju MHF aproksimativno oko 70%.

Faktor skrivanja atributa

Faktor skrivanja atributa (eng. Attribute Hiding Factor – AHF) je usko povezan sa MHF. Nevidljivost atributa je frakcija klasa za koje atributi nisu vidljivi, isključujući naslijeđene metode. AHF je prosječan faktor nevidljivost za sve atribute. U većini slučajeva, malo razloga postoji za AHF da bude ispod 100%.

Faktor nasljeđivanja metoda

Faktor nasljeđivanja metoda (eng. Method Inheritance Factor MIF) je mjera nasljeđivanja. MIF je broj naslijeđenih metoda u klasi podijeljen sa ukupnim brojem metoda u klasi, uključujući i metode nasljeđivanja. Interpretiranje MIF-a može biti teško. Velika MIF vrijednost indicira mnogo nasljeđivanja i može biti znak da se nasljeđivanje koristi neodgovarajuće. Mala vrijednost indicira manje nasljeđivanja i manje mogućnosti za ponovno iskorištenje kôda.

Faktor nasljeđivanja atributa

Faktor nasljeđivanja atributa (eng. Attribute inheritance factor AIF) je također mjera nasljeđivanja. AIF je broj naslijeđenih atributa u klasi podijeljen sa ukupnim brojem atributa u klasi uključujući i naslijeđene atribute. Kao i sa MIF, veća AIF vrijednost indicira veće korištenje nasljeđivanja, što može biti problem za testiranje i održavanje. Nasuprot tome, ako

potklase implementiraju većinu funkcionalnosti, tada će MIF i AIF vrijednost biti relativno niska, ali tada se može desiti da je klasa u velikoj mjeri odstupila od svoje apstrakcije, što može prouzrokovati i nedostatak ponovnog korištenja kôda.

Faktor polimorfizma

Faktor polimorfizma (eng. Polymorphism factor - PF) je definiran kao mjera aktualnog broja mogućih različitih polimorfnih situacija za neku klasu sa maksimalnim brojem mogućih različitih polimorfnih situacija za tu klasu.

Kao i sa MIF i AIF, velika PF vrijednost indicira mnogo nasljeđivanja i može indicirati da je nasljeđivanje korišteno neodgovarajuće. Nizak nivo indicira manje nasljeđivanje, ali i manje kôda koji se može ponovo koristiti.

Faktor međusobne ovisnosti

Faktor ovisnosti (eng. Coupling factor - CF) je mjera čvrstoće dizajna. CF je definiran kao mjera aktualnog broja mogućih veza u sistemu neke klase sa drugim i broja mogućih veza jedne klase sa drugim klasama sistema koje se ne odnose na nasljeđivanje. Broj mogućih veza je ustvari broj klasa-1.Visoka CF vrijednost je indikacija visoke ovisnosti. Niža CF vrijednost predstavlja malu ovisnost.

Npr. ako posmatramo sliku 7.17 na kojoj je prikazan dio dijagrama klase za `e-restoran` i ako želimo da odredimo faktor CF za grupu klasa, onda to možemo uraditi na sljedeći način:

Naziv klase	Aktualni broj mogućih veza (A)	Broj mogućih veza (B)	CF = A/B
`Narudzba`	4	9	0,44
`NarucilacJela`	2	9	0,22
`Jelovnik`	2	9	0,22
`StavkaNarudzbe`	2	9	0,22
`Jelo`	2	9	0,22
`SesijaZaNarucivanje`	6	9	0,67
`GUINarucivanje`	1	9	0,11
`DBInterfejs`	1	9	0,11
`InterfejsZaPlacanje`	1	9	0,11
`AplikacijaNarucivanje`	1	9	0,11

U tabeli je pobrojano 10 odabranih klasa za koje je određen CF faktor pojedinačno. Na osnovu izračunatih vrijednosti može se zaključiti da najlošiji faktor ima klasa `SesijaZaNarucivanje`, i zbog toga ona je najosjetljivija na promjene u sistemu.

Sljedeća tabela prikazuje faktor MIF:

Naziv klase	Naslijeđene metode (IM)	Definirane metode (DM)	IM+DM	IM /(IM+DM)	MIF
`Narudzba`	0	10	10	0/10	0
`NarucilacJela`	0	4	4	0/4	0
`Jelovnik`	0	9	9	0/9	0
`StavkaNarudzbe`	0	3	3	0/3	0
`Jelo`	0	7	7	0/7	0
`GostRestorana`	4	1	5	4/5	0,8
`...`	...	...	...	...	...

Iz ovog vidimo da nema puno nasljeđivanja metoda pa nema ni opasnosti po predvidivost ponašanja klasa. Klase kao što je klasa `GostRestorana` koja ima 4 naslijeđene metode, a samo jednu svoju imaju loš MIF faktor pa bi puno ovakvih klasa u sistemu bi predstavljalo loš dizajn.

Uspješan softverski projekt je onaj koji zadovolji ili premaši očekivanja naručioca, koji je razvijen u planiranom roku i ekonomično, i koji je pogodan za prilagodbe i zahtjeve za izmjenama. Uspjeh se zasniva, prije svega, na kvalitet objektno orijentirane analize i dizajna. Proces objektno orijentirane analize i dizajna zasniva se više na principima, nego na koracima koji se moraju izvršiti i u skladu s tim, prilikom dizajniranja sistema treba imati na umu principe dizajna koje smo izložili u ovom poglavlju.

Pitanja za ponavljanja

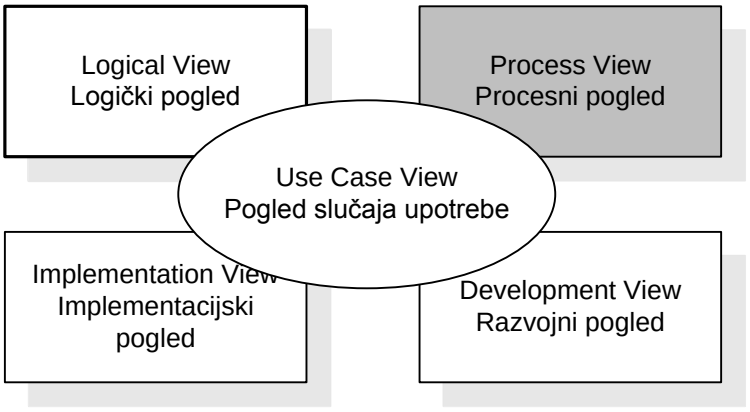
1.Kojim karakteristikama se odlikuje dobro dizajnirano objektno orijentirano softversko rješenje?
2. Navedite karakteristike lošeg dizajna?
3. Navedite osnovne principe dobrog dizajna?

4. Kako glasi princip pojedinačne odgovornosti ?
5. Kako glasi otvoreno zatvoren princip?
6. Kako glasi Liskov princip zamjene?
7. Kako glasi Princip inverzije ovisnosti ?
8. Kako glasi princip izoliranja interfejsa?
9. Navesti 3 metrike koje su definirali Chidamber i Kemerer, a koje se odnose na kvalitet objektno orijentiranog pristupa?
10. Što se kontrolira metrikom međusobna ovisnost između objekata klasa?
11. Objasniti metriku koja se odnosi na određivanje dubine drveta nasljeđivanja.
12. Koja dobivena vrijednost primjenom metrike određivanja broja djece znači da klase nisu dobro naslijeđene?
13. Nabrojite metrike objektno orijentiranog dizajna koje ispituju kvalitet na višem nivou.
14. Kako se određuje faktor skrivanja metoda?
15. Kako se određuje faktor skrivanja atributa?
16. Kako se određuje faktor nasljeđivanja metoda?
17. Kako se određuje faktor polimorfizma?
18. Kako se određuje faktor međusobne ovisnosti?

POGLAVLJE 10.

DIJAGRAM AKTIVNOSTI

Dijagram aktivnosti (eng. activity diagram) dozvoljava nam da specificiramo kako sistem radi. Dijagram aktivnosti je posebno dobar za opisivanje poslovnih procesa i poslovnih tokova. Ovaj dijagram je jedini dijagram koji pripada procesnom pogledu na sistem.



Slika 10.1: Dijagram aktivnosti pripada procesnom pogledu

Dijagram aktivnosti je jedan od najprihvatljivijih UML dijagrama pošto koristi simbole slične notaciji dijagrama toka. Ovi dijagrami imaju korijene, osim u dijagramima toka, u UML dijagramima stanja i Petrijevim mrežama. Ovi dijagrami su značajno mijenjani u svakoj verziji jezika UML, pa su i u verziji UML 2 ponovo dopunjeni i izmijenjeni. U verziji UML 1 postojali su kao specijalna vrsta dijagrama stanja što je izazivalo mnogobrojne probleme ljudima koji su modelirali poslovne tokove.

10.1. Namjena dijagrama aktivnosti

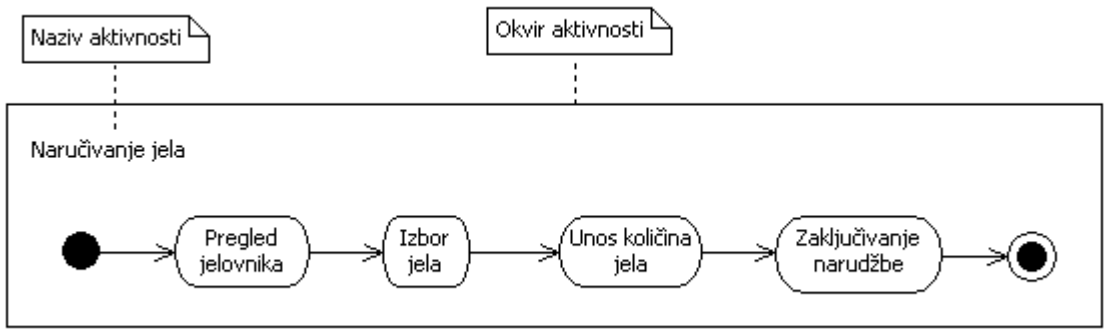
Dijagram aktivnosti može se koristiti tokom cijelog razvojnog procesa, od analize poslovnih procesa do programskog dizajna. Mogu se priložiti uz mnoge tipove objekata, to je ustvari generički tip dijagrama koji se može koristiti sa listom ispod:

- koriste se da modeliraju poslovne tokove;
- koriste se za identificiranje kandidiranih slučajeva upotrebe, za vrijeme ispitivanja poslovnih tokova;
- koriste se za identificiranje *pre*-i-*post* uvjeta za dijagram slučajeva upotrebe;
- koriste se za modeliranje radnih tokova između i unutar slučajeva upotrebe;
- koriste se za modeliranje kompliciranih operacija objekata;
- koriste se za detaljnije modeliranje kompleksnih aktivnosti prikazanim na dijagramom aktivnosti višeg nivoa.

10.2 Aktivnost i akcija

Aktivnost je proces koji se modelira, na primjer `Naručivanje jela`. Često se miješa termin aktivnosti i akcije. Akcija je korak u nekoj aktivnosti. U okviru aktivnosti `Naručivanje jela` postoji više akcija, neke od njih su `Pregled jelovnika`, `Izbor jela`. U samom dijagramu aktivnosti, aktivnost se sastoji od čvorova akcija, čvorova objekata i tokova koje vezuju prethodna dva tipa čvora.

Aktivnost se prikazuje sa pravougaonikom, koji predstavlja okvir aktivnosti, unutar kojeg je naziv aktivnosti. Akcija u dijagramu aktivnosti se prikazuje kao zaobljeni pravougaonik sa imenom akcije unutar njega. Na slici ispod je prikazan dijagram aktivnosti za aktivnost `Naručivanje jela`, sa akcijama (`Pregled jelovnika`, `Izbor jela`, `Unos količina jela`, `Zaključivanje narudžbe`).



Slika 10.2: Aktivnost u okviru pravougaonika (Naručivanje jela)

U praksi okvir aktivnosti često se izostavlja. Ekvivalenti prikaz dijagramu aktivnosti sa slike 10.2 je:

Slika 10.3: Aktivnost bez okvira

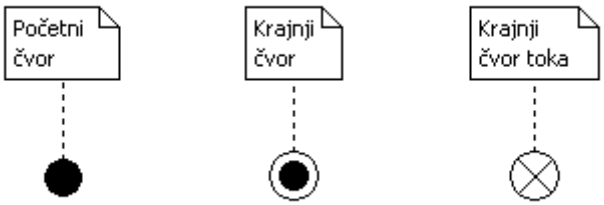
Pored simbola za uvedene pojmove za aktivnost i akciju na dijagramu vidimo i još neke dodatne simbole. Sintaksa dijagrama aktivnosti je veoma bogata ali i intuitivna, tako da možemo pretpostaviti da se strelice na prikazanoj aktivnosti odnose na tokove akcija, a da su tu i simboli za početnu i krajnju tačku dijagrama.

10.3 Početni, krajnji čvor aktivnosti

Postoje dva specijalna čvora koja pokazuju tačku gdje aktivnost počinje i tačke gdje aktivnost završava. Početni čvor je mjesto gdje počinje aktivnost, i obavezan je element svakog dijagrama aktivnosti. Svaki dijagram aktivnosti mora imati tačno jedan početni čvor. Prikazuje se kao crna tačka (slika 10.4).

Krajnji čvor aktivnosti se prikazuje kao crna tačka okružena sa krugom (slika 10.4). Pošto tok aktivnosti može imati više alternativnih ruta, aktivnost može terminirati na više različitih

tačaka, što uzrokuje da je moguće postojanje više krajnjih tačaka dijagrama. Na primjer, aktivnost Naručivanje jela ima dvije krajnje tačke: jednu nakon zaključivanja narudžbe, a drugu nakon odustajanja naručioca jela od naručivanja.
U okviru jedne aktivnosti može postojati više paralelnih tokova odnosno akcija. Neki tok može terminirati dok se drugi mogu nastaviti izvršavati. To se označava sa krajnjim čvorom toka. Krajnji čvor toka aktivnosti (slika 10.4) ne uzrokuje terminiranje aktivnosti, niti ima utjecaja na ostale tokove.



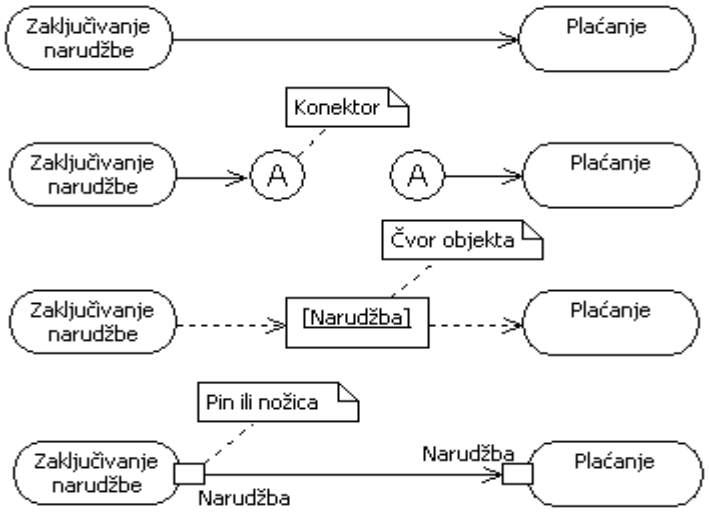
Slika 10.4: Početni, krajnji čvor aktivnosti i krajnji čvor toka aktivnosti

10.4 Tokovi i ivice

U jeziku UML 2, pojmovi tok i ivica su sinonimi i opisuju veze između dvije akcije. Najjednostavnija vrsta ivice je obična strelica između dvije akcije, kao što smo vidjeli u prethodnim primjerima.

Ukoliko je iz bilo kojeg razloga teško nacrtati ivicu, a da dijagram aktivnosti još uvijek ostane pregledan, možemo koristiti konektore. Konektori se uvijek crtaju u parovima, po jedan za ulazni i izlazni tok, i moraju biti isto obilježeni.

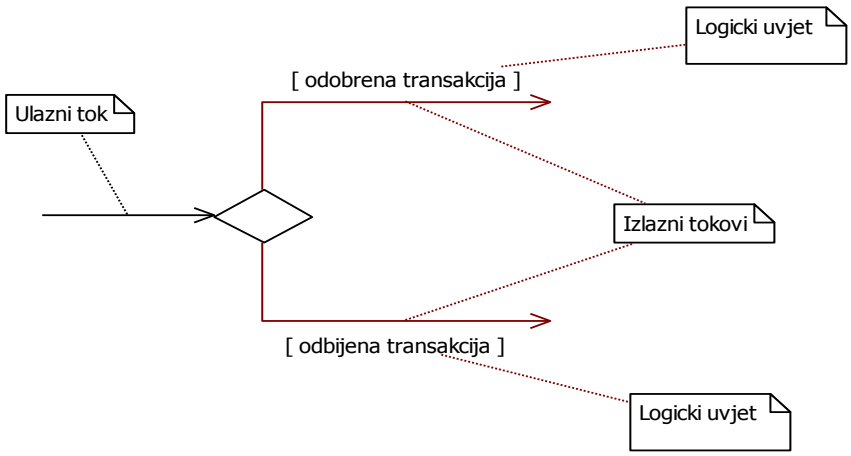
Ova dva načina označavanja ivica su prikazana na slici ispod.



Slika 10.5: Prikazivanje tokova strelicom i konektorom

10.5 Čvor odluke i čvor stapanja

Uvjetno ponašanje se pokazuje odlukama i stapanjima. Odluka (eng. decision), ima jedan ulazni tok i više uvjetnih izlaznih tokova. Svakom izlaznom toku je pridružen jedan logički uvjet, odnosno logički izraz između ugaonih zagrada. Pri svakom nailasku na odluku moguće je nastaviti samo jednim od izlaznih tokova, pa uvjeti trebaju biti uzajamno isključivi. Odluka se prikazuje simbolom romb i ima jedan ulazni tok i više izlaznih tokova što je i prikazano na slici ispod.



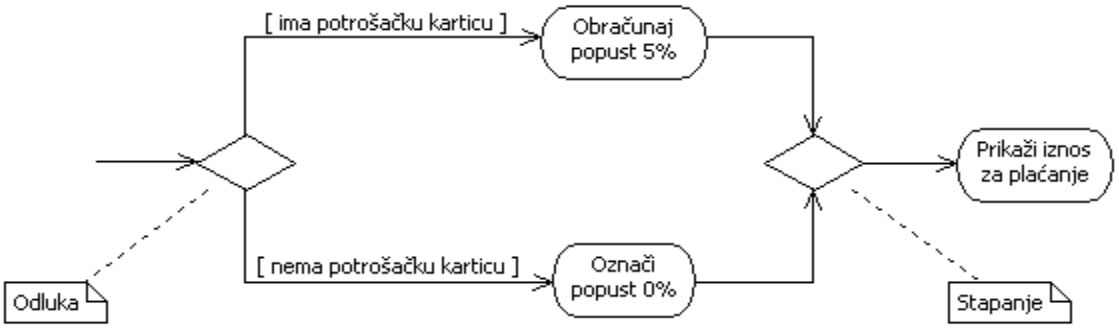
Slika 10.6: Čvor odluke

UML ne propisuje sintaksu za logičke uvjete vezane za tokove, to može biti kao i u drugim situacijama obilježavanja ograničenja obični tekst, sintaksa nekog programskog jezika, OCL.

Isti simbol se koristi za čvor stapanja čija uloga je stapanje svih alternativa.

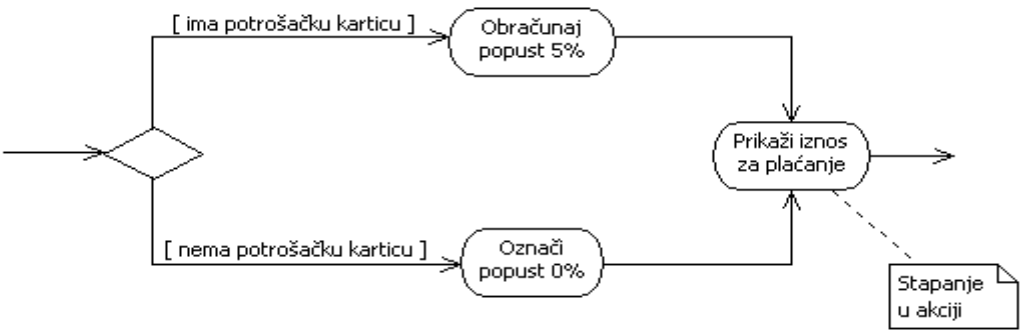
Stapanje (eng. merge) ima više ulaznih tokova i jedan izlazni. Stapanje označava kraj uvjetnog ponašanja započetog odlukom.

Na slici 10.7 je prikazan dio UML dijagrama koji pokazuje i odluku i stapanje. Odluka je o tome da li naručilac jela ima potrošačku karticu ili ne. Ako ima potrošačku karticu, onda mu se obračunava popust od 5%, a ukoliko nema potrošačku karticu onda se označi popust 0%. Alternative se stapaju i dalje slijedi zajednički tok, odnosno slijedi akcija `Prikaži iznos za plaćanje`.



Slika 10.7: Odluka i stapanje

Prilikom kreiranja dijagrama preporuka je da svakom čvoru odluke odgovara čvor stapanja, iako se često mogu vidjeti dijagrami gdje se alternativni tokovi sastaju u akciji što čini dijagram nepreglednim i zbunjujućim (slika 10.8).



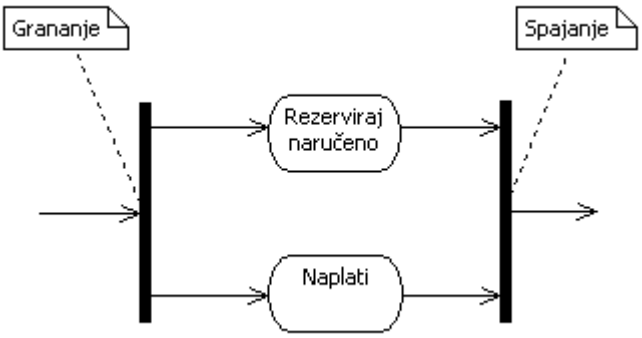
Slika 10.8: Stapanje u akciji

Potrebno je napomenuti da se simbol stapanja odnosi na alternativne tokove, ne na tokove koji se izvršavaju paralelno.

10.6 Grananje i spajanje

Aktivnosti koje modeliramo često sadrže akcije koje se izvršavaju paralelno. Tok aktivnosti se dijeli u više puteva i više puteva se spajaju u jedan tok korištenjem čvora grananja (eng. fork) i čvora spajanja (eng. join). Čvor grananja ima jedan ulazni tok i dva ili više izlazna toka. Čvor spajanja mora imati više ulaznih tokova i jedan izlazni tok.

Prilikom paralelnog izvršavanja aktivnosti redoslijed izvršavanja akcija neke aktivnosti nije bitan, neophodno je samo izvršiti sinhronizaciju što je, ustvari, i zadatak čvora spajanja.



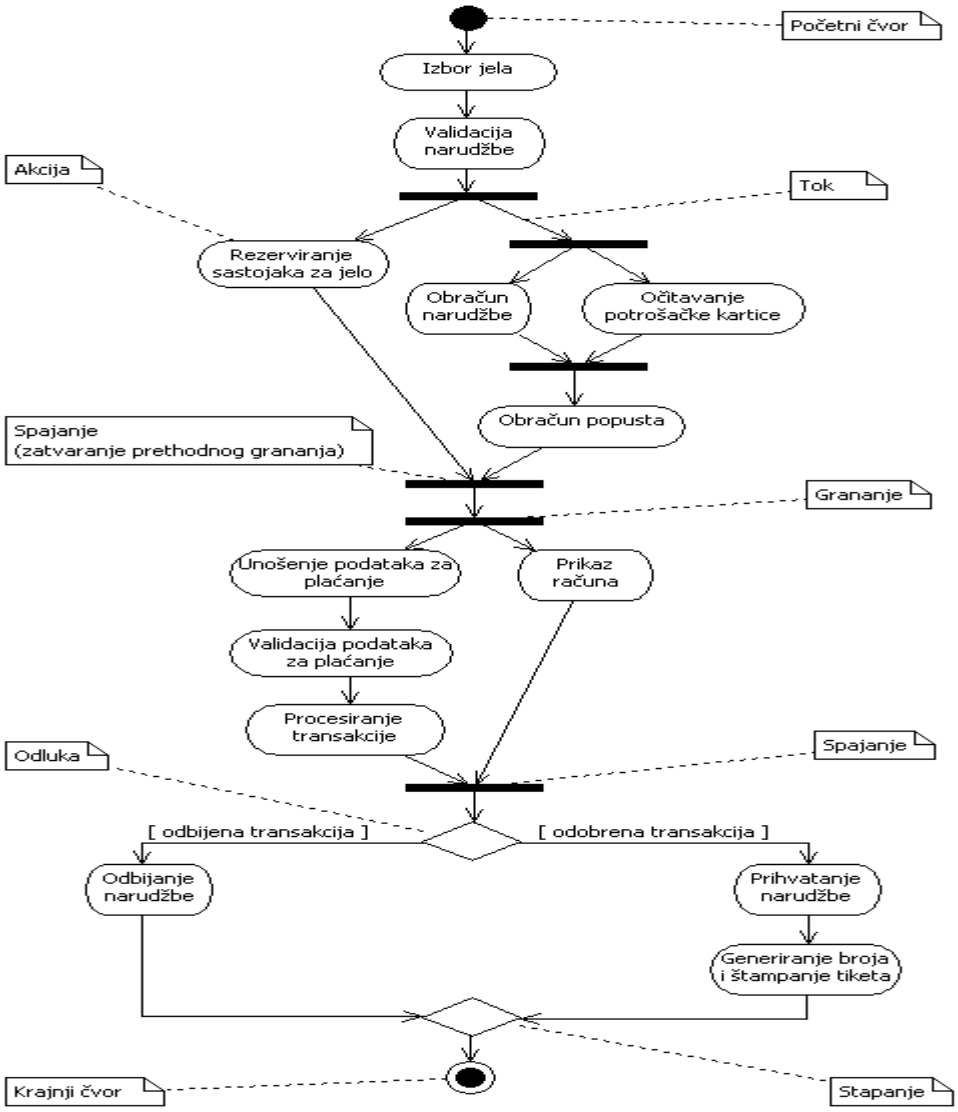
Slika 10.9: Simboli za čvor grananja i spajanja

10.7 Dijagram aktivnosti `e-restoran`

S obzirom na to da smo uveli već jedan broj elemenata neophodnih za konstrukciju dijagrama aktivnosti, vraćamo se na naš sistem. Pošto se dijagram aktivnosti najčešće koristi za detaljnije objašnjenje slučaja upotrebe, prikazat ćemo dijagram aktivnosti za dijagram slučaja upotrebe `Naručivanje jela`. Prva akcija je `Izbor jela`, nakon koje slijedi akcija `Validacija narudžbe`. Iza te akcije slijedi grananje koje ima jedan ulaz i dva izlaza. Prvi izlaz vodi do akcije `Rezerviranje sastojaka za jelo`. Paralelno s tim, drugi izlaz vodi do novog grananja, koje ima dva izlazna toka od kojih jedan vodi ka akciji `Obračun narudžbe`, dok istovremeno drugi tok vodi ka akciji `Očitavanje potrošačke kartice`. Potom se zadnje otvoreno grananje zatvara spajanjem iza kojeg slijedi akcija `Obračun popusta`. Nakon te akcije se vrši spajanje prvog grananja. Iza toga ponovno slijedi jedno grananje koje ima dva izlazna toka, od kojih jedan tok vodi

kroz akcije: Unošenje podataka za plaćanje, `Validacija podataka za plaćanje`, `Procesiranje transakcije`, a drugi tok vodi ka akciji `Prikaz računa`.

Poslije paralelnog izvršenja oba toka akcije se spajaju. Iza toga slijedi odluka kojom se bira jedan od dva toka u ovisnosti od toga je li „odbijena transakcija“ ili je „odobrena transakcija“. Ako je „odbijena transakcija“, onda slijedi akcija `Odbijanje narudžbe` i ide se u krajnji čvor, a ako je „odobrena transakcija“, onda slijedi akcija `Prihvaćanja narudžbe`, a zatim i akcija `Generiranje broja i štampanje tiketa` iza koje se ide u krajnji čvor. Svi tokovi koji su rezultat odluke se stapaju u jednoj tački iz koje slijedi dalje zajednički tok za sve njih.

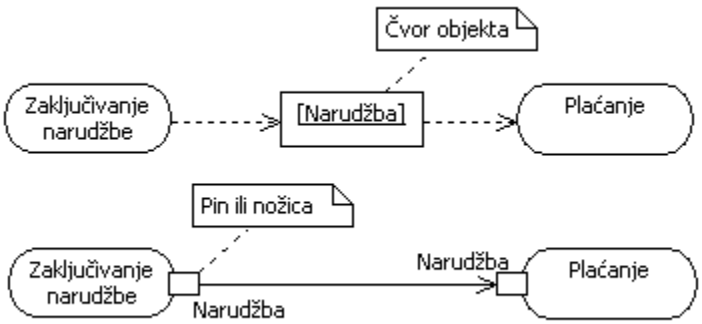


Slika 10.10: Dijagram aktivnosti za `e-restoran`

10.8 Objekti u aktivnosti

Objekti su veoma bitan aspekt prilikom modeliranja aktivnosti. Prilikom implementacije objektno orijentiranih sistema većina akcija vodi računa o objektima. Ponekad je veoma korisno u modelu aktivnosti označiti gdje akcija utiče na objekt. To se radi postavljanjem objekta na dijagram i uvezivanjem istog sa akcijom. Mjesto gdje se postavlja čvor naziva se čvor objekta i prikazuje se simbolom pravougaonika unutar kojeg je ime objekta. Veza između objekata i akcije je ovisnost.

Prikazivanje objekata se radi na jedan od dva načina: pomoću oznake objekta između dvije akcije ili uz pomoć nožica kao na slici 10.11.

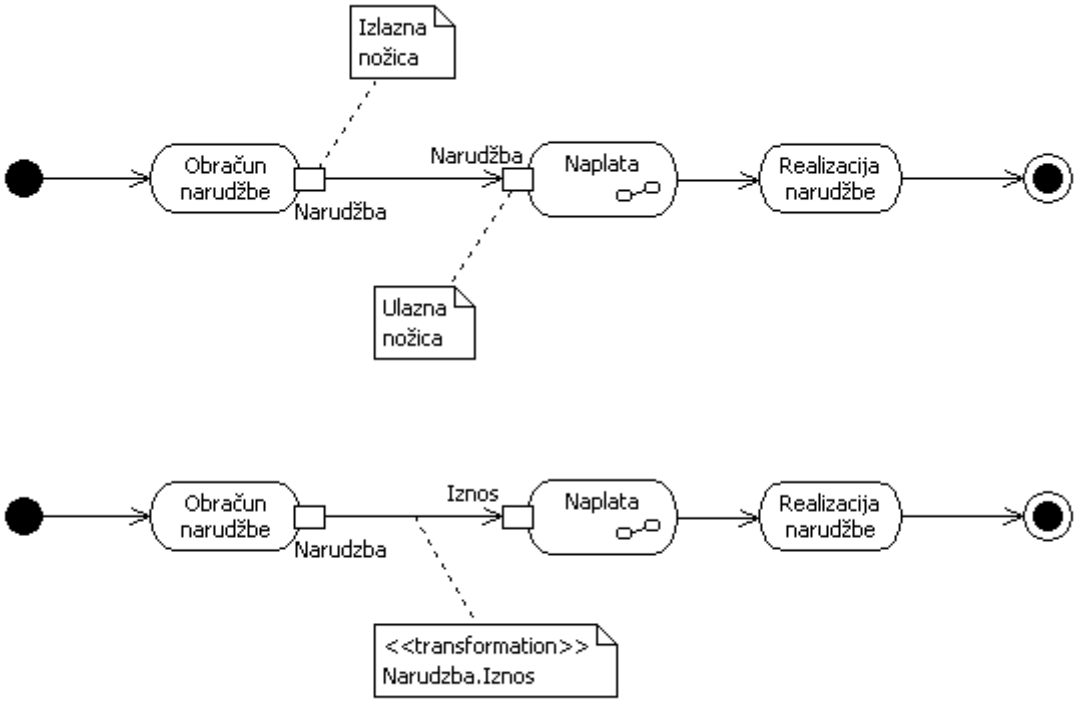


Slika 10.11: Prikazivanje objekata

Kao što vidimo na slici 10.11, ispod ili iznad grafičkog simbola za oznaku nožice se piše naziv objekta koji se prosljeđuje između akcija, međutim to ne mora nužno biti objekt, već i parametar bilo kojeg tipa. Moramo biti sigurni da izlazni parametri izlazne akcije odgovaraju ulaznim parametrima sljedeće akcije što je i slučaj na slici 10.11. Ako se parametri ne slažu, možemo naznačiti da je došlo do transformacije parametara tako što ćemo na dijagram uvrstiti obični tekstualni opis transformacije. Ova transformacija ne smije proizvoditi nikakve sporedne efekte, u suštini to je upit na izlaznom dijelu nožice, a tip rezultata treba da odgovara ulaznoj nožici.

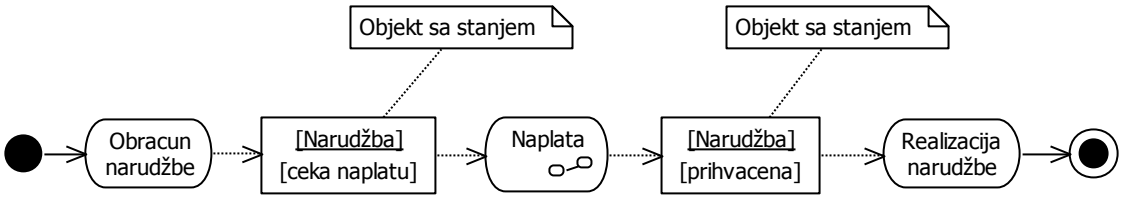
Na slici 10.12 je prikazan čvor koji predstavlja objekt `Narudzba` prikazan sa nožicama bez transformacije i sa transformacijom. Akcija `Obračun narudžbe` na izlazu generira objekt `Narudzba`, a podaktivnost `Naplata` koristi objekt `Narudzba` na svom ulazu, s tim da

podaktivnost `Naplata` koristi samo jedan atribut objekta `Narudzba` koji se zove `Iznos`. Ovu situaciju je moguće prikazati na dva načina. Prvi način je da se prikaže samo objekt `Narudzba` i kao izlazni i kao ulazni, a drugi način je da se eksplicitno naglasi da se kao ulazni objekt koristi samo `Iznos` te da tok predstavlja ujedno i transformaciju iz `Narudzba` u `Narudzba.Iznos`.



Slika 10.12: Objekt u dijagramu aktivnosti

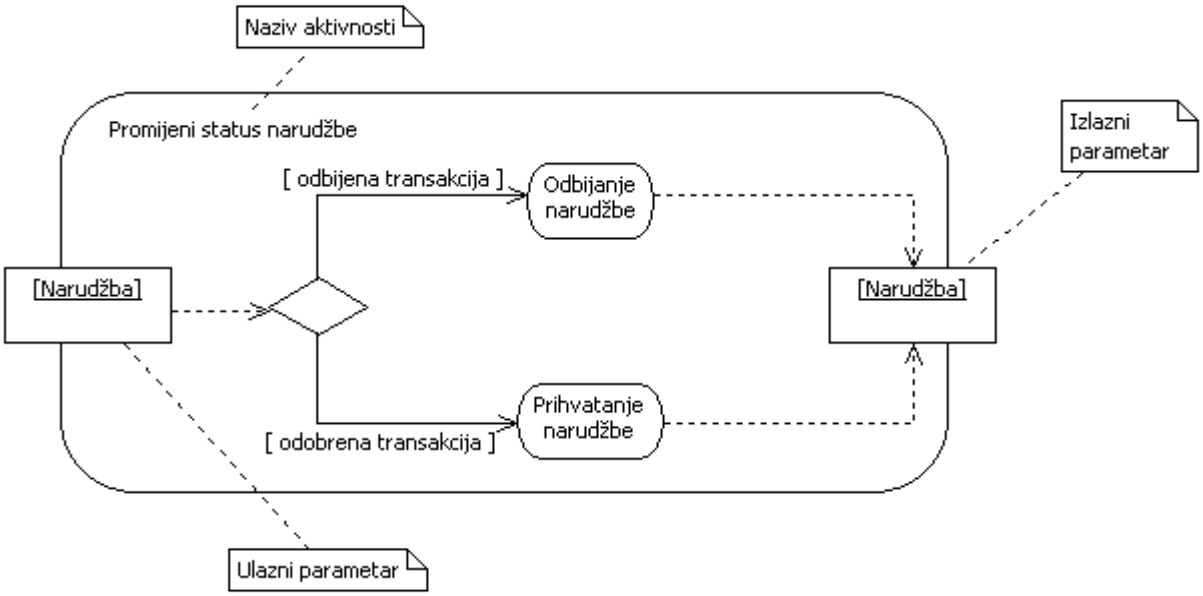
U okviru aktivnosti njen tok može uzrokovati promjenu stanja objekta, što se može naznačiti navođenjem stanja uz ime objekta.



Slika 10.13: Objekt sa stanjem

Objekt kao ulazni i izlazni parametar aktivnosti

Čvorovi objekata mogu biti ulazni i izlazni parametar aktivnosti. Prikazuje se na okviru aktivnosti kao na slici :



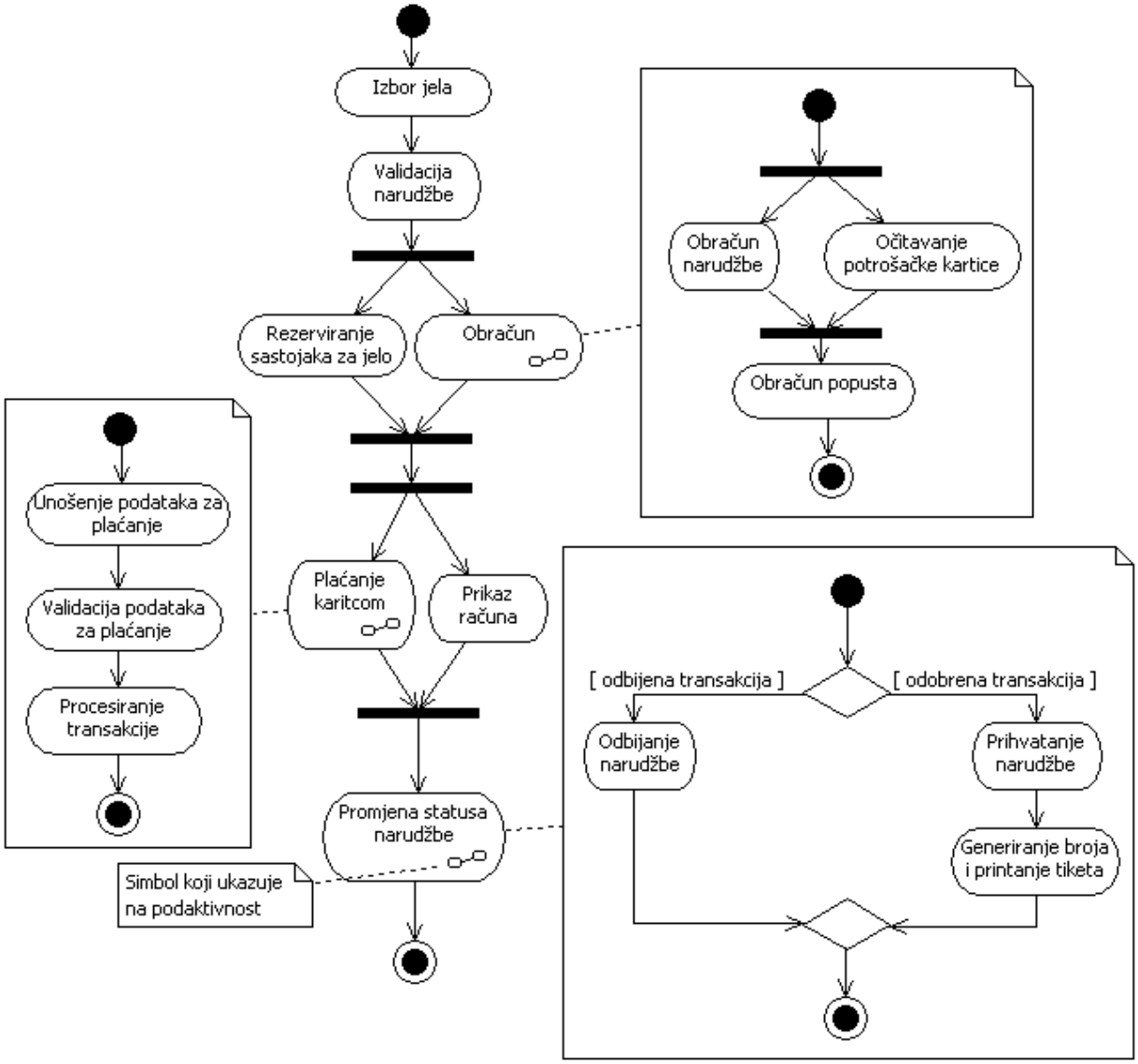
Slika 10.14: Čvor objekta kao ulazni i izlazni parametar aktivnosti

U slučaju prikazivanja ulaznih i izlaznih parametara aktivnosti ne prikazuje se inicijalni i krajnji čvor aktivnosti.

10.9 Pozivanje drugih aktivnosti

Dodavanjem detalja na dijagram aktivnosti, dijagram može postati prevelik. Isto tako neke sekvence akcija mogu se dešavati više puta. U takvim situacijama može se povećati preglednost dijagrama izdvajanjem sekvence akcija u poseban dijagram koji predstavlja podaktivnost glavne aktivnosti koju modeliramo. Podaktivnost se na osnovnom dijagramu aktivnosti prikazuje simbolom račve ili sličnim simbolom koji ukazuje na to. Akcije iz primjera na slici 10.10, koje se odnose na obračun narudžbe, zatim na plaćanje karticom i na promjenu statusa narudžbe nakon plaćanja mogu se grupirati kao podaktivnosti tako da se dobije jednostavniji i pregledniji dijagram koji je prikazan na slici 10.15. Akcije se mogu realizirati ili kao podaktivnosti ili kao metode klasa, a sintaksa za poziv metode je

ime_klase::ime_metode. Može se čak napisati i dio kôda unutar oznake akcije, ako se akcija ne može opisati pozivom jedne metode.

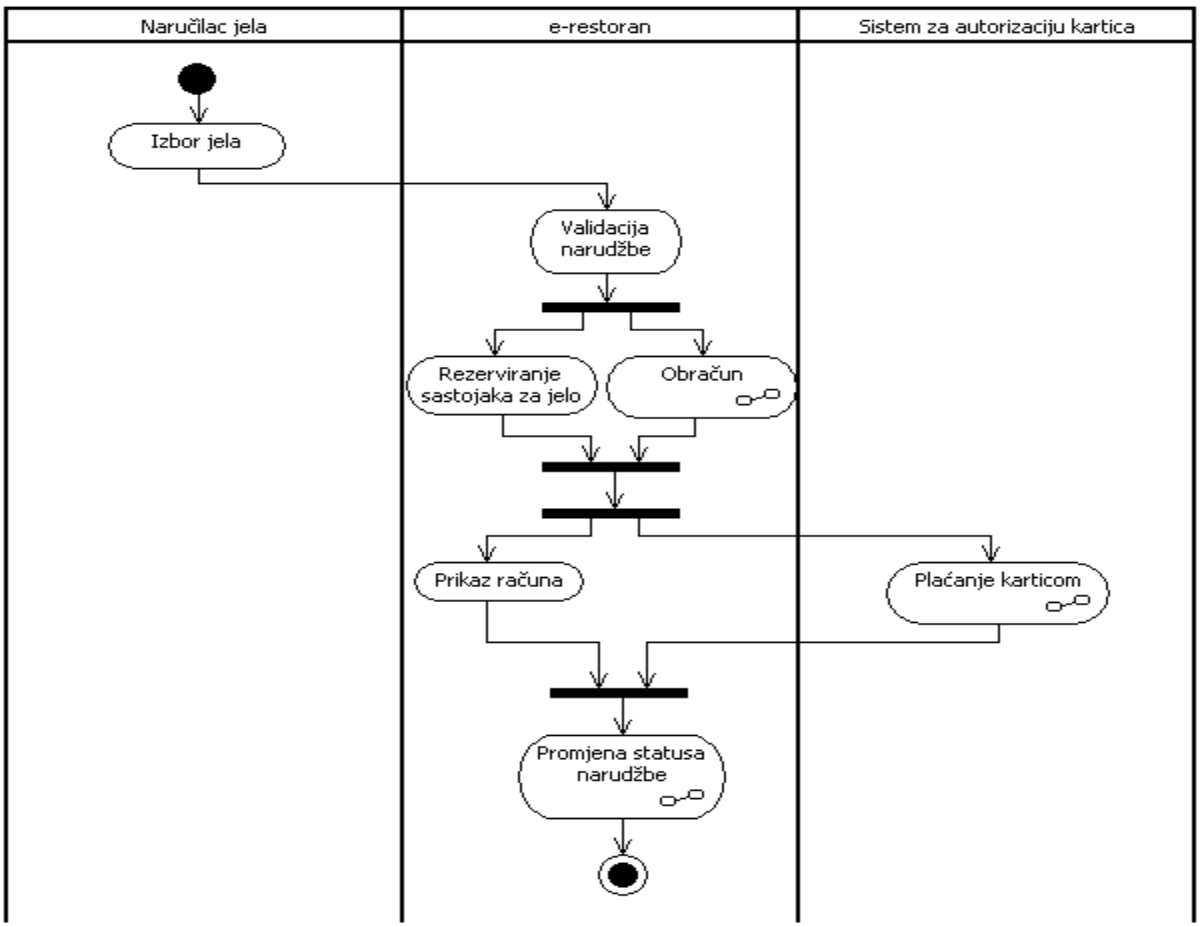


Slika 10.15: Dijagram aktivnosti sa podaktivnostima

10.10 Particije

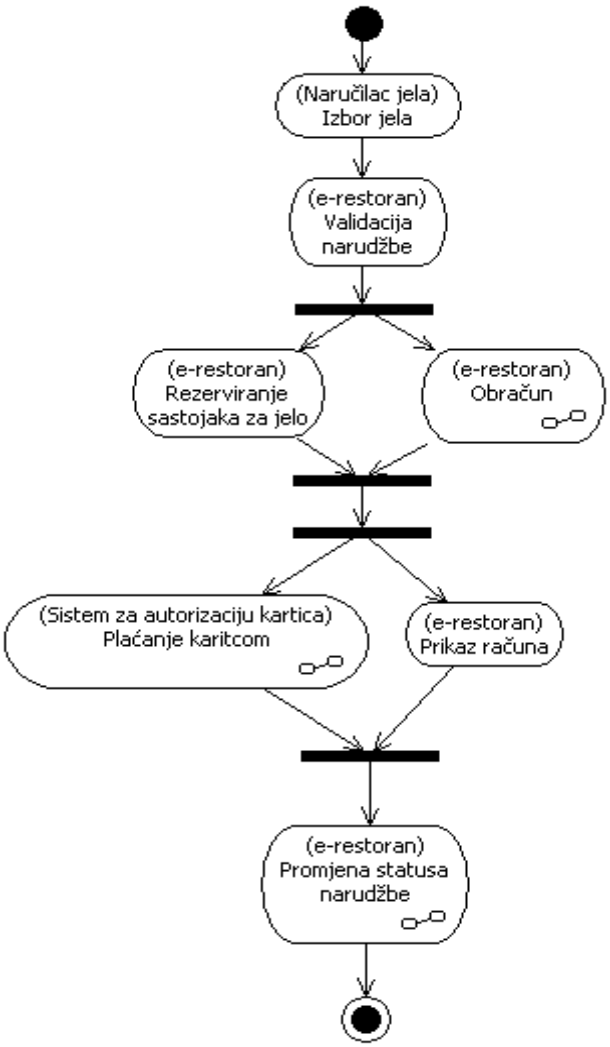
Dijagrami aktivnosti pokazuju što se dešava, ali ne i ko što radi. U programiranju to znači da ovaj dijagram ne pokazuje koja klasa je odgovorna za neku akciju. U modeliranju poslovnih procesa, dijagram ne prenosi informaciju o tome koji dio organizacije izvršava neku akciju. To ne mora biti problem, zato što se ponekad bolje koncentrirati na ono što se dešava, nego

na izvršioce pojedinih aktivnosti iako je naravno od velikog značaja prikazivanje samih izvršioca. Ako želimo prikazati izvršioce, dijagram aktivnosti možemo podijeliti u particije koje pokazuju koje akcije izvršava jedna klasa ili organizacijska cjelina, odnosno pokazuje odgovornost za pojedine akcije. Primjer zato je proces naručivanja u kojem učestvuje `Naručilac jela` koji vrši `Izbor jela`. Zatim učestvuje sistem `e-restoran` koji `Validira narudžbu`, vrši `Rezerviranje sastojaka za jelo`, vrši `Obračun` a zatim `Prikaz računa`. Paralelno sa akcijom `Prikaz računa` vrši se `Plaćanje karticom` u kome učestvuju svi (`Naručilac jela`, `e-restoran` i Sistem `za autorizaciju kartica`), ali s obzirom na to da je najznačajniji učesnik Sistem `za autorizaciju kartica` možemo njega smatrati nosiocem ove podaktivnosti. U primjeru sa slike 10.16 prikazana je jednostavna, jednodimenzionalna podjela na particije, i u ovakvoj podjeli, particija se često zove plivačka staza iz očiglednih razloga. U verzijama UML-a 1.x, to je bio jedini oblik particije. U verziji UML 2 možemo koristiti i dvodimenzionalnu mrežu, tako da plivačka metafora više ne vrijedi.



Slika 10.16: Particije na dijagramu aktivnosti

Postoji i drugi način označavanja odgovornosti za pojedine akcije a to je dodavanje odgovornosti uz ime akcije što je prikazano na slici 10.17.



Slika 10.17: Označavanje odgovornosti uz akciju

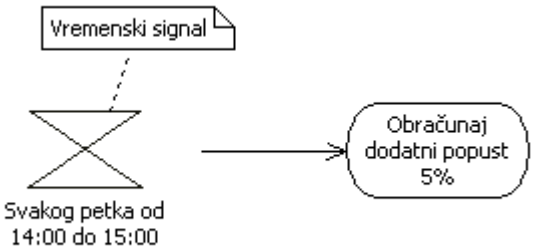
10.11 Signali

U primjerima pokazanim do sada, dijagrami aktivnosti imaju jasno definiranu početnu tačku koja odgovara pozivu nekog programa ili potprograma. Međutim, akcije mogu odgovarati i na signale.

Signali su specijalni tipovi objekta koji se koriste za upravljanje događajima. Jedna vrsta signala je vrijeme. Vremenski signali mogu ukazivati na kraj mjeseca ili računovodstvenog

perioda, ili na svaku mikrosekundu u upravljaču koji radi u realnom vremenu. Simbol vremenskog signala je pješčani sat.

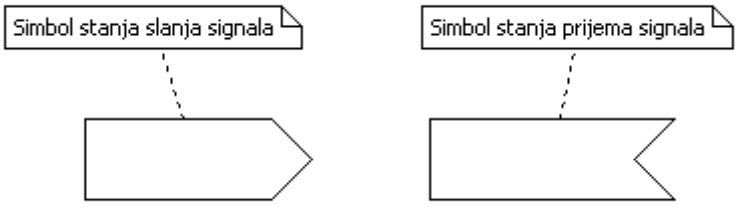
U primjeru, na slici 10.18 prikazana je upotreba vremenskog signala. Primjer pokazuje kako se svakog petka na sve narudžbe primljene u periodu od 14:00 h do 15:00 h obračunava dodatni popust od 5%.



Slika 10.18: Vremenski signal na dijagramu aktivnosti

Za vrijeme izvršavanja aktivnosti može biti potrebno odgovoriti na događaj koji se dešava izvan aktivnosti, ili sama aktivnost može prouzrokovati događaj za neku drugu aktivnost. Drugim riječima aktivnost može primati i slati signal. Signali predstavljaju interakcije sa eksternim učesnicima u obliku poruka.

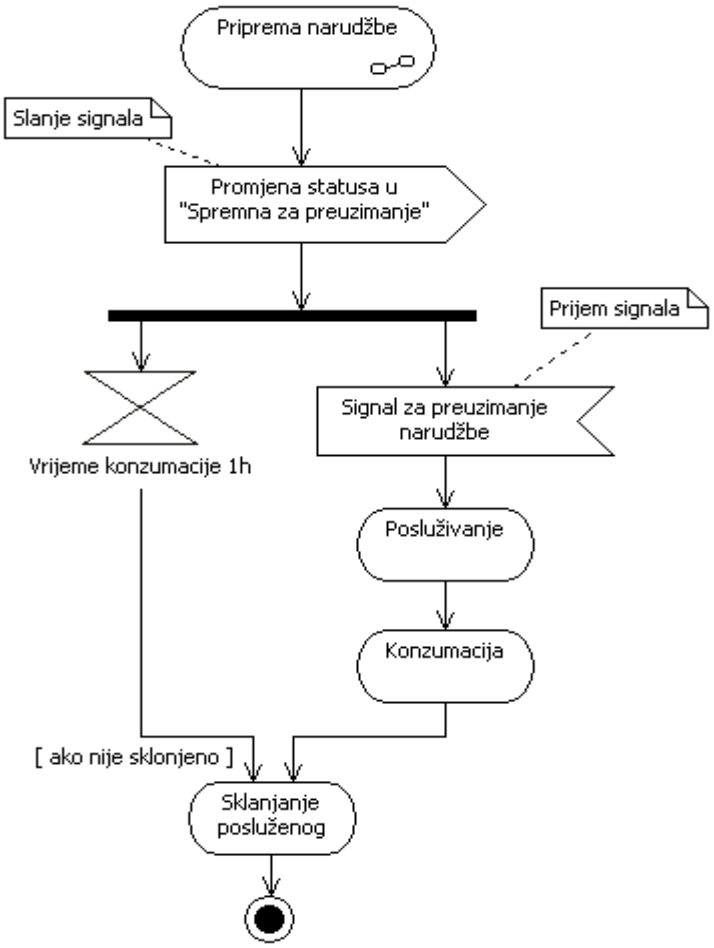
Simboli za prijem i slanje signalu su:



Slika 10.19: Simboli za prijem i slanje signala

Sljedeći primjer na slici 10.20 je nešto složeniji od gornjeg koji je demonstrirao upotrebu vremenskog signala jer pored vremenskog signala, prikazuje slanje i primanje signala. U ovom slučaju je to signal o promjeni statusa narudžbe (`Spremna za preuzimanje`). Kada

poslužilac vidi da je neka narudžba spremna za serviranje, on će istu odnijeti do naručioca jela i servirati za konzumaciju. Smatra se da je normalno potrebno vrijeme između promjene statusa narudžbe u `Spremna za preuzimanje` do završetka konzumacije iste 1 sat, stoga je vremenski signal sinhroniziran sa prijemom signala o promjeni statusa narudžbe, a obzirom na to da obje grane vode do akcije `Sklanjanje posluženog`, ne čekajući jedna drugu, onda nije prikazan i simbol spajanja. Ovo znači da će tok koji prvi stigne do akcije `Sklanjanje posluženog` prekinuti drugi tok.



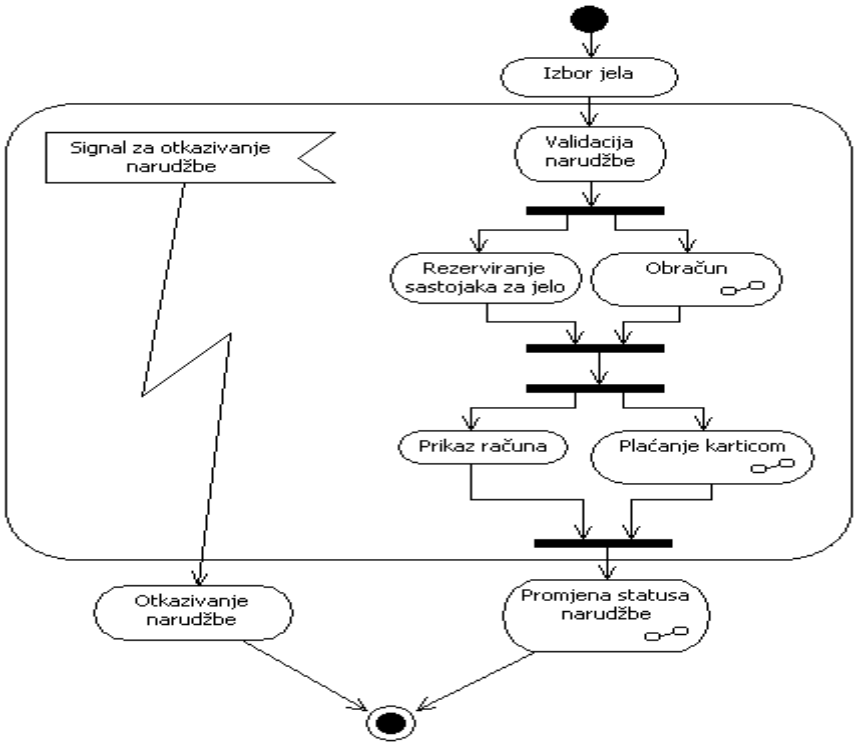
Slika 10.20: Slanje i primanje signala

Ako aktivnost počinje sa signalom tada čvor signala mijenja inicijalni čvor. Dakle, aktivnost može početi sa startnim čvorom ili nekom vrstom signala.

10.12 Prekid aktivnosti

Dijagrami aktivnosti koje smo prikazali do sada većinom imaju jedan krajnji čvor. U tom slučaju sve akcije u sklopu dijagrama imaju šansu da se okončaju.

Ponekad je potrebno da se modelira proces koji terminira sa događajem. To se može dogoditi u slučaju ako imamo neki dugi proces za vrijeme izvršavanja aktivnosti, koji se može prekinuti od strane korisnika. Da bi označili akcije koje se mogu prekinuti za vrijeme izvršavanja aktivnosti uvodimo područje prekida koji sadrži i akcije i događaj koji može uzrokovati prekid tih akcija. Područje prekida se označava sa pravougaonikom zaokruženih ivica. Događaj slijedi sa linijom koja izgleda kao munja što je vidljivo na slici ispod. Na slici 10.21 vidimo i da za vrijeme validacije narudžbe, njenog obračuna i drugih akcija do završteka plaćanja, naručilac jela može odustati od naručivanja i poslati `Signal za otkazivanje narudžbe`. Po prijemu tog signala, sve akcije koje se nalaze unutar područja prekida se prekidaju i ide se na akciju `Otkazivanje narudžbe`.

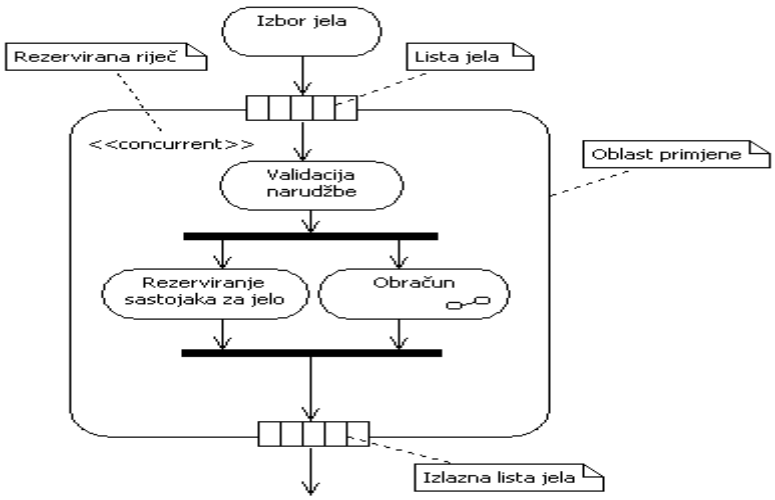


Slika 10.21: Područje prekida

10.13 Oblasti primjene

Na dijagramu aktivnosti, često ćemo dolaziti u situaciju da je potrebno prikazati da nakon neke akcije više puta treba izvršiti neku drugu akciju. To se može prikazati na nekoliko načina, ali je crtanje oblasti primjene najbolji način. Oblast primjene (eng. expansion region) jeste dio dijagrama aktivnosti u kome se akcije izvršavaju po jednom za svaki element kolekcije.

Prema primjeru sa slike 10.22, rezultat akcije `Izbor jela` jeste lista jela. Svaki element liste (jedno jelo) postaje ulazni žeton akcije `ValidacijaNarudžbe`, koja za svako jelo koje se dodaje u narudžbu, provjerava da li su raspoloživi sastojci za pripremu tog jela. Slično tome, izlaz iz akcije `Rezerviranje sastojaka jela` i podaktivnosti `Obračun` je lista jela za koje su rezervirani sastojci za pripremu i obračunati pojedinačni iznosi (cijena x količina). Kada se za svako jelo iz liste izvrši provjera raspoloživosti sastojaka, rezerviranje sastojaka za pripremu istog i obračun iznosa, te uveća ukupni iznos narudžbe i obračuna popust to jelo se stavi u listu jela na narudžbi. `Lista jela` na narudžbi je izlazna lista iz oblasti koja se daljim akcijama prosljeđuje u okviru jednog objekta `Narudžba`. U izlaznoj listi može se nalaziti isti broj žetona kao i u ulaznoj, ali ih može biti i manje, ukoliko sastojci za pripremu nekog jela nisu raspoloživi. U tom slučaju se oblast primjene ponaša kao filter.

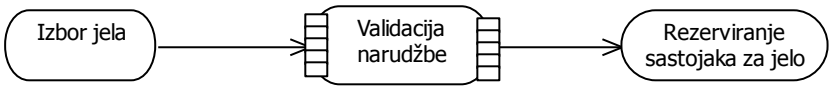


Slika 10.22: Oblast primjene

Rezervirana riječ <<concurrent>> označava da se akcije unutar oblasti primjene mogu obavljati paralelno. To znači da dok se za jedno jelo iz liste jela odvija akcija `Validacija`

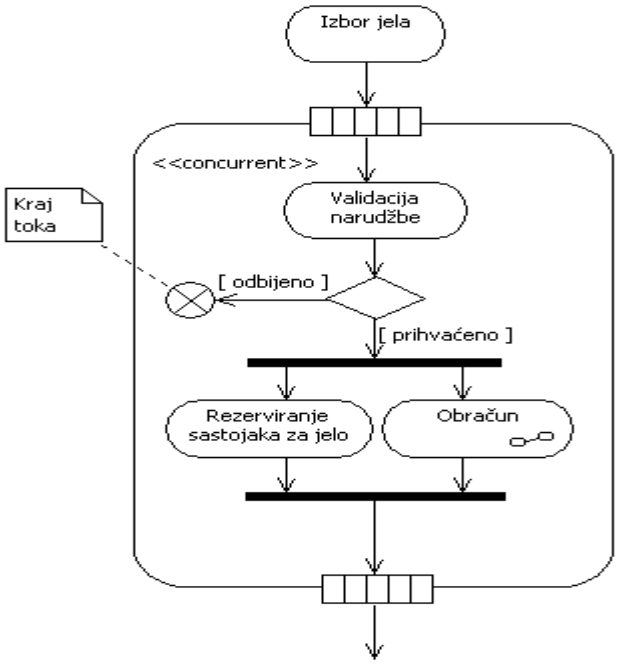
`narudžbe` za drugo jelo iz liste se može odvijati `Rezerviranje sastojaka za jelo`. Ipak, ovo ne znači da sa istim jelom iz liste dvije akcije mogu istovremeno manipulirati jer bi to moglo dovesti do kolizije.

Ako postoji samo jedna akcija koju treba pozivati više puta (npr. `Validacija narudžbe` kod koje bi se sva jela iz liste validirala pa bi onda cijela lista jela prešla na sljedeću akciju – `Rezerviranje sastojaka za jelo`), onda se može primijeniti skraćeni prikaz oblasti primjene kao na slici 10.23.



Slika 10.23: Skraćeni oblik prikaza oblasti primjene

Primjer iznad pokazuje oblast primjene koja se ponaša kao filter, odnosno izlazna kolekcija može biti manja od ulazne kolekcije oblasti primjene. Svako izabrano jelo se može odbiti sa značenjem njegovog poništavanja na narudžbi, ako u tom trenutku nema dovoljno sastojaka za njegovu pripremu. U slučaju odbijanja, potrebno je poništiti samo jedan žeton, odnosno završiti samo jedan tok, bez prekidanja cijele aktivnosti (slika 10.24).

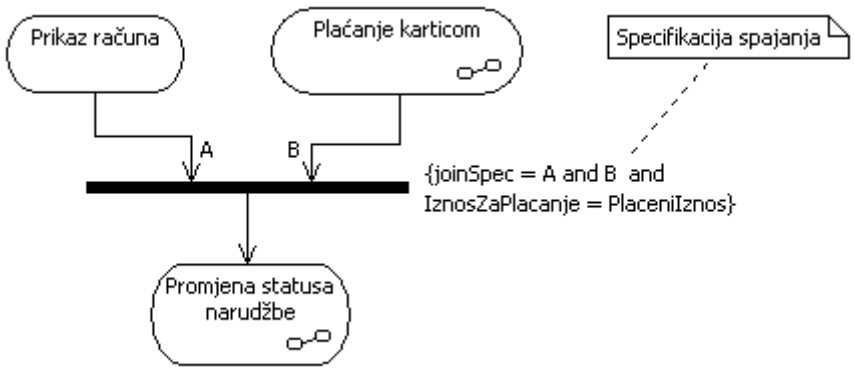


Slika 10.24: Tok se završava u aktivnosti

10.14 Specifikacije spajanja

Podrazumijeva se da spajanje dozvoljava izvršavanje izlaznog toka kada svi ulazni tokovi dođu do tačke spajanja. Formalnije, spajanje šalje žeton na izlazni tok kada stignu žetoni sa svih ulaznih tokova. Međutim, u nekim slučajevima je potrebno uvesti složenije pravilo.

Specifikacija spajanja je logički izraz koji se pridružuje spajanju. Vrijednost izraza se izračunava svaki put kada neki žeton stigne do spajanja i, ako je uvjet ispunjen, šalje se izlazni žeton. U primjeru na slici 10.25 to znači da se specifikacija spajanja izračunava svaki put kad dobijemo izlaze iz akcije `Prikaz računa` i podaktivnosti `Plaćanje karticom`. Prelazak na slijedeću akciju je moguć samo ako su akcija `Prikaz računa` i podaktivnost `Plaćanje karticom` završile, i ako je plaćeni iznos jednak iznosu za plaćanje.



Slika 10.25: Specifikacija spajanja

Završno razmatranje

Dijagrami aktivnosti opisuju radne tokove, fokusiraju se na sekvence akcija, opisuju kako sistem izvršava pojedine aktivnosti. Oni dozvoljavaju sekvencijalne i paralelne aktivnosti. Pored dijagrama komunikacije i sekvence, i ovi dijagrami daju opis dinamičkog ponašanja sistema. Oni se prilikom modeliranja sistema povezuju sa dijagramima slučajeva upotrebe, klasama i operacijama.

U nastavku slijede poglavlja posvećena dijagramima interakcije.

Pitanja za ponavljanje

1.Koja je osnovna namjena dijagrama aktivnosti?

2.Kojem pogledu pripada dijagram aktivnosti?

3.Koje vrste tokova se mogu opisati sa dijagramom aktivnosti?

4.Koja je razlika između aktivnosti i akcija?

5.Koja je UML notacija za aktivnost?

6.Koja je UML notacija za akciju?

7.Koja je UML notacija za početni i krajnji čvor aktivnosti?

8.Koliko može biti početnih čvorova u okviru dijagrama aktivnosti?

9.Koliko može biti krajnjih čvorova u okviru dijagrama aktivnosti?

10.Navesti osnovnu UML notaciju za prikazivanje tokova.

11.Navesti alternativne UML notacije za prikazivanje tokova.

12.Kako se prikazuje uvjetno ponašanje?

13.Prikazati UML notaciju za odluku stapanje.

14. Kako se predstavlja paralelno izvršavanje akcija?

15. Prikazati UML notaciju za grananje i stapanje.

16.Kako se u modelu aktivnosti označava utjecaj akcije na objekt?

17.Koja je UML notacija za prikazivanje podaktivnosti u okviru neke aktivnosti?

18.Što se postiže podjelom aktivnosti u podaktivnosti?

19. Što se postiže sa participama?

20. Što su signali i zašto ih koristimo?

21.Koja je UML notacija za prikazivanje vremenskog signala?

22.Koja je UML notacija za prikazivanje signala prijema i slanja?

23.Koja je osnovna namjena područja prekida u okviru dijagrama aktivnosti?

24.Kako se prikazuje konkurentno izvršavanje aktivnosti?

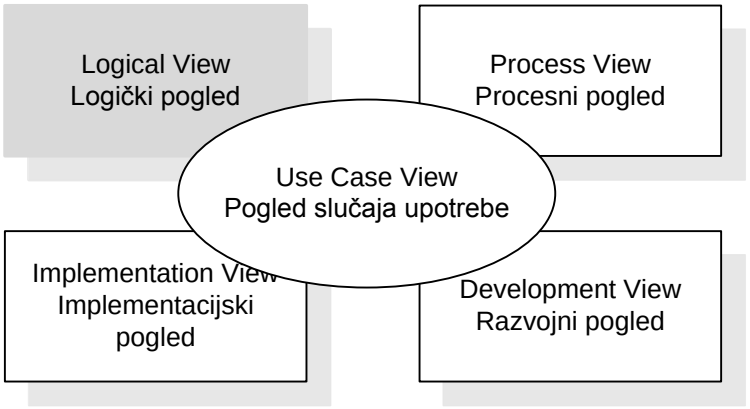
25.Sa kojim dijagramima tokom modeliranja se povezuju dijagrami aktivnosti?

POGLAVLJE 11.

DIJAGRAMI INTERAKCIJE – DIJAGRAM SEKVENCI

11.1 Dijagrami interakcije-uvod

U objektno orijentiranim sistemima da bi se postigla potpuna funkcionalnost sistema potrebno je da postoji interakcija objekata koja se ogleda u međusobnoj razmjeni poruka. Zajednički rad objekata i njihova međusobna komunikacija modelira se sa različitim varijantama UML dijagrama interakcije, koji modeliraju interakcije između pojedinih dijelova sistema. Ovi dijagrami formiraju dio logičkog pogleda modela koji sadrži apstraktne opise dijelova sistema uključujući i interakciju između tih dijelova.



Slika 11.1: Logički pogled uključuje interakcijske dijagrame

Grupi UML dijagrama interakcije pripadaju dijagrami sekvence, dijagrami komunikacije, dijagrami toka vremena i dijagrami pregleda interakcije.
Dijagrami sekvenci su najpopularniji od ova 4 tipa dijagrama interakcije.

Dijagrami interakcije su usko povezani i sa ostalim do sada uvedenim dijagrama. Ovim dijagramima prethode u modeliranju dijagrami slučajeva upotrebe. Dijagrami interakcije pokazuju tokove kroz slučajeve upotrebe, i to korak po korak, pokazuju koji su objekti potrebni da bi se tok obavio, koje se poruke razmjenjuju između objekata, koji akter inicira tok i koji je redoslijed slanja poruka.
Za `e-restoran`, postoji više različitih mogućih tokova za slučaj upotrebe `Naručivanje jela`, zbog toga je potrebno realizirati više dijagrama interakcije za ovaj slučaj upotrebe. Na jednom dijagramu interakcije se prikaže što se dešava pod uvjetom da se sve odvija kako treba (glavni tok događaja). Na drugim dijagramima interakcije se prikazuju alternativni tokovi, kao na primjer, što se dešava kada se odbije nečija kreditna kartica. Pomoću dijagrama interakcije, moguće je dokumentirati sve različite scenarije u sistemu.

Modeliranje interakcije korištenjem dijagrama komunikacije i dijagrama sekvence može rezultirati i u prepoznavanju novih klasa, atributa i operacija.

11.2 Dijagram sekvenci

Najčešći oblik dijagrama interakcije koji se koristi u praksi jeste dijagram sekvenci (eng. sequence diagram). Ovaj dijagram obično prikazuje jedan scenarij koji obuhvaća izvjestan broj objekata i poruka koje oni razmjenjuju u okviru slučaja upotrebe. Korištenjem dijagrama sekvenci može se opisati koje interakcije se izvršavaju kada se pojedinačni slučaj upotrebe izvršava i u kojem redoslijedu se ove interakcije izvršavaju.

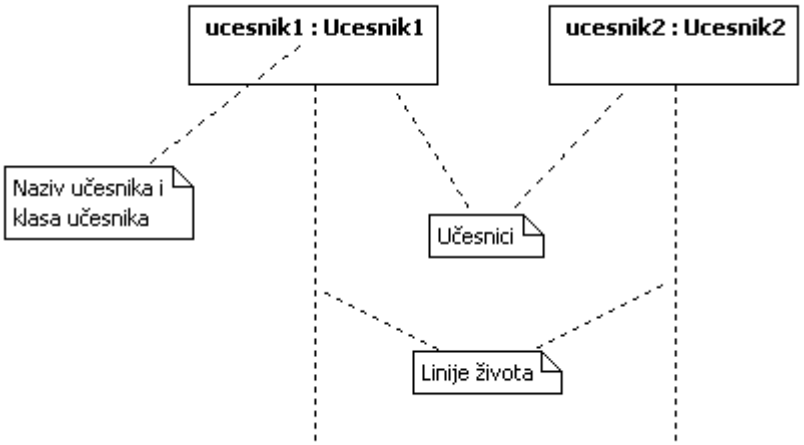
Dijagrami sekvenci mogu se koristiti za:

- modeliranje interakcija visokog nivoa između aktivnih objekata u sistemu;
- modeliranje interakcija visokog nivoa između podsistema;
- modeliranje interakcija između instanci objekata koji realizira slučaj upotrebe;
- modeliranje interakcija između objekata realizirane pomoću operacija (metoda).

Dijagrami sekvenci se koriste i za modeliranje ponovno iskoristivih fragmenata interakcije koji se mogu koristiti u drugim dijagramima sekvence ili u preglednim dijagramima interakcije.

11.3 Učesnici dijagrama sekvenci

Dijagram sekvenci je sačinjen kao kolekcija učesnika odnosno dijelova sistema koji su u interakciji jedan sa drugim za vrijeme sekvence.
Svaki učesnik ima odgovarajuću liniju života (eng. lifeline) i obično predstavlja instancu objekta. Linija života učesnika pokazuje postojanje učesnika u sekvenci i ista je povezana sa dijelom kreiranja i/ili brisanja učesnika u toku sekvence.



Slika 11.2: Osnovni elementi dijagrama sekvence

Veoma je važno gdje se postavljaju učesnici na dijagramu sekvence. Učesnici se uvijek postavljaju horizontalno i tako da se nikad ne preklapaju vertikalno, kao što je prikazano na slici 11.2. Po konvenciji, na sami vrh se postavljaju učesnici koji predstavljaju eksterne aktere i linije života koje predstavljaju objekte interfejsa.

Imenovanje učesnika/linija života

Učesnici, odnosno linije života na dijagramima sekvenci mogu se imenovati na različite načine na osnovu formata:

ime [selektor]: ime_klase ref dekompozicija

ime predstavlja ime objekta ili nekog podsistema

selektor predstavlja indeks selekcije objekta ako postoji skup objekata kao što je niz objekata

ime_klase je ime klase koja predstavlja tip za objekt

dekompozicija dekompozicija se koristi za predstavljanje dekompozicije sistema u okviru linije života. Referenciranje dekompozicijskog dijagrama sekvence je pomoću ključne riječi *ref*

Elementi koje ćemo izabrati za korištenje prilikom imenovanja ovise od informacija poznatih o učesniku. Slijedi tabela primjera imenovanja učesnika na osnovu date sintakse.

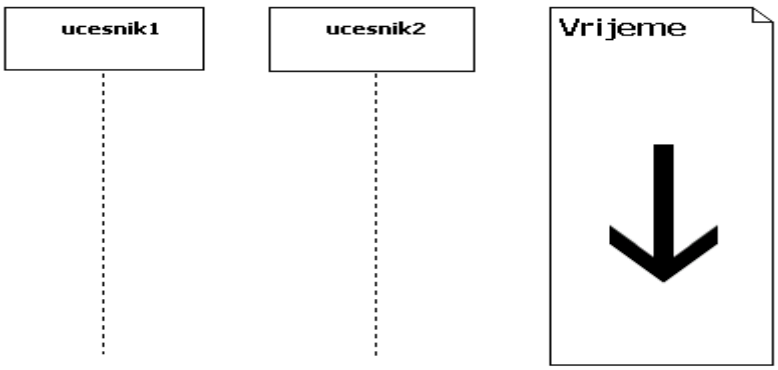
Tabela 11.1: Imenovanje učesnika

Ime učesnika/linije života	**Objašnjenje**
o	Objekt imena o
o:C	Objekt imena o, klase C
:C	Anonimni objekt klase C
o[i]	Objekt imena o koji se selektira sa indeksom
s ref sd3	Podsistem čija je interna struktura prikazana sa sd3 dijagramom sekvence ili komunikacije

Konvenciju imenovanja određujemo sami, a najčešće se koriste pravila kao u ovom slučaju: prva riječ u imenu učesnika piše se malim slovom, a druga koja se odnosi na ime klase piše se s velikim slovom.

11.4 Vrijeme

Dijagram sekvence opisuje redoslijed odvijanja interakcija, tako da je vrijeme bitan faktor. Vrijeme se prikazuje na dijagramu sekvence na način kako je to na slici 11.3.



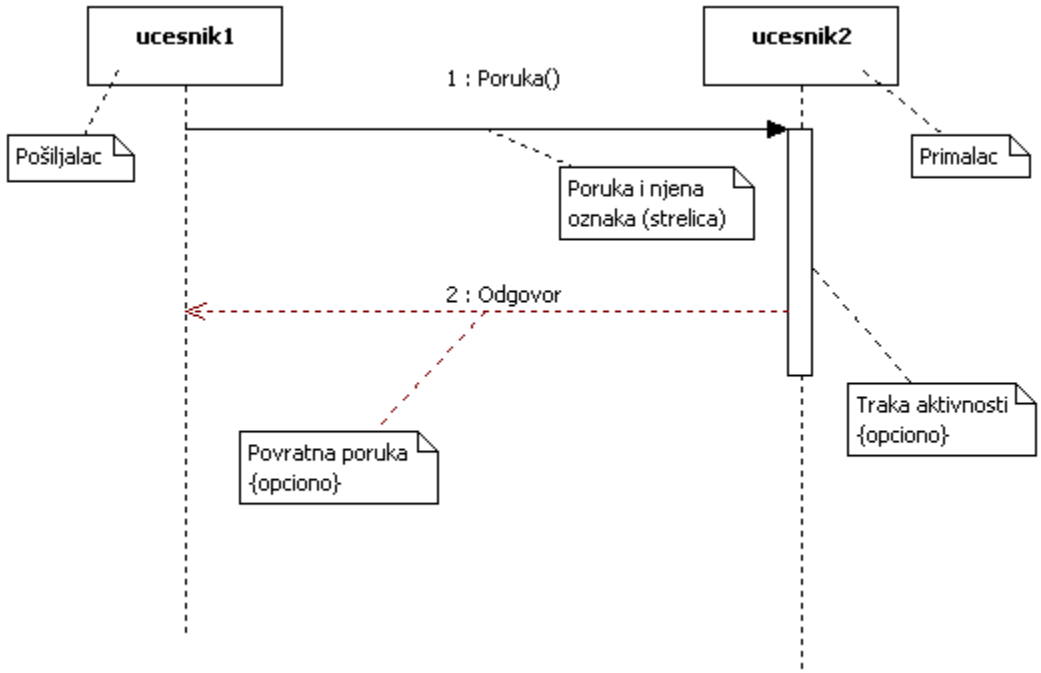
Slika 11.3: Označavanje vremena

Vrijeme na dijagramu sekvence počinje na početku stranice sa zaglavljem učesnika i ima progres duž stranice. Redoslijed smještanja interakcija koje su smještene na dijagramu sekvenci indicira redoslijed u kome se interakcije odvijaju u vremenu. Vrijeme na dijagramu sekvence je vezano za redoslijed dešavanja poruka, a ne trajanja istih.

11.5 Događaji, signali i poruke

Najmanji dio interakcije je događaj. Signali i poruke su različita imena za isti koncept: signal je terminologija koju najčešće koriste dizajneri sistema, dok softverski dizajneri više preferiraju izraz poruka. Na primjer događaj je izbor jela na jelovniku. Taj događaj generira poruku prema sistemu koja prosljeđuje, kao argument, informaciju o tome koje je jelo izabrano. Prijem te poruke u sistemu je također događaj koji uzrokuje sljedeću poruku u interakciji. Ta poruka može biti interna u sistemu ili pak usmjerena ka nekom od učesnika u interakciji.

Interakcija na dijagramu sekvence se dešava kada jedan učesnik ima potrebu da pošalje poruku drugom učesniku. Poruka se predstavlja strelicom usmjerenom od pošiljalaca poruke ka primalaca poruke što je i prikazano slikom ispod.



Slika 11.4: Prikaz interakcije

11.6 Trake aktivnosti

Kada se poruka proslijedi učesniku uzrokuje se uključivanje učesnika u neku aktivnost u cilju obavljanja nekog zadatka. Za učesnika koji je primio poruku kažemo da je aktivan. Da bi pokazali da je učesnik aktivan i da radi nešto, može se koristi traka aktivnosti kao na slici 11.4.

Trake aktivnosti jasnije naglašavaju metode koje se izvršavaju prilikom interakcije učesnika. Trake aktivnosti nisu obavezne u UML-u, ali mogu biti korisne u razjašnjavanju ponašanja.

11.7 Označavanje poruka

Format za poruku je:

[atribut=]ime_signala_poruke[(argumenti)]:[povratni_tip]

Sve je opcionalno osim imena poruke.
atribut = i *povratni_tip* se koristi samo u povratnim porukama, koje ćemo objasniti u nastavku. *Atribut* se koristi da bi se smjestila povratna vrijednost poruke, a povratni_tip da bi se specificirao tip te povratne vrijednosti.

Može se specificirati više argumenata poruke, svaki odvojen sa zarezom. Format argumenta je:

<ime>:<klasa>

Izbor elemenata prilikom imenovanja poruke ovise od poznatih informacija o poruci, a ilustracija imenovanja je data u tabeli ispod.

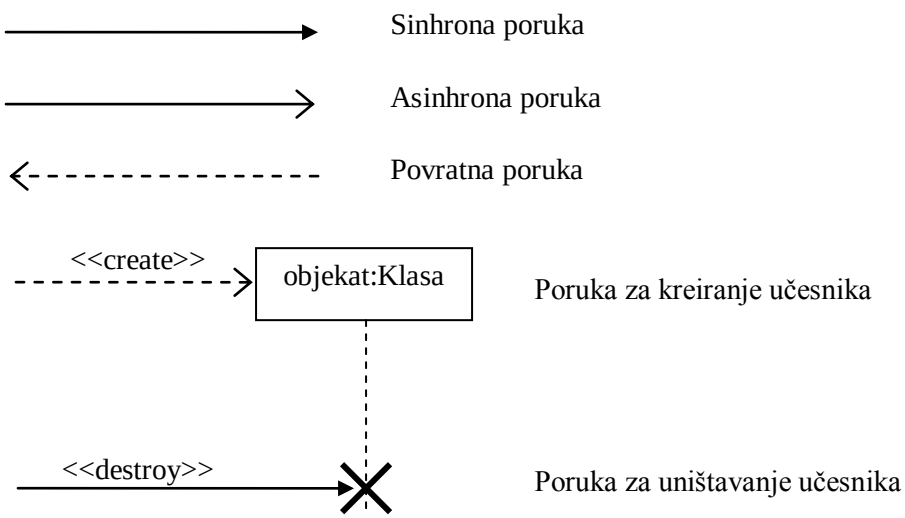
Tabela 11.2: Imenovanje poruka

Primjer poruke	Objašnjenje
uradiNesto()	Ime poruke je uradiNesto ali dodatne informacije nisu poznate u ovom trenutku
uradiNesto(broj1:Broj, broj2:Broj)	Ime poruke je uradiNesto, i ima dva argumenta, broj1 i broj2 i oba su tipa klase Broj
uradiNesto():Klasa1	Ime poruke je uradiNesto; nema argumente i vraća objekt tipa klase Klasa1
rez=uradiNesto():Klasa1	Ime poruke je uradiNesto; nema argumente i vraća objekt klase Klasa1 koji je dodijeljen atributu rez poruke pošiljalaca

11.8 Tipovi poruka

Veoma je bitno razumjeti koji tip poruke se razmjenjuje između učesnika. Pošiljalac poruke može čekati dok se poruka koju je poslao obradi i u tom slučaju govorimo o sinhronim porukama. Suprotno, pošiljalac poruke može poslati poruku bez čekanja na bilo kakvu povratnu informaciju i tada govorimo o asinhronim porukama.

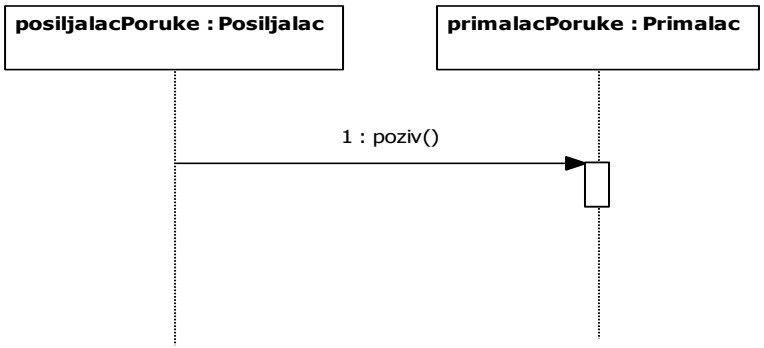
Na dijagramu sekvenci možemo prikazati različite tipove poruka korištenjem različitih linija i strelica, kao što je pokazano na slici 11.5.



Slika 11.5: Tipovi poruka

11.9 Sinhrone poruke

Kao što smo već objasnili sinhrona poruka se uključuje kada pošiljalac poruke čeka da primalac poruke obradi poruku, izvrši specificiranu metodu i vrati informaciju da je obradio tu poruku.



Slika 11.6: Sinhrona poruka

Prilikom mapiranja u kôd objekt označen kao `posiljalacPoruke` ustvari poziva metod (Java, C++) na objektu `primalacPoruke` i čeka da `primalacPoruke.poziv()` metoda vrati povratnu vrijednost prije nego što nastavi sa budućom interakcijom.

Java :

```
public class Primalac
{
     public void poziv()
     {
          // ...
     }
}

public class Posiljalac
{
    private Primalac primalacPoruke;

    // ...

    public radiNesto()
    {
          // Posiljalac aktivira poziv() metod
          This.primalacPoruke.poziv();
          // Poslije čeka da metod vrati vrijednost
          // prije nego što nastavi sa ostatkom posla
    }
}
```

C++ :

```
// Primalac.h
// ----------------------------------
class Primalac
{
public:
       void Poziv();
};

// Primalac.cpp
// ----------------------------------
void Primalac::Poziv
{
       // ...
};

// Posiljalac.h
// ----------------------------------
#include "Primalac.h"
class Posiljalac
{
private:
     Primalac *primalacPoruke;

     // ...

public:
       RadiNesto();
};

// Posiljalac.cpp
// ----------------------------------
void Primalac::RadiNesto()
{
       // Posiljalac aktivira poziv() metod
```

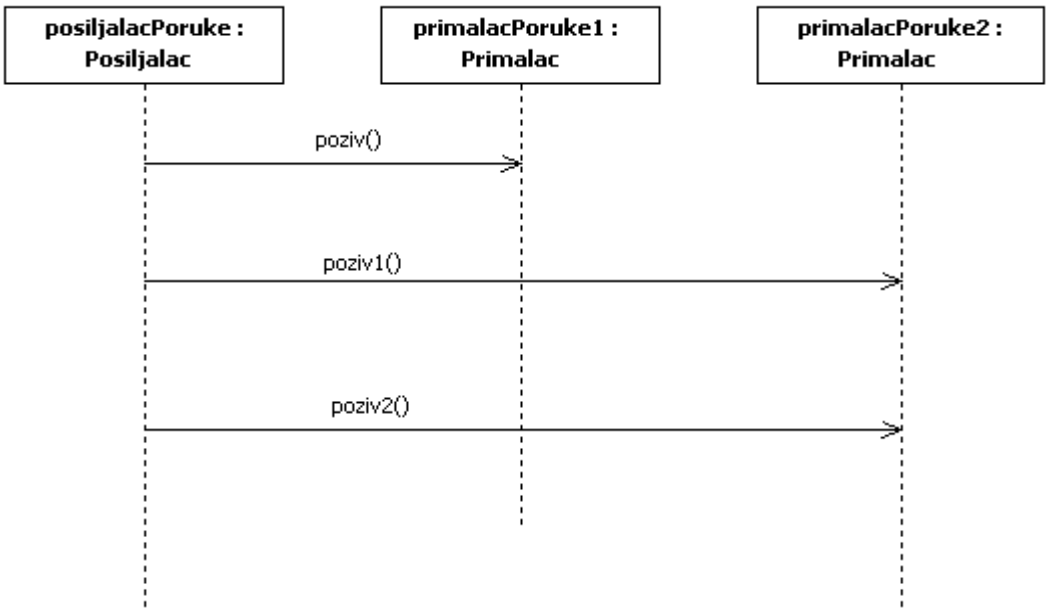
Implementacija sinhrone interakcije

11.10 Asinhrone poruke

Bilo bi veoma jednostavno i uređeno kada bi se sve interakcije dešavale jedna za drugom. Međutim, to nije kako sistem radi. Interakcije se mogu dešavati u svakom trenutku vremena i ponekad je potrebno inicijalizirati kolekciju interakcija u isto vrijeme na koje nećemo čekati odgovor.

Na primjer, aplikacija za naručivanje jela, na korisničkom interfejsu ima mogućnost ispisa računa. Klikom na dugme korisnik aplikacije aktivira ispis. Nakon aktiviranja ispisa, korisnik aplikacije može nastaviti koristiti njene druge funkcionalnosti. Regularna sinhrona poruka nije dovoljna da prikaže ovu vrstu interakcije. Mi trebamo drugi tip poruke, a to je asinhrona poruka.

Na slici 11.7 pokazana je asinhrona interakcija. Pošiljalac poruke šalje poruku jednom učesniku, i ne čeka odgovor već nastavlja sa sljedećim porukama prema drugim učesnicima.



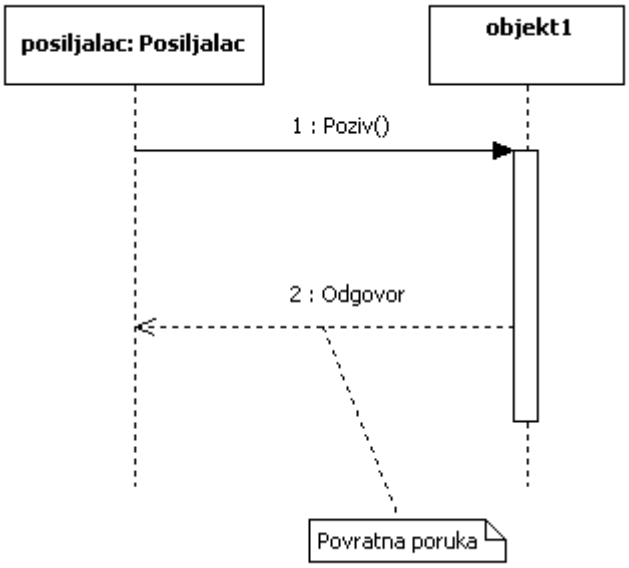
Slika 11.7: Asinhrone poruke

Asinhrone poruke se rješavaju sa nitima u programskim jezicima koji to podržavaju. Na primjer, u sistemu `e-restoran` neophodno je omogućiti istovremeno opsluživanje više naručilaca jela. Zbog toga se za svakog od naručilaca jela prilikom prvog pozivanja aplikacije otvara posebna sesija, koja je klase `SesijaZaNarucivanje`, a koja je neovisna nit koju kreira i uništava glavni proces, koji je klase `NarucivanjeServer`. Naručilac jela dalje nastavlja svoju interakciju sa svojom sesijom, posredstvom glavnog procesa, a glavni proces može

obavljati druge poslove. Poruke koje se razmjenjuju su asinhrone jer aplikacija ne čeka da jedan naručilac jela završi izbor jela nakon što mu je prikazan jelovnik, nego nastavlja da opslužuje ostale naručioce jela i tek po prijemu naredne poruke od istog naručioca jela reagira na istu i nastavi interakciju od mjesta gdje je prethodno bila.

11.11 Povratne poruke

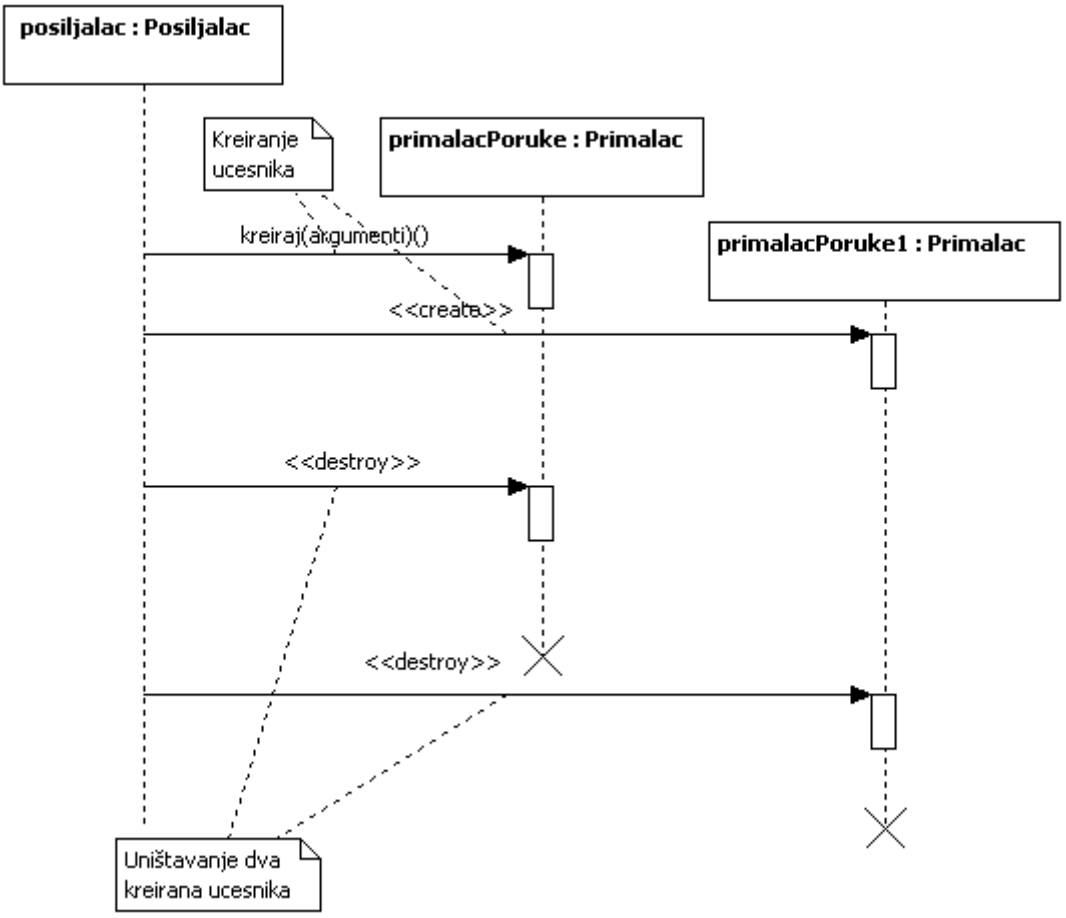
Povratne poruke su opcionalni dio notacije koji se može koristiti na traci aktivnosti da se pokaže kontrolni tok poruke koju vraća učesnik koji je primio poruku učesniku koji je poslao poruku. Ponekad je nepotrebno koristiti povratne poruke na dijagramu sekvenci jer čine dijagram previše 'zaposlenim' i konfuznim.



Slika 11.8: Povratna poruka

11.12 Kreiranje i brisanje učesnika

Učesnici ne moraju postojati sve vrijeme izvršavanja dijagrama sekvenci. Oni se mogu kreirati i uništavati (brisati) u skladu sa porukama koje primaju kao što je i pokazano na slici 11.9. Da bi pokazali da je učesnik kreiran možemo direktno poslati poruku kreiranja učesnikovoj liniji života u obliku kreiraj(argumenti) ili korištenjem stereotipa <<create>>. Uništavanje učesnika se pokazuje tako što na kraju učesnikove linije života postavimo znak za (x) a poruku kojoj smo izvršili njegovo brisanje označimo stereotipom <<destroy>>.



Slika 11.9: Kreiranje i uništavanje učesnika

Bitno je naglasiti da se pri dizajnu softvera često ne zna tačno način implementacije pa je poželjno da se na dijagramima uvijek eksplicitno naglasi poruka za uništavanje objekta.
Mogući način implementacije kreiranja učesnika je:

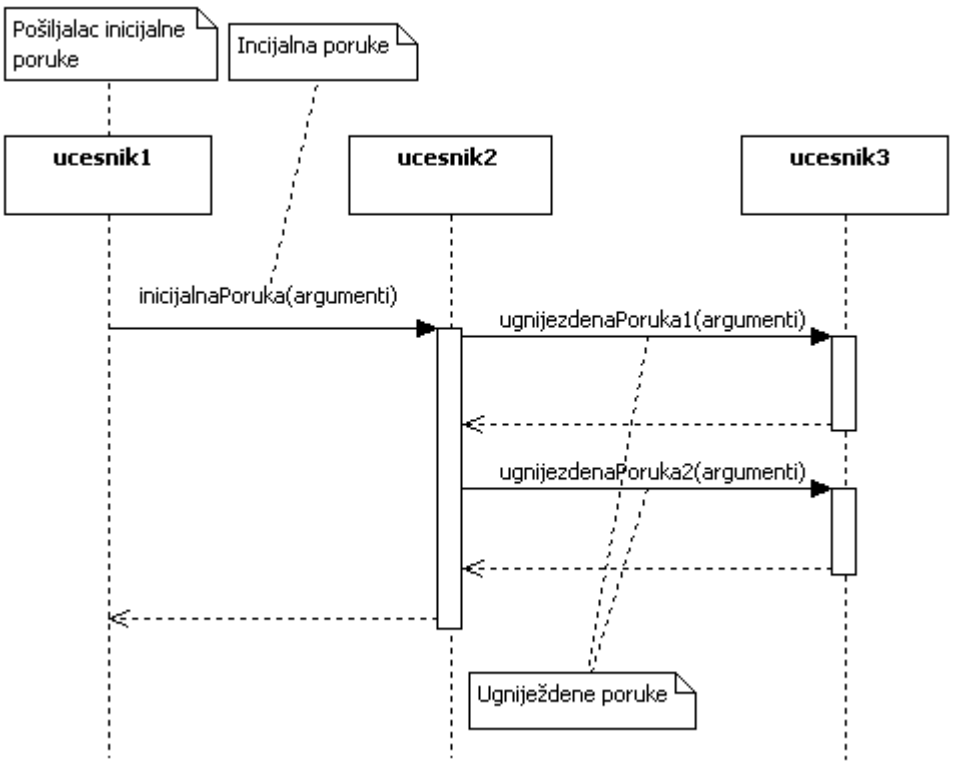
Java :
```
public class Primatelj
{
    // ...
}

public class Posiljatelj
{
    // ...

    public radiNesto()
    {
         Primatelj primateljPoruke = new Primatelj();
    }
}
```

C++ :
```
// Primatelj.h
// ------------------------------------
class Primatelj
{
    // ...
};

// Primatelj.cpp
// ------------------------------------
// ...

// Posiljatelj.h
// ------------------------------------
#include "Primatelj.h"
class Posiljatelj
{
// ...
public:
       RadiNesto();
};

// Posiljatelj.cpp
// ------------------------------------
void Primatelj::RadiNesto()
{
       Primatelj *primateljPoruke = gcnew Primatelj();
};
```

Implementacija kreiranja i uništavanja poruka

11.13 Ugniježđene poruke

Kada poruka od nekog učesnika rezultira u jednu ili više poruka, tada govorimo o ugniježđenim porukama, kao što je prikazano na slici 11.10.

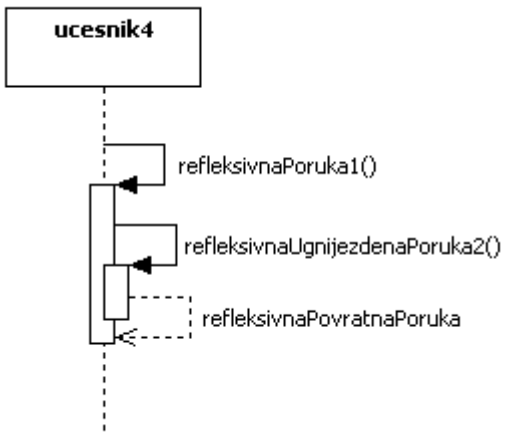


Slika 11.10: Ugniježđene poruke

Na slici 11.10 možemo vidjeti da `ucesnik1` šalje inicijalnu poruku `inicijalnaPoruka(argumenti)` učesniku `ucesnik2`. Kada `ucesnik2` primi poruku `inicijalnaPoruka(argumenti)` on postaje aktivan i šalje ugniježđenu poruku `ugnijezdenaPoruka1(argumenti)` učesniku `ucesnik3` koji na nju odgovara povratnom porukom. Po primitku povratne poruke, `ucesnik2` šalje novu ugniježđenu poruku `ugnijezdenaPoruka2(argumenti)` učesniku `ucesnik3` i čeka na povratnu poruku. Kada dobije povratnu poruku `ucesnik2` šalje povratnu poruku učesniku `ucesnik1`. Može biti više ugniježđenih poruka kao i više nivoa ugniježđene.

11.14 Refleksivne poruke

Učesnik može slati poruku sam sebi ili pozivati jednu od svojih vlastitih operacija. To se može eksplicitno pokazati postavljanjem odvojenog pravougaonika sa pomakom na desno na već postojeći, koji, ustvari, predstavlja traku aktivnosti kao što je pokazano na slici 11.11.



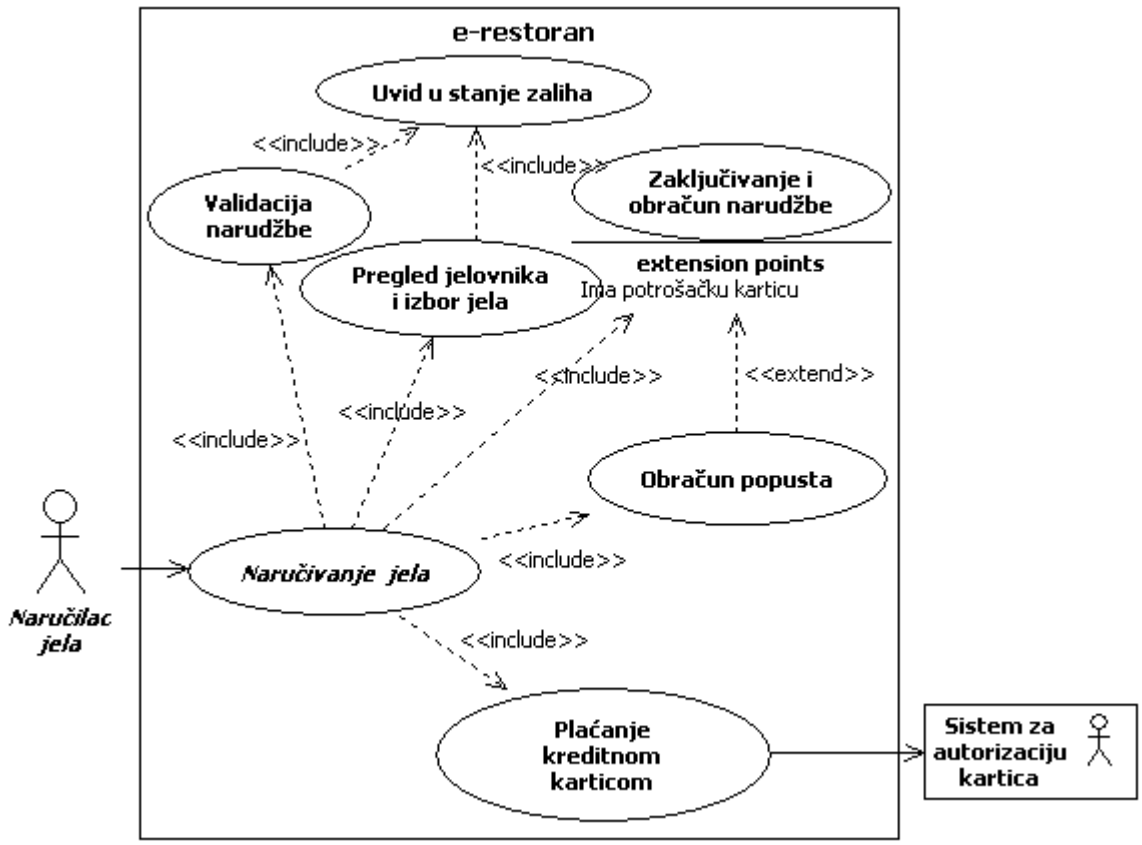
Slika 11.11: Refleksivne poruke

11.15 Povezivanje slučajeva upotrebe i dijagrama sekvence

Već smo spominjali da se dijagram sekvenci koristi da bi se opisale interakcije koje se uzrokuju kada se izvršava neki slučaj upotrebe i u kojem redoslijedu.

Da bi pokazali vezu između dijagrama slučajeva upotrebe i dijagrama sekvenci vratit ćemo se ka našem primjeru `e-restorana`.

Na slici 11.12 prikazan je dijagram slučajeva upotrebe – `Naručivanje jela`. Ovaj slučaj upotrebe je generalizacija slučajeva upotrebe `Naručivanje jela u restoranu` i `Naručivanje jela putem Interneta`. Na dijagramu su prikazani i ostali slučajevi upotrebe koje uključuju `Naručivanje jela`, a koji su potrebni da bi se prikazalo naručivanje jela i plaćanje karticom (specijalan slučaj plaćanja).

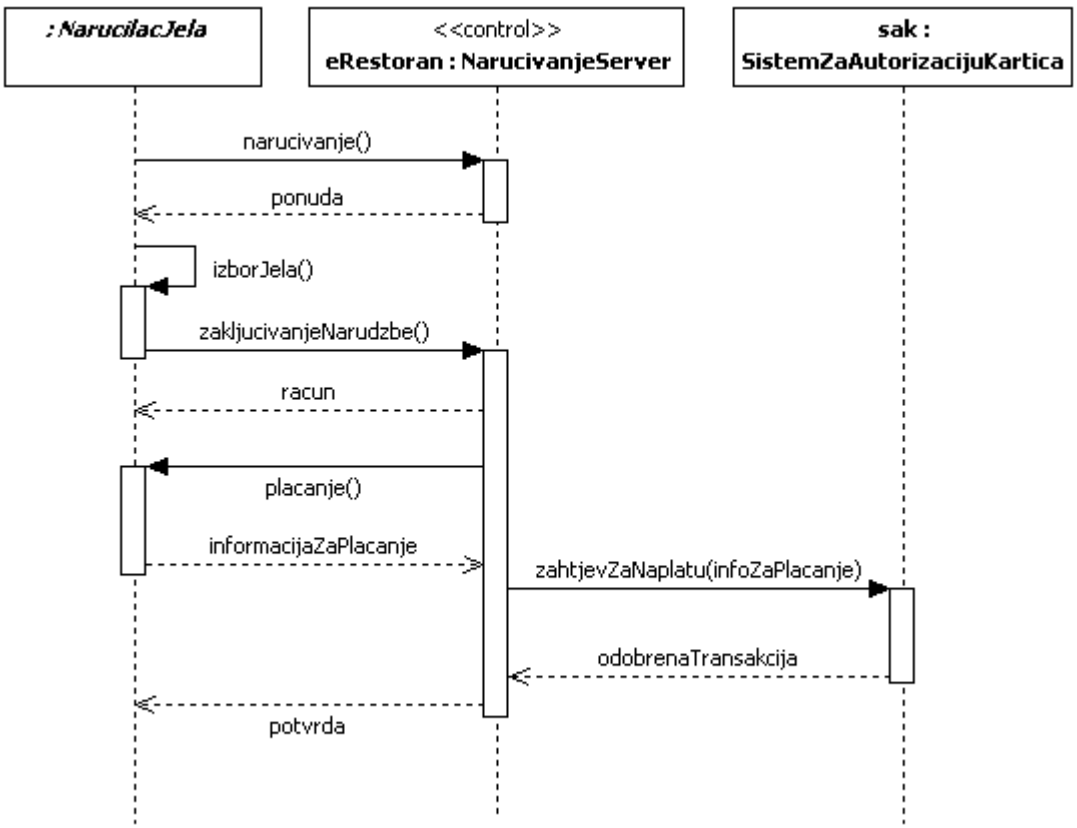


Slika 11.12: Dijagram slučajeva upotrebe – `Naručivanje jela`

Za gornji dijagram slijedi prikaz dvije varijante dijagrama sekvenci.

Dijagram sekvenci najvišeg nivoa

Na najvišem nivou apstrakcije imamo interakciju između naručioca jela, sistema `e-restoran` i eksternog sistema za autorizaciju kartica. Na slici 11.12 je prikazana moguća sekvenca poruka između spomenutih učesnika u interakciji. Važno je uočiti nekoliko bitnih elemenata na dijagramu: klasa `NarucilacJela` je apstraktna klasa pa je naziv klase na dijagramu prikazan italik stilom, `eRestoran` je kontrola (<<control>>) koja predstavlja sistem. Također, na dijagramu postoje i razni tipovi poruka, kao što je ugniježđena poruka `zahtjevZaNaplatu(infoZaPlacanje)`, refleksivna poruka `izborJela()`, povratna poruka `ponuda` i asinhrona poruka `narucivanje()`.



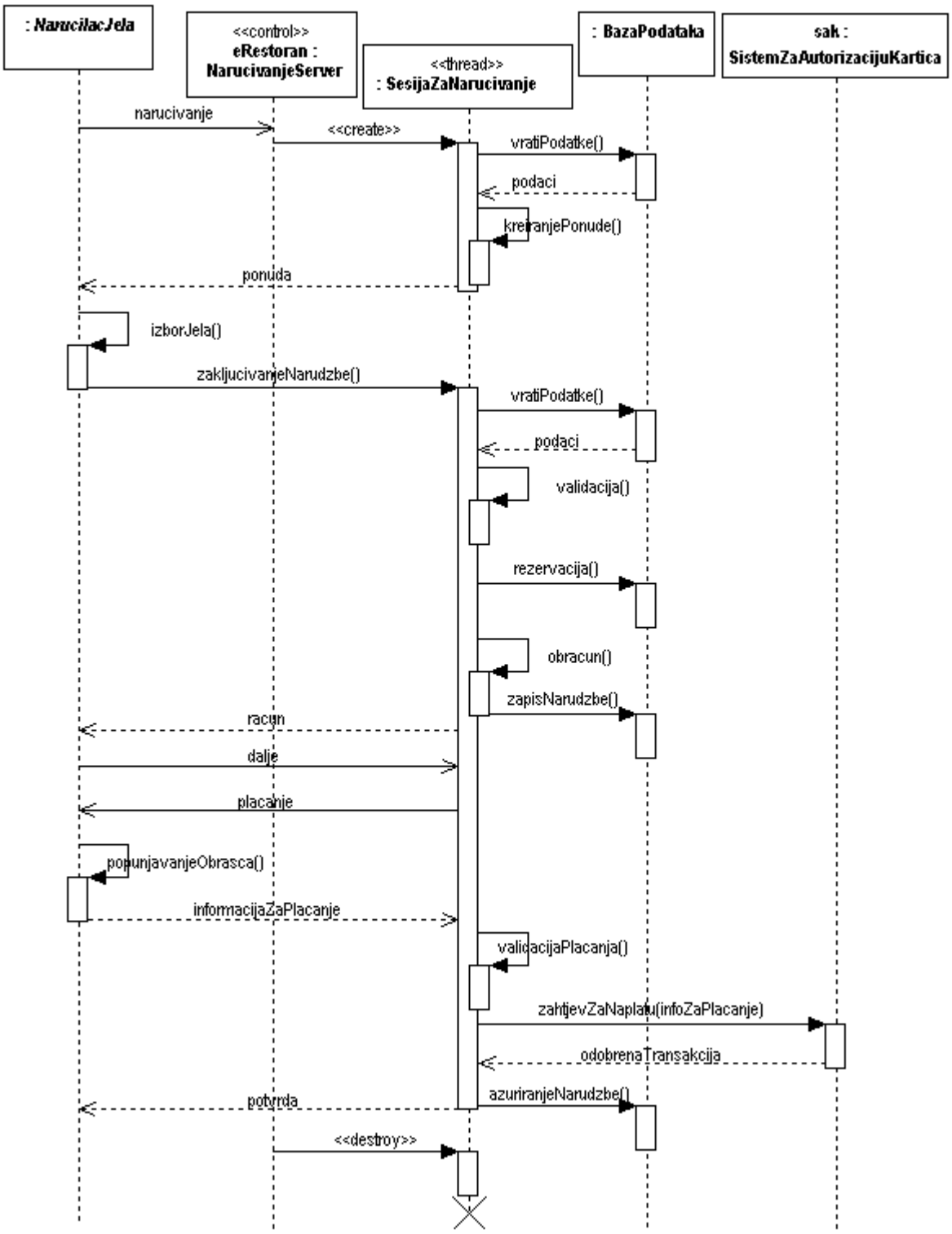
Slika 11.13: Dijagram sekvence za proces naručivanje jela (najviši nivo)

Podjela interakcije na više učesnika

Na slici 11.13 prikazana je samo interakcija koja se dešava između eksternih aktera i našeg sistema. Tu je naš sistem predstavljen kao `eRestoran`. Na slici 11.14 je dijelom prikazana i interakcija unutar našeg sistema, da bi preciznije pojasnili što sistem radi. Bitno je uočiti nekoliko značajnih elemenata na tom dijagramu:

- Objekt `eRestoran` po prijemu poziva za naručivanje, otvara novu sesiju, kao zasebnu programsku nit, i predaje joj dalju interakciju sa naručiocem jela, a sam `eRestoran` nastavlja da osluškuje da li ima novih zahtjeva za naručivanje.
- Kreirana sesija komunicira sa bazom podataka da bi formirala trenutnu ponudu i istu vraća naručiocu jela da bi on mogao izvršiti izbor jela.
- Dok naručilac jela vrši izbor jela, sesija je neaktivna. Aktivira se od momenta zaključivanja narudžbe.

- Nakon prijema poruke za zaključivanje narudžbe, sesija učitava ponovno trenutno stanje iz baze da bi izvršila validaciju narudžbe. Po završetku validacije se izvrši rezervacija sastojaka za naručena jela, a zatim se izvrši obračun.



Slika 11.14: Dijagram sekvence za proces naručivanje jela (nivo sesije za naručivanje)

- Po završetku obračuna narudžba se zapisuje u bazu podataka, a račun se prosljeđuje naručiocu jela. Naručilac jela, nakon pregleda računa, šalje poruku da se nastavi interakcija.
- Sesija šalje obrazac za unošenje informacije za plaćanje, koji naručilac jela popunjava i vraća sesiji.
- Sesija provjerava da li su sva polja unesena i prosljeđuje zahtjev za naplatu sistemu za autorizaciju kartica. U okviru zahtjeva za naplatu prosljeđuje se, kao argument, informacija za plaćanje, u kojoj je sadržan i iznos za naplatu.
- Sistem za autorizaciju, po završetku naplate, vraća poruku da je transakcija odobrena. Na osnovu te poruke sesija mijenja status narudžbe i vraća potvrdu naručiocu jela.
- Sesija je privremeni objekt koji se kreira s ciljem formiranja narudžbe i izvršenja njene naplate. Nakon što se narudžba zabilježi u bazu podataka, sesija nije neophodna jer više nema potrebe za interakcijom sa korisnikom. Zbog toga se sesija može uništiti. S obzirom na to da je sesija nit, ona se automatski uništava kada dođe do svog kraja. Međutim, zbog mogućnosti da je neka sesija aktivna, a da se u tom momentu desi uništavanje objekta `eRestoran`, objekt `eRestoran` treba pri svom uništenju eksplicitno pozvati uništenje otvorenih sesija.

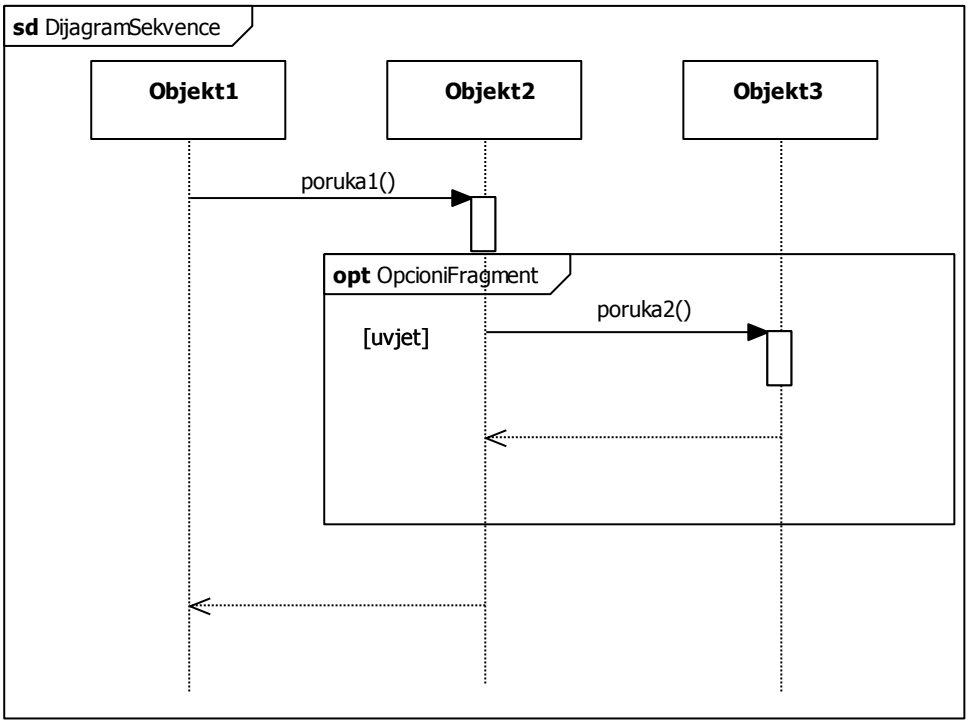
Ovakav način crtanja dijagrama može postati vrlo kompleksan i nepregledan zbog velikog broja poruka koje se crtaju. U nastavku slijedi objašnjenje elemenata UML notacije koji omogućavaju preglednije, tačnije i potpunije prezentiranje interakcije među učesnicima.

11.16 Upravljanje kompleksnim interakcijama sa fragmentima sekvence

Prije UML 2.0 dijagrami sekvenci su brzo postojali veliki i pretrpani, i sadržavali su puno detalja što je otežavalo razumijevanje i održavanje. Nije postojalo standardnih načina da bi se omogućilo zajedničko grupiranje poruka, interakcije i alternativni tokovi. UML 2.0 je odgovorio na te zahtjeve i uveo fragmente sekvenci.

Fragment sekvence se predstavlja kao okvir koji uokviruje dio interakcija unutar dijagrama sekvenci što je pokazano na slici 11.15. Uz svaki fragment veže se ključna riječ ili kako se često naziva operator fragmenta, a koja označava namjenu fragmenta. Opcionalno uz operator se navodi i uvjetni izraz.

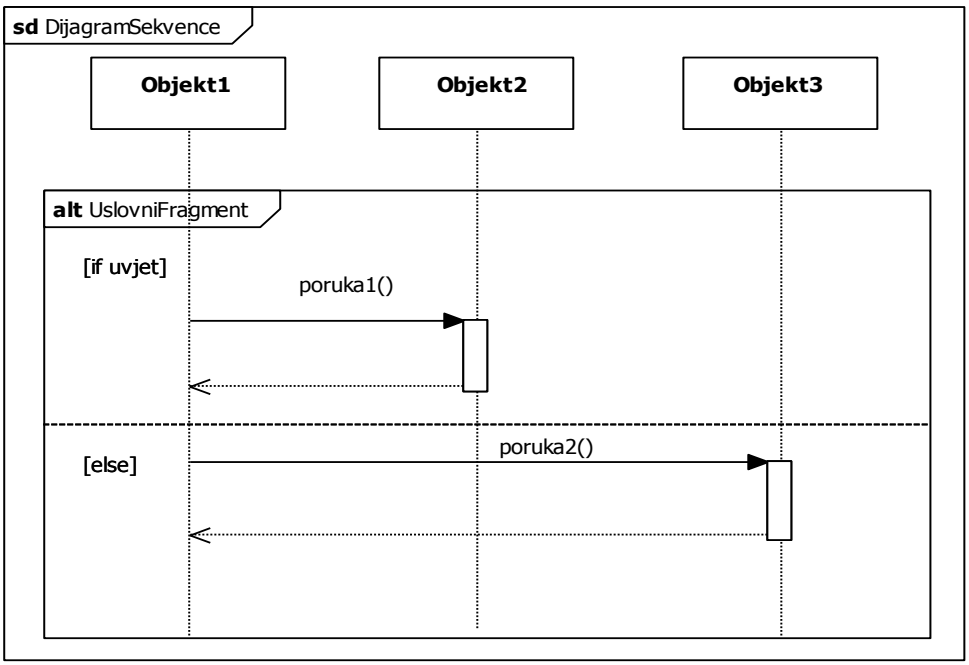
Na slici 11.15 operator je *opt* što znači da je ovo opcionalni fragment. Sve interakcije koje su sadržane unutar fragmenta se izvršavaju u ovisnosti od uvjetnog parametra u okviru fragmenta.



Slika 11.15: Notacija za opcionalni fragment

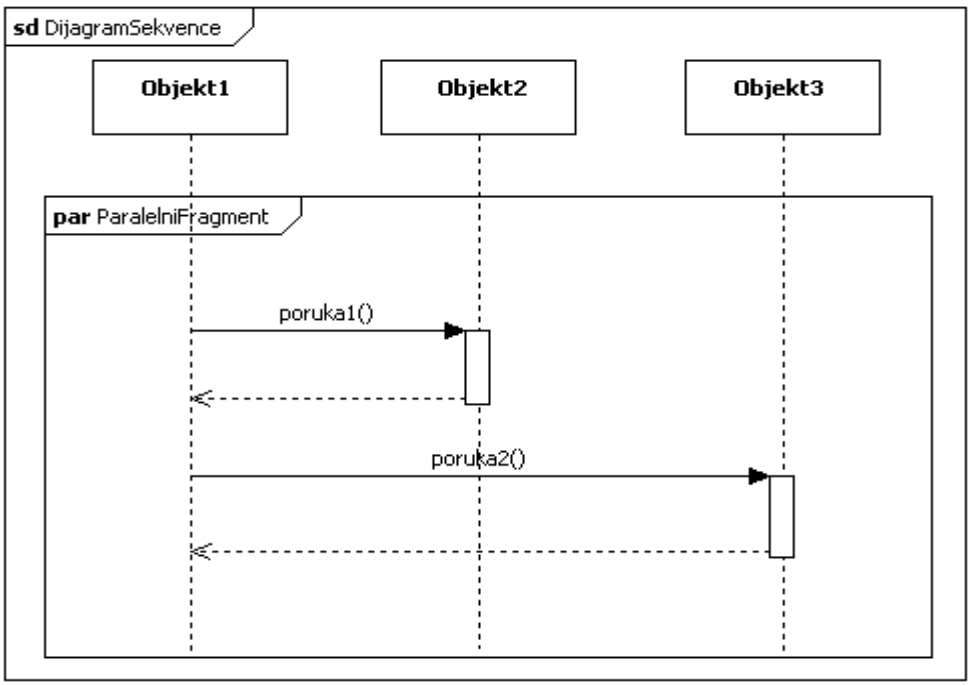
Na slici 11.15 se može primijetiti i upotreba sd fragmenta koji se koristi da bi se cijeli dijagram sekvence imenovao i uokvirio u cjelinu.

Ponavljanje se može prikazati fragmentom čija je ključna riječ *loop*. Grananje se može prikazati u okviru dijagrama sekvenci sa fragmentom čija je ključna riječ *alt* i koji je podijeljen na dva ili više dijelova. Ovi dijelovi su poznati i kao operandi fragmenta kojem pripadaju. Uvjetna klauzula se postavlja blizu vrha svakog operanda i pokazuje uvjet na osnovu čije procjene se određuje koji dio (grana) se treba izvršiti. Ključna riječ else se koristi za obilježavanje grane koja će se izvršiti ako nijedan prethodni uvjet nije ispunjen.



Slika 11.16: Notacija za uvjetni fragment

Paralelno izvršavanje akcija se prikazuje pomoću par fragmenta, a paralelne akcije se odvajaju u posebne dijelove fragmenata. Alternativa prikazivanju par fragmenta je postavljanje coregion notacije.



Slika 11.17: Notacija za paralelni fragment

Fragment može sadržavati veći broj interakcija kao na slikama iznad, ali i ugniježđene fragmente.
UML 2.0 sadrži širok skup različitih tipova fragmenata. Neki su prikazani u tabeli ispod.

Tabela 11.3: Najčešći operatori fragmenata

Operator	Značenje
alt	Alternativni izbor između više fragmenata; izvršava se samo onaj dio fragmenta čiji je uvjet ispunjen. Ako je u okviru jednog dijela else operand naveden, on će se izvršiti ako se nijedan od ostalih operanda nije izvršio.
assert	Označena interakcija se mora izvršiti tačno redoslijedom kako je prikazano
break	Ako se desi interakcija u break fragmentu, onda se napušta fragment koji u sebi sadrži break fragment (najčešće je to *loop*)
opt	Opcionalni izbor fragment se izvršava samo ako je navedeni uvjet ispunjen
par	Paralelno se izvršavaju svi dijelovi fragmenta
loop	Fragment se može izvršiti više puta u petlji
critical	Kritični region sa značenjem da ima prednost u odnosu na ostale u skupu paralelnih operanda specificiranih u fragmentu
neg	Koristi se da pokaže poruke koje nisu dozvoljene
ref	Referenca: odnosi se na interakciju koja je definirana na drugom dijagramu
region	Označava region, odnosno mjesto pristupa zajedničkom resursu, gdje može doći do kolizije koja može izazvati nekonzistentnost resursa
sd	Dijagram sekvence služi za postavljanje okvira za cijeli dijagram sekvence i njegovo imenovanje

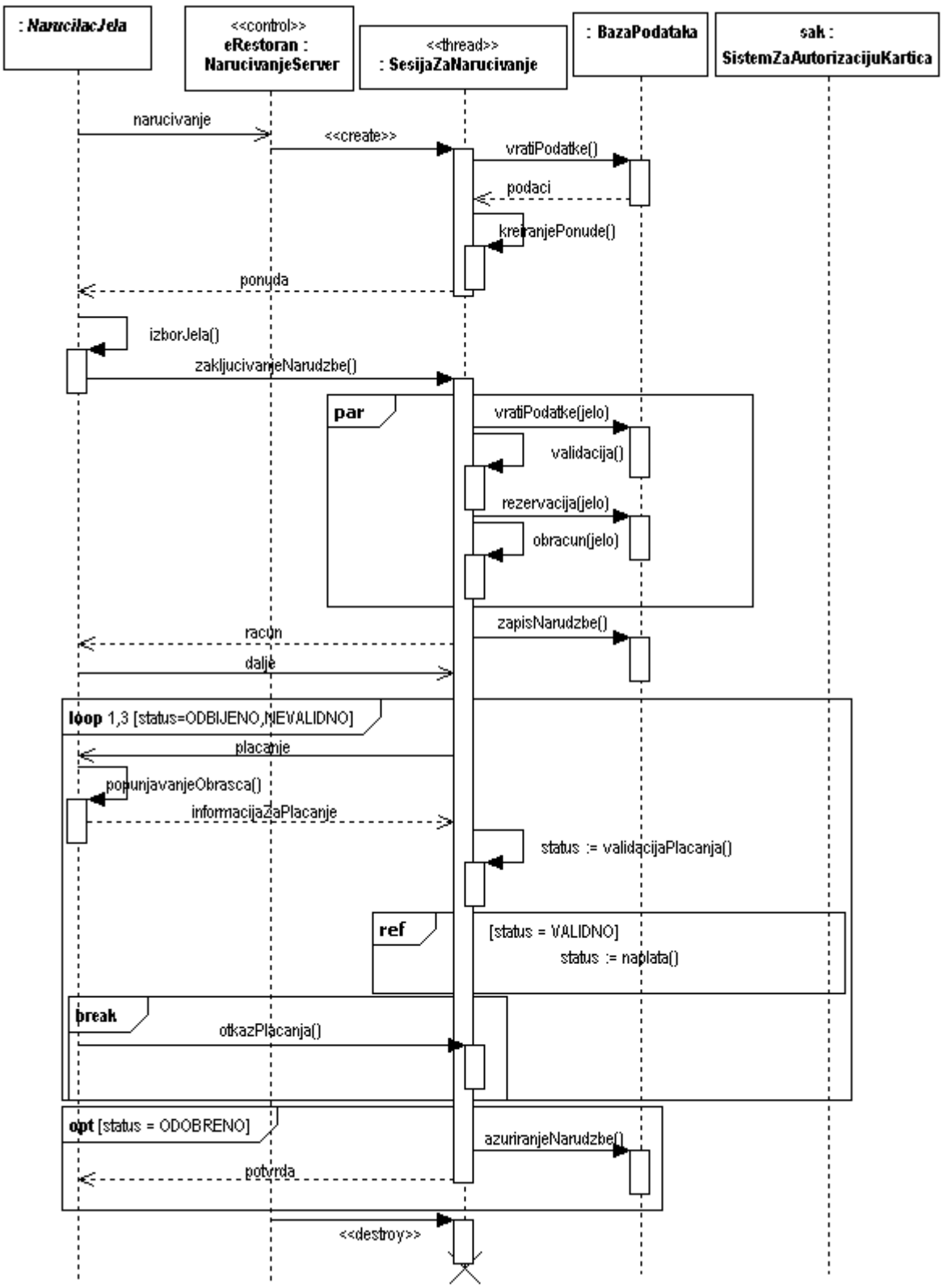
Neki tipovi fragmenata ne zahtijevaju dodatne parametre kao dio svoje specifikacije kao npr. ref parametar čija je namjena da se referencira na neki drugi dijagram sekvenci. Korištenjem prethodno objašnjene notacije fragmenata poboljšat ćemo dijagram sekvence za `e-restoran`.

11.17 Fragmenti na `e-restoran` dijagramima sekvence

Na slici 11.18 prikazana je primjena fragmenata na dijagramu sekvence za proces naručivanje jela. Uočimo sljedeće fragmente:

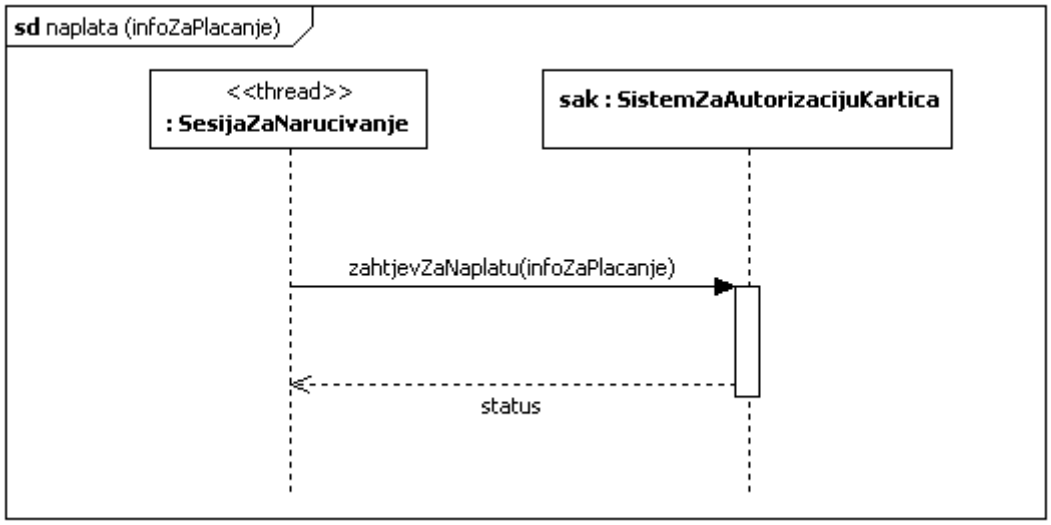
- par, koji označava da se validacija, rezervacija i obračun za pojedina jela sa narudžbe mogu izvršavati paralelno;
- loop, koji označava da se u slučaju da je informacija za plaćanje nevalidna ili da je transakcija odbijena od strane sistema za autorizaciju kartica može se najviše 3 puta ponoviti unošenje informacija za plaćanje;
- ref, koji pokazuje da je interakcija u okviru tog fragmenta prikazana na odvojenom dijagramu sekvence; ovaj se fragment izvršava uz uvjet da je status informacije za plaćanje `VALIDNO`;
- break, koji pokazuje da se u slučaju odustajanja od plaćanja treba prekinuti loop u kome se nalazi;
- opt, koji pokazuje da se ažuriranje statusa narudžbe i vraćanje potvrde naručiocu jela trebaju desiti samo ako je naplata uspješno završena (status `ODOBRENO`).

Možemo se zapitati da li je možda trebalo postaviti još neki od fragmenata na prikazani dijagram, ali uvijek treba imati na umu da je svrha dijagrama da što jednostavnije pojasni sistem, a ne da zakomplicira prikaz sistema. Tako, na primjer, rezervacija jela, odnosno sastojaka za pripremu tog jela, je operacija koja predstavlja kritični region (dvije sesije mogu istovremeno pokušati rezervirati jedno jelo, a sastojaka ima samo za pripremu jednog od njih) pa bi bilo dobro da je označena fragmentom region, međutim to bi previše zakompliciralo prikaz. Isto tako, možemo se zapitati, što u slučaju da se ne može izvršiti rezervacija, gdje se dešava uklanjanje rezervacije sa jela kad se desi otkaz naručivanja ili plaćanja i slično. Skoro je nemoguće sve predstaviti na dijagramima pa se na njima prikazuje samo ono što se smatra važnim za razumijevanje interakcije, a ukoliko se ukaže potreba da se neki dijelovi interakcije još dublje i detaljnije pojasne onda je najbolje koristiti ref fragment i nacrtati tu interakciju odvojeno, na posebnom dijagramu.



Slika 11.18: `e-restoran` sa primjenom notacije fragmenata

Na sljedećoj slici prikazano je kako se referencirani fragment može prikazati na odvojenom dijagramu sekvence.



Slika 11.19: Referencirani dijagram sekvenci za interakciju sa sistemom za autorizaciju kartica

Završno razmatranje

Dijagram sekvence se koristi da bi se realizirali slučajevi upotrebe, ili operacije, ili klase. Svaka operacija koja se modelira mora postojati u dijagramu klase. Poruke mogu biti signali, ili operacije klase. Dijagrami sekvenci i dijagrami komunikacije modeliraju iste aspekte sistema, a to je kolaboracija objekata i poruke koje se razmjenjuju da bi se postigli ciljevi.

U sljedećem poglavlju ćemo se upoznati i sa dijagramom komunikacije koji, također, pripada grupi dijagrama interakcije.

Pitanja za ponavljanje

1.Što se opisuje sa dijagramima interakcije?

2.Koji dijagrami se ubrajaju u grupu dijagrama interakcije?

3.Koja je osnovna namjena dijagrama sekvence?

4.Kojem pogledu pripada dijagram sekvence?

5.Što se podrazumijeva pod interakcijom?

6.Kako dijagram sekvence predstavlja vrijeme?

7.Što predstavlja linija života objekta?

8. Kako se imenuju učesnici odnosno linije života u okviru dijagrama sekvenci?

9. Što predstavlja događaj, signal i poruka?
10.Koja je uloga trake aktivnosti na liniji života učesnika?
11.Koja je notacija za traku aktivnosti?
12. Koje vrste poruka postoje u okviru dijagrama sekvenci?
13.Objasniti notaciju poruka.
14. Što je asinhrona poruka?
15.Što je sinhrona poruka?
16. Koja je notacija za asinhrone, a koja za sinhrone poruke?
17. Koje je značenje povratnih poruka?
18. Kako se obilježavaju poruke kreiranja i uništavanja učesnika?
19. Kako se prikazuju poruke koje učesnik sam sebi šalje?
20. Kako nastaju ugniježđene poruke?
21. Koja je namjena fragmenata u okviru dijagrama sekvence?
22. Navesti najčešće korištene operatore fragmenata?
23. Koja je namjena alt fragmenta?
24. Koja je namjena loop fragmenta?
25. Pomoću kojeg fragmenta se može prikazati paralelno izvršavanje interakcija?
26. Što se predstavlja sa fragmentom sd?
27.Koja je veza između dijagrama sekvence i dijagrama klase?
28.Koja je veza između dijagrama sekvence i dijagrama slučajeva upotrebe?

POGLAVLJE 12.

DIJAGRAM KOMUNIKACIJE

Osnovna svrha dijagrama sekvence jeste da prikažu redoslijed događaja između dijelova sistema koji su uključeni u određenu interakciju. Dijagrami komunikacije (eng. communication diagram) pružaju još jedan pogled na interakciju fokusiranjem na linkove (veze) između učesnika.

Dijagrami komunikacije su posebno korisni za prikazivanje koji linkovi su potrebni između učesnika kako bi se proslijedila interakcijska poruka. Letimičnim pogledom na dijagram komunikacije, može se reći koji učesnici trebaju biti uključeni kako bi se određena interakcija mogla obaviti.

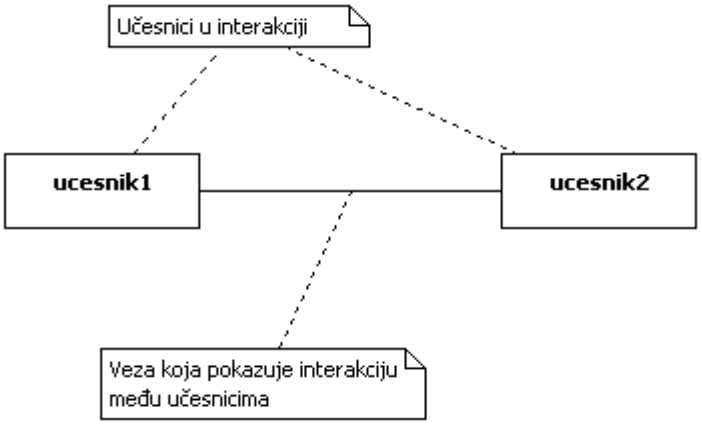
12.1 Namjena dijagrama komunikacija

Osnovna namjena korištenja dijagrama komunikacija je kako slijedi:

- Koriste se za modeliranje interakcija između objekata koji isporučuju funkcionalnost za slučaj upotrebe i pri tome pokazuje poruke koje se razmjenjuju između linija života.
- Koriste se za modeliranje interakcija između objekata koji isporučuju funkcionalnost operacija i pri tome pokazuju poruke koje se razmjenjuju između linija života.
- Koriste se za modeliranje mehanizama unutar dizajna arhitekture sistema.
- Koriste se za modeliranje alternativnih scenarija unutar slučaja upotrebe ili operacija koje uključuju kolaboraciju različitih objekata i različitih interakcija.
- Koriste se u ranim koracima projekta prilikom identificiranja objekata i njihovih klasa, koji učestvuju u slučaju upotrebe.
- Koriste se da pokažu učesnike u dizajn paternima.

12.2 Elementi dijagrama komunikacije

Dijagrame komunikacije čine tri elementa: učesnici, komunikacijski linkovi (veze) između pojedinih učesnika i poruke koje se mogu proslijediti kroz ove komunikacijske linkove, kao što je prikazano na sljedećoj slici.



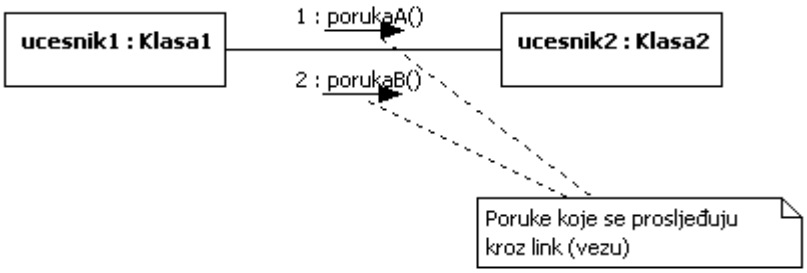
Slika 12.1: Dijagrami komunikacije sastoje se od učesnika i linkova (veza)

12.3 Učesnici

Učesnici se na dijagramu komunikacije predstavljaju pomoću pravougaonika. Ime učesnika i klasa smještaju se u sredinu pravougaonika i formira se na isti način kao i kod dijagrama sekvence.

12.4 Linkovi

Komunikacijski link (veza) je prikazan jednom linijom koja povezuje dva učesnika. Svrha veze je da dozvoli porukama da se prosljeđuju između različitih učesnika; bez linka dva učesnika ne bi mogla komunicirati. Komunikacijski link je prikazan na slici 12.2.



Slika 12.2: Poruke na linku

Linkovi na dijagramu komunikacije mogu se instancirati na osnovu asocijacije koja je prikazana na dijagramima klasa, ili mogu biti privremeni linkovi između linija života koji omogućavaju razmjenu poruka.

12.5 Poruke

Poruka se na dijagramu komunikacije prikazuje korištenjem linije sa ispunjenom strelicom na njenom kraju usmjerene od primalaca poruke ka pošiljalaca poruke. Slično porukama na dijagramu sekvence, poruke se označavaju sa nazivom operacije i listom parametara. Međutim, za razliku od obilježavanja poruka na dijagramima sekvence, na dijagramu komunikacije treba prikazati redoslijed u kojem se poruke pozivaju u toku interakcije.

Potpuna sintaksa prikaza poruka na dijagramu komunikacije je:

sekvenca-izraz[atribut=]ime_signala_poruke[(argumenti)]:[povratni_tip]

Svi elementi poruke osim *sekvenca-izraz* su objašnjeni prilikom objašnjavanja notacije poruka na dijagramu sekvence. U nastavku slijedi detaljnije razmatranje formiranja *sekvenca-izraza* prilikom numeriranja poruka.

Dijagrami komunikacije ne moraju obavezno teći od vrha ka dnu stranice na kojoj su prikazani kao što je slučaj kod dijagrama sekvence. Zbog toga se redoslijed poruka na dijagramu komunikacije obavezno prikazuje korištenjem broja ispred svake poruke. Broj ispred poruke označava redoslijed u kojem se poruke pozivaju, počevši od 1 i povećavajući se sve dok se sve poruke na dijagramu ne označe. Slijedeći ovo pravilo na slici 12.2 najprije se poziva poruka `porukaA()`, a zatim poruka `porukaB()`.

Stvari postaju kompliciranije kada poruka koja je poslana ka učesniku direktno uzrokuje da učesnik poziva još jednu poruku. Kada jedna poruka uzrokuje poziv druge poruke, tada se za drugu poruku kaže da je ugniježđena unutar originalne poruke.

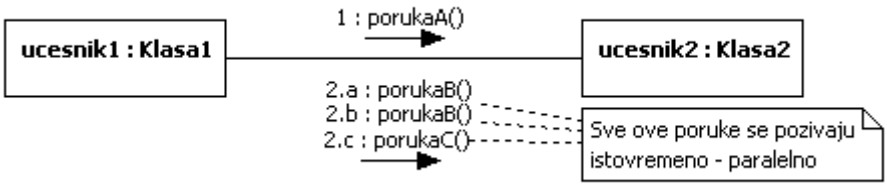
Dijagrami komunikacije koriste shemu brojnog označavanja kako bi pokazali redoslijed izvršavanja ugniježđenih poruka. Ako kažemo da je inicijalna poruka označena sa 1., tada bilo koja poruka ugniježđena unutar inicijalne počinje sa 1., dodajući nakon decimalne tačke broj za označavanje ugniježđenih poruka.

Ako je broj inicijalne poruke 1., tada će prva ugniježđena poruka biti 1.1, a druga ugniježđena poruka na osnovu poruke 1. će biti 1.2. Primjer numeriranja ugniježđenih poruka je prikazan na slici 12.3.

Slika 12.3: Ugniježđene poruke

Poruke koje se pojavljuju u isto vrijeme

Dijagrami komunikacije imaju jednostavan odgovor na problem poruka koje se šalju u isto vrijeme. Dijagrami sekvence za ovo trebaju kompliciraniju konstrukciju, kao što su fragmenti, dok dijagrami komunikacije koriste prednosti njihovog brojno baziranog označavanja poruka, dodavanjem notacije *broj i slovo* kako bi označili da se jedna poruka izvršava u isto vrijeme kada i druga poruka, kao što je prikazano na slici 12.4.



Slika 12.4: Istovremene poruke

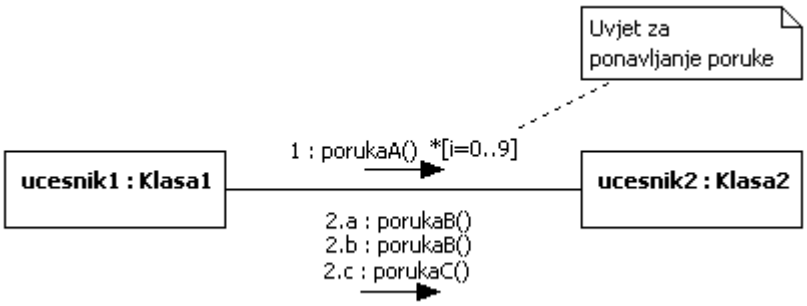
Iniciranje poruke više puta – ponavljanje poruke

Kada se opisuju poruke na dijagramu komunikacije, često je potrebno i da se pokaže ako je poruka inicirana više puta.

Iako UML ne definira striktno kako prikazati na dijagramu komunikacije da se neka poruka poziva više puta, rečeno je da se zvjezdica treba koristiti prije nego što se primijeni uvjet za ponavljanje (eng. looping constraint).

Sljedeći primjer je način da se specificira da će se nešto desiti 10 puta: *[i = 0 .. 9]

U gornjem primjeru uvjeta za ponavljanje, *i* predstavlja brojač koji raste od 0 do 9, radeći sve ono što je neophodno u svakoj od 10 iteracija. Slika 12.5 prikazuje kako ovaj uvjet za ponavljanje može biti primijenjen na dijagramu komunikacije.

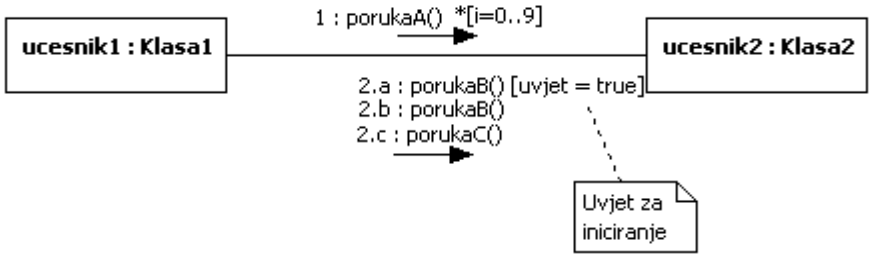


Slika 12.5: Uvjet za ponavljanje poruke

Slanje poruke na osnovu uvjeta

Ponekad poruka treba biti inicirana samo ako je određeni uvjet ispunjen ili ako je prethodna poruka izvršena korektno. Slično kao kod fragmenata dijagrama sekvenci, poruke dijagrama komunikacije mogu imati skup uputa o tome koji uvjeti trebaju biti evaluirani i ispunjeni kako bi se poruka inicirala.

Skup uputa su logički izrazi. Kada se njihovom evaluacijom dobije "tačno", poruka će biti inicirana, u suprotnom ne. Slika 12.6 prikazuje kako se ispitivanje skupa uputa primjenjuje na jednu od tri poruke koje se konkurentno izvršavaju.

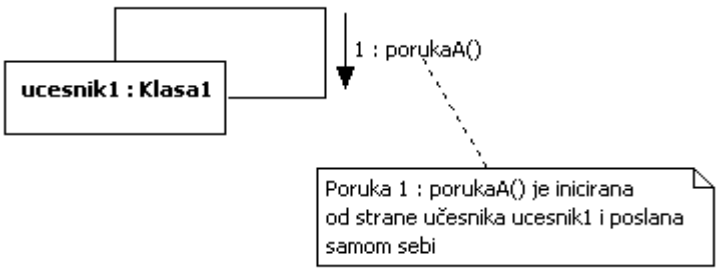


Slika 12.6: Uvjet za iniciranje poruke

Na slici se uočava da poruka `2.a:porukaB()` će biti inicirana u isto vrijeme kada i `2.b:porukaB()` i `2.c:porukaC()` samo ako je izraz "uvjet = true" evaluiran kao tačan; ako je izraz "uvjet = true" netačan, tada se `2.a:porukaB` neće inicirati, ali hoće poruke `2.b:porukaB()` i `2.c:porukaC()`.

Kada učesnici šalju poruke sami sebi

Pojam učesnika koji sam sa sobom razgovara može u prvi mah zvučati čudno, ali ako se razmišlja u terminologiji softverskih objekata poziv vlastitog metoda je čak i uobičajen način komunikacije. Slično kao i na dijagramima sekvence, učesnici na dijagramima komunikacije mogu slati poruke sami sebi. Sve što je potrebno je link od učesnika ka samom sebi kako bi se omogućilo iniciranje ovakve poruke, kao što je prikazano na slici 12.7.

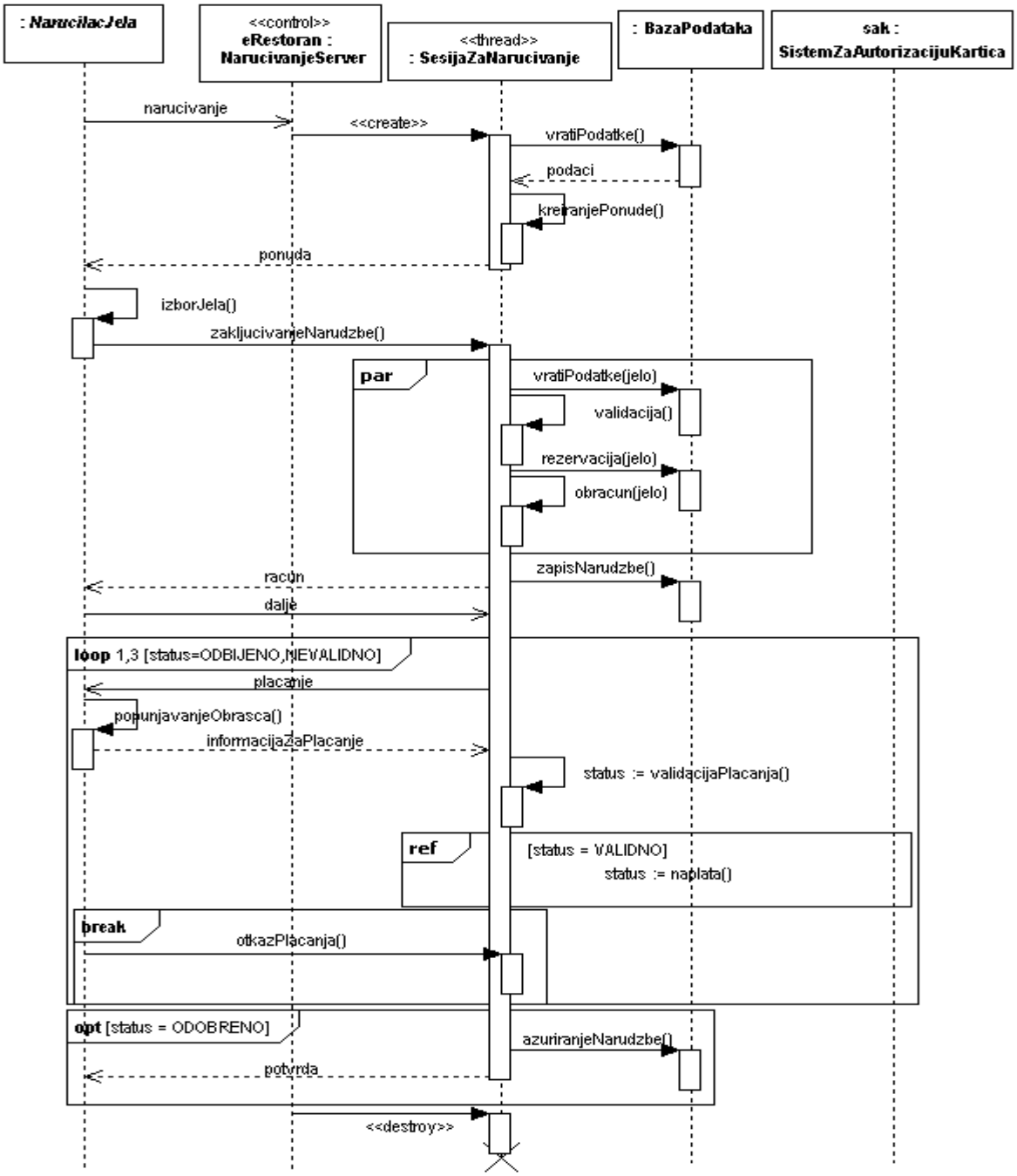


Slika 12.7: Slanje poruke samom sebi

Učesnik može inicirati `1:porukaA()` ka samom sebi jer ima komunikacijsku liniju usmjerenu ka sebi.

12.6 Prikaz interakcija sa dijagramom komunikacije

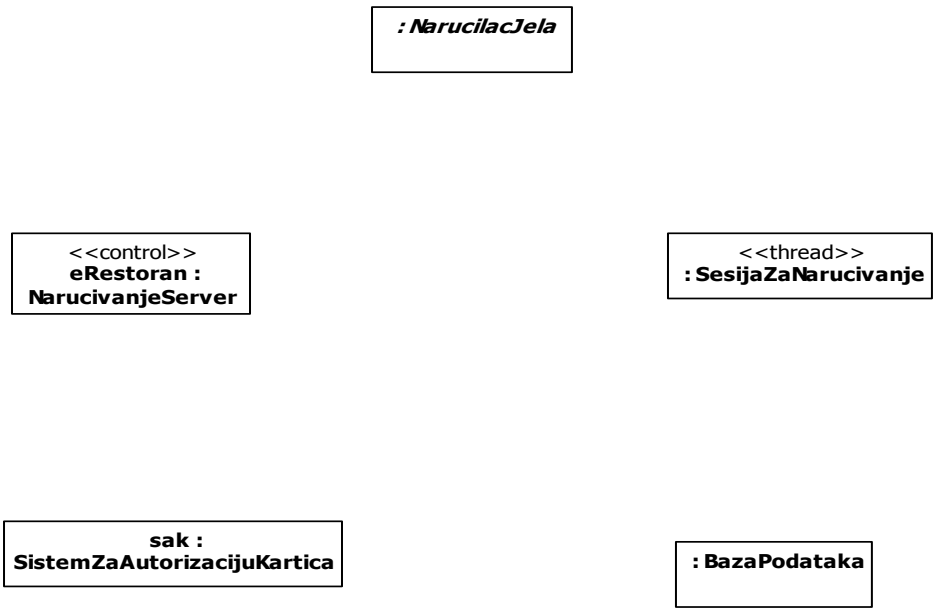
Nakon što je objašnjena notacija dijagrama komunikacije, slijedi na jednom praktičnom primjeru prikaz interakcije. Za primjer dijagrama sekvenci sa slike 12.8, objašnjen u sklopu poglavlja o dijagramima sekvenci, slijedi prikaz kako se njegove interakcije mogu modelirati na dijagramu komunikacije.



Slika 12.8: Dijagram sekvenci za kojeg se pravi dijagram komunikacije

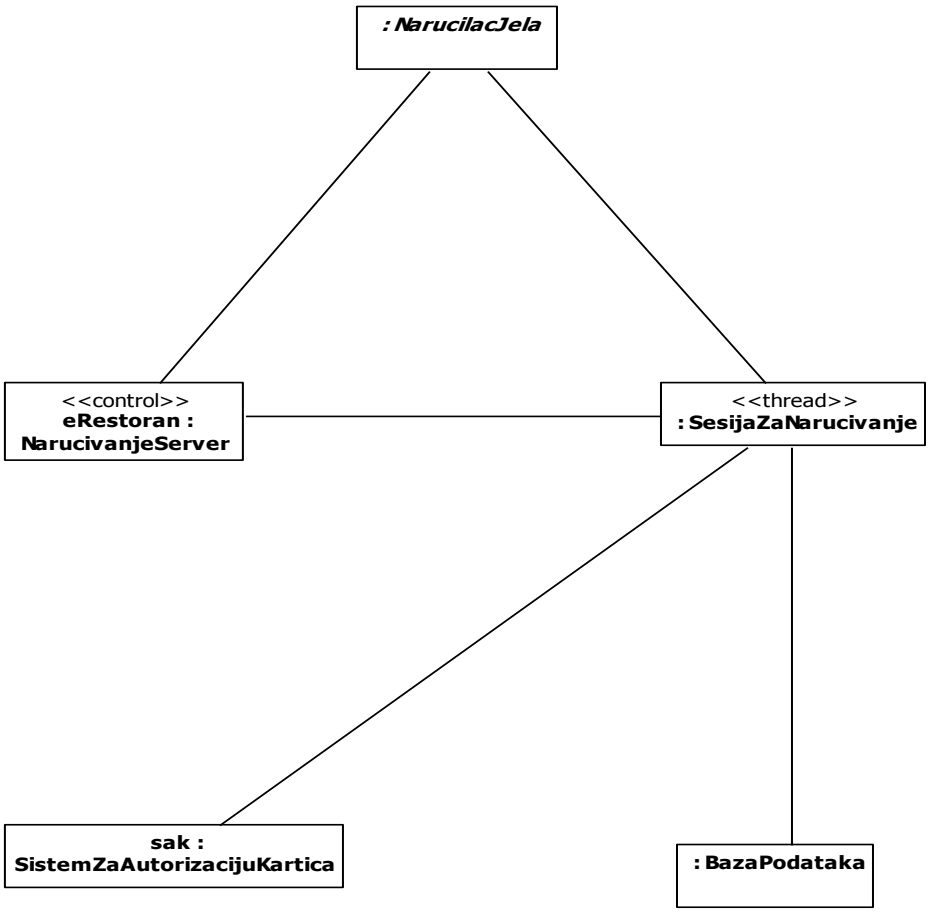
Dijagrami sekvence sadrže mnogo specifičnih oznaka i njihovo savladavanje zahtijeva određeno vrijeme. Nije neophodno imati dijagram sekvence prije nego što se kreira dijagram komunikacije. Dijagrami komunikacije i/ili dijagrami sekvence mogu se kreirati u redoslijedu koji nama odgovara.

Prvi korak u formiranju dijagrama komunikacije je identificiranje učesnika. U našem slučaju učesnici su već identificirani i jednostavno na naš dijagram komunikacije dodamo učesnike sa slike 12.8.



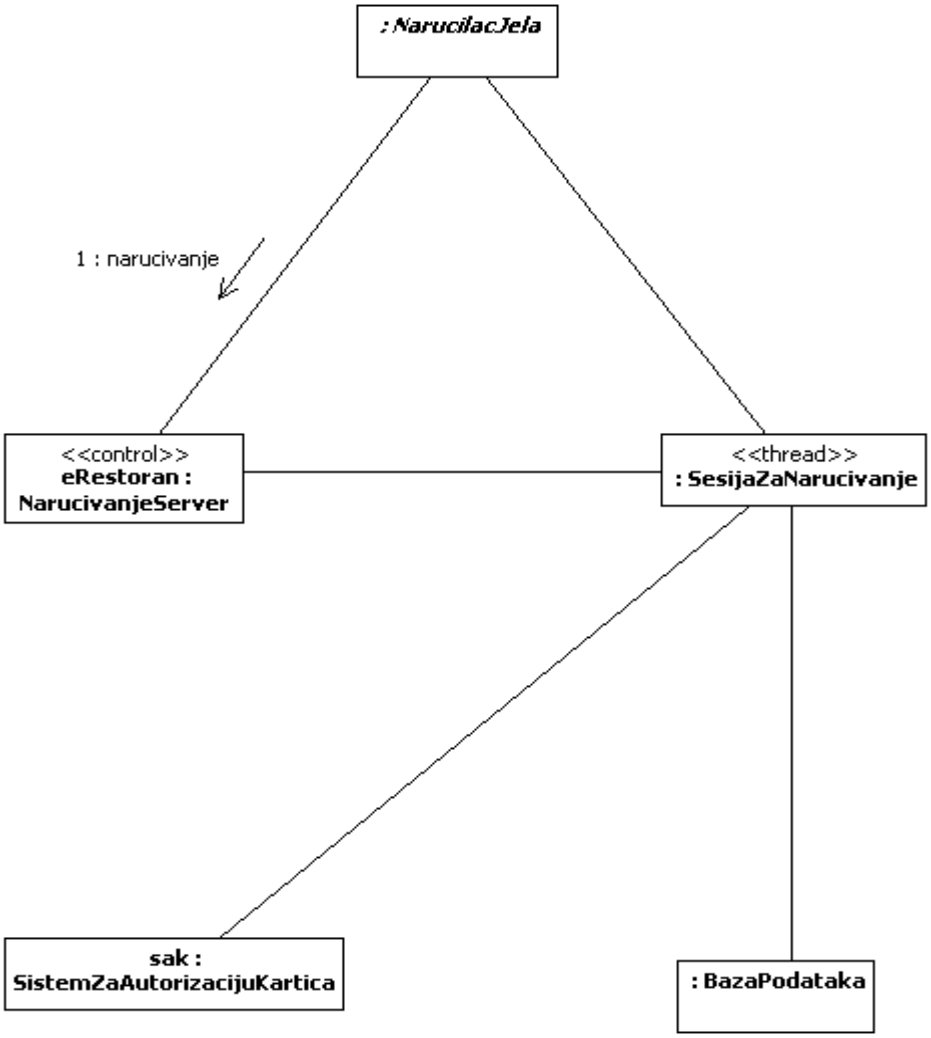
Slika 12.9: Prvi korak – crtanje učesnika u interakciji

Drugi korak je identificiranje linkova između učesnika, a u našem slučaju je to crtanje linkova na osnovu dijagrama sekvenci na slici 12.8. Učesnici i linkovi između svih učesnika kako bi oni mogli međusobno komunicirati prikazani su na slici 12.10.



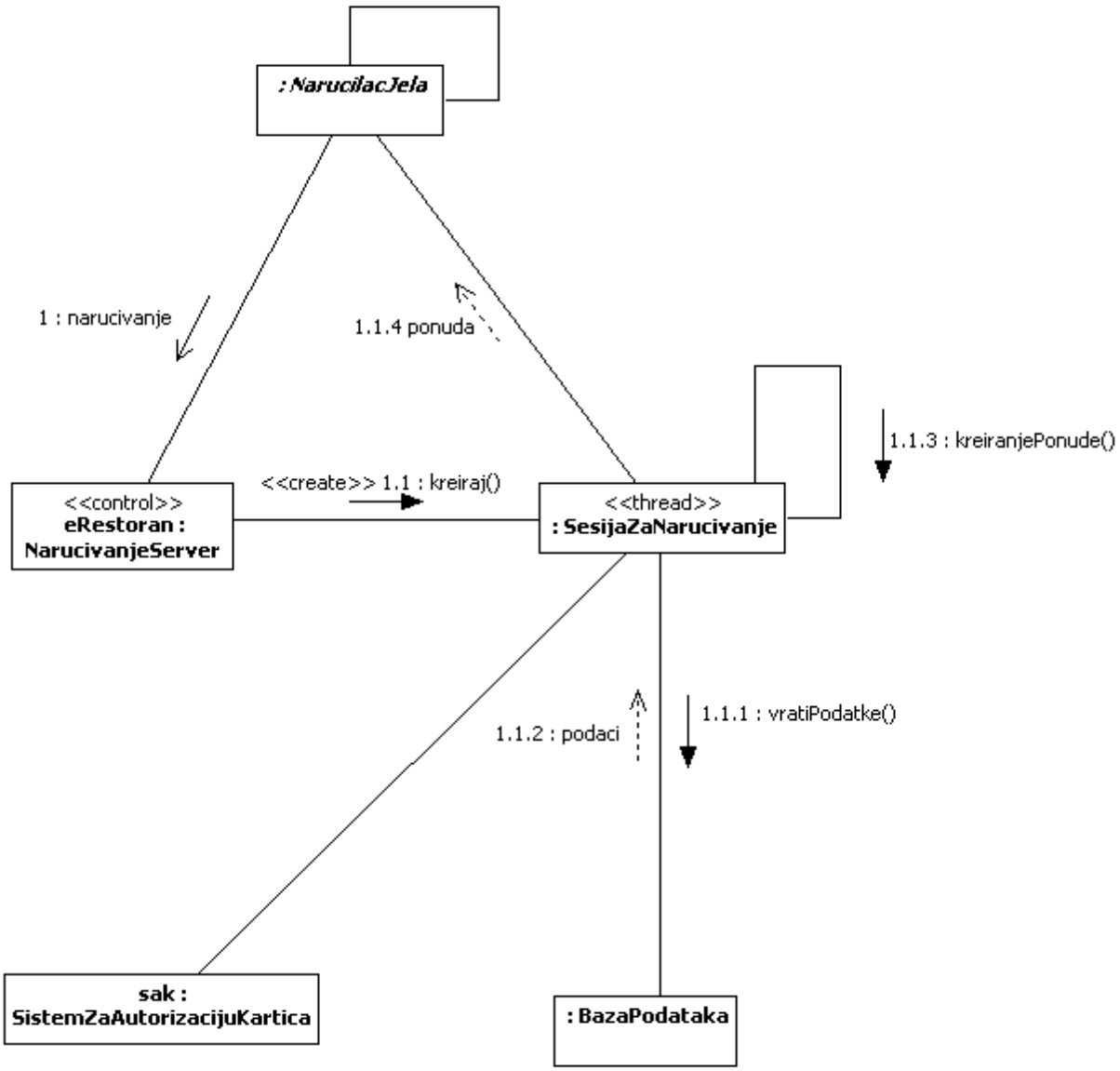
Slika 12.10: Drugi korak – crtanje veza

Treći korak je da se dodaju poruke koje se šalju između učesnika tokom životnog vijeka interakcije, kao što je prikazano na slici 12.11. Kada se dodaju poruke na dijagram komunikacije, najčešće je najbolje početi sa učesnikom ili događajem koji pokreće interakciju.



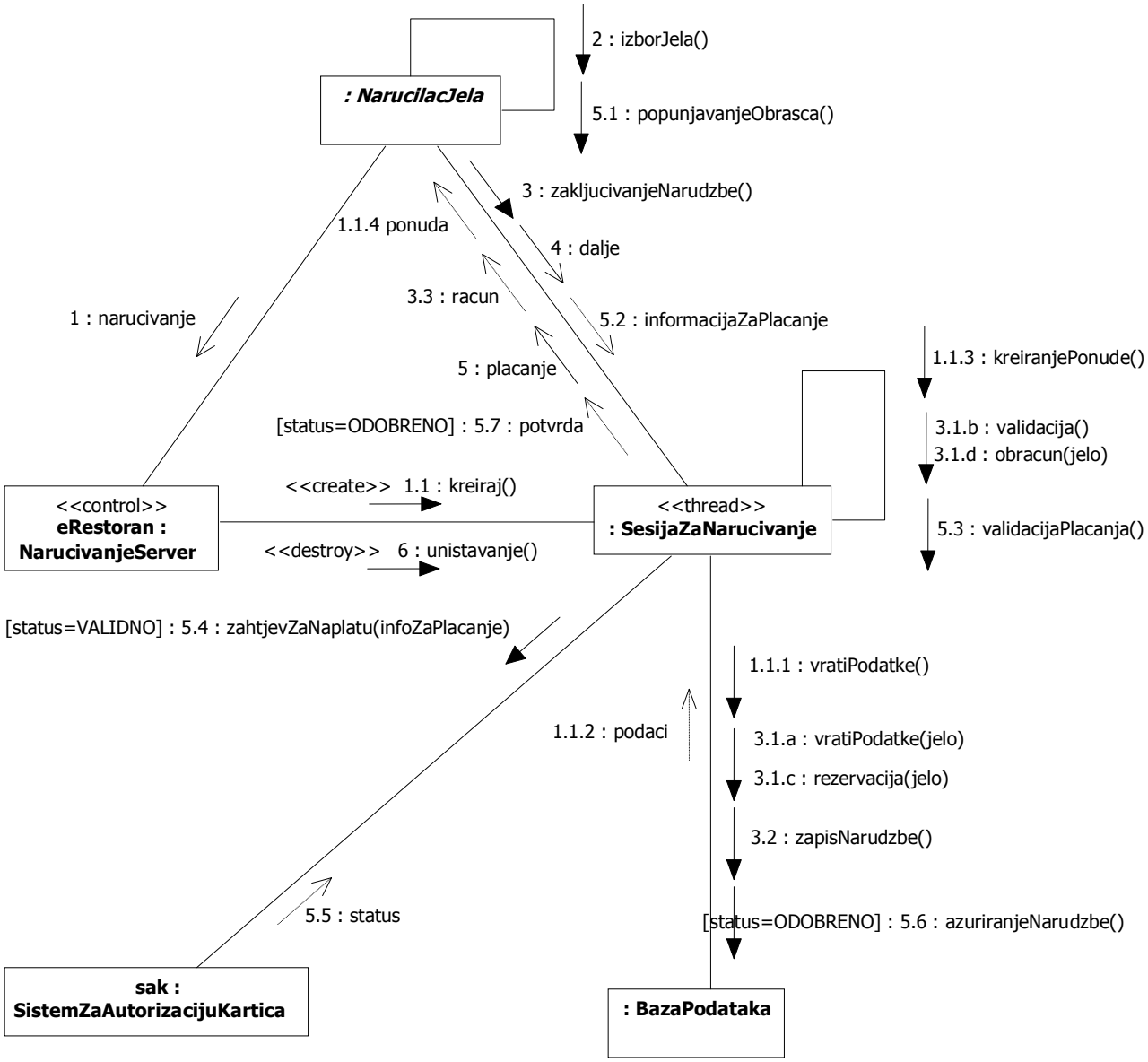
Slika 12.11: Treći korak – inicijalna poruka

Nakon što se inicijalna poruka ili poruke dodaju na dijagram komunikacije, stvari se počinju komplicirati. Poruka *1:narucivanje* pokreće ugniježđenu poruku koju šalje učesnik `eRestoran:NarucivanjeServer` kako bi se kreirao novi učesnik `:SesijaZaNarucivanje`. Ugniježđena poruka dobiva dodatnu decimalnu tačku na osnovu broja poruke koja je pokreće. Dalje ta poruka pokreće sljedeće Ugniježđene poruke, do formiranja odgovora na prvu inicirajuću poruku, što je prikazano na slici 12.12. Osim toga vidimo da je na slici dodana i veza samog na sebe za učesnika `:SesijaZaNarucivanje`. Često se desi da se u koraku dodavanja veza ne uoči neka veza pa se takve veze dodaju naknadno.



Slika 12.12: Četvrti korak – ostale poruka i neuočene veze u prethodnim koracima

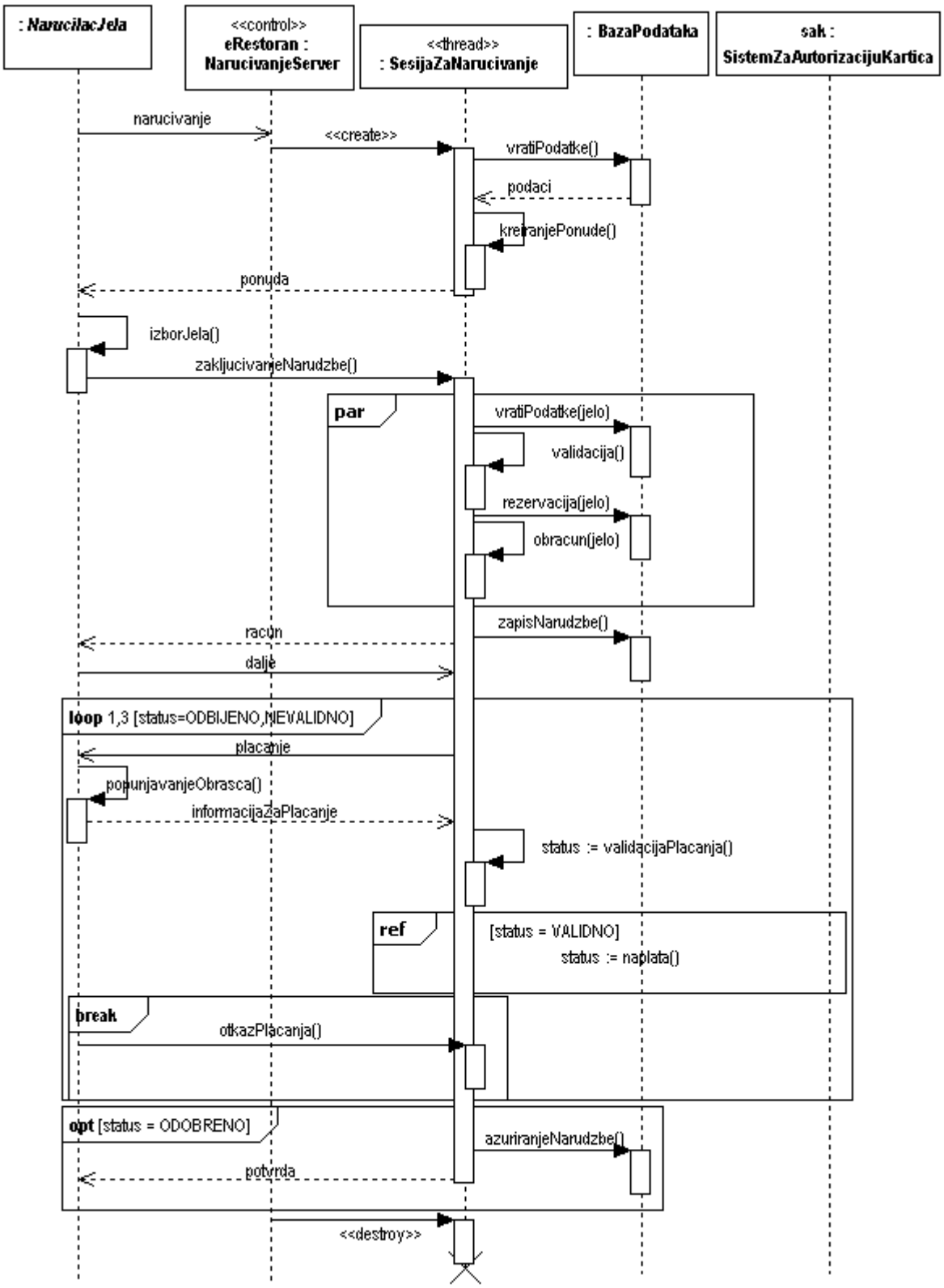
Uz mali dodatni napor mogu se dodati i preostale poruke na dijagram komunikacije. Slika 12.13. prikazuje kompletan skup poruka unutar interakcije naručivanje jela, u skladu sa onim što je prikazano na originalnom dijagramu sekvence.

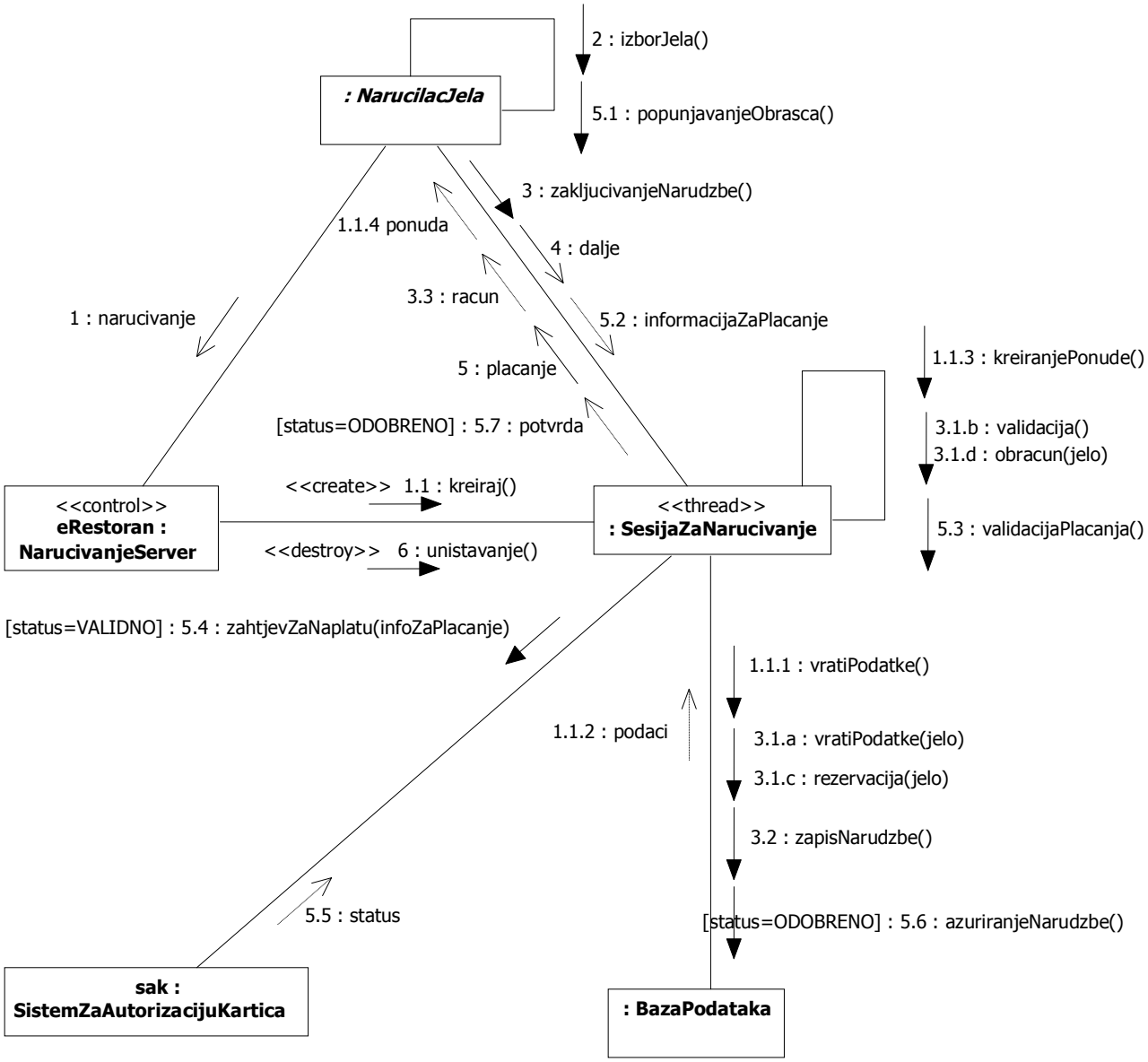


Slika 12.13: Kompletirani `e-restoran` dijagram komunikacije

12.7 Usporedba dijagrama komunikacije i dijagrama sekvence

Dijagrami komunikacije i dijagrami sekvence prezentiraju slične informacije tako da je njihova usporedna skoro neizbježna. U tu svrhu pogledajmo na slici 12.14 dvije različite reprezentacije iste interakcije – naručivanje jela.


: NarucilacJela
<<control>>
eRestoran :
NarucivanjeServer
<<thread>>
: SesijaZaNarucivanje
: BazaPodataka
sak :
SistemZaAutorizacijuKartica
narucivanje
<<create>>
vratiPodatke()
podaci
kreiranjePonude()
ponuda
izborJela()
zakljucivanjeNarudzbe()
par
vratiPodatke(jelo)
validacija()
rezervacija(jelo)
obracun(jelo)
zapisNarudzbe()
racun
dalje
loop 1,3 [status=ODBIJENO,NEVALIDNO]
placanje
popunjavanjeObrasca()
informacijaZaPlacanje
status := validacijaPlacanja()
ref
[status = VALIDNO]
status := naplata()
break
otkazPlacanja()
opt [status = ODOBRENO]
azuriranjeNarudzbe()
potvrda
<<destroy>>



Slika 12.14: Dijagram sekvence i dijagram komunikacije za istu interakciju – naručivanje jela

Na dijagramu sekvence veze između pojedinih učesnika se zasnivaju na činjenici da se između njih prosljeđuje poruka. Dijagrami komunikacije imaju intuitivni način prikaza veza između učesnika čije učešće je neophodno u događaju koji čini interakciju. Na dijagramu komunikacije, redoslijed događaja uključenih u interakciju je skoro drugorazredna informacija.

Dijagrami sekvence i dijagrami komunikacije su tako slični da većina UML alata može automatski konvertirati jedan tip dijagrama u drugi. Odabir jednog od ova dva pristupa je

uglavnom vlastita sklonost. Ako je važniji pogled na interakcije iz perspektive linkova, onda su dijagrami komunikacije bolji izbor; međutim ako više želimo što je moguće jasnije vidjeti redoslijed interakcija, tada će vjerovatno naš odabir biti dijagram sekvence.

Možda najvažnija pouka koju treba izvući iz ove diskusije je da iako oba tipa dijagrama sadrže slične informacije, dijagrami komunikacije i dijagrami sekvence nude različite prednosti, stoga je najbolji pristup koristiti oba.

Završno razmatranje

Dijagrami komunikacije mogu se koristiti za realizaciju ili slučajeva upotrebe ili operacija ili klasa. Svaka operacija koja se modelira mora postojati u dijagramu klase. Poruke mogu biti događaji ili operacije klase. Ako su stanja označena sa uvjetima, oni moraju biti validna stanja relevantnih klasa i moraju se pojaviti u dijagramima stanja, koji će se prikazati nakon ostalih dijagrama koji pripadaju grupi, dijagrama interakcije, a to je dijagram toka vremena i dijagram pregleda interakcija.

Pitanja za ponavljanje

1.Koja je osnovna namjena dijagrama komunikacije?
2. Koji su osnovni elementi dijagrama komunikacije?
3. Koja je notacija za učesnike u okviru dijagrama komunikacije?
4.Kako se imenuju učesnici?
5.Koja je namjena linkova između učesnika?
6.Koja je notacija za obilježavanje poruka?
7.Kako se vrši numeriranje poruka koje se izvršavaju jedna za drugom?
8.Kako se numeriraju poruke koje se dešavaju istovremeno?
9.Kako se obilježavaju poruke koje učesnik šalje sam sebi?
10. Kako se obilježava kada se poruka ponavlja određen broj puta?
11.Kako se obilježava izvršavanje poruka na osnovu uvjeta?
12.Kako se numeriraju ugniježđene poruke?
13. Koji su koraci kreiranja dijagrama komunikacije na osnovu dijagrama sekvenci?
14.Koja je osnovna razlika između dijagrama komunikacije i dijagrama sekvence?

POGLAVLJE 13.

DIJAGRAM TOKA VREMENA

Dijagrami toka vremena (eng. timing diagram) su, kao što im i ime kaže, vezani za vremenski aspekt. Dijagrami sekvence se fokusiraju na redoslijed poruka, dijagrami komunikacije prikazuju veze između učesnika, tako da na ovim dijagramima nije bilo mjesta za detaljno modeliranje vremenskih informacija.

13.1 Namjena dijagrama toka vremena

Dijagrami toka vremena se koriste za:

- Specificiranje vremenski ovisnog ponašanja objekata, podsistema i sistema.
- Modeliranje veza između linija života čije interakcije ovise od vremena.

Trajanje interakcije je najčešće vezano za sisteme koji rade u realnom vremenu, ali sigurno nije limitirano samo na njih. Ustvari potreba da se evidentiraju tačne vremenske informacije o interakciji može biti važna bez obzira kojeg je tipa sistem koji se modelira.

U dijagramima toka vremena, svaki događaj ima vremenske informacije vezane za sebe koje precizno opisuju kada se događaj desio i koliko je dugo trebalo dok drugi učesnik nije izvršio operacije prouzrokovane događajem. Ako nam nisu potrebni vremenskih dijagrami za određenu interakciju, to je kao da kažemo, "Zna se koji se događaj treba dogoditi, ali nije bitno kada će se dogoditi ili kako će trajati".

13.2 Elementi dijagrama toka vremena

Dijagram toka vremena je jedna vrsta dijagrama sekvence i crta se u okviru koji se obilježava sa ključnom riječi sd i imenom interakcije u lijevom uglu. Nazivi linija života pišu se lijevo u okviru i mogu se pisati u pravcima lijevo-desno ili odozgo-dolje. Ako je potrebno prikazati više od jedne linije života, tada se one odvajaju sa horizontalnim linijama.

Vrijeme u dijagramu toka vremena teče od lijeva prema desno, ili od vrha prema dnu, pri čemu se više preferira prikazivanje vremena od lijeva prema desno.

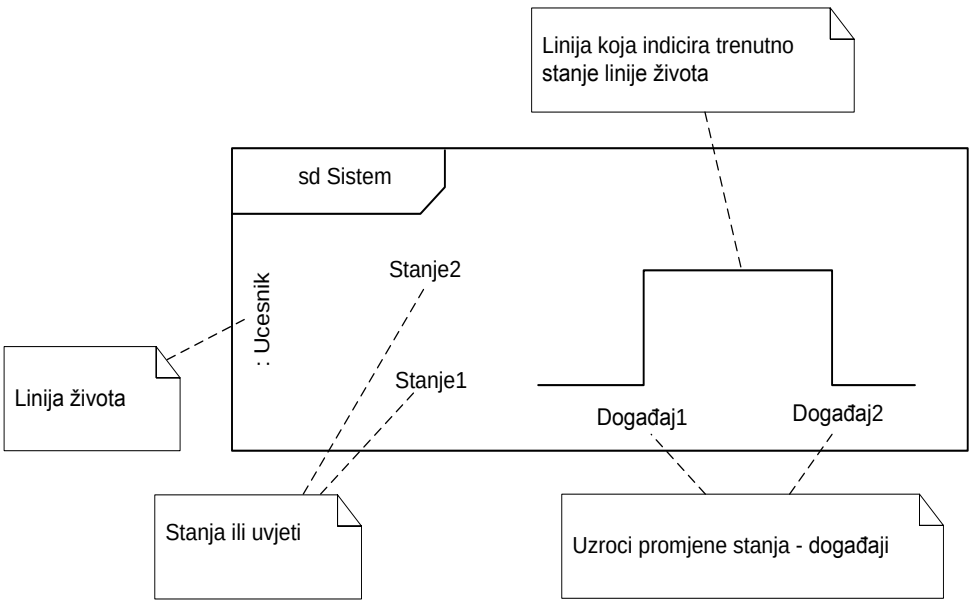
13.3 Učesnici

Učesnika dijagrama toka vremena je moguće obilježivati sa punim nazivom <ime>:<tip> pri čemu je <ime> ime učesnika a <tip> klasa na osnovu koje je učesnik instanciran. Moguće je i da se radi upravljivosti dijagrama ne navodi puni naziv učesnika.

13.4 Stanja

Tokom interakcije, učesnik može biti u različitim stanjima. Učesnik je u određenom stanju kada primi događaj odnosno poruku. Učesnik je u tom stanju dok ne nastupi drugi događaj. Stanja su na dijagramu toka vremena postavljena do odgovarajućeg učesnika odnosno njegove linije života.

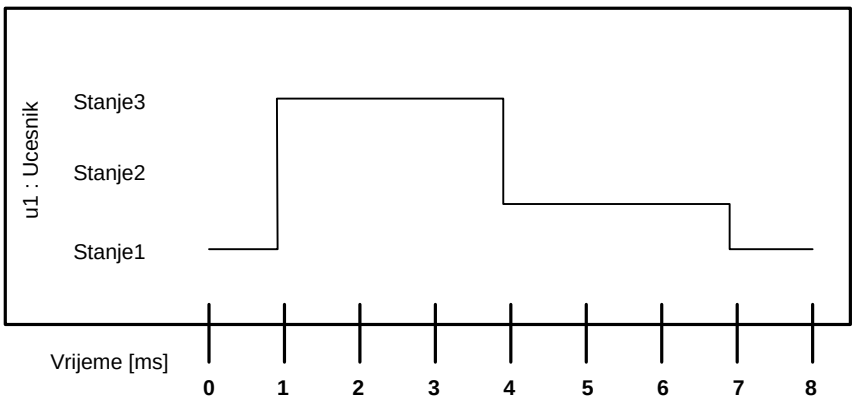
Na slici 13.1 je prikazan jednostavni dijagram toka vremena sa linijom života jednog učesnika. Stanja linije života su Stanje1 i Stanje2 i pokazana su lijevo od linije života. Prijelazi između stanja kao promjene na liniji uzrokovane sa događajima su `Događaj1` i `Događaj2`. Tok promjena koje se dešavaju je od lijeva prema desno i prikazani su sa linijom života koja indicira trenutno stanje. U ovom slučaju linija se naziva stanje ili uvjetna vremenska linija.



Slika 13.1: Linija stanja na dijagramu toka vremena

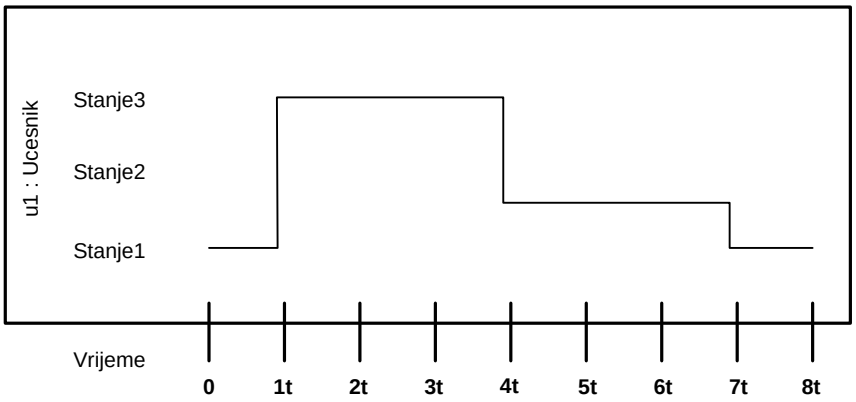
13.5 Vrijeme

Vrijeme na dijagramu toka vremena ide s lijeva na desno preko stranice, kao što je prikazano na slici 13.2.



Slika 13.2: Mjerenje vremena je smješteno na dijagram toka vremena kao ruler na dnu stranice

Mjerenje vremena može se iskazati na mnogo različitih načina; možemo imati precizno mjerenje vremena kao što je prikazano na slici 13.2 ili indikatore za relativno vrijeme kao što je prikazano na slici 13.3.



Slika 13.3: Indikatori relativnog vremena

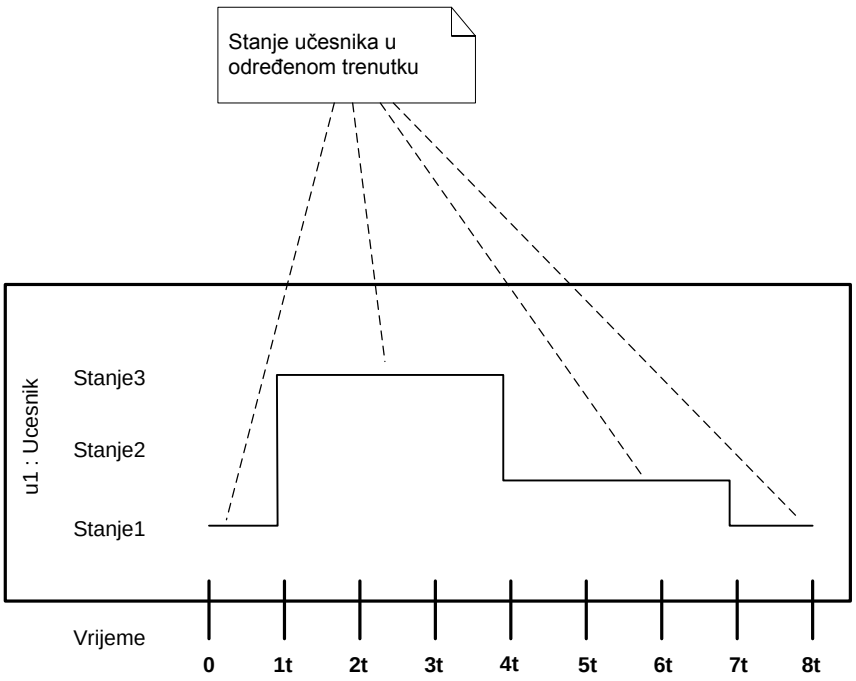
Indikatori realnog vremena su posebno bitni kada je prisutan vremenski aspekt tipa jedan učesnik će biti u nekom stanju pola vremena za koje će drugi učesnik biti u nekom drugom stanju.

Na dijagramu toka vremena, t predstavlja tačku vremena koje je u fokusu posmatranja. Ne zna se kada će se dogoditi taj vremenski trenutak, jer je moguće da se dogodi kao odgovor na događaj ili poruku, ali t se odnosi na taj moment bez predznanja kada će se dogoditi. Sa referencom t mogu se specificirati vremenska ograničenja za tu tačku t.

13.6 Linija stanja učesnika

Nakon što se doda vrijeme na dijagramu toka vremena može se prikazati u kojem stanju je učesnik u određenom trenutku. Ako pogledamo prethodne slike može se vidjeti da je učesnikovo trenutno stanje označeno sa horizontalnom linijom koja se zove linija stanja (state-line).

U bilo kojem trenutku interakcije, učesnikova linija stanja je poravnata sa jednim od učesnikovih stanja (slika 13.4).



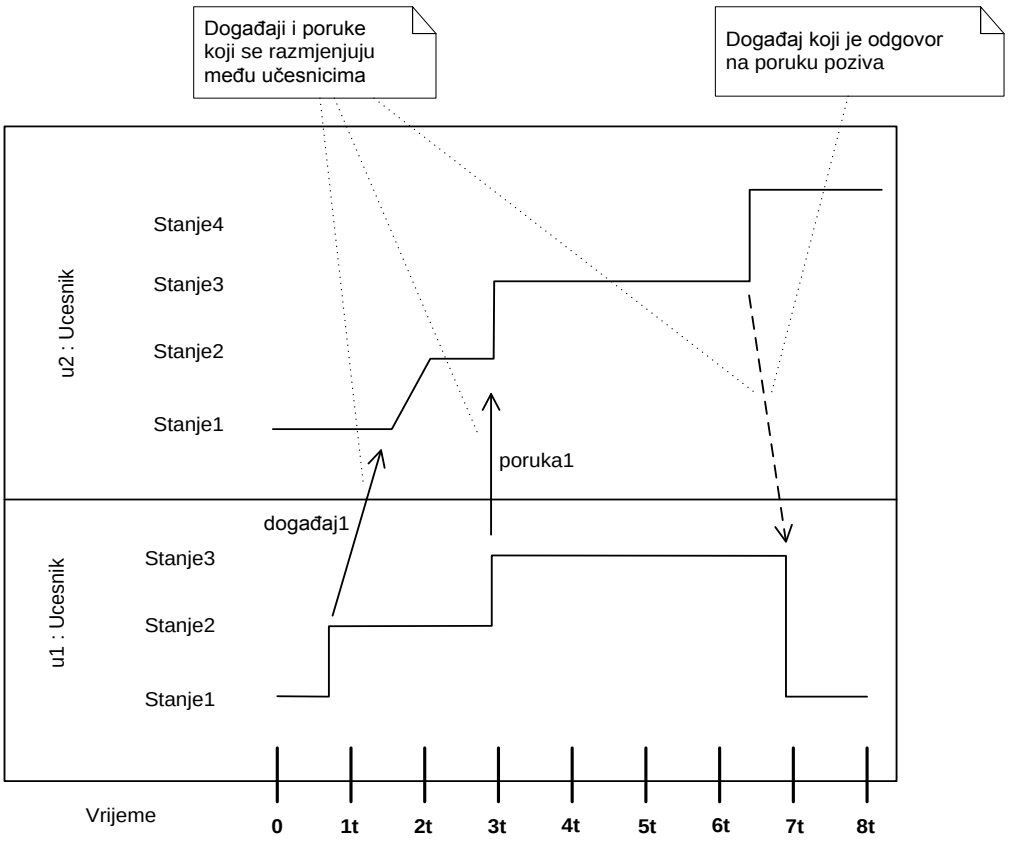
Slika 13.4: Stanje učesnika u određenom trenutku

U ovom primjeru, `u1:Učesnik` učesnikova linija stanja označava da je u stanju `Stanje1` jednu vremensku jedinicu, stanju `Stanje3` tri vremenske jedinice, i stanju `Stanje2` otprilike tri jedinice vremena, prije povratka u stanje `Stanje1` na kraju interakcije.

To je sve što se tiče prikazivanja da je učesnik u određenom stanju u određenom trenutku. Vrijeme je pogledati zašto učesnik mijenja stanje, što nas vodi ka događajima i porukama.

13.7 Događaji i poruke

Učesnici mijenjaju stanje na dijagramu toka vremena kao odgovor na događaje. Ovi događaji mogu biti poruke, ili nešto drugo. Razlika između poruka i događaja nije važna na dijagramu toka vremena kao na dijagramu sekvenci. Događaj se prikazuje na dijagramu toka vremena da se prikaže promjena stanja učesnika i prikazuje se kao strelica od linije stanja jednog učesnika koji je izvor događaja do linije stanja drugog učesnika koji je primalac događaja.

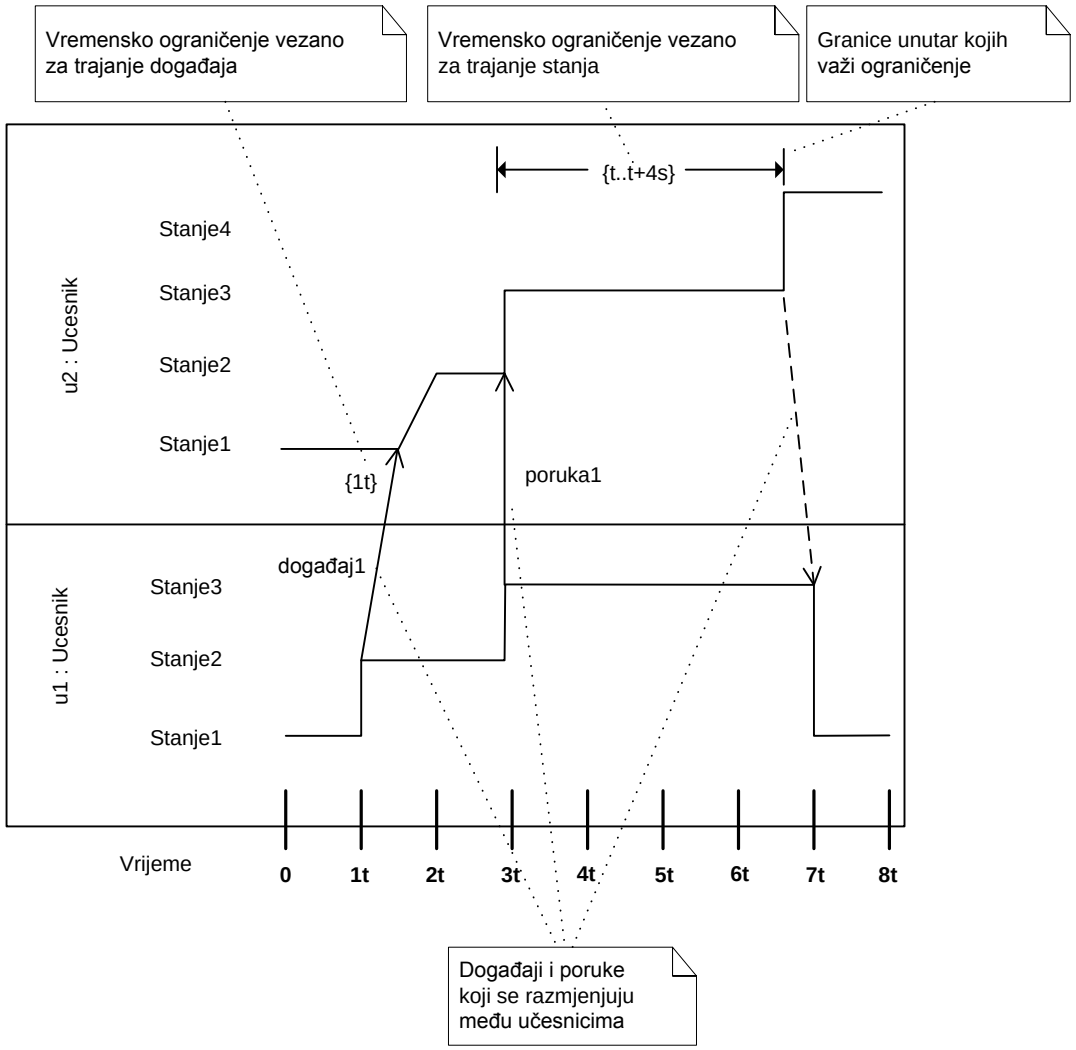


Slika 13.5: Događaj na dijagramu toka vremena

Na slici se vidi da događaj može imati svoje trajanje, kako je pokazano događaj1 uzima jednu jedinicu vremena. `u1:Ucesnik1` poziva događaj, a primalac događaja je `u2:Ucesnik2`.

13.8 Vremenska ograničenja

Do ove tačke, već smo savladali osnove dijagrama toka vremena i osnovne elemente u koje se ubrajaju učesnici, stanje, vrijeme, događaji i poruke. Vremenska ograničenja opisuju detaljno koliko dugo dâti dio interakcije može trajati. To se obično primjenjuje na količinu vremena koje učesnik treba provesti u pojedinačnom stanju ili koliko vremena će događaj uzeti za poziv i primanje, kao na slici 13.6.



Slika 13.6: Vremenska ograničenja

Primjenom ograničenja na dijagramu toka vremena, trajanje događaja `događaj1` mora biti manje od vrijednosti relativne mjere t, i `u2:Ucesnik2` može biti u stanju `Stanje4` maksimalno 4 sekunde.

Format vremenskih ograničenja

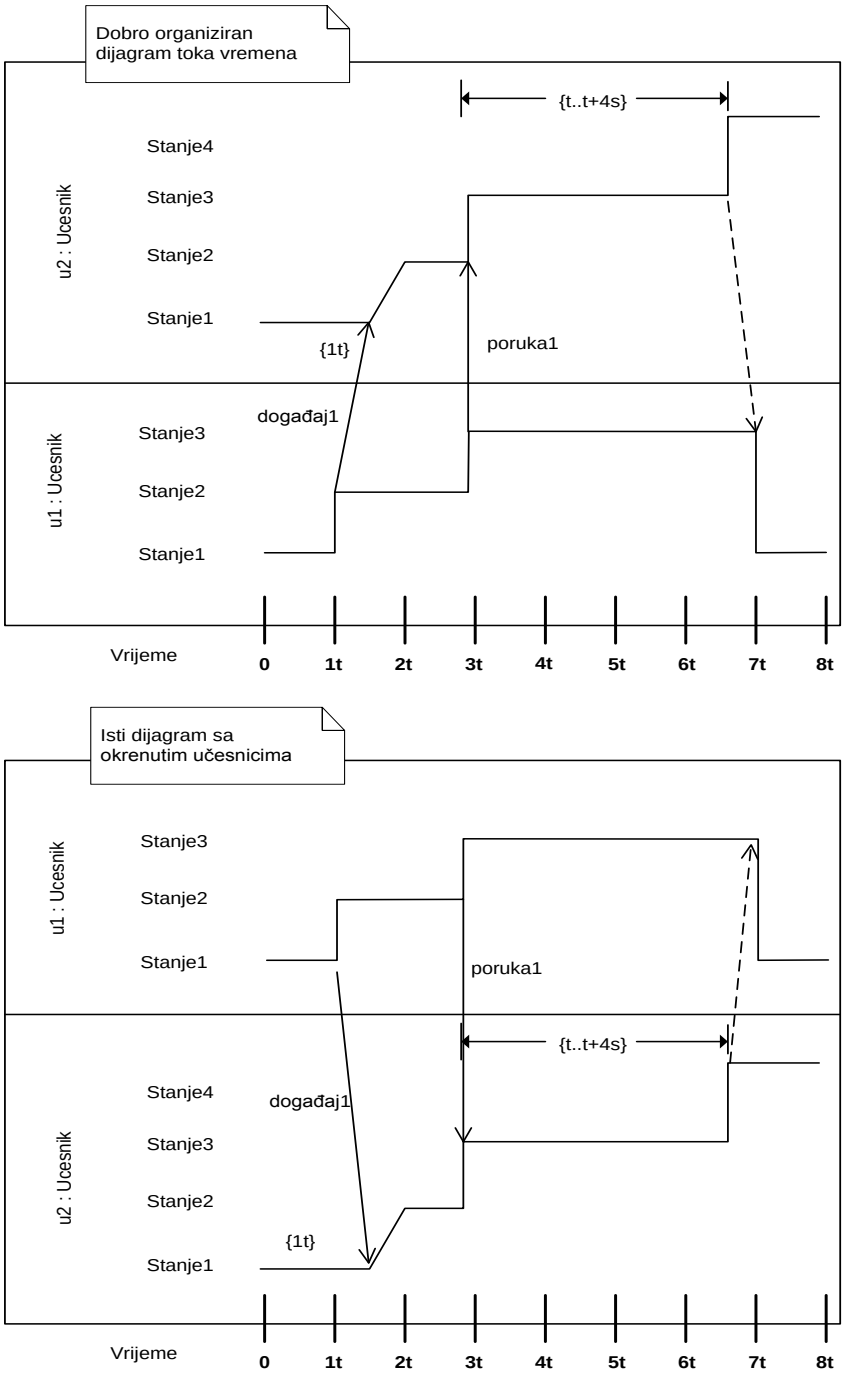
Vremenska ograničenja mogu se specificirati na više načina, u ovisnosti od informacija koje se modeliraju. Tabela 13.1 prikazuje uobičajne primjere vremenskih ograničenja.

Tabela 13.1: Različiti načini prikazivanja vremenskih ograničenja

Vremenska ograničenja	**Opis**
{t..t+5s}	Trajanje događaja ili stanja može biti 5 sekundi ili manje
{<5s}	Trajanje događaja ili stanja može biti manje od 5 sekundi. Manje formalna notacija od {t..t+5s}, ali ekvivalentna notacija
{>5s, <10s}	Trajanje događaja ili stanja može biti veće od 5 sekundi, ali manje od 10 sekundi
{t}	Trajanje događaja ili stanja treba biti jednako vrijednosti t. To je relativna mjera, gdje t može biti bilo koja vrijednost.
{t..t*5}	Trajanje događaja ili stanja može biti t*5 vremenskih jedinica (t može biti bilo koja vrijednost vremena)

13.9 Organiziranje učesnika na dijagramu toka vremena

Kada se dodaju učesnici na dijagram toka vremena, to i nije na prvi pogled veoma važno. Međutim, kada se dodaje više detalja na dijagram u formi događaja i vremenskih informacija, mjesto gdje je učesnik lociran na dijagramu toka vremena može uzrokovati probleme. Pogledajmo sliku 13.7.



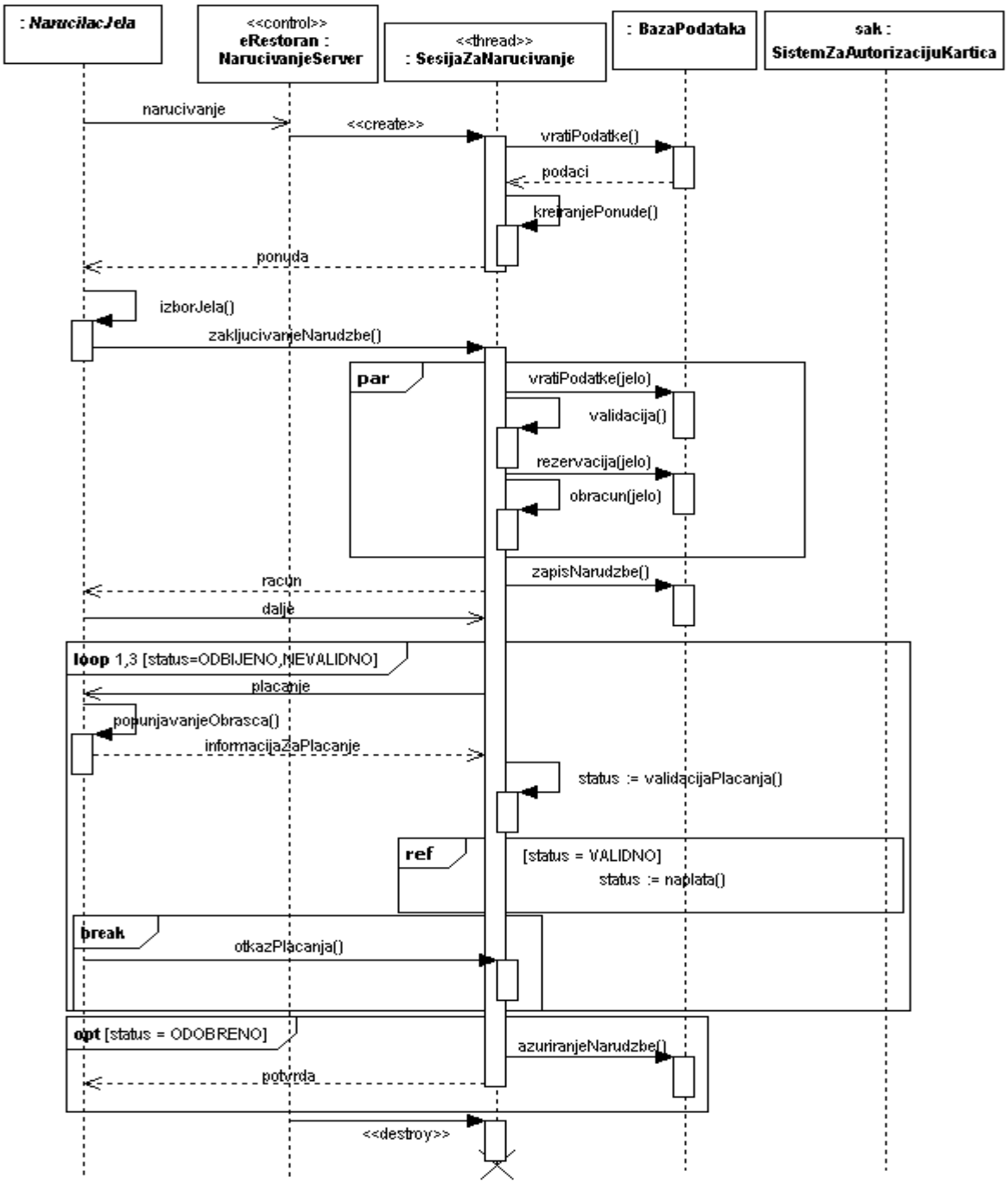
Slika 13.7: Postavljanje učesnika na dijagramu

Uočava se da je donji dijagram teži za čitanje i da se detalji preklapaju.

Ako u okviru modela već ima dobro organiziran dijagram sekvenci, tada imamo dobru podlogu za postavljanje učesnika na dijagramu toka vremena. Jednostavno treba preuzeti redoslijed učesnika kako se oni prikazuju na vrhu stranice dijagrama sekvenci i zaokrenuti listu učesnika za 90 stepeni suprotno od smjera kazaljke na satu, kao što je prikazano na slici 13.9.

13.10 Dijagrama toka vremena na osnovu dijagrama sekvence za `e-restoran`

Sklopimo dijagram toka vremena za `e-restoran`. Nastavljamo na istom primjeru na kojem smo radili u poglavlju dijagrama sekvence i dijagrama komunikacije. Interakcija u procesu naručivanja je prikazana na slici 13.8.



Slika 13.8: Dijagram sekvence za `e-restoran`

Dijagram sekvence sadrži vrlo malo, skoro ništa, informacija o vremenu, i uglavnom je fokusiran na redoslijed izvršavanja događaja unutar interakcije.

Vremenska ograničenja zahtjeva

Interakcija prikazana na slici 13.8 je rezultat specificiranih zahtjeva kao što je sljedeći zahtjev:

Sistem `e-restoran` *treba omogućiti naručivanje izborom jela sa trenutno važećeg jelovnika, pri čemu se podaci o trenutno važećem jelovniku uzimaju iz baze podataka.*

Specificirani zahtjevi kao što je navedeni nemaju sadržan vremenski aspekt pa ih je potrebno proširiti dodavanjem vremenskih ograničenja koja su važna za proces koji se modelira. Nakon ažuriranja prethodni zahtjev može dobiti sljedeću formulaciju:

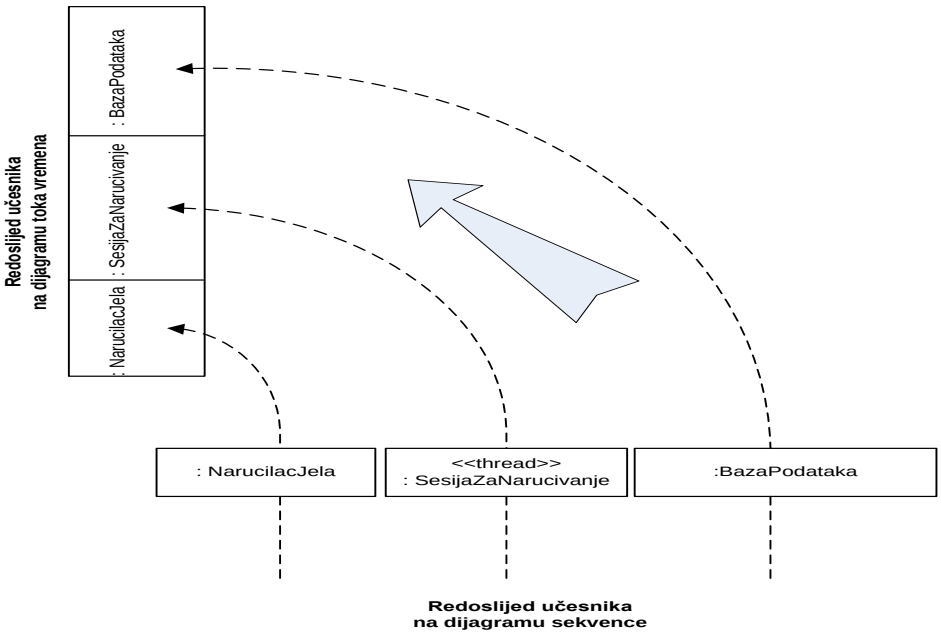
Sistem `e-restoran` *treba omogućiti naručivanje izborom jela sa trenutno važećeg jelovnika, pri čemu se podaci o trenutno važećem jelovniku moraju pripremiti za manje od 5 sekundi.*

Nakon uvođenja i vremenskih ograničenja u zahtjeve, koja diktiraju trajanje određenih interakcija, moguće ih je implementirati na dijagramu toka vremena.

Primjena učesnika na vremenskom dijagramu

Najprije treba kreirati dijagram toka vremena koji uključuje sve učesnike koji učestvuju u interakciji, a za koje su vremenski aspekti specificirani u zahtjevima, kao što je prikazano na slici 13.9.

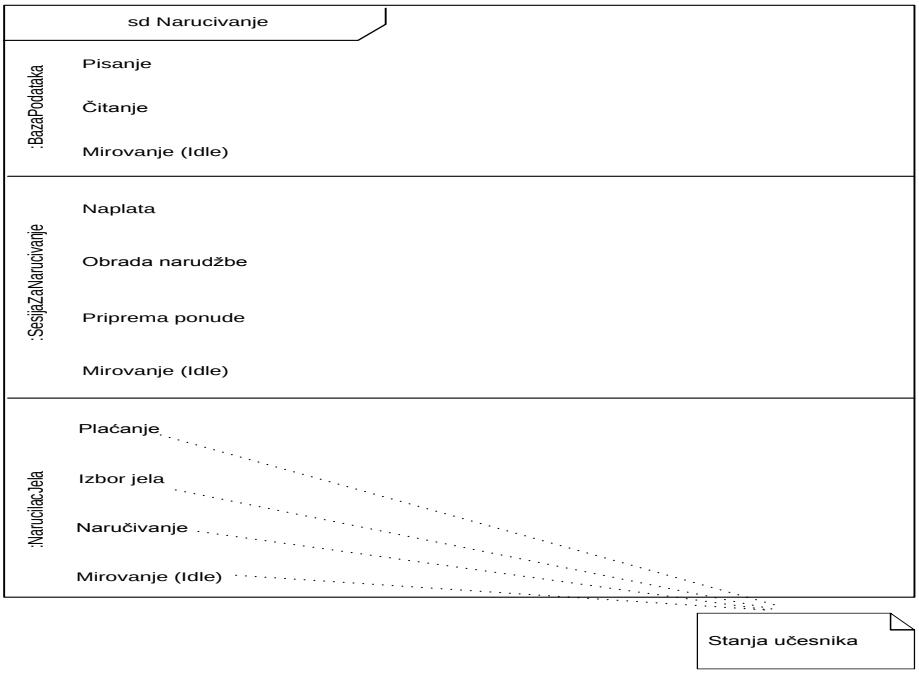
Učesnike koji nisu značajni za razumijevanje vremenskih aspekata interakcije specificiranih u zahtjevima, možemo izostaviti sa dijagrama, jer se dijagrami toka vremena fokusiraju na vrijeme u odnosu na promjenu stanja. Zbog toga su izostavljeni `eRestoran:NarucivanjeServer` i `sak:SistemZaAutorizacijuKartica`.



Slika 13.9: Učesnici sa dijagrama sekvence na dijagram toka vremena

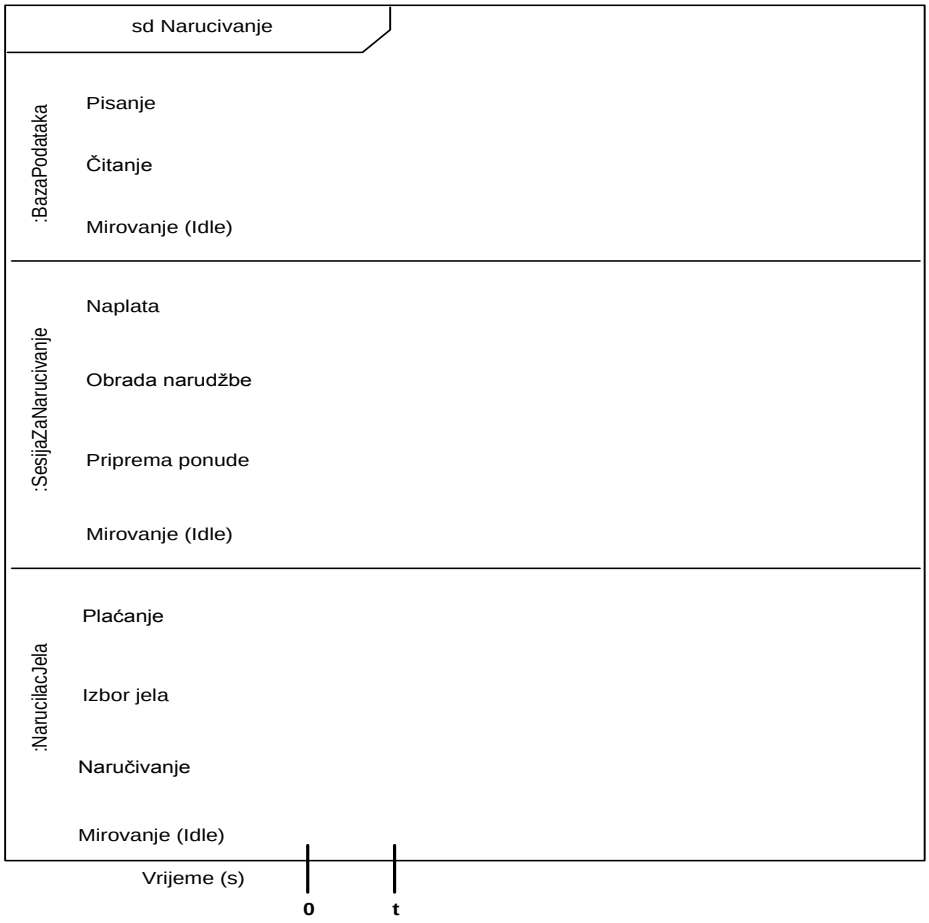
Stanja

Stanja su na dijagramu toka vremena postavljena do odgovarajućeg učesnika kao što pokazuje slika 13.10.



Slika 13.10: Stanja su napisana horizontalno na dijagramu toka vremena pored učesnika

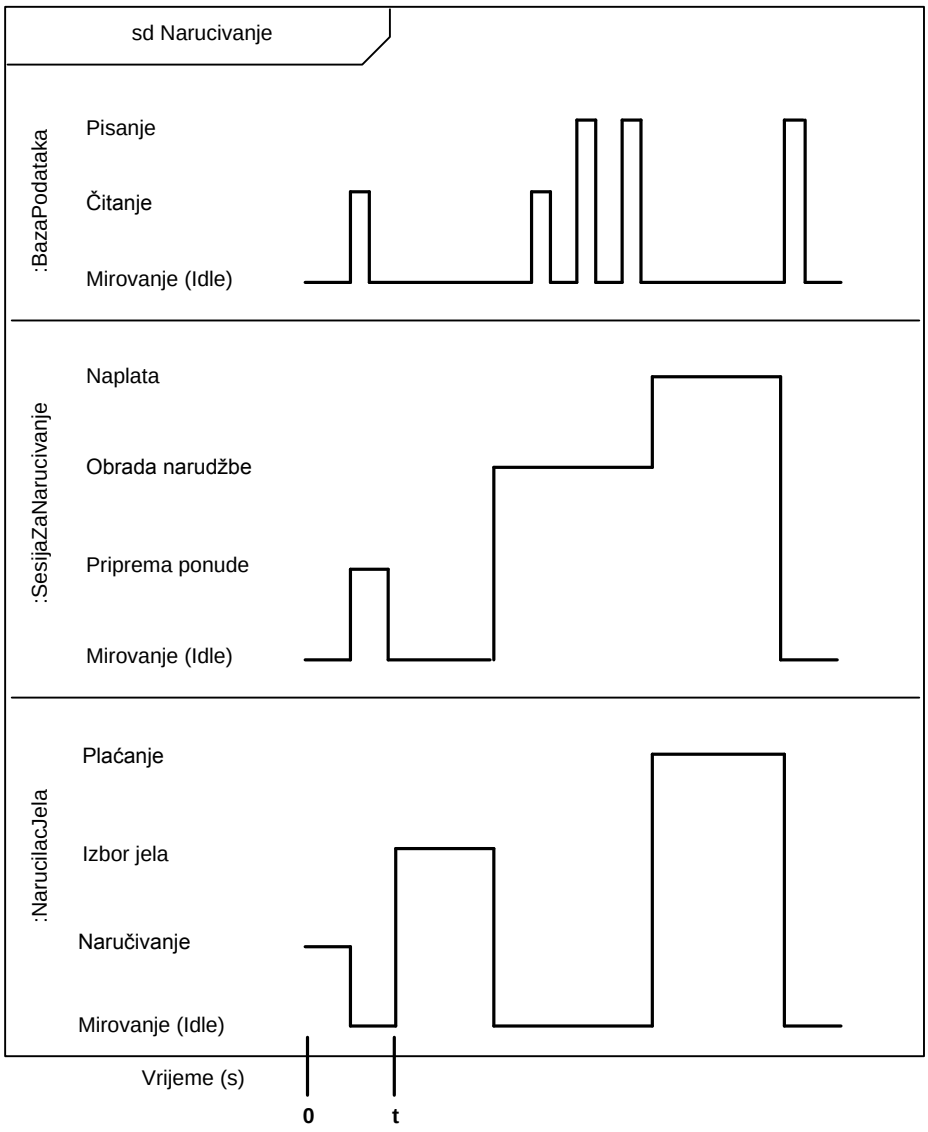
Nakon toga dodajemo i vremenski aspekt na slici 13.11.



Slika 13.11: Mjerenje vremena – vremenska skala

Mjerenje vremena je u sekundama. Početak je označen sa nulom, a mjesto od kog se počinju primjenjivati vremenska ograničenja je označeno relativno sa t.

Sljedeći dijagram toka vremena (slika 13.12) je ažuriran kako bi prikazao stanje svakog od učesnika u određenom trenutku u toku interakcije.

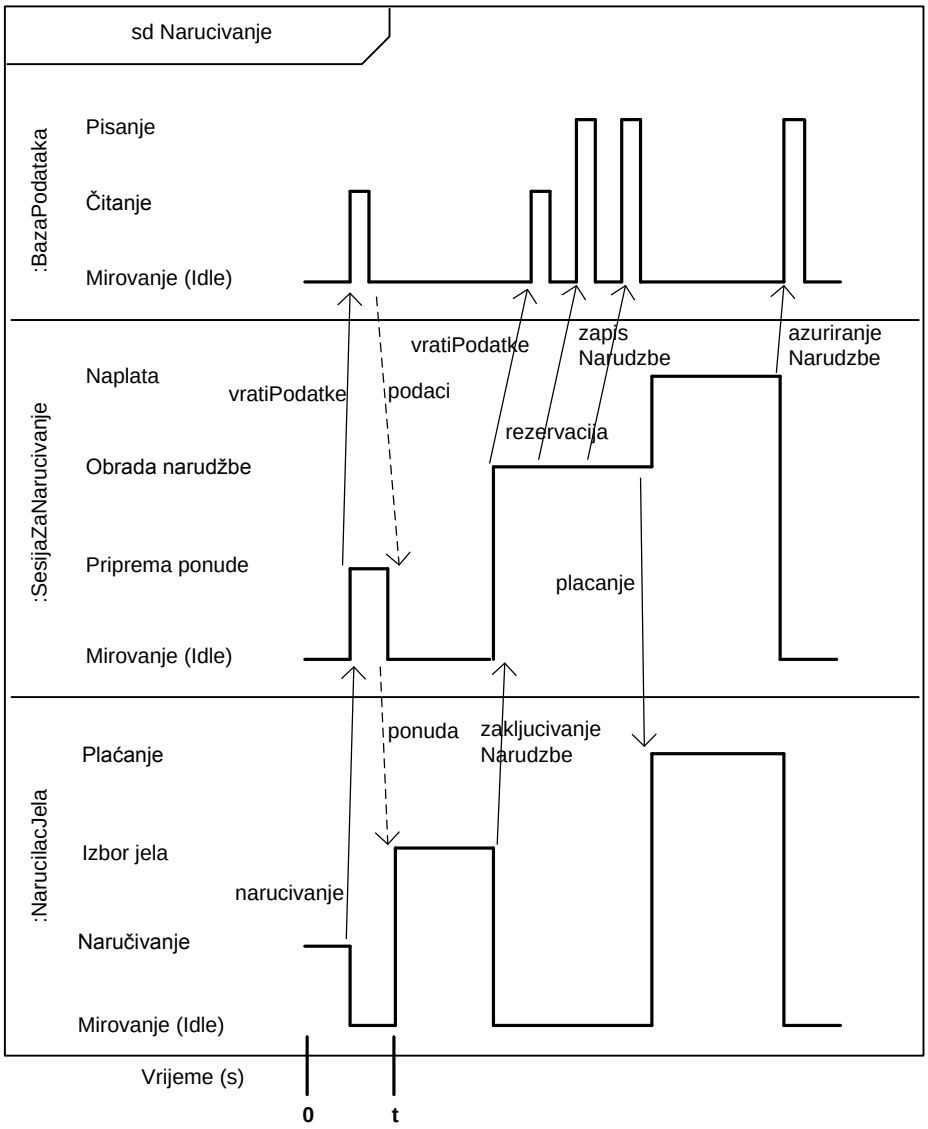


Slika 13.12: Dijagram toka vremena ažuriran sa stanjima svakog od učesnika u određenom trenutku u toku interakcije

U praksi se često dodaju u isto vrijeme i događaji i stanja na dijagram toka vremena. Mi smo ovdje razdvojili ove aktivnosti jednostavno zato što je lakše pratiti kako se dva dijela notacije primjenjuju bez miješanja jednog s drugim.

Dodavanje događaja na dijagram toka vremena je ustvari veoma jednostavan zadatak, jer imamo referentni dijagram sekvence (slika 13.8). Dijagram sekvence već pokazuje poruke koje se razmjenjuju između učesnika, tako da se jednostavno ove poruke mogu dodati na

dijagram toka vremena kao na slici 13.13. Promjena stanja učesnika ima više smisla kada se vide događaji koji ih uzrokuju.

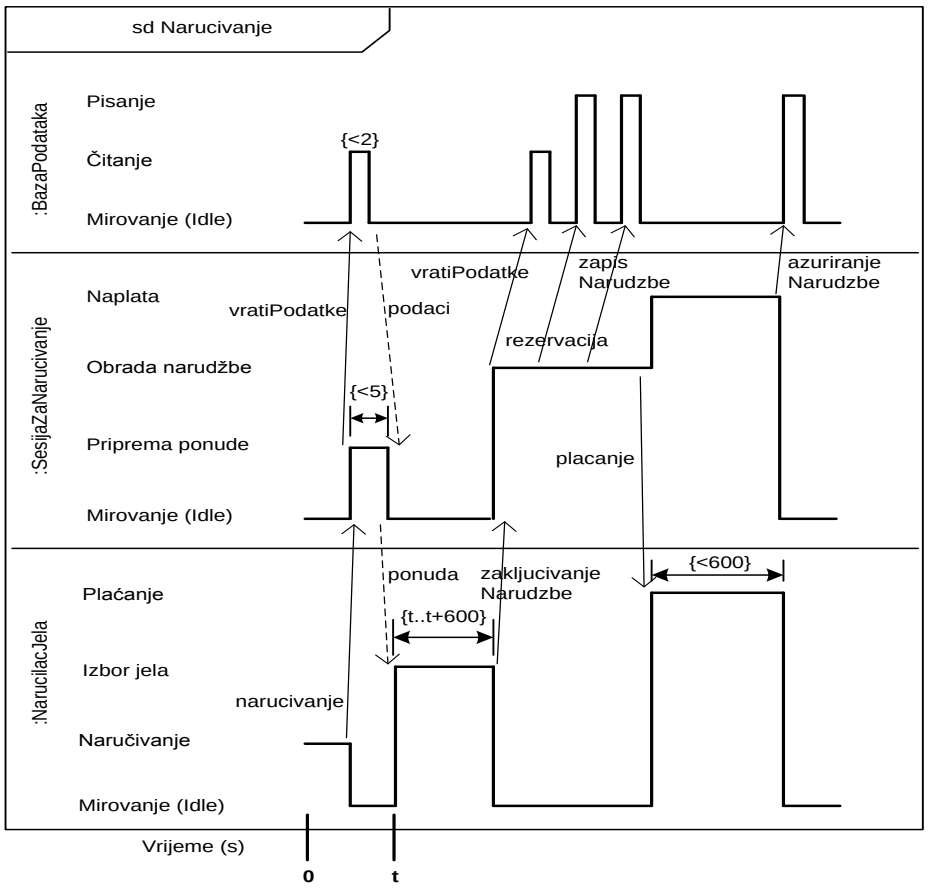


Slika 13.13: Dijagram toka vremena sa događajima koji uzrokuju promjenu stanja

Primjena vremenskih ograničenja na stanja i događaje

Na početku ovog dijela govorili smo o proširivanju zahtjeva da bi uzeli u obzir i vremenska ograničenja. Dodavanjem i vremenskih ograničenja dobivamo cjelovitu sliku sistema u ovisnosti od vremenskih aspekata. Sa slike 13.14 vidimo da ponuda mora biti formirana za manje od 5 sekundi. Zbog toga i čitanje iz baze mora biti za manje od 2 sekunde. Nakon

formiranja ponuda se vraća naručiocu jela (momenat t, na dijagramu), a on ima 600 sekundi da izvrši naručivanje i zaključi narudžbu (ograničenje t..t+600, na dijagramu). Vrijeme koje protekne između zaključivanja narudžbe i plaćanja nije strogo određeno, ali vrijeme u kome se mora izvršiti naplata/plaćanje je određeno i može trajati do 600 sekundi (ograničenje <600, na dijagramu). Kada postoje ovakva ograničenja, onda to znači da sistem mora reagirati u slučaju da se ista naruše. Npr. u slučaju da korisnik ne izvrši naručivanje i zaključivanje narudžbe u roku od 600 sekundi smatra se da je odustao i sesija za naručivanje će biti uništena.



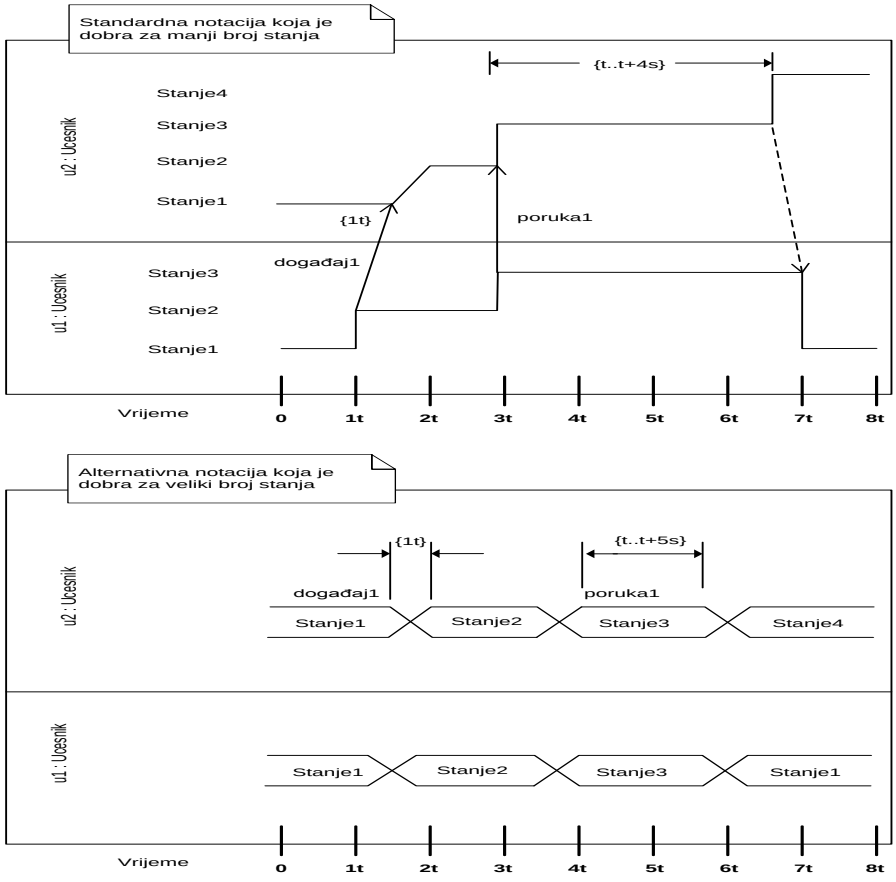
Slika 13.14: Vremenska ograničenja na stanje i događaje

13.11 Alternativna notacija

Dijagram toka vremena za proces naručivanja je prilično jednostavan primjer. Međutim, pitanje je kako će izgledati dijagram ukoliko je riječ o većem broju stanja nego što je to na trivijalnom primjeru.

Zašto imati drugu notaciju za dijagrame toka vremena? Jednostavan je odgovor na ovo pitanje. Notacija uobičajenih dijagrama jednostavno nije dovoljno dobra kada imamo mnogo učesnika koji mogu biti u puno različitih stanja tokom interakcije. Kao što je odgovor jednostavan tako i pravilo koju notaciju koristiti je prilično jednostavno. Ako se učesnik smješta u puno različitih stanja tokom trajanja interakcije, tada vrijedi razmisliti o korištenju alternativne notacije. Uobičajena notacija je u ovom trenutku šire prihvaćena u korisničkoj zajednici.

Alternativna, jednostavnija notacija za interakcije u kojima je uključen veći broj stanja je prikazan na slici 13.15.

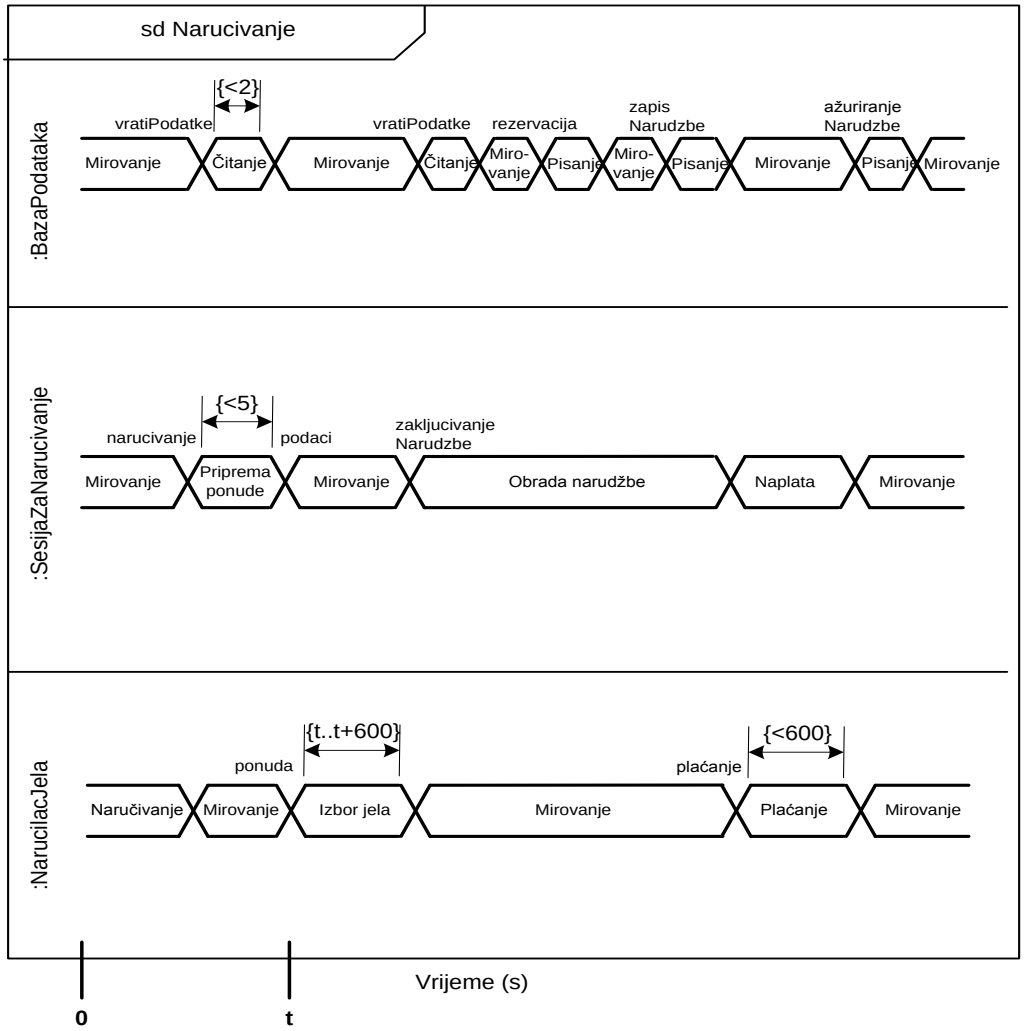


Slika 13.15: Standardna i alternativna notacija dijagrama toka vremena

Ukoliko se pogleda pažljivije može se primijetiti da alternativna notacija ne izgleda drastično drugačije od uobičajene notacije. Označavanje učesnika i vremena se nije promijenilo. Velika razlika u notaciji između uobičajne notacije dijagrama toka vremena i alternativne je u načinu prikaza stanja i prijelaza u stanja.

Uobičajena notacija prikazuje stanja kao listu pored relevantnog učesnika. Nakon toga je linija stanja potrebna da se prikaže u kojem je stanju učesnik u određenom trenutku. Nažalost ako učesnik ima mnogo različitih stanja, tada količina prostora potrebnog za modeliranje učesnika na dijagramu toka vremena brzo raste. Alternativna notacija rješava ovaj problem uklanjanjem vertikalne liste različitih stanja, koja smješta učesnikova stanja direktno u tačku u vremenu kada je učesnik u tom stanju. Zbog toga linija stanja više nije neophodna i sva stanja za određenog učesnika mogu biti smještena na jednu liniju na dijagramu.

Da bi se prikazale promjene stanja učesnika uslijed nekog događaja, znak x se postavlja između dva stanja i događaja koji uzrokuje promjenu stanja. Vremenska ograničenja mogu se primijeniti na skoro isti način kao kod uobičajene notacije.



Slika 13.16: Interakciju u procesu naručivanja koristeći alternativnu notaciju dijagrama toka vremena

Tokom procesa modeliranja trebat će odlučiti što će, a što neće biti eksplicitno prikazano na dijagramu. Pitajte se: "Da li je ovaj detalj neophodan za shvaćanje onog što se modelira" i "da li ovaj detalj nešto dodatno pojašnjava". U slučaju pozitivnog odgovora na jedno od ovih pitanja, najbolje je uključiti detalj u dijagram, a ako ne, treba ga izostaviti. Ovo može zvučati kao grubo pravilo, ali je izuzetno efikasno kada zbrku na dijagramu pokušavamo održati na minimumu.

Završno razmatranje

Dijagram toka vremena je povezan sa dijagramom sekvence. Koncept stanja objekta je povezan sa dijagramom toka vremena pošto pokazuje stanje objekta u određenom vremenu. Dijagram stanja pokazuje detaljno stanja objekata i događaje koji uzrokuju promjenu tih stanja. I dijagram stanja i dijagram toka vremena je povezan sa modeliranjem sistema realnog vremena.

Pitanja za ponavljanje

1.Koja je osnovna namjena dijagrama toka vremena?
2.Za koju vrstu sistema se koriste dijagrami toka vremena?
3.Koji su osnovni elementi dijagrama toka vremena?
4.Kako se predstavljaju učesnici u okviru dijagrama toka vremena?
5.Što se predstavlja stanjem u okviru dijagram toka vremena?
6.Što uzrokuje promjenu stanja?
7.Kako se predstavlja vrijeme na dijagramu toka vremena?
8.Koja je osnovna namjena linije stanja učesnika?
9.Kako se predstavljaju događaji i poruke?
10.Šta se može predstaviti sa vremenskim ograničenjima?
11.Navedite formate predstavljanja vremenskih ograničenja.
12.Diskutirati važnost dobre organizacije učesnika u okviru dijagrama toka vremena.
13.Koji su koraci formiranja dijagrama toka vremena na osnovu dijagrama sekvence?
14. Kada se preporučuje korištenje alternativne notacije za dijagram toka vremena?
15.Kako se prikazuju pojedini elementi dijagrama toka vremena sa alternativnom notacijom?
16.Koja je veza dijagrama toka vremena i ostalih dijagrama?

POGLAVLJE 14.

DIJAGRAM PREGLEDA INTERAKCIJA

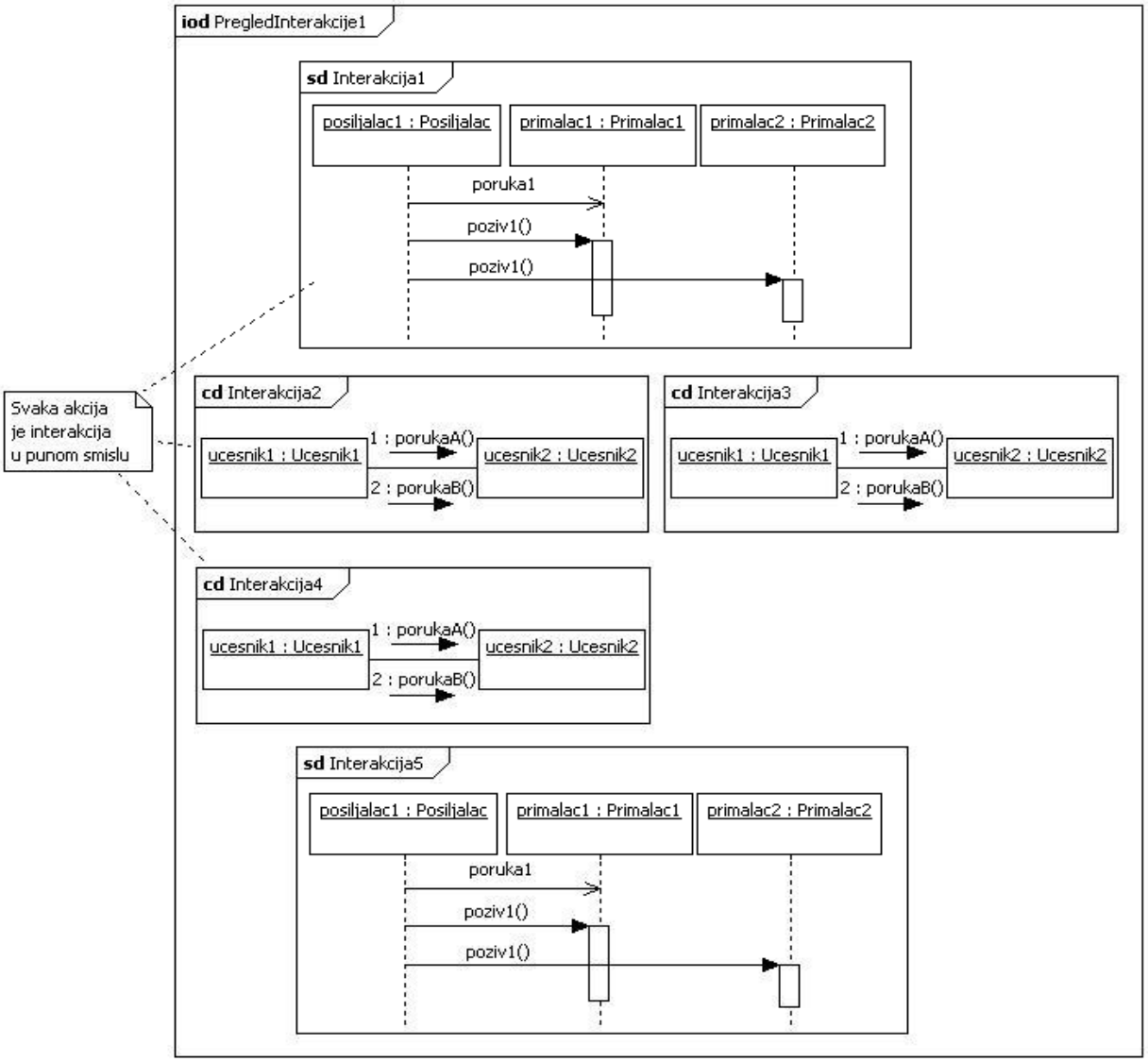
Dijagrami pregleda interakcija su uvedeni sa UML 2.0. Dijagrami pregleda interakcija daju pogled visokog nivoa kako više interakcija radi zajedno da bi implementirali sistem. Dijagrami sekvenci, dijagrami komunikacije i dijagrami toka vremena fokusiraju se na detalje koje se tiču poruka, dok dijagram pregleda interakcija povezuje zajedno različite interakcije u jednu kompletnu sliku interakcija.

Dijagram pregleda interakcije se povezuje i sa dijagramom aktivnosti. Dijagrami aktivnosti koji su uvedeni u poglavlju 10 pokazuju tok procesa. Osnovni elementi dijagrama aktivnosti su akcije, objekti, tokovi između akcija ili između akcija i objekata, odluke i stapanje, račvanje i spajanje. U dijagramu pregleda interakcija, čvorovi se mijenjaju sa slučajevima interakcije. Ako je interakcija u dijagramu pregleda interakcija u vezi sa vremenom, tada se koristi dijagram toka vremena, a ako je u vezi sa redoslijedom poruka, tada se koristi dijagram sekvence.

Namjena dijagrama pregleda interakcije je da prikaže pregled visokog nivoa na kontrolni tok unutar interakcije bez pokazivanja detalja o linijama života i porukama. Linije života i poruke se ne pojavljuju u dijagramu pregleda interakcija, umjesto toga dijagram se imenuje sa imenima linija života za koje je namijenjen.

14.1 Dijelovi dijagrama pregleda interakcija

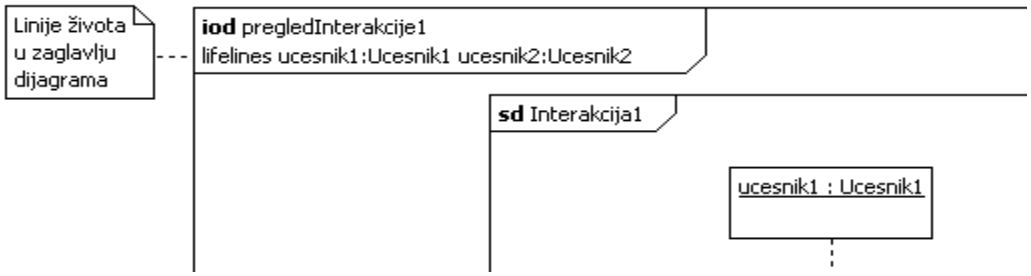
Najbolji način da se razumije dijagram pregleda interakcija jeste da se razmišlja o njemu kao dijagramu aktivnosti, ali umjesto opisa akcija ovdje se opisuju kompletne interakcije kako je pokazano na slici 14.1.



Slika 14.1: Dijagram pregleda interakcija sa individualnim interakcijama

O individualnim interakcijama na dijagramu pregleda interakcija se razmišlja kao o akcijama na dijagramu aktivnosti.

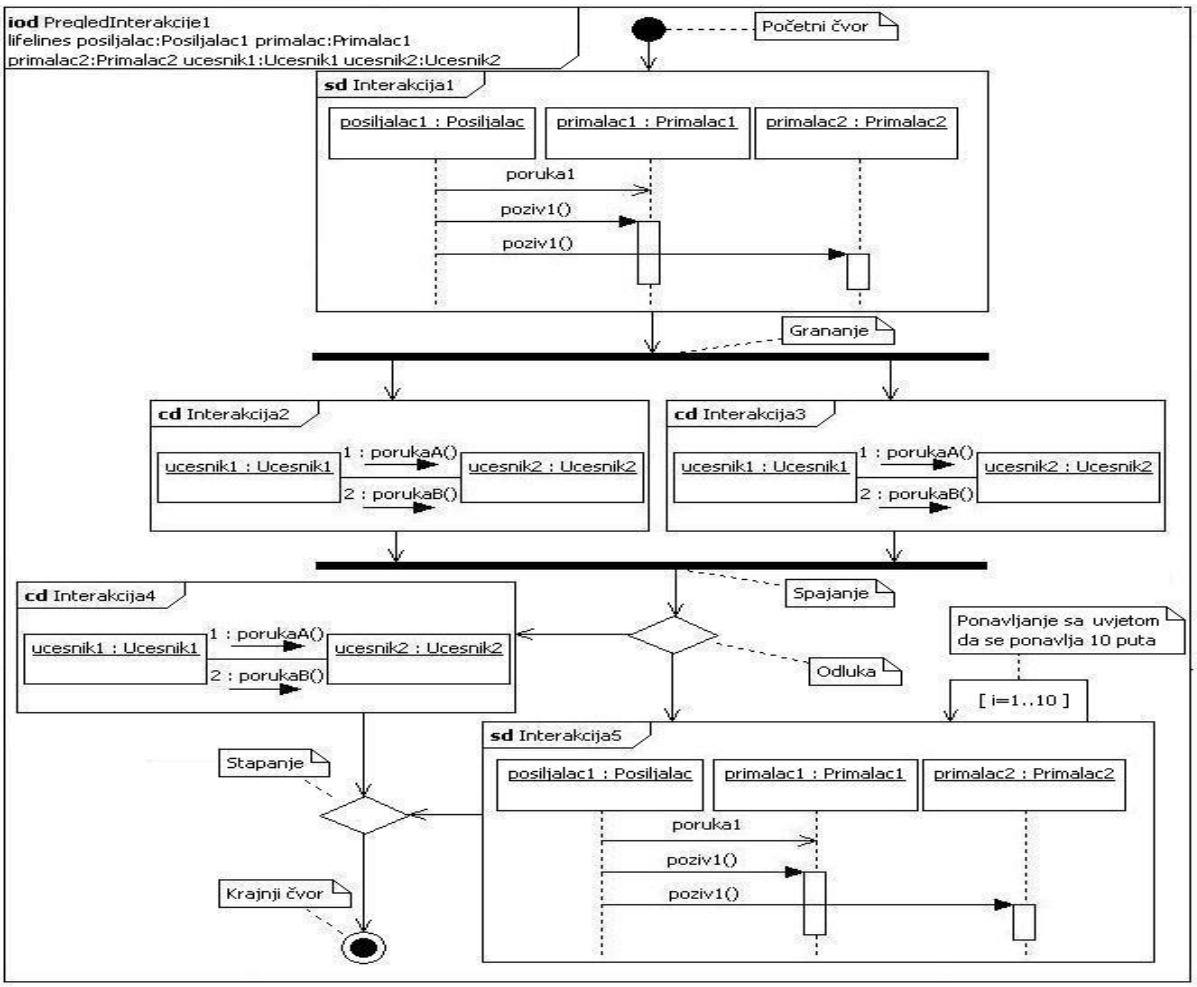
Više participanata može se uključiti u interakcije koje se dešavaju u okviru dijagrama pregleda interakcija. Da bi se vidjelo koji učesnici su uključeni u okviru dijagrama pregleda interakcija linije života dodaju se kao naslov dijagrama, kao što je pokazano na slici 14.2.



Slika 14.2: Linije života na dijagramu pregleda interakcija

Slično kao dijagram aktivnosti dijagram pregleda interakcija počinje sa inicijalnim čvorom i završava se sa krajnjim čvorom.

Kontrolni tok između ova dva čvora su interakcije koje su objašnjene u okviru dosadašnjih poglavlja namijenjenim dijagramima interakcije. Pošto kontrolni tok dijagrama aktivnosti može sadržavati odluke, paralelno izvršavanje, petlje, grananja, dijagram pregleda interakcija može sadržavati elemente kao na slici 14.3.



Slika 14.3: Notacija dijagrama pregleda interakcija

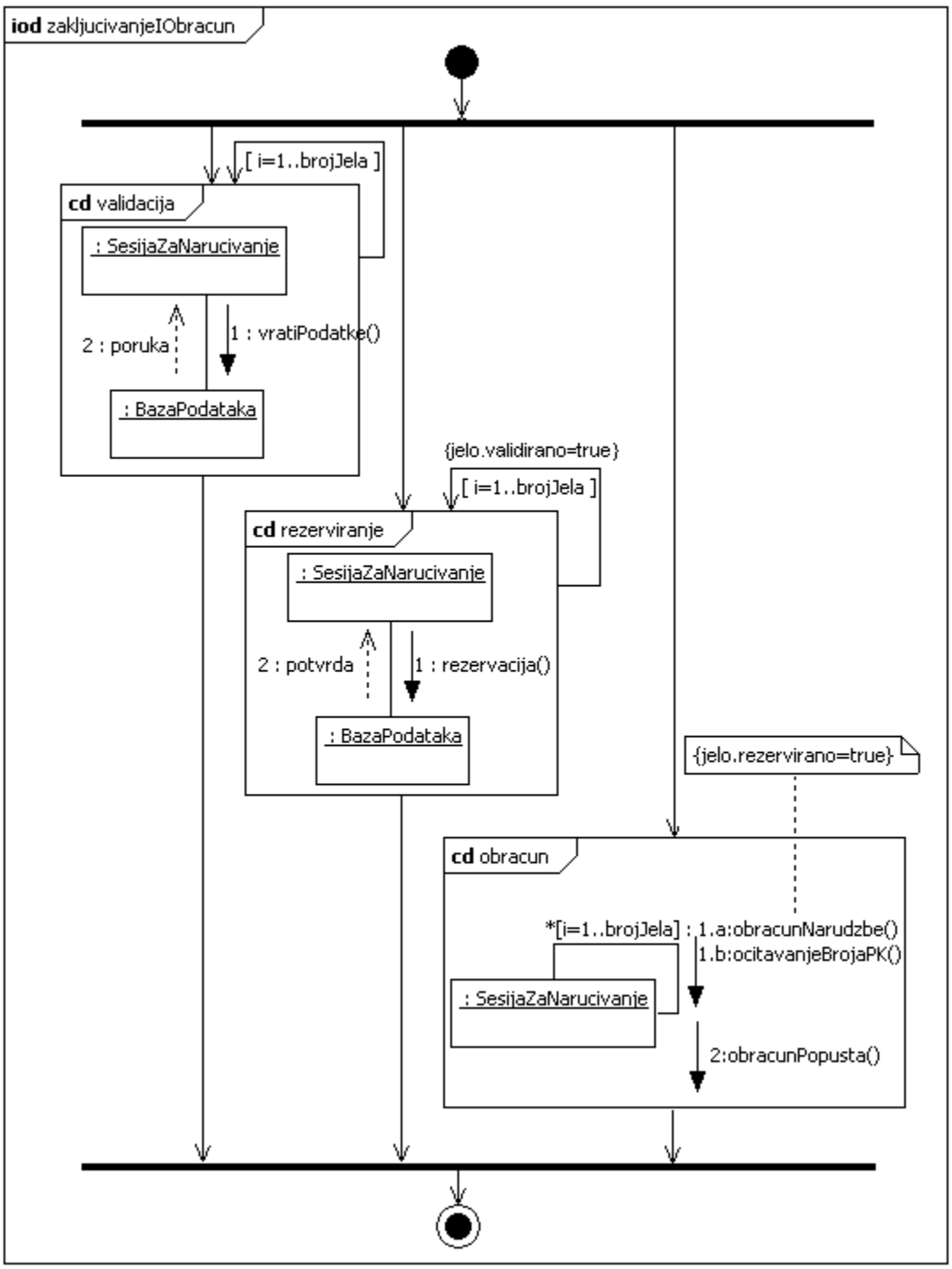
Dijagrami sekvence se najčešće nalaze u okviru dijagrama pregleda interakcija. Mogu se uočiti i neka pravila zamjene između dijagrama sekvence i njegovog prikaza u okviru dijagrama pregleda interakcije:

- Alternativni i opcionalni fragmenti se mijenjaju sa čvorom odluke i čvorom stapanja.
- Paralelni fragmenti se mijenjaju sa čvorom račvanja i spajanja.
- Grananje i spajanje grana mora biti odgovarajuće ugniježđeni.

U okviru dijagrama pregleda interakcija najčešće se koristi alternativna notacija dijagrama toka vremena.

14.2 Dijagram pregleda interakcija za `e-restoran`

Na slici 14.4 prikazana je pojednostavljena verzija dijagrama pregleda interakcije za `e-restoran`, odnosno prikazan je pregled interakcije za samo nekoliko akcija sa dijagrama aktivnosti. Slika pokazuje kako se dijagrami pregleda interakcije koriste da bi akcije sa dijagrama aktivnosti predstavili zasebnim interakcijama. Sa dijagrama možemo uočiti da se tri akcije: validacija, rezerviranje i obračun odvijaju paralelno na način da se validacija vrši za svako jelo posebno, dok se istovremeno može za sva jela koja su već validirana vršiti rezerviranje, a za sva rezervirana vršiti obračun narudžbe (uvećavanje iznosa narudžbe za svako sljedeće jelo). Istovremeno sa obračunom narudžbe može se vršiti očitavanje potrošačke kartice. Nakon toga se vrši obračun popusta i sve interakcije se spajaju. Time završava dio interakcije vezan za zaključivanje i obračun narudžbe.



Slika 14.4: Dijagram pregleda interakcija za `e-restoran`

Završno razmatranje

Dijagrami pregleda interakcija kombiniraju zajedno dijagrame sekvence, dijagrame komunikacija i dijagrame toka vremena da bi pokazali kompletnu sliku interakcija na visokom nivou.

Pitanja za ponavljanje

1. Koja je osnovna namjena dijagrama pregleda interakcija?
2. Diskutirajte vezu dijagram pregleda interakcija i dijagrama aktivnosti.
3. Diskutirajte vezu dijagrama pregleda interakcija i ostalih dijagrama interakcije.

POGLAVLJE 15.

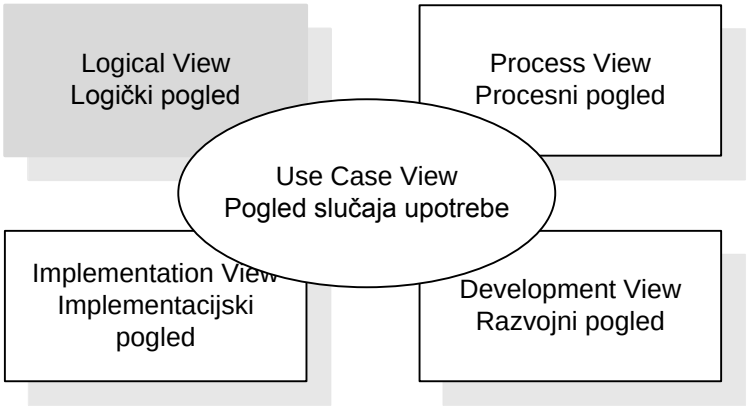
DIJAGRAM STANJA

Dijagrami aktivnosti i dijagrami interakcije su korisni za opisivanje ponašanja, ali još uvijek postoji nešto što nedostaje u prikazu sistema, a što se prikazuje sa dijagramima stanja.

Dijagram stanja je u veoma bliskoj relaciji sa dijagramom aktivnosti. Dok dijagram aktivnosti opisuju tok između pojedinih akcija neke aktivnosti, dijagram stanja opisuje tok između stanja. Ponekad je stanje objekta važan faktor ponašanja.

Ako neki sistem zahtijeva da potencijalni korisnik prijavi aplikaciju za otvaranje korisničkog računa, sama aplikacija može proći kroz razna stanja, može biti u redu za obradu, prihvaćena ili odbijena. U ovakvim situacijama je korisno modelirati stanja objekta i događaje koji prouzrokuju promjenu stanja. Upravo to dijagram stanja (eng. state machine) radi najbolje.

Dijagrami stanja se koriste u fazi analize za opisivanje kompleksnih ponašanja elemenata u okruženju sistema, i fazi dizajna za opisivanje ponašanja kompleksnih klasa i objekata. Na osnovu toga možemo ustanoviti da je dijagram stanja mašine dio logičkog modela sistema.



Slika 15.1: Dijagram stanja pripada logičkom pogledu

Puno je objekata koji mijenjaju stanje, gost u nekom restoranu može čekati da naruči, ili na isporuku naručenog, također, može čekati da plati račun. Automat za novac može biti u

raznim stanjima prilikom obrade neke transakcije: provjerava karticu, štampa priznanicu, daje novac.

Uobičajna je praksa u poslovnom svijetu razmišljati o entitetima koji prolaze kroz razna stanja.

15.1 Namjena dijagrama stanja

Dijagram stanja, koji se ponekad naziva i dijagram stanja mašine, veoma je koristan za sljedeće namjene:

- Opisivanje kompleksnih poslovnih entiteta, kao što su mušterija, računi.
- Modeliranje ponašanja podsistema.
- Modeliranje realizacije slučajeva upotrebe.
- Modeliranje interakcije za klase koje su namijenjene predstavljanju korisničkih interfejsa.
- Modeliranje kompleksnih objekata koji realiziraju kompleksne aktivnosti.

Dijagrami stanja često se koriste za modeliranje specijalnih slučajeva softverskih i hardverskih sistema uključujući:

- Sisteme u realnom vremenu/vremenski kritične sisteme kao sisteme za kontrolu rada srca;
- namjenske uređaje čije je ponašanje definirano preko stanja, kao što su na primjer mašine za podizanje novca;
- razne igre.

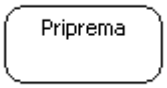
Dijagrami stanja se ne koriste kada je više objekata uključeno u interakciju, u tom slučaju su bolji dijagrami interakcije.

U okviru ove knjige više ćemo se fokusirati na dijagram stanja mašine, kojim se mogu prikazati stanja, tranzicije i ponašanje. Drugi tip mašine stanja, koji se naziva protokol mašina stanja (eng. protocol state machine), ne modelira ponašanje, ali je korisna za modeliranje protokola kao što je mrežni komunikacijski protokol.

15.2 Stanje i pseudostanje

Stanje (eng. state) u okviru dijagrama stanja je tačka životnog ciklusa određenog model elementa koja zadovoljava neke uvjete, gdje se pojedinačna akcija izvršava, ili gdje se čeka neki događaj.

Objekti unutar sistema mogu se posmatrati kako prelaze iz stanja u stanje. Eksterne aktivnosti uzrokuju neke akcije koje dovode do promjene stanja sistema ili nekih njegovih dijelova. Stanje može biti označeno kao aktivno ukoliko objekt nešto radi ili kao neaktivno ukoliko objekt ništa ne radi. Stanje se prikazuje kao pravougaonik sa zaobljenim ivicama i sa imenom u centru, kao što je označeno na slici 15.2. Ime treba biti jednostavno, ali deskriptivno. Na slici stanje se naziva `Priprema` i može da predstavlja stanje vezano za klasu `Narudzba`.



Slika 15.2: Stanje

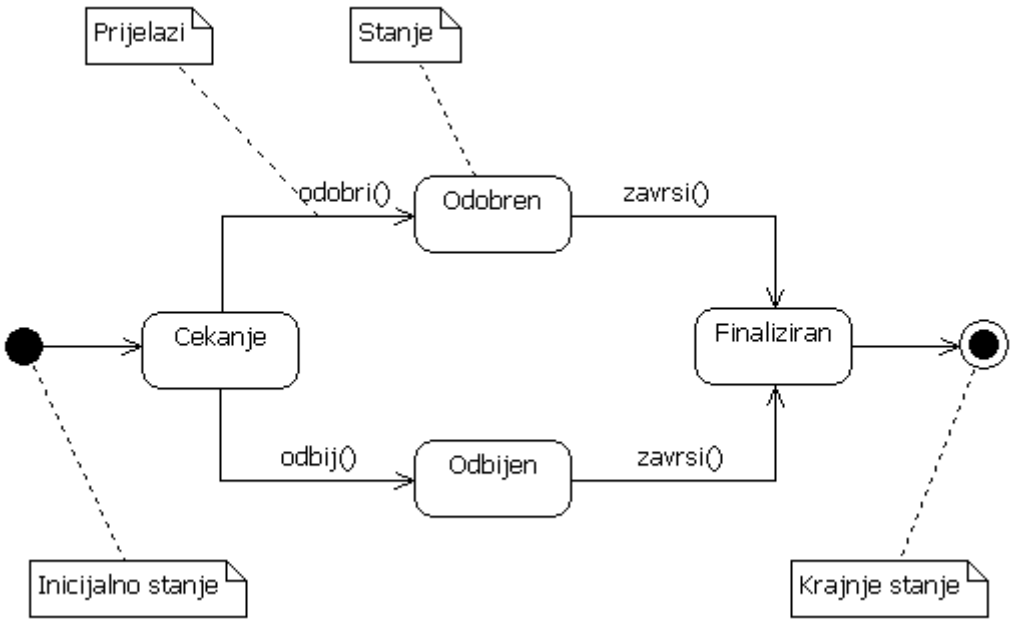
Stanje može biti jednostavno ili kompozitno (sastavljeno) stanje. Jednostavno stanje se ne može rastaviti i za objekt to je slično kao da predstavlja jednostavan skup vrijednosti atributa. Kompozitna stanja mogu se rastavljati, obično su to ugniježđene mašine stanja ili ponekad umetnuto stanje unutar nekog stanja na dijagramu. O ovim stanjima će biti više u posebnoj sekciji ovog poglavlja.

Postoje i specijalna pseudostanja. Pseudostanja su specijalni markeri koji usmjeravaju tok saobraćaja u dijagramima stanja.

Dijagrami stanja obično imaju inicijalno pseudostanje i krajnje stanje, koja označavaju startnu i krajnju tačku dijagrama stanja, respektivno. Inicijalno pseudostanje se prikazuje sa ispunjenim krugom, a krajnje stanje se prikazuje sa dva koncentrična kruga pri čemu je unutrašnji krug ispunjen, kao što je prikazano na slici 15.3.

Inicijalno (početno) stanje mora biti označeno sa događajem koji uzrokuje početak mašine stanja. Može postojati više inicijalnih početnih stanja. Može biti više i krajnjih stanja.

Pored početnog i krajnjeg pseudostanja postoje i druga pseudostanja koja modeliraju kompleksne tranzicije između stanja.



Slika 15.3: Elementi dijagrama stanja

U softveru, dijagrami stanja modeliraju ciklus života objekta, ili stanja kroz koja objekt prolazi tokom svog života. Slika 15.3 prikazuje životni ciklus objekta kako prolazi iz stanja čekanja označenog na slici `Cekanje` u stanje `Odobren` ili `odbijen` i zatim u stanje `Finaliziran`.

Dijagram stanja se sastoji od stanja i prijelaza o kojima slijedi u nastavku.

15.3 Prijelaz

Prijelaz predstavlja promjenu stanja, ili kako se prelazi iz jednog stanja u drugo. Prikazuju se kao linije sa strelicom od izvornog stanja u ciljno stanje, što je vidljivo na slici 15.3. Opis tranzicije, koji je napisan duž strelice opisuje okolnosti, odnosno događaj koji uzrokuje

promjenu stanja. Više od jedne tranzicije može se desiti iz jednog stanja. Stanje je aktivno kada tranzicija ulazi u njega, a neaktivno kada tranzicija izlazi iz njega.

Dijagrami stanja su korisni za modeliranje objekata koji se ponašaju različito ovisno od svog stanja. Za objekt na slici 15.3 pozivanje operacije `zavrsi`() kada je objekt u stanju čekanja ne bi imalo smisla. Finaliziraj stanje obavlja završno ponašanje i tada operacija `zavrsi`() ima smisla, jer za kreiranje korisničkog računa prvo treba znati da li je on odobren. Dijagrami stanja su efektan način da se ova informacija učini eksplicitnom.

Prethodni dijagrami imaju prilično jednostavne tranzicijske opise jer se sastoje samo od događaja koji prouzrokuju tranziciju. Ali tranzicijski opisi mogu biti puno kompleksniji.

Opisi prijelaza

Potpuna notacija za opise prijelaza je:

događaj[izraz] /operacija

Svaki element gornje notacije je opcionalan. Slijedi definiranje uloge svakog od elemenata date notacije .

Događaj

Sistem je upravljan događajima koji mogu biti eksterni ili interni. Sistem odgovara na događaje koji izazivaju promjene u sistemu, a najčešće prijelaze objekata iz jednog stanja u drugo. Eksterni događaj može uzrokovati seriju internih događaja, a ponekad oni mogu također uzrokovati eksterne događaje. Vrijeme također može biti izvor događaja. Kao primjer možemo uzeti generiranje računa za plaćanje na neki datum. Događaj u okviru UML-a prosljeđuje informacije sa parametrima. Ti parametri se mogu koristiti da se izrazi koje je sljedeće stanje kada se događaj desi. Pored događaja, prijelazi mogu također biti izazvani kompletiranjem internog ponašanja, što ćemo opisati kasnije.

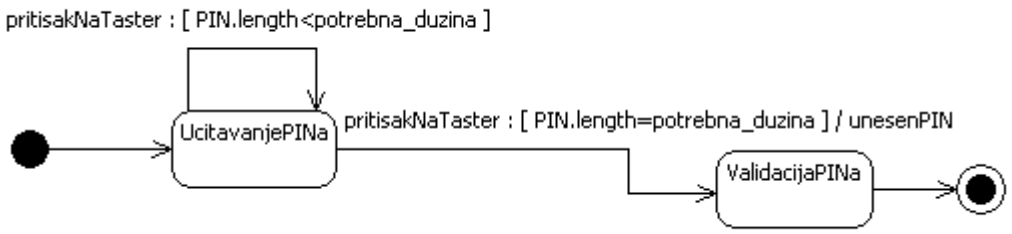
Izraz

Izraz je logički uvjet koji dozvoljava ili blokira prijelaz. Kada je izraz prisutan, prijelaz se desi ako se izraz procjeni na tačno, ali je prijelaz blokiran ako je izraz netačan.

Operacija

Operacija je neprekidna aktivnost koja se izvršava dok se prijelaz dešava. Može se nazvati i prijelazno ponašanje.

Slika 15.4. pokazuje prijelaze sa primjenom potpune notacije pri njihovom opisu.



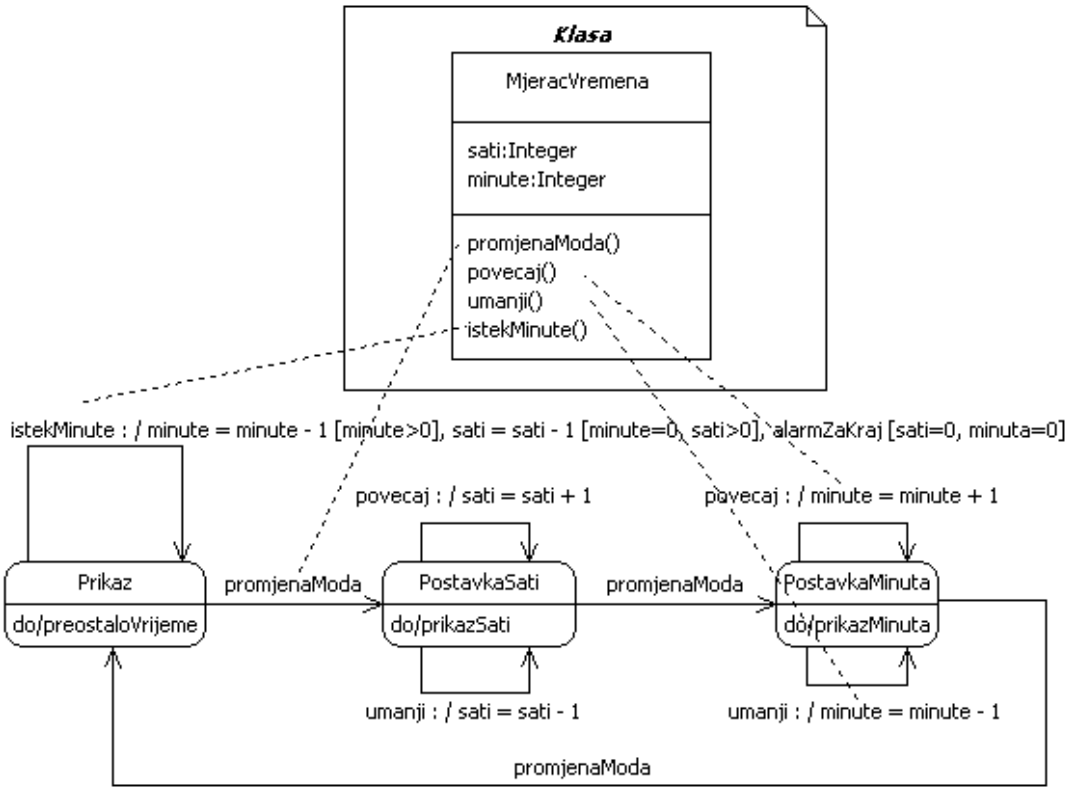
Slika 15.4: Opisi prijelaza

Pri unosu PIN-a na bankomatu, bankomat je u stanju `UcitavanjePINa` sve dok se pritiskom na taster unose brojevi PIN-a i dok je dužina unesenih brojeva manja od potrebne dužine. Kada se potvrdi unos PIN-a pritiskom na taster i kada je dužina unesenog PIN-a jednaka potrebnoj dužini uzrokuje se događaj `unesenPIN` koji vodi u stanje `ValidacijaPINa`.

Stanje može da izvrši i prijelaz u samo sebe. Na slici 15.4 je to vidljivo za stanje `UcitavanjePINa`.

15.4 Klasa i stanja

Često je veoma korisno pokazati vezu između klase i dijagrama stanja. Slika 15.5 pokazuje klasu `MjeracVremena` sa odgovarajućim dijagramom stanja. Na slici se jasno vidi kako se događaji mašine stanja vežu sa operacijama unutar klasa.



Slika 15.5: Klasa i stanja

U kuhinji, kuhari imaju mogućnost da podese vrijeme kuhanja nekog jela da bi im se po isteku tog vremena automatski alarmiralo da pogledaju da li je kuhanje završeno. Taj mehanizam je realiziran kroz klasu `MjeracVremena` koja ima atribute kao što su: `sati` i `minute`, ali ima i operacije koje dovode do promjene stanja mjerača vremena: `promjenaModa`, `povecaj`, `umanji`, `istekMinute`.

15.5 Interno ponašanje

Interno ponašanje je bilo koje ponašanje koje se desi kada je objekt u nekom stanju. Interno ponašanje je više generalni koncept koji također uključuje ulazno i izlazno ponašanje i veže se sa akcijama koje stanje može uzrokovati.

Sintaksa akcije je: *labela-akcije/akcija*.

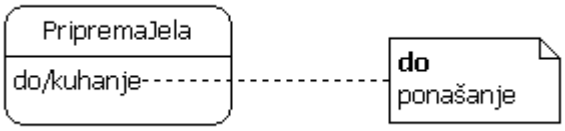
Labele akcije su entry, do, exit i označavaju akcije koje se dešavaju čim stanje postane aktivno, dok je stanje aktivno i kada se stanje napušta, respektivno.

Ukoliko se prikazuje i interno ponašanje stanja, simbol stanja se dijeli na dva dijela. Ime stanja se postavlja na vrh i odvaja se horizontalnom linijom od liste akcija vezanih za to stanje.

Na slici ispod je prikazano stanje `PripremaJela` sa akcijama koje se uzrokuju pri aktiviranju stanja (entry), dok je stanje aktivno (do) i kada stanje postaje neaktivno (exit).

Slika 15.6: Interno ponašanje stanja

Na slici 15.7 je prikazano stanje sa imenom `PripremaJela` i **do** ponašanjem i već smo naglasili da se njegova pripadajuća akcija dešava toliko dugo koliko je stanje aktivno.



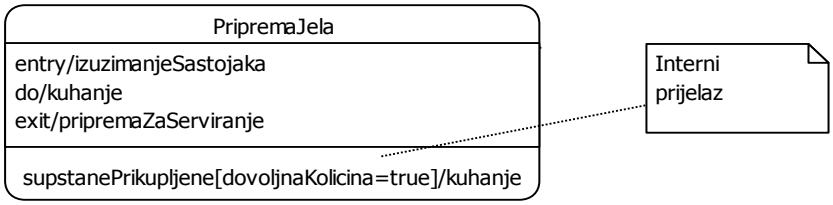
Slika 15.7: Radno **do** ponašanje stanja

Interni prijelazi

Interni prijelaz je prijelaz koji uzrokuje reakciju unutar stanja, ali ne uzrokuje da objekt promijeni stanje. Interni prijelaz je različito od prijelaza stanja u samo sebe jer taj prijelaz uzrokuje da se ulazno i izlazno ponašanje dogode dok interni prijelaz to ne radi.

Notacija za prijelaze je *događaj[izraz]/akcije*.

Interni prijelazi su prikazani unutar stanja, u okviru odvojenog dijela, kao na slici 15.8.



Slika 15.8: Donji dio pokazuje interne prijelaze

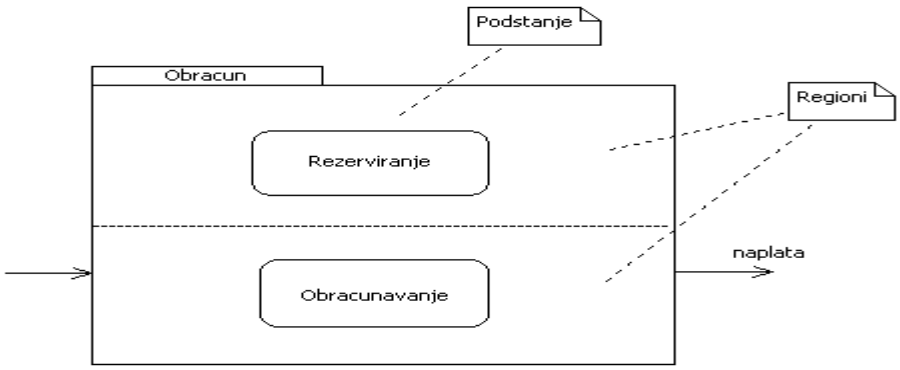
Na slici, stanje `PripremaJela` ima interni prijelaz koji možemo tumačiti kada se prikupe supstance za pripremu jela i kada je dovoljna količina supstanci za pripremu onda se prelazi na kuhanje jela.

15.6 Kompozitna stanja i regioni

Kompozitna stanja prikazuju da je jedan objekt u više stanja istovremeno. Kompozitno stanje sadrži jedan ili više dijagrama, pri čemu svaki dijagram stanja pripada jednom regionu. Regioni su odvojeni isprekidanom linijom. Stanje u regionu je podstanje kompozitnog stanja.

Kompozitno stanje radi na sljedeći način: kada kompozitno stanje postane aktivno, inicijalno pseudostanje svakog regiona postaje aktivno i odgovarajući dijagram stanja se počne izvršavati. Odgovarajući dijagram se prekida ako se desi neki događaj koji uzrokuje prekid tog kompozitnog stanja ili ako su se izvršile sve akcije do kompletiranja stanja.

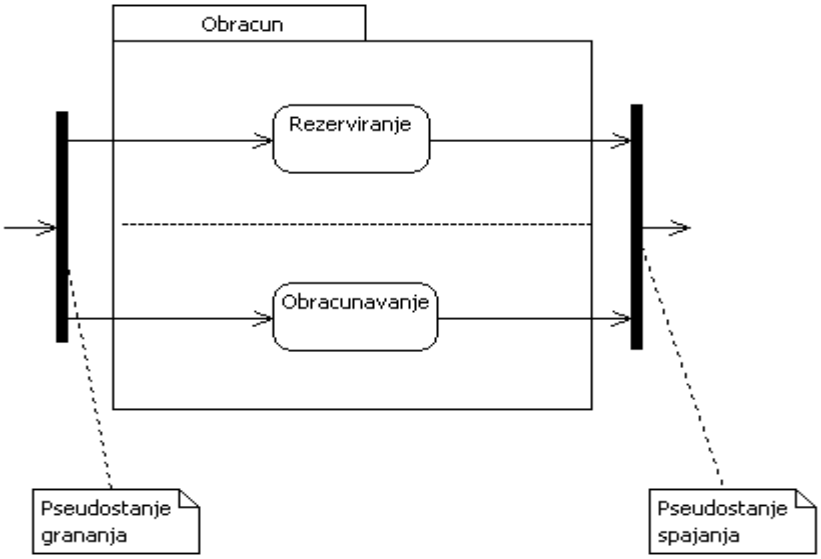
Možemo modelirati dva istovremena stanja koristeći kompozitna stanja, kako je prikazano na slici 15.9. Pretpostavimo da `Obracun` stanje radi dvije stvari u isto vrijeme `Rezerviranje` i `Obracunavanje`.



Slika 15.9: Kompozitna stanja

Grananje i spajanje

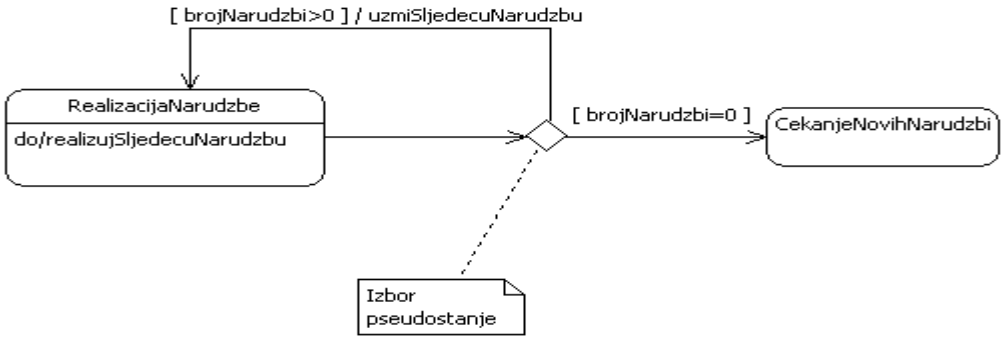
Grananje i spajanje pokazuju grananje u više konkurentnih stanja i potom njihovo spajanje u jedno stanje. Na slici 15.10, pseudostanje grananja prekida dolazeću tranziciju, dozvoljavajući da se `Rezerviranje` i `Obracun` dese istovremeno. Pseudostanje spajanja potom udružuje dvije dolazeće tranzicije u izlaznu tranziciju.



Slika 15.10: Pseudostanja grananje i spajanje

15.7 Napredna pseudostanja

Postoje i dodatna pseudostanja koja su korisna za usmjeravanje toka saobraćaja između stanja. Pseudostanje izbora je korisno da se naglasi logički uvjet koji određuje koji prijelaz slijedi. Logički uvjet se postavlja na svakoj mogućoj varijanti prijelaza. Na slici 15.11 je prikazano napredno pseudostanje izbora.

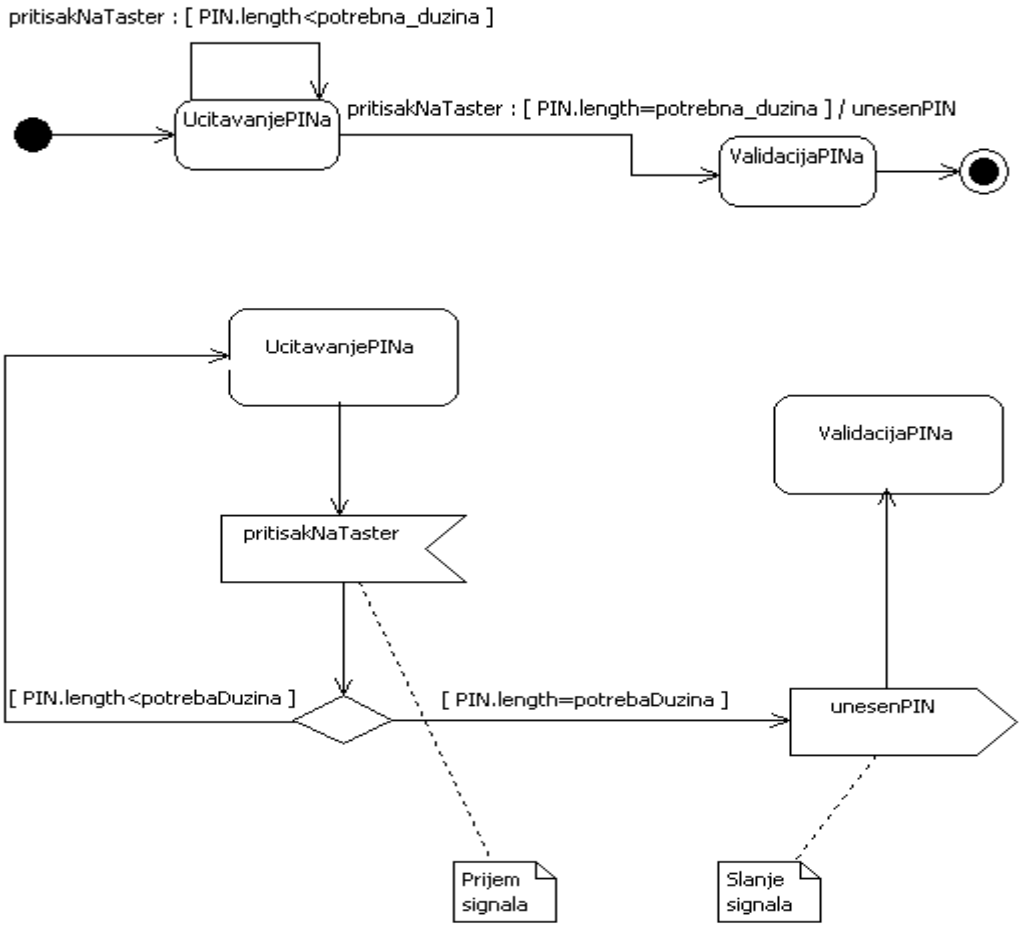


Slika 15.11: Put koji slijedi poslije izbora ovisi od uvjeta

15.8 Signali

Možemo koristiti specijalne ikone, simbole za prijem i slanje, da privučemo pažnju na prijelaze i prijelazno ponašanje. Ovo se naziva prijelazno orijentiran pogled.

Slika 15.12 prikazuje kako se koriste simboli prijema i slanja za primjer sa slike 15.4.

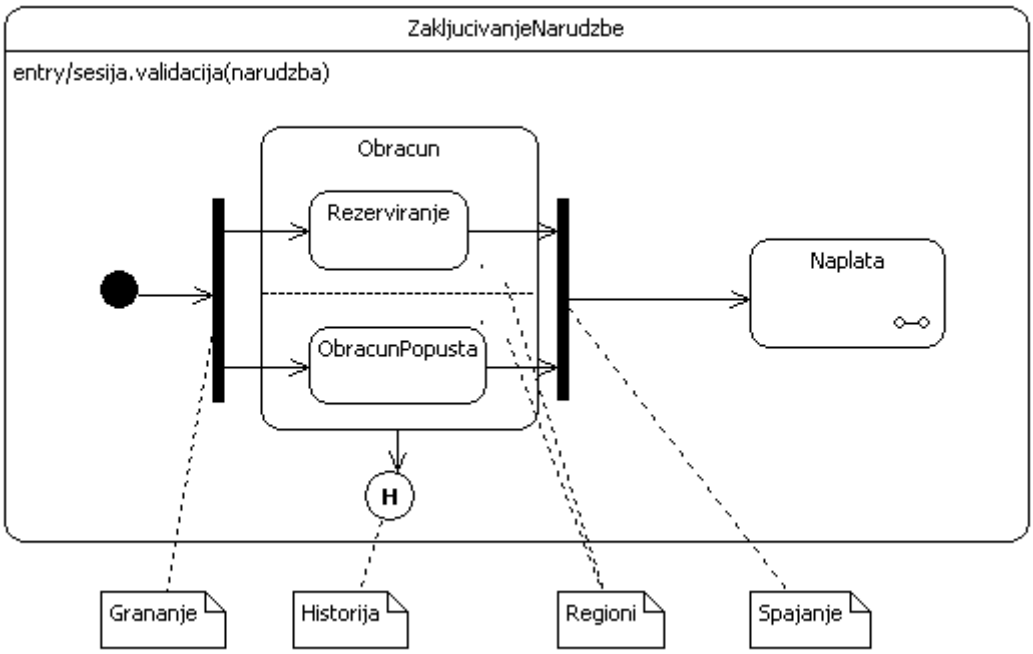


Slika 15.12: Prikaz prijelaza i prijelaznog ponašanja kao prijemni i poslani signal

Glavna namjena ove notacije je da vizualno naglasi prijemni i poslani signal. Iako oba dijagrama prikazuju isto, verzija sa ikonama signala je čitljivija.

15.9 Stanje historije

Ponekad je veoma korisno vratiti kompozitno stanje u njegovo prethodno stanje. To se radi sa dodavanjem stanja historije koje se obilježava kao krug sa H unutar njega. Ta historija je poznata kao kratka historija. Ako je historija obilježena sa H*, umjesto H, to je poznato kao duboka ili cijela historija čija namjena je pamćenje svih prethodnih stanja nekog stanja i u slučaju potrebe vraćanje svih podstanja u stanja u kojima su se nalazili. Slika 15.13 ilustrira primjenu i označavanje stanja historije.



Slika 15.13: Stanje historije

Slika 15.13 pokazuje promjene stanja u okviru stanja `ZakljucivanjeNarudzbe`. Nakon toga slijedi ulazna akcija validacija narudžbe. Zatim, u okviru stanja `Obracun` dešavaju se dva paralelna stanja: `Rezerviranje` i `ObracunPopusta` koji su sinhronizirani grananjem na ulazu i spajanjem na izlazu, a nalaze se u regionima odvojenim isprekidanom crtom. Nakon spajanja slijedi kompozitno stanje `Naplata` čija interna struktura nije predstavljena. Obračun koristi stanje kratke historije koje, ukoliko se tranzicijom dosegne, dovodi do toga da se vrati zadnje podstanje `Obracun` stanja. Ova vrsta historije ne vraća i podstanja u njihova zadnja stanja.

Završno razmatranje

U softveru, dijagrami stanja modeliraju ciklus života objekta, ili stanja kroz koja prolazi tokom svog života. Ako objekt ima jednostavan životni ciklus, tada nije vrijedno modelirati životni ciklus sa dijagramima stanja. Na primjer, objekt čija je namjena da sadrži autorske kontakt informacije i ne mijenja stanje osim da je kreiran i uništen ne treba dijagram stanja. Dijagram stanja je usko povezan sa dijagramom aktivnosti. Povezuju se i sa slučajevima upotrebe, klasama i operacijama.

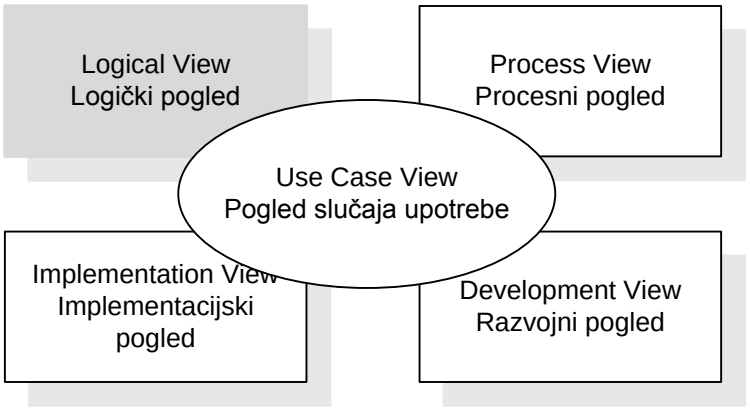
Pitanja za ponavljanje

1.Koja je osnovna namjena dijagram stanja?
2.Kojem pogledu pripada dijagram stanja?
3.Za koje sisteme se koristi dijagram stanja?
4.Kako se predstavlja stanje u dijagramu stanja?
5.Što su pseudostanja?
6.Kako se obilježava početno, a kako krajnje pseudostanje?
7.Koja je osnovna namjena prijelaza?
7.Koja je potpuna notacija za opis prijelaza?
8.Što je aktivno, a što pasivno pseudostanje?
9.Koja je veza između klase i dijagrama stanja?
10. Što je interno ponašanje stanja?
11. Koja je sintaksa za akcije internog ponašanja stanja?
12. Koje labele se koriste za opis akcija internog ponašanja?
13. Kako se obilježavaju akcije koje se izvršavaju čim stanje postane aktivno?
14.Kako se obilježava radno ponašanje stanja?
15.Kako se obilježavaju akcije koje se dešavaju kada stanje postane neaktivno?
16. Koja je notacija za opis internih prijelaza?
17. Koja je namjena pseudostanja izbora?
18. Kako dobivamo prijelazno orijentiran pogled sa dijagramom stanja?
19.Što je kompozitno stanje?
20. Koja je uloga grananja i spajanja pri predstavljanju konkurentnih stanja?
21. Koja je razlika između stanja historije koje je obilježeno sa H i H*.

POGLAVLJE 16.

DIJAGRAM SLOŽENE STRUKTURE

Ponekad osnovni UML dijagrami, kao što su dijagrami klase i dijagrami sekvenci ne mogu da daju pregledno sve detalje o sistemu. Dijagram složene strukture pomaže da se pregledno prikaže složena struktura i veza između klasifikatora visokog nivoa kao što su slučajevi upotrebe, klase i komponente. Ovaj tip UML dijagrama pokazuju i kako objekti rade zajednički da bi se postigla određena funkcionalnost sistema. Dijagram složene strukture pomaže u formiraju sveobuhvatnog logičkog pogleda na sistem.



Slika 16.1: Dijagram složene strukture formira logički pogled na sistem

16.1 Namjena dijagrama složene strukture

Namjena dijagrama složene strukture je:

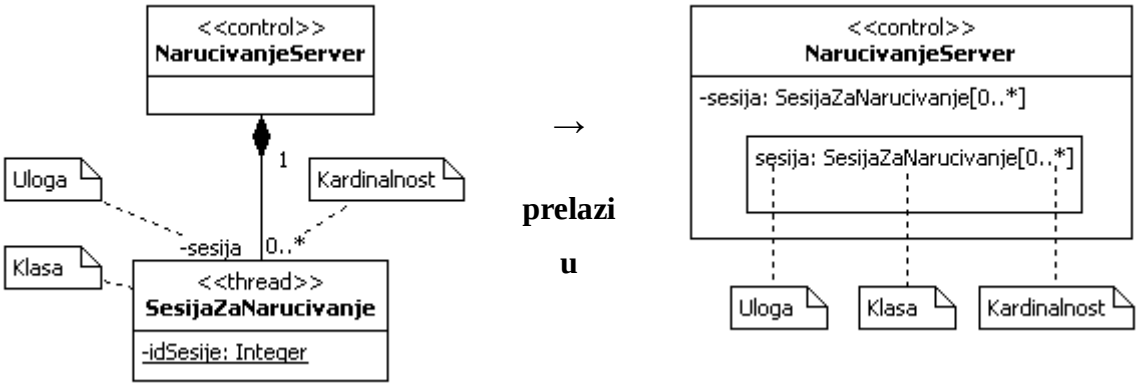
- Prikaz interne strukture pojedinih elemenata modela, kao što su klase, komponente ili slučajevi upotrebe, uključujući i interakcijske tačke pojedinih elemenata sa drugim dijelovima sistema.
- Prikaz kako se klase koriste u sistemu sa portovima.

- Prikaz kako kolekcija surađujućih instanci obavlja zadatak ili skup zadataka.
- Opis i grafički prikaz dizajn paterna.

16.2 Interna struktura

Dijagram klase koji smo već upoznali sadrži veze asocijacije i kompozicije. Dijagram složene strukture nudi alternativni način pokazivanja ovih relacija tako da se unutar klase mogu prikazati i njeni sastavni dijelovi.

Da bi bolje razumjeli dijagram složene strukture na slici 16.2 prikazan je dijagram složene strukture za dio dijagrama klase koji se odnosi na prikaz `NarucivanjeServer` klase i njene relacije sa klasom `SesijaZaNarucivanje`. Dijagram složene strukture ustvari predstavlja internu strukturu klase.



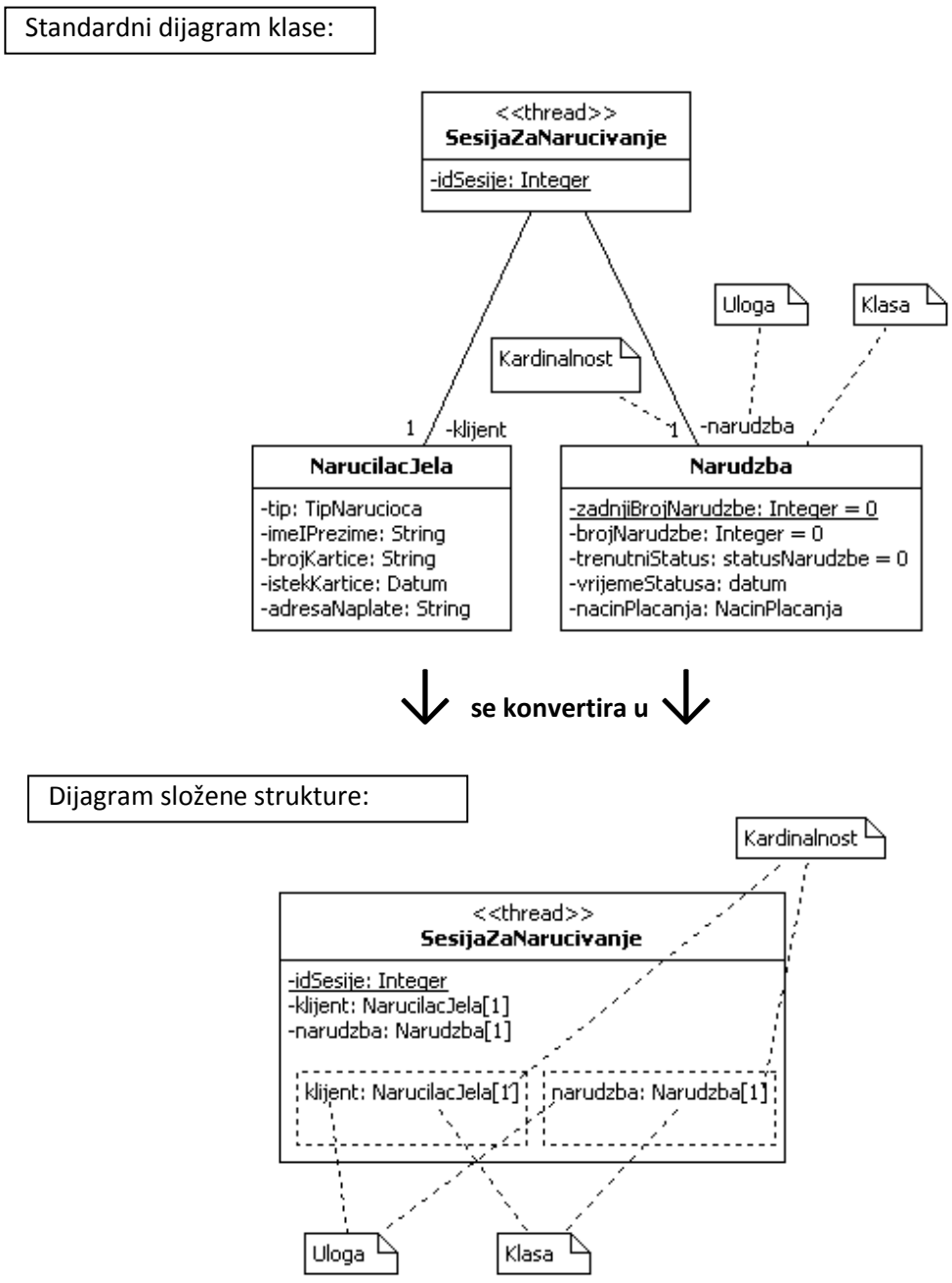
Slika 16.2: Dijagram klase i dijagram složene strukture za klasu `SesijaZaNarucivanje`

Interna struktura klase se prikazuju tako da se sve klase sa kojima je ta klasa u vezi kompozicije navedu unutar klase. Za klase koje čine internu strukturu neke klase navodi se njena uloga i tip u formatu *<ulogaIme>:<tip>*. Kardinalnost ili broj instanci dijelova se piše u gornjem desnom uglu samog dijela ili alternativno pored naziva uloge dijela u okviru srednjih zagrada.

Na slici 16.2 se vidi kako se konvertira dijagram klase u dijagram složene strukture sa naglaskom na ulogu `sesija`, kardinalnost 0..* i klasu `SesijazaNarucivanje`.

16.3 Osobine

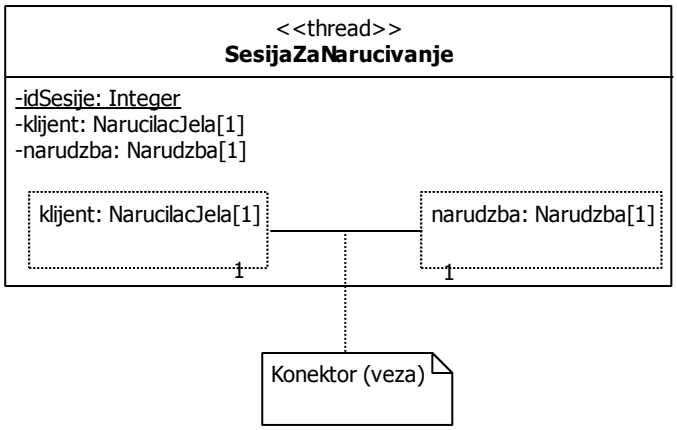
Pored pokazivanja dijelova, koji su u vezi kompozicije sa klasom čija se struktura prikazuje, mogu se prikazati i osobine koje su referencirane kroz asocijaciju. Osobine imaju istu notaciju označavanja kao i dijelovi s tim što se prikazuju u okviru od isprekidanih linija. Notacija i konvertiranje asocijacije iz standardnog dijagrama klase u dijagram interne strukture je prikazan na slici 16.3.



Slika 16.3: Interna struktura klase sa osobinama

16.4 Konektori

Slično kao i dijagram klase, dijagram složene strukture pokazuje dijelove i konektore. Dijelovi nisu uvijek model elementi i ne moraju predstavljati pojedinačne instance, oni mogu biti i uloge koju imaju ti elementi. Dijelovi se prikazuju na sličan način kao i objekti, samo ime nije podvučeno. Veze između dijelova mogu se prikazati i sa konektorima između dijelova. Konektor je veza koji omogućava komunikaciju između dijelova u izvršnom okruženju i predstavlja instancu asocijacija ili dinamički link uspostavljen prilikom izvršavanja sistema. Na svakom kraju konektora može se naznačiti i kardinalnost veze. Notacija označavanja kardinalnosti je ista kao notacija označavanja asocijacije na dijagramu klase.

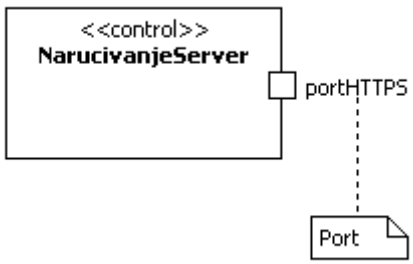


Slika 16.4: Interna struktura klase sa konektorom (vezom)

16.5 Port

Dok se interna struktura klase fokusira na sadržaj klase, portovi se fokusiraju na to kako se klasa koristi sa drugim klasama. Port je tačka interakcije između klasa i ostatka svijeta. Pokazuje različite načine korištenja klase, obično sa različitim tipovima klijenata.

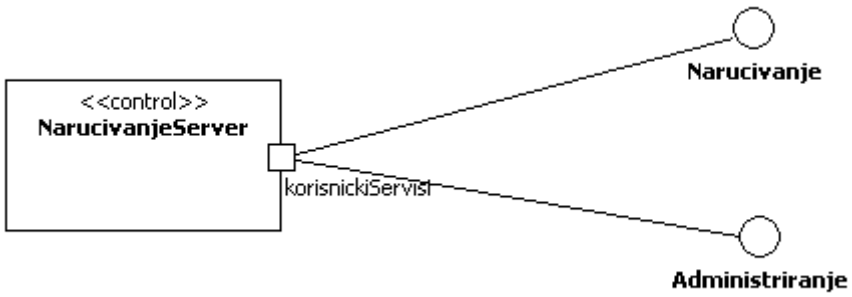
Na slici 16.5 je prikazana klasa `NarucivanjeServer` koja omogućava komuniciranje putem HTTPS protokola i to je jedan vid korištenja ove klase pa je stoga prikazan portom.



Slika 16.5: Port

Svako različito korištenje klase se predstavlja sa portom, koji se crta kao mali pravougaonik na granicama klase. Ime se piše uz port da bi se pokazala namjena porta.

Za klase je uobičajno da ima interfejse povezane sa portovima. Portovi se mogu koristiti i za grupiranje ovisnih interfejsa da bi se pokazao servis koji je omogućen na portu kao što je na slici 16.6.



Slika 16.6: Portovi u ulozi kao interfejsi

Završno razmatranje

Dijagram složene strukture prikazuje na kompaktniji način prvenstveno dijelove dijagrama klasa, iako se mogu koristiti za prikaz i komponenti ili čak i slučajeva upotrebe. U vezi sa klasama dijagrami složene strukture služe za dekompoziciju složenih klasa u hijerarhijske strukture, čime se dobiva na jednostavnosti modela. Koriste se i za prikazivanje suradnje i dizajniranje paterna. Njihova uloga u sklopu dizajniranja paterna će biti objašnjena u poglavlju 21.

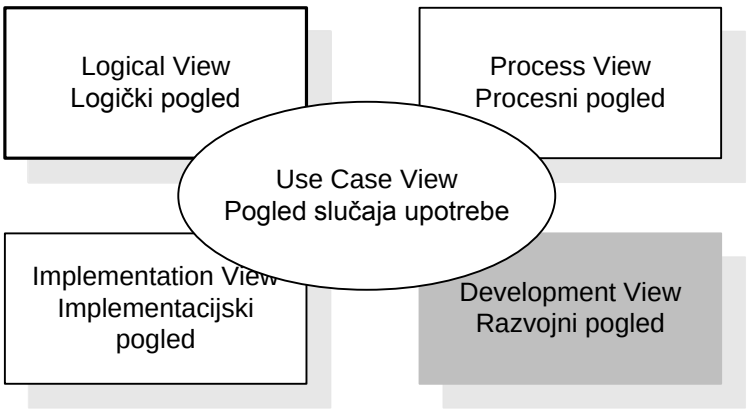
Pitanja za ponavljanje

1. Koja je osnovna namjena dijagrama složene strukture?
2. Kojem pogledu na sistem pripada dijagram složene strukture?
3. U odnosu na dijagram klase kako dijagram složene strukture prikazuje klase i veze?
4. Koja je notacija prikaza i imenovanja klase u okviru dijagrama složene strukture?
5. Kako se prikazuje kardinalnost u okviru dijagrama složene strukture?
6. Kako se prikazuju osobine koje su referencirane kroz asocijaciju?
7. Kako dijagram složene strukture pokazuje dijelove i konektore?
8. Kako se prikazuje port koji ima ulogu interfejsa?

POGLAVLJE 17.

DIJAGRAM KOMPONENTI

UML dijagram komponenti (eng. component diagram) modelira komponente sistema i na taj način formira dio razvojnog pogleda, kao što vidimo na slici 17.1. Komponente mogu biti individualne klase ili mogu predstavljati kolekciju klasa. Razvojni pogled opisuje kako su dijelovi sistema organizirani u module i komponente, te pomaže u upravljanju nivoima arhitekture sistema.



Slika 17.1: Razvojni prikaz opisuje organiziranje dijelova sistema u module i komponente

17.1 Namjena dijagrama komponenti

Pri razvoju softverskih sistema postoje dvije mogućnosti, između mnogih drugih, koje se mogu koristiti da se smanje rizici povezani sa procesom razvoja softvera.

To su:

- Reduciranje i jasno definiranje ovisnosti između softverskih komponenti aplikacije.
- Ponovno korištenje komponenti čije su funkcionalnosti već poznate i testirane.

Dijagrami komponenti unutar procesa modeliranja:

- Koristi se za modeliranje planiranih softverskih komponenti i interfejsa između njih na početku projekta čime se omogućava da se razmatranje detaljnog dizajna ostavi za bolje vrijeme.
- Koristi se za skrivanje specifikacije detalja komponenti, a fokusiraju se na međusobne relacije komponenti koje se specificiraju kao interfejsi između komponenti.
- Koriste se da pokažu kako se prethodno dokazane komponente integriraju u tekući dizajn sistema.

17.2 Komponenta

Komponenta je enkapsulirani, ponovno upotrebljivi i zamjenjivi dio nekog sistema. Možemo smatrati komponente kao gradivne elemente sistema, koje se kombiniraju tako da se uklapaju međusobno i da formiraju sistem. Komponente mogu biti različite veličine, od relativno malih, veličine prosječne klase do velikih podsistema.

Dobri kandidati za komponente su dijelovi koji izvršavaju ključne funkcionalnosti i koji se često koriste u sistemu. Softver kao što je XML parseri, online šoping korpe i slično su često korištene komponente. Ove komponente su nam ujedno i primjeri komponenti trećih lica koje možemo preuzeti bez vlastitog kreiranja.

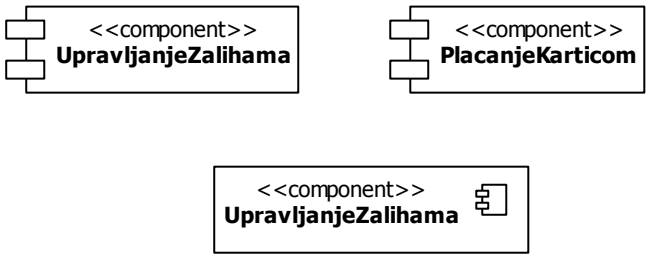
U sistemu koji razvijamo možemo kreirati komponente koje pružaju servise ili pristupaju podacima. Primjer za to je `e-restoran` koji može imati komponentu za naručivanje, komponentu za upravljanje zalihama, komponentu za plaćanje kreditnom karticom. UML komponente mogu imati isto ponašanje kao klase, biti u relacijama generalizacije i asocijacije sa drugim klasama i komponentama, implementirati interfejse, imati operacije. One, također, mogu imati i portove te pokazivati internu strukturu. Osnovna razlika između klase i komponente je ta da komponente uglavnom imaju veću odgovornost od klasa. Na primjer, možemo kreirati klasu za korisničke informacije koja sadrži informacije o kontakt podacima korisnika kao što su ime i e-mail adresa, te komponentu za korisničke informacije koja

dozvoljava kreiranje korisničkih računa i provjeru autentičnosti. Također je za komponentu uobičajeno da sadrži i koristi druge klase i komponente da bi izvršila svoj zadatak.

Kako komponente predstavljaju osnovne elemente dizajna softvera, bitno je da su one slabo povezane tako da se promjena na komponenti ne odražava na ostatak sistema. Da bismo ostvarili takvu enkapsulaciju, komponentama bi se trebalo pristupati preko interfejsa. Interfejse koristimo da bismo odvojili ponašanje od implementacije. Dozvoljavajući komponentama da pristupaju jedne drugim preko interfejsa smanjujemo mogućnost da promjena jedne komponente uzrokuje pad sistema.

17.3 Komponenta u UML-u

Komponenta se uglavnom predstavlja kao tabovani pravougaonik, unutar kojeg je naziv komponente. Naziv komponente je unutar simbola komponente i treba biti deskriptivan i odgovarati domeni problema. Pravila imenovanja komponenti su u velikoj mjeri slična pravilima imenovanja klasa. Postoji više varijacija predstavljanja komponenti, neke od njih koriste <<component>> stereotip. Slika 17.2 prikazuje neke od mogućih komponenti u sistemu `e-restoran` te varijacije označavanja komponenti. Odabrana notacija treba u cijelom modelu biti konzistentna.



Slika 17.2: Simboli za komponentu i varijacije označavanja komponenti

Ako želimo pokazati da je naša komponenta zapravo podsistem veoma velikog sistema, tada umjesto <<component>> može se pisati <<subsystem>>, kao što pokazuje slika 17.3. Podsistem je sekundarni sistem koji je dio većeg sistema. UML posmatra podsistem kao posebnu vrstu komponente i fleksibilan je po pitanju kako koristiti ovaj stereotip, ali najbolje je da se rezervira samo za najveće dijelove cjelokupnog sistema.

Slika 17.3: <<subsystem>> stereotip se koristi da pokaže najveće dijelove sistema

17.4 Ponuđeni i zahtijevani interfejsi komponenti

Komponente trebaju biti slabo povezane da bismo ih mogli mijenjati bez promjene drugih komponenti. Da bismo ovo ostvarili koristimo interfejse. Komponente komuniciraju međusobno preko ponuđenih i zahtijevanih interfejsa. To omogućava kontroliranje ovisnosti između komponenti i čini komponente zamjenjivim.

Ponuđeni interfejs komponente je interfejs koji komponenta realizira. Druge komponente i klase komuniciraju sa komponentom kroz njene ponuđene interfejse. Ponuđeni interfejs komponente zapravo pokazuje koje servise komponenta nudi.

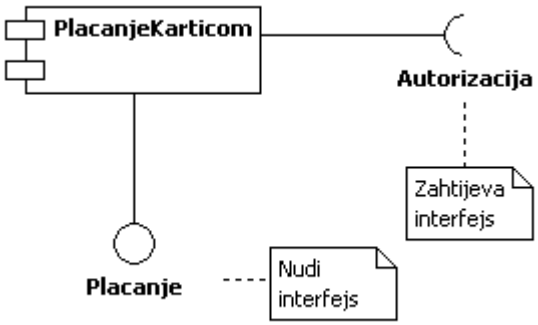
Zahtijevani interfejs komponente je interfejs koji komponenta treba da bi funkcionirala. Preciznije rečeno, komponenta za svoje funkcioniranje može trebati drugu klasu ili komponente. Ali da bi se ostvario cilj slabog vezivanja komponenti, to se postiže kroz zahtijevani interfejs, a ne direktno korištenje klase od strane naše komponente. Zahtijevani interfejs dakle predstavlja servise koji su potrebni komponenti od strane drugih komponenti ili klasa.

Postoje tri standardna načina da pokažemo zahtijevane i ponuđene interfejse komponente u UML-u: krug i polukrug (eng. ball i socket), notacija stereotipa i tekst listing.

Krug i polukrug notacija za interfejse

Zahtijevani interfejs komponente u ovoj notaciji pokazujemo sa simbolom polukruga, a ponuđeni interfejs simbolom kruga. U blizini simbola se piše ime interfejsa.

Slika 17.4 pokazuje da komponenta `PlacanjeKarticom` nudi interfejs `Placanje` i zahtjeva interfejs `Autorizacija`.

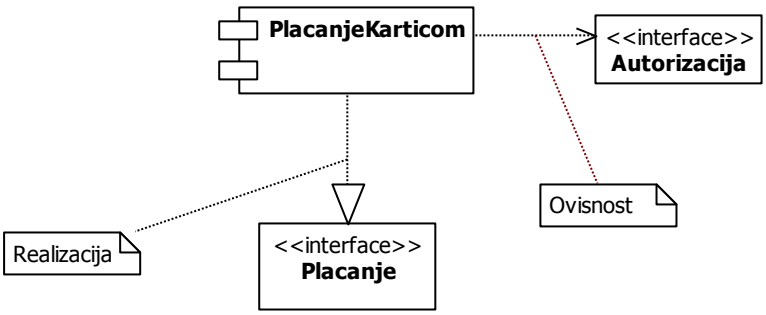


Slika 17.4: Krug i polukrug notacija za ponuđeni i zahtijevani interfejs komponenti

Ova notacija je najčešći način prikazivanja interfejsa komponenti.

Notacija stereotipa za interfejse

Ponuđene i zahtijevane interfejse komponenti možemo predstaviti kao klase sa stereotipom <<interface>>. Ako komponenta nudi interfejs, tada je potrebno nacrtati strelicu realizacije od komponente prema interfejsu. Ako komponenta zahtjeva interfejs, tada crtamo strelicu ovisnosti od komponente prema interfejsu. Slika 17.5 prikazuje upotrebu ove notacije.

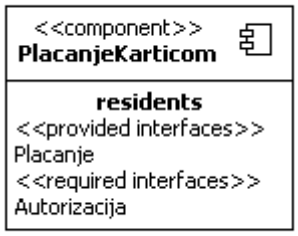


Slika 17.5: Notacija stereotipa koja pokazuje zahtijevane i ponuđene interfejse

Ova notacija je korisna ukoliko želimo pokazati i operacije interfejsa. Ako nemamo potrebu za tim, tada je najbolje koristiti krug i polukrug notaciju, jer se iste informacije pokazuju kompaktnije.

Listing interfejsa komponente

Najkompaktniji način prikazivanja zahtijevanih i ponuđenih interfejsa komponente jeste popis odnosno listing istih unutar komponente. Zahtijevani i ponuđeni interfejsi se popisuju odvojeno, kao što vidimo na slici 17.6.



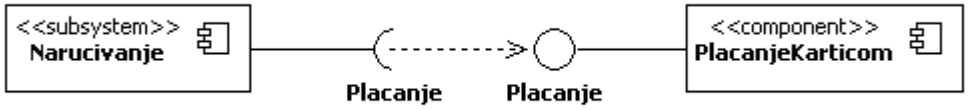
Slika 17.6: Popis zahtijevanih i ponuđenih interfejsa unutar komponente

Ovdje se mogu pobrojati svi elementi komponente, uključujući i fizičke fajlove (<<artifacts>>), koji implementiraju ovu komponentu, a o kojima će više riječi biti u dijagramima raspoređivanja.

Odluka o tome koju notaciju koristiti ovisi o tome što želimo prikazati.

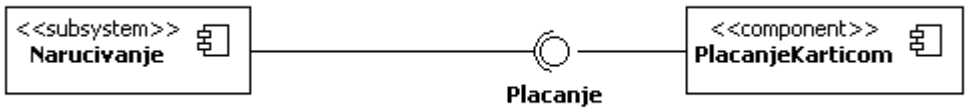
17.5 Prikazivanje komponenti koje rade zajedno

Ako komponenta ima zahtijevane interfejse, tada ona treba drugu klasu ili komponentu unutar sistema da bi funkcionirala. Da bismo pokazali da komponenta sa zahtijevanim interfejsom ovisi o drugoj komponenti ili klasi koja ga nudi, crta se strelica ovisnosti od simbola polukruga ovisne klase do simbola kruga komponente koja pruža taj interfejs, kao što je prikazano na slici 17.7.



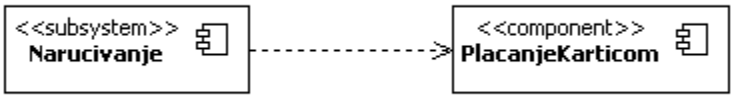
Slika 17.7: Podsistem `Narucivanje` zahtjeva interfejs `Placanje` a komponenta `PlacanjeKarticom` nudi taj interfejs

UML dopušta i spajanje simbola kruga i polukruga zajedno kao što pokazuje slika 17.8.



Slika 17.8: Opcija prezentacije koja spaja krug i polukrug zajedno

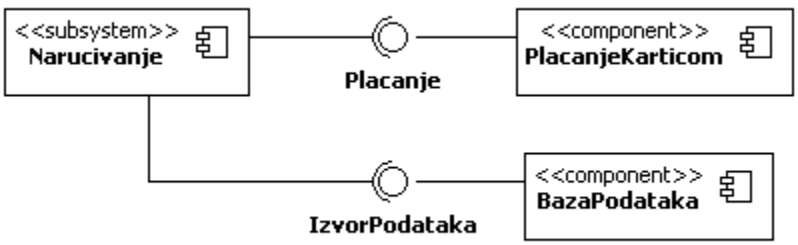
Također mogu se izbjeći interfejsi i ovisnost prikazati direktno između komponenti kao što pokazuje slika 17.9.



Slika 17.9: Prikaz ovisnosti na višem nivou prikazujući samo ovisne komponente bez prikaza konkretnih interfejsa

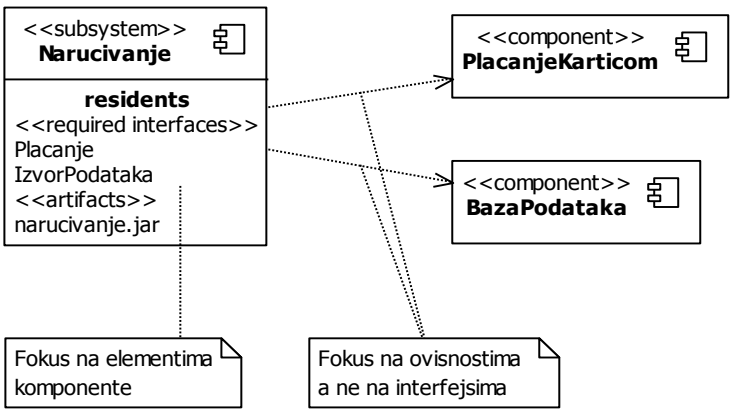
Ova zadnja notacija koja isključuje prikaz interfejsa je jednostavnija, ali ipak pri odluci koju koristiti treba razmisliti o nekoliko faktora.

Podsjetimo se da interfejsi omogućavaju da komponente ostanu slabo vezane, pa predstavljaju važan faktor u arhitekturi komponenti. Prikazujući ključne komponente sistema i njihove ovisnosti kroz pripadajuće interfejse je odličan način za opisivanje arhitekture sistema, te je zbog toga prva notacija dobra, kao što možemo vidjeti na slici 17.10.



Slika 17.10: Fokusiranje na ključne komponente i interfejse

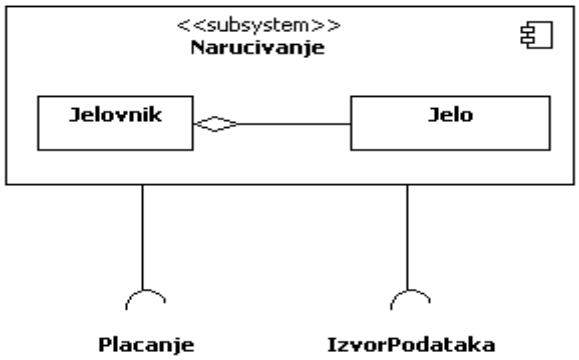
Druga notacija je dobra za pojednostavljeno prikazivanje ovisnosti među komponentama sistema na višem nivou. Ova notacija može biti korisna za razumijevanje konfiguracije ili razvoja jer isticanje ovisnosti među komponentama i popis manifestirajućih artefakta dopušta da vidimo koje komponente i prateći fajlovi su nam potrebni za vrijeme razvoja, kao što vidimo na slici 17.11.



Slika 17.11: Fokusiranje na ovisnost među komponentama i artefaktima

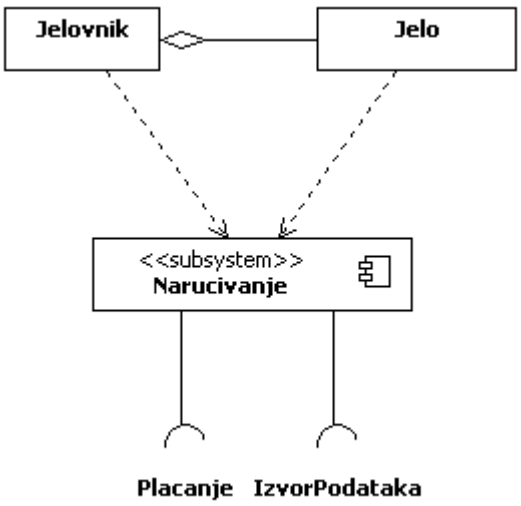
17.6 Klase koje realiziraju komponente

Komponenta često sadrži ili koristi druge klase da bi implementirala svoju funkcionalnost. Za takve klase kažemo da realiziraju komponente. Realizirajuće klase zajedno sa njihovim relacijama možemo prikazati unutar komponente. Slika 17.12 pokazuje da komponenta `Narucivanje` sadrži klase `Jelovnik` i `Jelo`. Također prikazuje relaciju agregacije među ovim klasama.



Slika 17.12: `Jelovnik` i `Jelo` klase učestvuju u realizaciji komponente `Narucivanje`

Realizirajuće klase možemo pokazati i njihovim crtanjem izvan komponente sa strelicom ovisnosti od pojedine realizirajuće klase prema komponenti, kao što vidimo na slici 17.13.



Slika 17.13: Alternativni pogled, realizirajuće klase izvan komponente uz relacije ovisnosti

Posljednji način za prikazivanje ovih klasa je da se navedu u <<realizations>> dijelu unutar komponente, kao što je prikazano na slici 17.14.

Slika 17.14: Popis realizirajućih klasa unutar <<realizations>> dijela komponente

Opet se postavlja pitanje kako se odlučiti za jednu od notacija. Ponekad je UML alat koji se koristi sam ograničen i nameće određenu notaciju, ali ako nema takvih ograničenja, tada možemo slijediti iskusne moderatore koji najčešće koriste prvu notaciju.

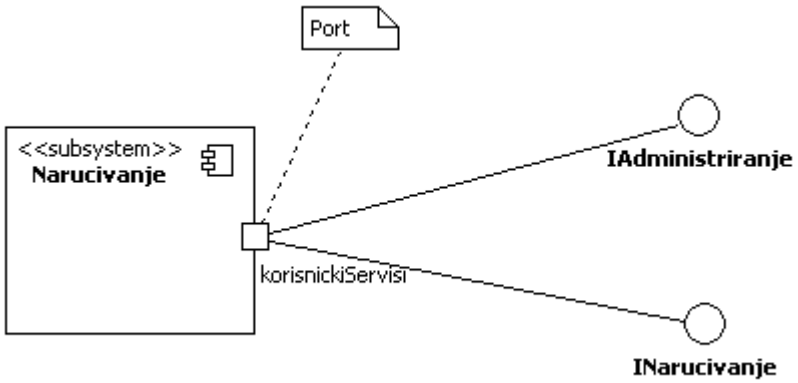
Prednost prve dvije notacije u odnosu na treću koja je najkompaktnija je što one pokazuju relacije među realizirajućim klasama.

17.7 Portovi i interna struktura

Interna struktura pokazuje dijelove koje sadrži klasa i relacije među dijelovima, što omogućava prikazivanje relacija u ovisnosti od samog konteksta, ili relacija koje su sadržane unutar konteksta pripadajućih klasa. Portovi pokazuju kako se klasa koristi na sistemu sa portovima.

Portovi

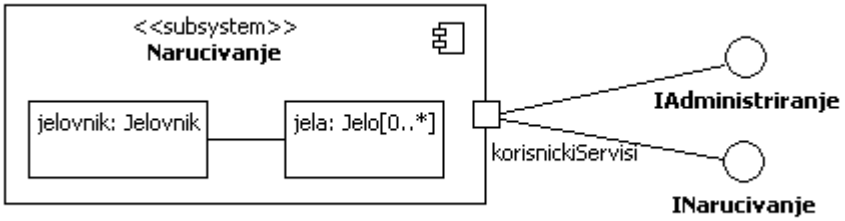
Komponente također mogu imati portove i interne strukture. Portove možemo koristiti da bismo modelirali različite načine na koje komponenta može biti korištena sa povezanim interfejsom na port. Na slici 17.15 je prikazana komponenta `Narucivanja` koja ima port `korisnisnickiServisi` sa dodijeljenim interfejsima.



Slika 17.15: Port prikazuje jednu od mogućih upotreba komponente i interfejsa

Interna struktura

Internu strukturu komponenti koristimo za modeliranje njenih dijelova, osobina i konektora. Slika 17.16 prikazuje dio interne strukture komponente `Narucivanje`.



Slika 17.16: Interna struktura komponente

Komponente imaju svoje jedinstvene elemente pri prikazivanju portova i interne strukture, a to su delegacijski konektori i konektori spajanja. Konektori se koriste za prikaz kako se interfejsi komponenti slažu sa internim dijelovima i kako interni dijelovi rade skupa.

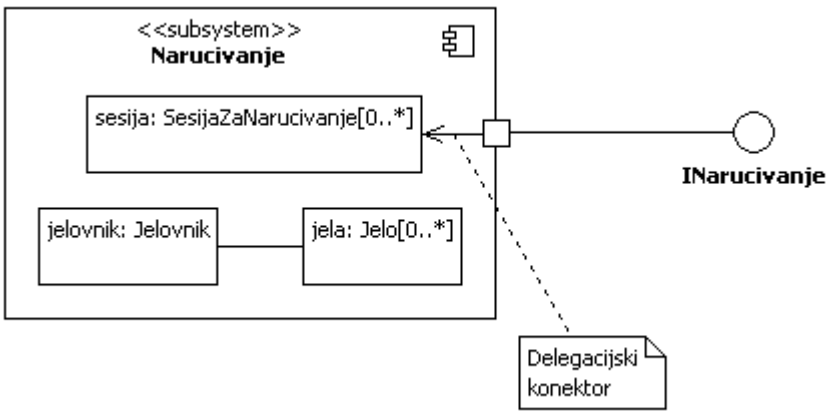
17.8 Konektori

Razlikujemo dvije vrste konektora: delegacijski konektori i konektori spajanja.

Delegacijski konektori

Ponuđeni interfejsi komponente mogu biti realizirani preko nekog od internih dijelova, a zahtijevani interfejsi su zahtijevani isto tako od nekog dijela druge komponente. U ovakvim slučajevima možemo koristiti delegacijske konektore da bismo prikazali da interni dijelovi realiziraju ili koriste interfejse komponente. Delegacijski konektori se prikazuju sa strelicom koja pokazuje smjer "saobraćaja", i spaja port koji je povezan na interfejs sa internim dijelom.

Ako dio komponente koristi zahtijevani interfejs, tada je strelica usmjerena od internog dijela do porta. Slika 17.17 pokazuje upotrebu delegacijskih konektora za spajanje interfejsa sa internim dijelovima.



Slika 17.17: Upotreba delegacijskih konektora

Na slici delegacijski konektori pokazuju kako interfejsi odgovaraju internim dijelovima i da klasa `SesijaZaNarucivanje` realizira `INarucivanje` interfejs.

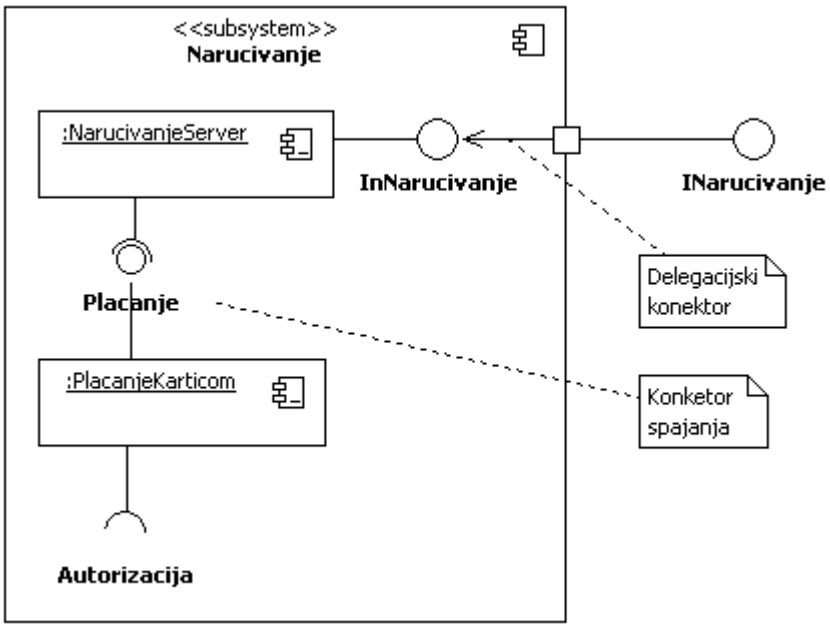
Delegacijski konektori mogu se posmatrati na sljedeći način: port predstavlja otvor prema komponenti kroz koji se obavlja komunikacija, a delegacijski konektori prikazuju smjer komunikacije. Dakle delegacijski konektor usmjeren od porta ka internom dijelu prikazuje da se poruke prosljeđuju izvana, dijelu kojem su potrebne.

Konektori spajanja

Konektori spajanja pokazuju kako komponente surađuju kroz interfejse. Konektori spajanja ustvari pokazuju da komponenta zahtijeva interfejs koji realizira druga komponenta.

Konektor spajanja je predstavljen spojenim simbolima kruga i polukruga da bi prikazao zahtijevani i ponuđeni interfejs. Na slici 17.18 je prikazana notacija konektora spajanja koji spaja komponente `NarucivanjeServer` i `PlacanjeKarticom` koje su u okviru komponente `Narucivanje`. `NarucivanjeServer` i `PlacanjeKarticom` koriste notaciju kojom je specificirano da su instance komponenti. Komponenta `Narucivanje` nudi interfejs `INarucivanje`, ali njega zapravo realizira interna komponenta `NarucivanjeServer`.

Konektori spajanja su specijalne vrste konektora koje se koriste pri pokazivanju složene strukture komponenti.



Slika 17.18: Konektor spajanja

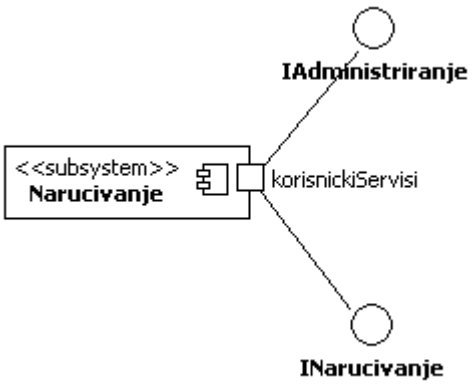
Na slici 17.18. je prikazana još jedna uloga delegacijskog konektora a to je da mogu spajati i interfejse internih dijelova sa portovima.

17.9 Pogled na komponente: crna kutija i bijela kutija

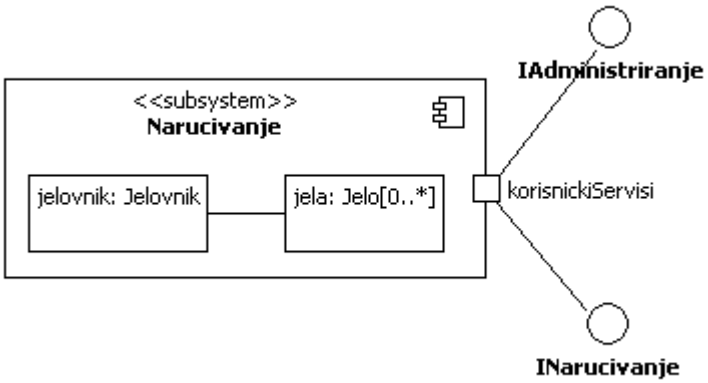
UML omogućava dva pogleda na komponente, pogled kroz crnu kutiju i pogled kroz bijelu kutiju. Pogled kroz crnu kutiju pokazuje kako komponenta izgleda izvana, uključujući zahtijevane i ponuđene interfejse komponenti, te odnos komponente prema drugim komponentama. Pogled kroz crnu kutiju ne specificira ništa vezano za internu implementaciju komponente. Pogled kroz bijelu kutiju pokazuje koje klase, komponente i interfejsi pomažu komponenti da izvršava svoje funkcionalnosti.

U ovom poglavlju je korišten i jedan i drugi pogled na komponente. Razlika između ova dva pogleda prikazana je na slici 17.19, gdje se uočava da bijela kutija pokazuje i interne dijelove komponente, za razliku od crne kutije.

Crna kutija



Bijela kutija



Slika 17.19: Crna i bijela kutija

Pogled kroz crnu kutiju je dobar za prikaz krupne slike komponenti sistema, dok se pogled kroz bijelu kutiju fokusira na unutarnji rad komponenti.

Pri modeliranju sistema, prikaz kroz crnu kutiju je dobro koristiti za fokusiranje na arhitekturu sistema na visokom nivou. Crna kutija je dobra za prikaz ključnih komponenti u sistemu i njihove relacije. Bijele kutije su s druge strane dobre da bismo vidjeli kako komponenta postiže svoju funkcionalnosti kroz klase koje koristi.

Prikaz kroz crnu kutiju obično sadrži više komponenti, dok prikaz kroz bijelu kutiju fokusira na sadržaj jedne komponente. U primjeru na slici 17.9 vidimo da se u prikazu kroz crnu kutiju komponente `Narucivanje`, vide samo interfejsi `IAdministracija` i `INarucivanje`, dok se u prikazu kroz bijelu kutiju vide i interni dijelovi komponente: `jelovnik` i `jela`. Naravno ovdje je dat pojednostavljen prikaz bez svih internih dijelova.

Završno razmatranje

Dijagrami komponenti osiguravaju odličnu mogućnost dokumentiranja sistema od samog početka razvoja odnosno od nekog prvog modela do zadnjeg modela sistema. Prvi model sistema je sačinjen od velikih dijelova komponenti koje su planirane i omogućavaju samo razumijevanje sistema. Zadnji pogled na sistem pomoću ovog dijagrama skriva dijelove implementacije, a pokazuje funkcionalne komponente i interfejs između komponenti.

Dijagrami komponenti su strogo povezani sa dijagramima klase i dijagramima složene strukture, na početku i kraju razvojnog procesa. Usko su povezani i sa dijagramima raspoređivanja koji će biti objašnjeni u poglavlju 19. i koji će pokazati kako se komponente raspoređuju u okviru računarske platforme.

Pitanja za ponavljanja

1.Koja je osnovna namjena dijagrama komponenti?
2.Kojem pogledu pripada dijagram komponenti?
3. Što se podrazumijeva pod komponentom?
4.Koje su različite UML notacije za komponentu?
5.Kada se koristi stereotip <<subsystem>> ?
6.Kako se interfejsi koriste u dijagramima komponenti?
7.Što je ponuđeni interfejs komponenti?
8.Što je zahtijevani interfejs komponenti?
9.Navesti tri standardna načina za prikaz zahtijevanog i ponuđenog interfejsa komponenti.
10.Navesti UML notaciju prikaza da komponente rade zajedno?
11.Koji su načini prikaza klasi koje realiziraju komponentu?
12. Koja je namjena portova u dijagramima komponenti?
13.Kako se portovi uvezuju sa internim elementima komponente?
14.Što su i koja je UML notacija za delegacijski konektor?
15.Što su i koja je UML notacija za konektore spajanja?
16. Koja dva pogleda na komponente omogućava UML?

17. Što pokazuje pogled kroz crnu kutiju na komponente?

18.Što pokazuje pogled kroz bijelu kutiju na komponente?

19.U kojim fazama projekta se koriste dijagrami komponenti?

20.Koja je veza dijagrama komponenti i ostalih dijagrama?

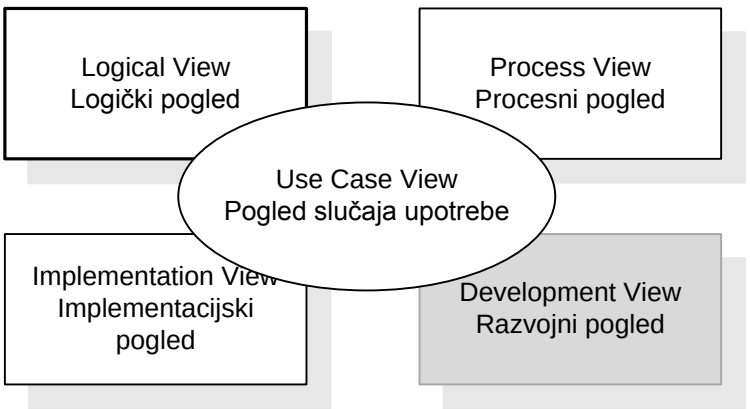
POGLAVLJE 18.

DIJAGRAM PAKETA

Kompleksni softverski sistemi mogu sadržavati stotine klasa. Tu se postavlja pitanje kako će se organizirati takav sistem. Jedan način strukturiranja je organiziranje klasa u logički povezane grupe. Klase koje se vežu za korisnički interfejs možemo smjestiti u jednu grupu, a uslužne klase u drugu grupu.

U UML-u, grupe klâsa se modeliraju sa paketima. Većina objektno orijentiranih jezika imaju analogiju UML paketima da bi organizirali i izbjegli kolizije s imenima među klasama. Na primjer, Java ima pakete, a C++ i C# ima imenovane prostore.

Dijagrami paketa (eng. package diagram) često se koriste da bi se pokazala ovisnost među paketima. Razumijevane ovisnosti među paketima je ključna za stabilnost softvera zato što u slučaju promjene jednog paketa može doći do raskida paketa koji o njemu ovisi. Paketima se može organizirati većina UML elemenata, a ne samo klase. Na primjer, pakete često koristimo za grupiranje slučajeve upotrebe. Dijagrami paketa formiraju razvojni pogled na sistem, koji se bavi time kako su dijelovi sistema organizirani u module i pakete.



Slika 18.1: Dijagram paketa pripada razvojnom pogledu

18.1 Namjena dijagrama paketa

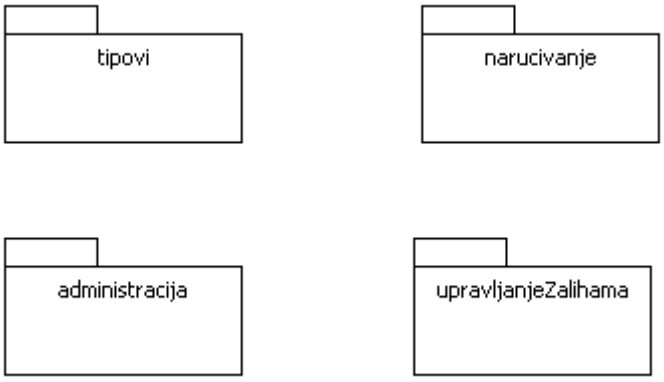
Sumarno, namjena dijagrama paketa je:

- Prikaz pregleda zahtjeva na visokom nivou.
- Prikaz dizajna na visokom nivou.
- Logička modularna organizacija kompleksnih dijagrama.
- Organizacija izvornog kôda.

18.2 Paketi

Simbol paketa (eng. package) ima izgled pravougaonika sa tabom, što je prikazano na slici 18.2. Ime paketa je napisano unutar pravougaonika.

Prilikom dizajna `e-restoran` sistema, klase se organiziraju u pakete na osnovu njihove srodnosti imajući u vidu aspekt poslovnih procesa ili aspekt razvoja softvera. U skladu s tim klase koje definiraju složene tipove podataka kao što su `Datum`, `StatusNarudzbe` možemo grupirati u jedan paket koji se naziva `tipovi`. S druge strane, klase koje su vezane za proces naručivanja, kao što su `Narudzba`, `Jelovnik`, `Jelo` možemo grupirati u paket `narucivanje`.

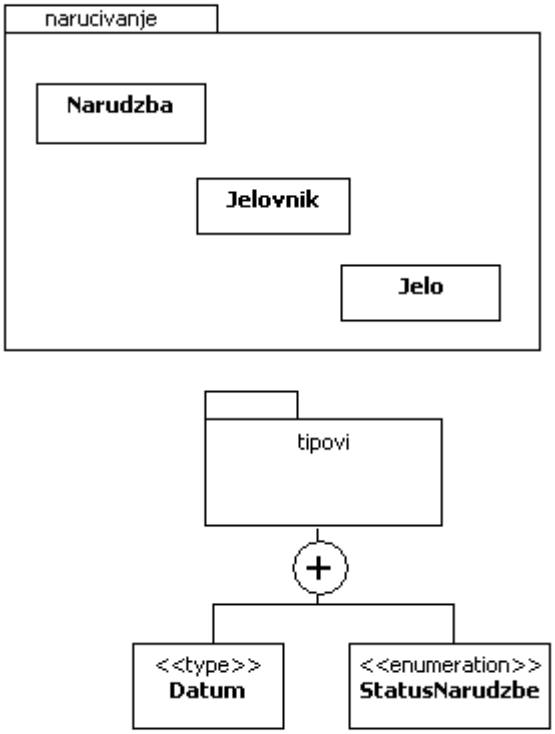


Slika 18.2: Notacija paketa

Na slici su prikazani paketi `e-restorana`. Svaki paket predstavlja određenu grupu funkcionalnosti.

18.3 Sadržaj paketa

Paketi organiziraju UML elemente kao što su klase. Sadržaj paketa može biti prikazan unutar paketa ili izvan paketa. U slučaju prikazivanja sadržaja izvan paketa taj sadržaj se povezuje linijom sa okruženim znakom plus sa paketom. Obje notacije se vide na slici 18.3. Ako crtamo elemente unutar paketa, imena paketa se pišu u tabu foldera.



Slika 18.3: Dva načina prikazivanja sadržaja paketa

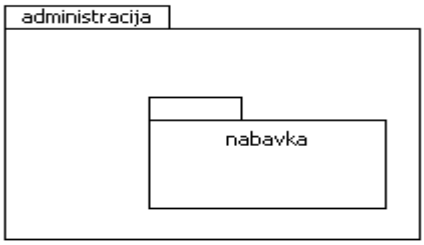
Razmotrimo programski kôd koji odgovara paketu `narucivanje`. U Javi, ključna riječ *package* na početku klase specificira da je klasa u paketu, a u C++ ključna riječ *namespace* označava da je klasa u paketu.

```
Java :  package narucivanje;

        public class Narudzba
        {
               ...
        }
        ...
```

```
C++ :   namespace narucivanje
        {
            class Narudzba
            {
               ...
            };
        }
```

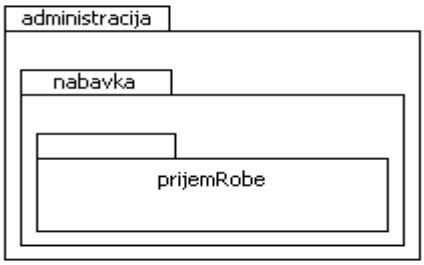
Klasa `Narudzba` je locirana u paketu `narucivanje`

Paketi također mogu sadržavati i druge pakete, kao što je pokazano na slici 18.4.



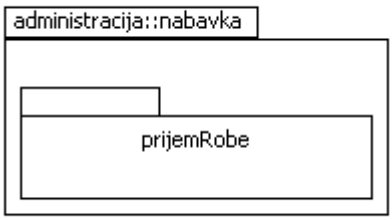
Slika 18.4: Paket koji sadrži drugi paket

Imena paketa trebaju biti jednostavna i deskriptivna, koja u slučaju ugniježđavanja veoma jasno daju do znanja šta je sadržano u paketu. Imena su tipa: `administracija`, `nabavka`, `prijemRobe`. Sve što se smješta u paket mora imati smisla u odnosu na ostatak sadržaja paketa odnosno sadržaj paketa mora biti kohezivan. Dobar test da li je paket kohezivan jeste da li se može paketu dati kratko, opisno ime. Ako se ne može dati takvo ime, onda je u paketu sigurno više nepovezanih stvari.



Slika 18.5: Duboko ugniježđeni paketi su vrlo česti u velikim aplikacijama

Ovako prikazani paketi zauzimaju puno mjesta i ako želimo da pokažemo klase unutar paketa `prijemRobe`, svaki paket koji sadrži tu klasu bi rastao u prikazu. Zbog toga postoji druga notacija označavanja ugniježđenih paketa. Ugniježđeni paketi se mogu prikazati sa notacijom *paketA:: paketB:: paketC*. To je prikazano na slici 18.6.



Slika 18.6: Skraćena notacija ugniježđenih paketa

Varijacije prikaza

Većina alata može prikazati da klasa pripada paketu pomoću jedne od notacija na slici 18.7. Prva notacija na slici je nama već poznata i predstavlja standardnu UML notaciju. Da bi se specificiralo da klasa pripada paketu većina UML alata dozvoljava da se unese ime paketa pri specifikaciji klase ili da se klasa prevuče u paket kojem pripada.

Slika 18.7: Uobičajeni prikazi da klasa pripada paketu

18.4 Organizacija klasa u pakete

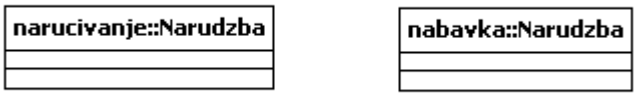
Klase u istoj hijerarhiji nasljeđivanja tipično pripadaju istom paketu. Klase koje su vezane agregacijom ili kompozicijom također pripadaju istom paketu. Također klase koje su u kolaboraciji jedna sa drugom često pripadaju istom paketu.

UML uspostavlja logiku paketa i imenovanih prostora, tako da ako element unutar paketa želi da koristi element iz drugog paketa mora da specificira gdje je taj element lociran.

Da bi se specificirao kontekst UML elementa treba se dati puno ime elementa, koje uključuje ime paketa i ime elementa odvojeno duplim dvotačkama kao, *imePaketa::imeKlase*.

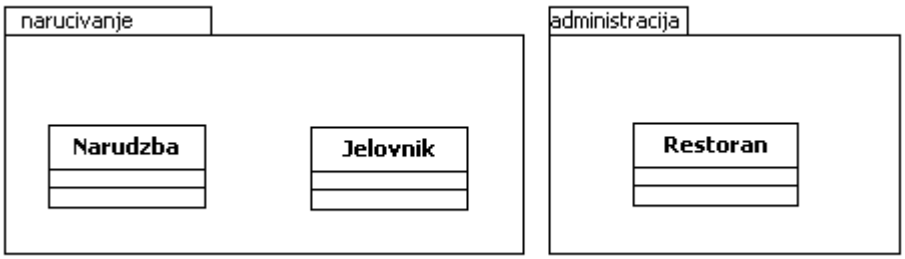
Puno ime za klasu `Narudzba` koja pripada paketu `narucivanje` je `narucivanje::Narudzba`.

Ako postoje dvije klase istog imena u različitim paketima, korištenje punog imena osigurava razlikovanje. Elementi unutar jednog paketa moraju imati jedinstvena imena. Ovo znači da paket `narucivanje` ne može imati dvije klase sa imenom `Narudzba`, ali da dvije klase sa istim imenima mogu pripadati različitim paketima.



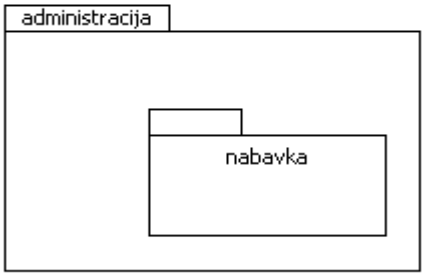
Slika 18.8: Prikaz klase sa potpunim imenom: oba paketa imaju klasu sa istim imenom

Klase u istom paketu mogu pozivati jedna drugu bez navođenja potpunog imena. Kako su klase `Narudzba` i `Jelovnik` unutar istog paketa, `Narudzba` može imati atribut tipa `Jelovnik` bez navođenja imena paketa. S druge strane klasa izvan paketa `narucivanje` kao što je `Restoran` mora definirati potpuno ime pri pristupu klasi `Jelovnik`.



Slika 18.9: Klase iz različitih paketa moraju navoditi potpuno ime pri pristupu jedna drugoj

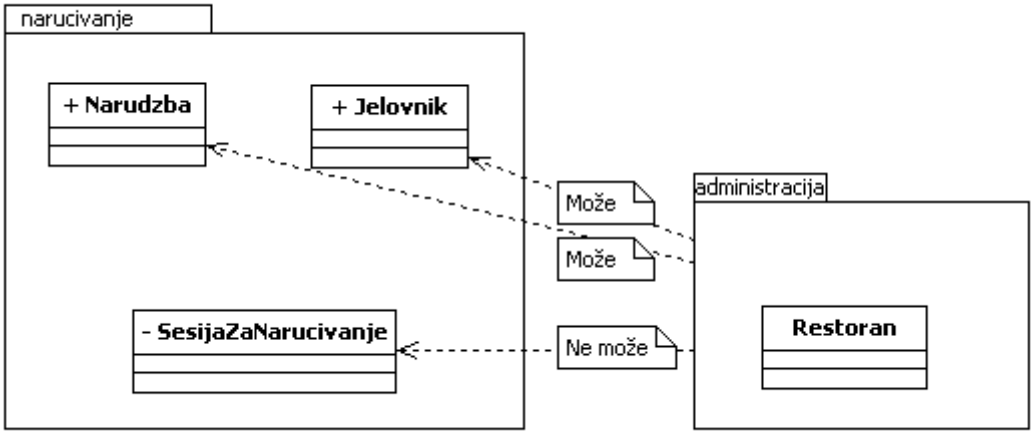
Ako su u pitanju ugniježđeni paketi, tada elementi u paketima mogu pozivati elemente u ugniježđenim paketima unutar njih bez navođenja potpunog imena. Na slici 18.10 znači da element u paketu `administracija` može pristupiti elementu u `nabavka` paketu bez navođenja potpunog imena.



Slika 18.10: Ugniježđeni paketi

18.5 Vidljivost elemenata

Elementi u paketu mogu imati javnu ili privatnu vidljivost. Elementi sa javnom vidljivošću su dostupni i izvan paketa. Elementi sa privatnom vidljivošću su dostupni samo drugim elementima iz paketa. Privatnu vidljivost u UML-u modeliramo simbolom - a javnu simbolom +. Simboli vidljivosti se pišu ispred imena elementa, što se vidi na slici 18.11.



Slika 18.11: Vidljivost paketa

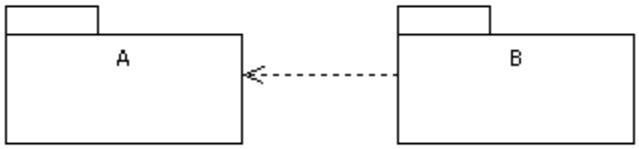
Na slici se vidi da klasa `SesijaZaNarucivanje` ima privatnu vidljivost i nije dostupna izvan paketa `narucivanje`, dok klase `Narudzbe` i `Jelovnik` imaju javnu vidljivost i dostupne su i ostalim elementima modela koji su izvan paketa. Ukoliko vidljivost elementa nije označena podrazumijeva se da imaju javnu vidljivost.

Mnogi UML alati ne nude simbole + i – za označavanje vidljivosti, ali je moguće napisati spomenute simbole pored naziva klase, ukoliko se na osnovu toga ne generira kôd. Neki alati

omogućavaju da se specificira vidljivost, ali je ne prikazuju na slici, nego se to samo koristi pri generiranju kôda.

18.6 Ovisnost paketa

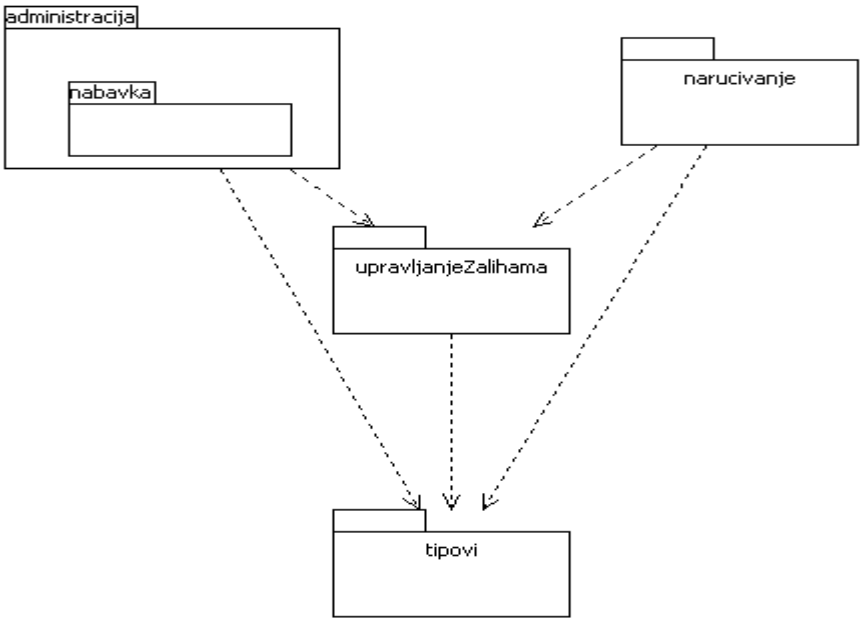
Prethodna razmatranja pokazuju da ponekad klasa iz jednog paketa koristi klasu iz drugog paketa. Ovo dovodi do ovisnosti među paketima: ako element iz paketa B koristi element iz paketa A, tada paket B ovisi od paketa A, kao što vidimo na slici 18.12.



Slika 18.12: Paket B ovisi o paketu A

Razumijevanje ovisnosti između paketa korisno je za analiziranje stabilnosti softvera.

Osnovna upotreba UML dijagrama paketa je prikaz osnovnih paketa u sistemu i ovisnosti među njima, kao na slici 18.13.



Slika 18.13: Tipični dijagram paketa koji prikazuje osnovne pakete i njihovu ovisnost

18.7 Uvoz paketa i pristup elementima paketa

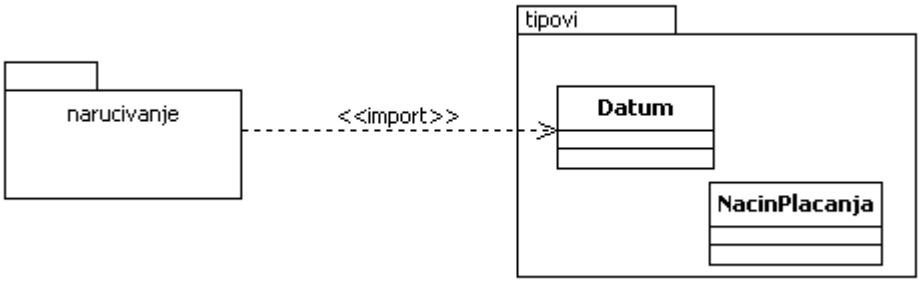
Kada paket uvozi (eng. import) drugi paket, taj paket može koristiti elemente uvezenog paketa bez navođenja punog imena. Ova funkcionalnost je slična uvozu u Javi, gdje klasa može uključiti paket i onda koristiti njegove elemente bez navođenja punog imena.

Relacija uvoza paketa se prikazuje sa relacijom ovisnosti od paketa koji uvozi do ciljnog paketa koji se uvozi. Ova relacija se dodatno obilježava sa stereotipom <<import>> kao na slici 18.14.

Slika 18.14: Paket `narucivanje` uvozi paket tipovi

Relacijom prikazanom na slici iznad, klase iz paketa `narucivanje` mogu koristiti javne klase iz paketa `tipovi` bez navođenja punog imena.

Paket može uvoziti pojedinačne elemente iz drugog paketa umjesto cijelog paketa, kao što se vidi na slici 18.15.



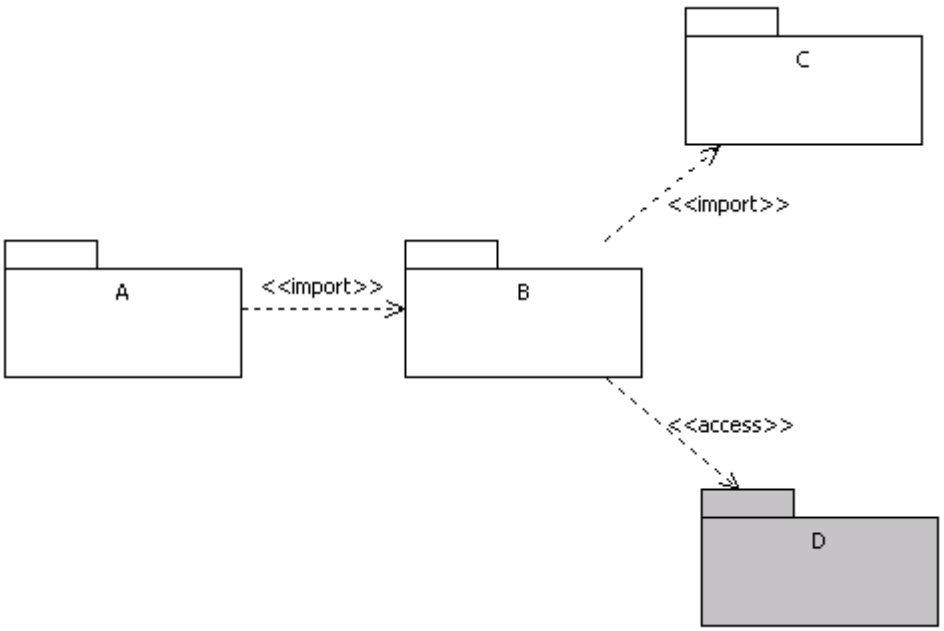
Slika 18.15. Paket `narucivanje` uvozi samo element `Datum` iz paketa tipovi

Pri uvozu paketa ili pojedinih elemenata, samo su javni elementi dostupni u okviru paketa koji vrši uvoz.

Vidljivost relacije uvoza

I sama relacija uvoza ima vidljivost. Uvoz može biti privatni ili javni, pri čemu, ako nije izričito naglašeno, podrazumijeva se javni uvoz. Javni uvoz znači da uvezeni elementi imaju javnu vidljivost u okviru paketa koji je izvršio uvoz dok privatni uvoz znači da uvezeni elementi imaju privatnu vidljivost. Javni uvoz se označava sa stereotipom <<import>> a privatni uvoz sa stereotipom <<access>>.

Razlika između <<import>> i <<access>> uključivanja se vidi tek kada postoji više paketa u međusobnim relacijama uvoza. Na slici 18.16, paket B vrši javni uvoz (<<import>>) paketa C i privatni uvoz (<<access>>) paketa D, tako da paket B može vidjeti javne elemente i paketa C i paketa D. Paket A javno uvozi paket B tako da može vidjeti njegove javne elemente. Paket A također može vidjeti javne elemente paketa C, jer C je javno uvezen u B, ali A ne može vidjeti elemente paketa D jer je isti privatno uvezen u B.



Slika 18.16: Paket A može vidjeti javne elemente paketa C ali ne i paketa D

18.8 Implementacija ovisnosti paketa

Relacije javnog i privatnog uvoza mogu se koristiti za modeliranje programskog koncepta uvoza i imenovanja klasa u okviru paketa ili imenovanih prostora. Relacije među paketima na slici 18.14 prikazane u programskim jezicima Java i C++:

Java :
```
package narucivanje;

// importovanje svih public elemenata iz paketa tipovi
import tipovi.*;

public class Narudzba
{
       Datum datumNarudzbe;
       ...
}
...
```

C++ :
```
namespace narucivanje
{
    using namespace tipovi;

    class Narudzba
    {
       Datum datumNarudzbe;
       ...
    };
}
```

Implementacija uvoza paketa `tipovi` u paket `narucivanje`

Pošto klasa `Narudzba` uvozi paket `tipovi`, može da pristupa klasi `Datum` bez korištenja punog imena.

Uvoz elementa na slici 18.15 odgovara sljedećoj Java i C++ implementaciji.

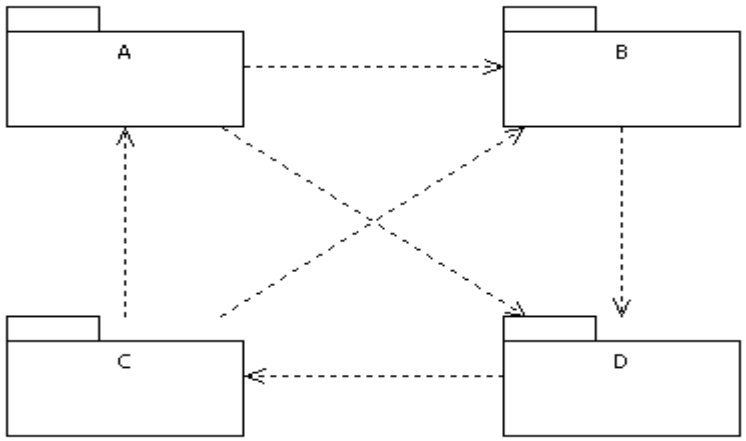
```
Java :  package narucivanje;

        // importovanje svih public elemenata iz paketa tipovi
        import tipovi.Datum;

        public class Narudzba
        {
               Datum datumNarudzbe;
               ...
        }
        ...
```

```
C++ :   namespace narucivanje
        {
            using namespace tipovi::Datum;

            class Narudzba
            {
               Datum datumNarudzbe;
               ...
            };
        }
```

Implementacija uvoza elementa `Datum` paketa `tipovi` u paket `narucivanje`

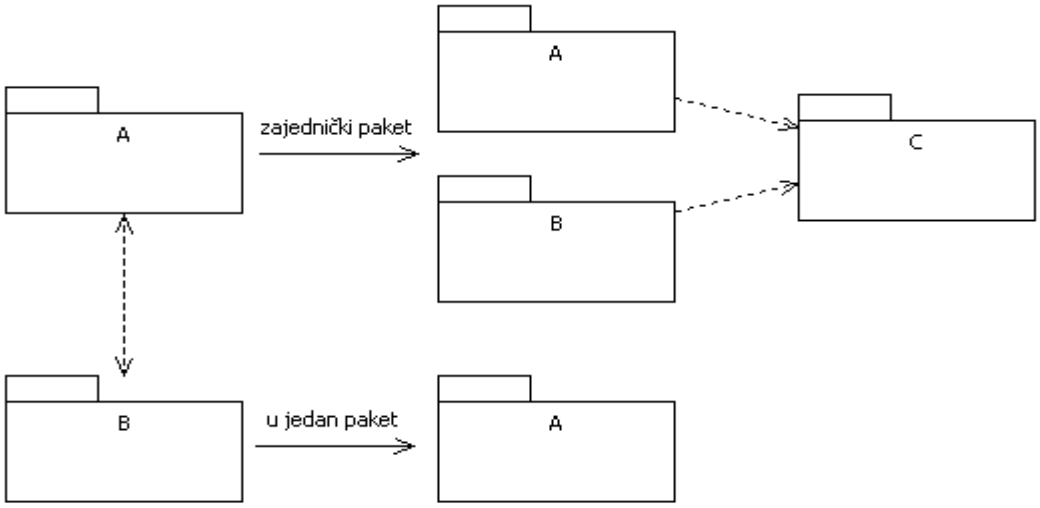
18.9 Upravljanje ovisnostima paketa

Ukoliko imamo kompliciranе ovisnosti među paketima, može doći do nestabilnog softvera, jer promjena u jednom paketu može uzrokovati probleme. Slika 18.17 pokazuje problem sa ovisnostima: promjena u bilo kojem paketu može utjecati na bilo koji paket.



Slika 18.17: Direktno ili indirektno promjena u bilo kom paketu može utjecati na sve druge pakete

Ako postoji cikličnost u ovisnostima među paketima, postoji nekoliko načina da se ukloni. Možemo uvesti novi paket o kojem će ovisiti oba paketa ili možemo svesti dva paketa na jedan i u njega smjestiti sve klase, kao što je prikazano na slici 18.18.

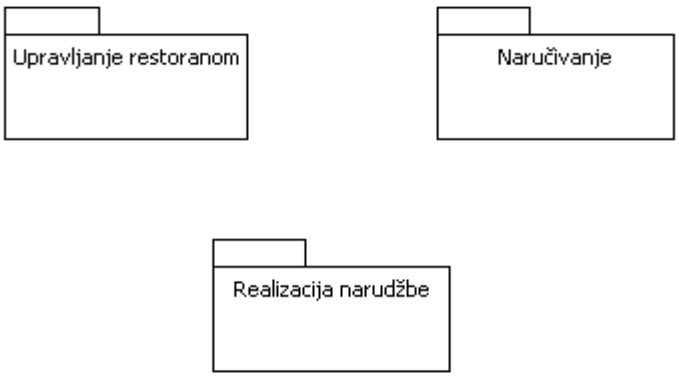


Slika 18.18: Uklanjanje cikličnosti u ovisnostima paketa

Ovisnost u poretku stabilnosti podrazumijeva da paket može ovisiti o paketu koji je stabilniji od njega. Nestabilni paket ovisi o mnogo drugih paketa dok stabilan paket ovisi o nekoliko paketa. Proučavanje dijagrama paketa može pomoći pri otkrivanju potencijalno lošeg dizajna uzrokovanog time da osnovni paketi sistema, kao što su oni koji sadrže interfejse ovise o nestabilnim paketima.

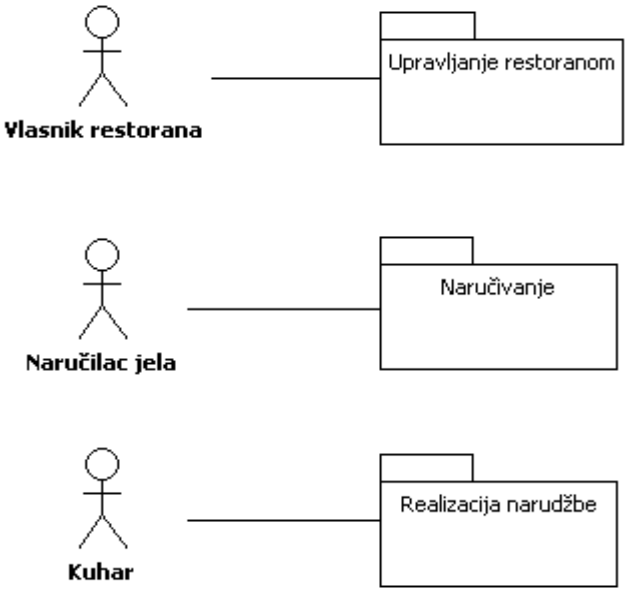
18.10 Paketi i slučajevi upotrebe

Kako koristimo pakete za grupiranje klasa sličnih funkcionalnosti, tako ih možemo koristiti i za grupiranje drugih UML elemenata, kao što su slučajevi upotrebe. Slika 18.19 pokazuje neke pakete slučajeva upotrebe za `e-restoran` sistem.



Slika 18.19: Grupiranje glavnih slučajeva upotrebe `e-restoran` sistema u pakete

Grupiranje slučajeva upotrebe u pakete pomaže u organiziranju modela i prikazivanju istog na višem nivou, te lakšem uočavanju koji akteri imaju interakciju s kojim dijelom sistema, kao što se vidi na slici 18.20.



Slika 18.20: Paketi omogućavaju prikaz interakcije aktera sa sistemom na višem nivou

Prilikom organiziranja slučajeva upotrebe u pakete potrebno je grupirati u jedan paket slučajeve upotrebe povezane relacijom generalizacije, relacijom uključivanja i proširivanja. Također je dobro grupirati slučajeve upotrebe na osnovu potrebe glavnih aktera i uključiti aktere na dijagramu paketa.

Glavna publika ovih dijagrama su svi učesnici projekta, pa organizacija ovih dijagrama treba reflektirati njihove potrebe.

Završno razmatranje

Dijagram paketa je povezan sa dijagramom klase i predstavlja logičku organizaciju klasa i dijagramom slučajeva upotrebe. Ako hoćemo fizičku organizaciju dizajna, tada koristimo dijagram komponenti.

Pitanja za ponavljanje

1.Koja je osnovna namjena dijagrama paketa?
2.Kojem pogledu na sistem pripada dijagram paketa?
3.Koja je UML notacija za paket?
4.Što je paket?
5.Kako se može prikazati klase koje čine sadržaj paketa?
6.Kako se prikazuju paketi unutar paketa?
7.Kako ime pomaže u određivanju da li je paket kohezivan?
8.Koje su UML varijacije prikaza klase unutar paketa?
9. Da li klase u istom paketu mogu pozivati jedna drugu bez navođenja potpunog imena?
10.Kako se međusobno pozivaju klase koje su u različitim paketima?
11.Koju vidljivost mogu imati elementi u paketu?
12.Kako se prikazuje relacija ovisnosti paketa?
13.Koje vrste uvoza vežemo za pakete?
14.Kako se obilježava javni uvoz paketa?
15.Objasniti vidljivost elemenata pri javnom uvozu.
16.Kako se obilježava privatni uvoz paketa?

17.Objasniti vidljivost elemenata pri privatnom uvozu.

18.Za koje pakete kažemo da su nestabilni paketi?

19.Kako možemo ukloniti cikličnu ovisnost?

20.Koja je namjena grupiranje slučajeva upotrebe u pakete?

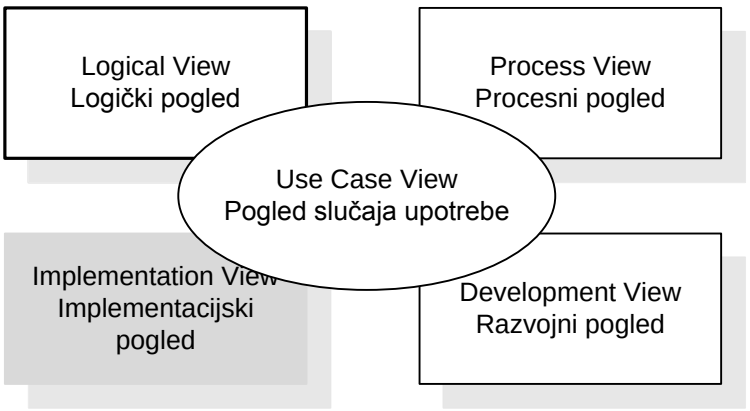
22. Kako se određuje koje slučajeve upotrebe grupirati u jedan paket?

23.Koja je veza između dijagrama klase i dijagrama paketa?

POGLAVLJE 19.

DIJAGRAM RASPOREĐIVANJA

Do sada smo vidjeli sve poglede na sistem osim jednog. Taj preostali pogled je implementacijski pogled i opisuje se pomoću dijagrama raspoređivanja (eng. deployment diagram). Ovim pogledom i dijagramom dobivamo model potreban za implementaciju sistema a koji se sastoji od hardverskih elemenata, njihovih međusobnih veza i softverskih elemenata raspoređenih na taj hardver.



Slika 19.1: Dijagram raspoređivanja pripada implementacijskom pogledu

19.1 Namjena dijagrama raspoređivanja

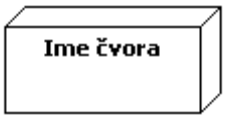
Dijagrami raspoređivanja se koriste za:

- Modeliranje fizičkih elemenata i komunikacijskih puteva između njih.
- Izradu plana arhitekture sistema.
- Dokumentiranje raspoređivanja softvera na hardverske čvorove.

U svojoj osnovnoj konfiguraciji dijagram raspoređivanja se sastoji od čvorova koji su međusobno spojeni komunikacijskim putevima.

19.2 Čvor

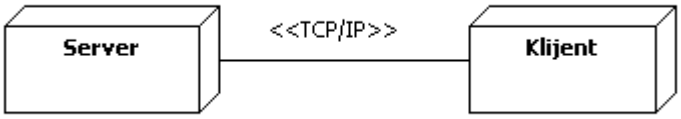
Notacija za čvor je kocka unutar koje se upisuje ime čvora, kao na slici 19.2. Čvorovi tipično predstavljaju kompjutere ali se mogu koristiti i za predstavljanje senzora, raznih periferija ili ugrađenih sistema.



Slika 19.2: Notacija za čvor

19.3 Komunikacijski putevi

Čvorovi su povezani sa komunikacijskim putevima, na kojima se mogu postaviti stereotipi da pokažu prirodu komunikacije između spojenih čvorova. Slika 19.3 pokazuje dva čvora koji se nazivaju `Klijent` i `Server` koji su spojeni sa komunikacijskim putem koji je obilježen sa stereotipom <<TCP/IP>>.

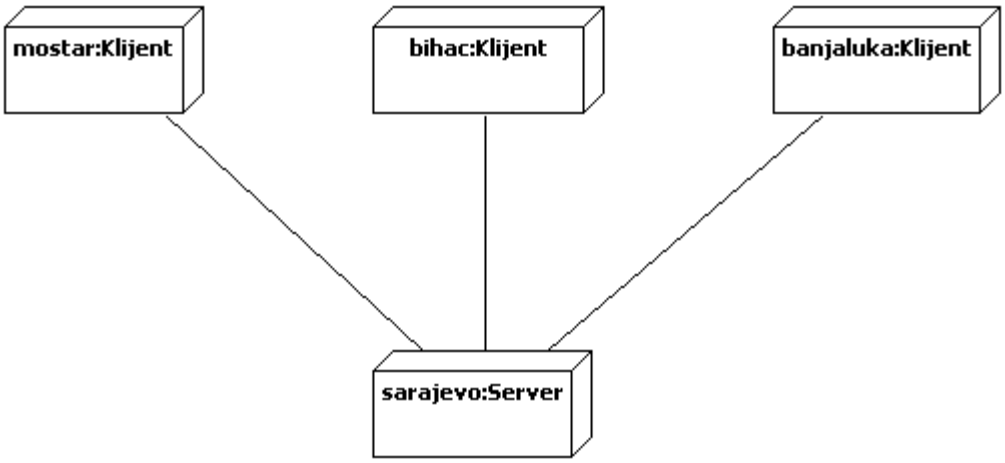


Slika 19.3: Komunikacijski put sa stereotipom

19.4 Instance čvora

Dijagram raspoređivanja može se koristiti da prikaže tip čvora, ali i instancu čvora, kao što je prikazano na slici 19.4, gdje su navedeni specifični klijenti i serveri koji će biti u implementiranom sistemu.

Možemo prikazati instance čvora koristeći *ime:tip* notaciju, kako je prikazano na slici 19.4.



Slika 19.4: Prikaz instanci

19.5 Čvorovi hardvera i čvorovi izvršnog okruženja

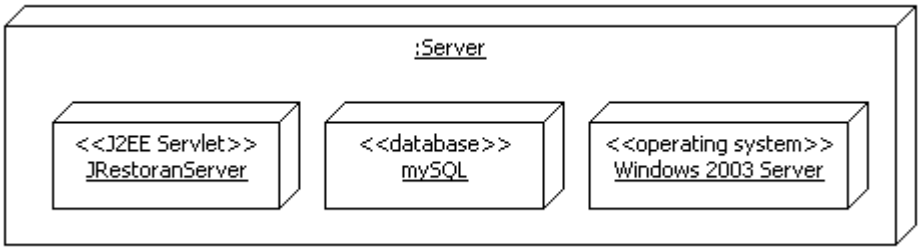
Već smo vidjeli da možemo koristiti čvor da prikažemo hardver u našem dijagramu raspoređivanja, međutim čvor ne mora uvijek biti hardver. Određeni tipovi softvera koji prikazuju okruženje unutar kojeg se ostale softverske komponente mogu izvršavati su također čvorovi. Dakle, čvor može biti hardverski ili softverski resurs na koji se može rasporediti dio aplikativnog softvera ili neki ovisni fajlovi.

Primjeri hardverskih čvorova su: Server, Desktop PC, Disk drajv.

Primjeri čvorova koji predstavljaju čvorove softvera odnosno izvršno okruženje:Operativni sistem, J2EE kontejner, Web server, Aplikativni server.

Čvor može imati stereotip <<device>> ili neki drugi stereotip koji će preciznije objasniti ulogu čvora. Na primjer može se koristiti stereotip <<database>> da bi označio čvor koji je namijenjen smještanju baze podataka, ili stereotip <<J2EEContainer>>, <<J2EEServlet>> za obilježavanje specifičnih različitih izvršnih okruženja.

Slika 19.5 prikazuje server za `e-restoran` na kome su operativni sistem, baza podataka i dinamička web-aplikacija. S obzirom na to da je `e-restoran` mali sistem, onda su i server baze podataka i web-server na istom hardveru.



Slika 19.5: Čvor sa više stereotip čvorova

Nije striktno potrebno u UML 2.0 odvajati čvorove uređaja od čvorova izvršnog okruženja, ali je dobar običaj da se to radi jer može dodatno pojasniti model.

19.6 Raspoređivanje softvera

Već je rečeno da se dijagram raspoređivanja koristi za prikazivanje raspoređenog softvera na hardveru sistema. Vidjeli smo kako se modelira hardver sistem, a sada ćemo se posvetiti raspoređivanju softvera odnosno modula kôda na taj hardver. UML raspoređeni softver se prikazuje pomoću artefakta (eng. artifact**)** koji su ustvari fizički fajlovi koji se izvršavaju ili se koriste od strane našeg softvera.

Uobičajni artefakti su:

Izvršni fajlovi, kao .exe ili .jar datoteke.

Biblioteke, kao što su .dll (ili podrške .jar fajlovima).

Izvorni fajlovi, kao što su .java ili .cpp fajlovi.

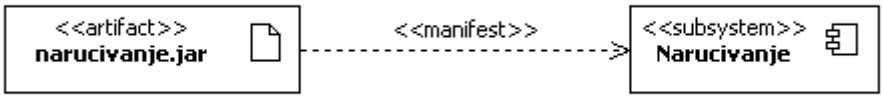
Konfiguracijski fajlovi korišteni od strane softvera tokom izvršavanja, uobičajeno u formatu kao .xml, .txt.

UML notacija za artefakt je pravougaonik sa stereotipom <<artifact>>, ili sa ikonom dokumenta u gornjem desnom uglu, ili kombinacijom ova dva prikaza, kako je pokazano na

slici 19.6. U nastavku ćemo artefakt označavati sa stereotipom <<artifact>> i ikonom dokumenta.

Slika 19.6: Ekvivalentna prezentacija `narucivanje.jar` artefakta

Kad projektiramo softver, mi ga dijelimo u odvojene funkcionalne grupe, kao što su komponente i paketi, koje se eventualno prevode u jedan ili više artefakta. Manifestirana ovisnost je prikazana sa strelicom ovisnosti od artefakta do komponente, označena sa stereotipom <<manifest>> kako je prikazano na slici 19.7.



Slika 19.7: Artefakt `narucivanje.jar` manifestira podsistem `Narucivanje`

Slika 19.7 pokazuje da aplikacijski kôd koji je dio podsistema `Narucivanje` je za svrhu raspoređivanja manifestiran u artefakt `narucivanje.jar`. Artefakt može manifestirati klase i pakete isto kao i komponente.

Kako artefakt može biti dodijeljen čvoru, manifest relacija daje i dodatnu informaciju o vezi softverskih komponenti i hardvera. Međutim, vezivanje komponenti sa artefaktima može dovesti do konfuznih dijagrama, pa je uobičajeno da su manifest relacije odvojene od relacija raspoređivanja, čak i ako su na istom dijagramu raspoređivanja.

Od UML verzije 2.0, artefakti predstavljaju fizičke fajlove.

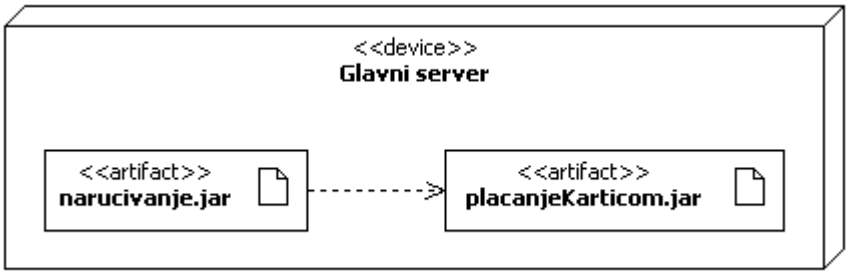
19.7 Raspoređivanje artefakta na čvor

Artefakt koji je raspoređen na čvor, što u principu znači da je on instaliran na tom čvoru, možemo modelirati na tri različita načina.

Jedan od načina je smještanje artefakt simbola unutar simbola čvora. Slika 19.8 prikazuje taj način pri raspoređivanju `narucivanje.jar` artefakta na `Glavni server`.

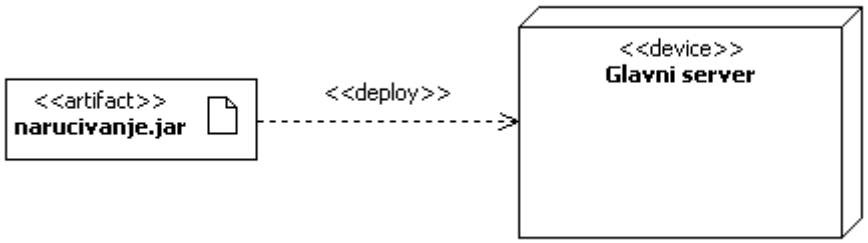
Slika 19.8: Artefakt `narucivanje.jar` artefakt raspoređen na `Glavni server`

U slučaju raspoređivanja više artefakta unutar čvora možemo prikazati i njihovu međusobnu relaciju kao na slici 19.9.



Slika 19.9: Više artefakta unutar čvora

Drugi način je prikaz pomoću strelice ovisnosti od artefakta prema ciljnom čvoru sa stereotipom <<deploy>> kako je prikazano na slici 19.10.



Slika 19.10: Alternativni način modeliranja artefakta

Treći, najkompaktniji način označavanja artefakta je prikaz samo imena artefakta unutar ciljnog čvora, kako je prikazano na slici 19.11.

Slika 19.11: Ime artefakta unutar čvora

Svi ovi načini prikazuju iste odnose rasporeda, pa ćemo navesti neke upute za izbor notacije.

Navođenje artefakta bez upotrebe simbola može sačuvati prostor ako imamo puno artefakta. Ako želimo da pokažemo da jedan artefakt ovisi od drugog, moramo prikazati artefakte i njihovu relaciju ovisnosti što se postiže sa prve dvije notacije.

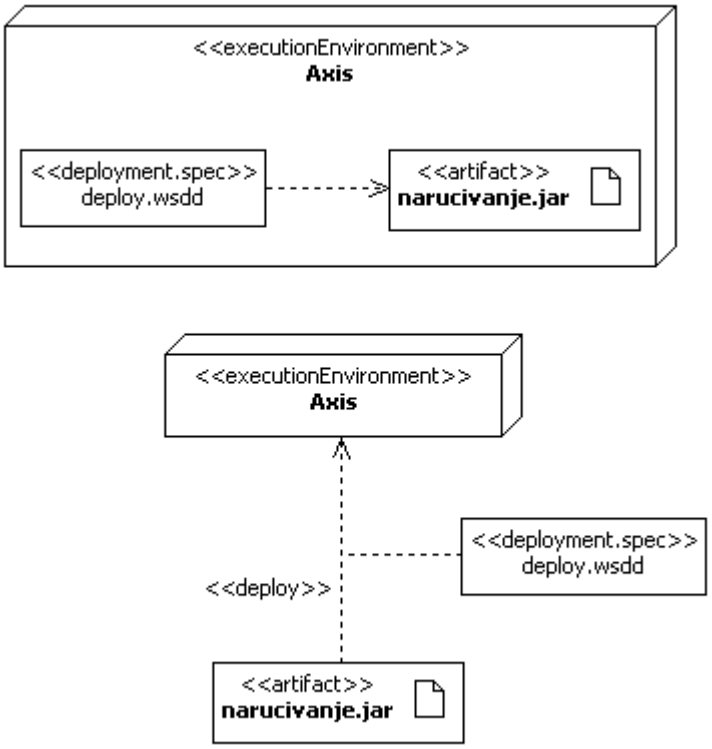
19.8 Specifikacija raspoređivanja

Instalacija softvera nije jednostavno kopiranje fajla na računar, obično treba specificirati konfiguracijske parametre neophodne za uspješno izvršavanje softvera. Specifikacija raspoređivanja je specijalni artefakt koji specificira kako je jedan artefakt raspoređen na čvor.

Specifikacija raspoređivanja se predstavlja kao pravougaonik sa stereotipom <<deployment spec>>. Postoje dva načina da se veže specifikacija raspoređivanja uz artefakt.

- Relacijom ovisnosti od specifikacije raspoređivanja do artefakta, pri čemu su i jedno i drugo ugniježđeni u ciljni čvor.
- Dodavanjem specifikacije raspoređivanja relaciji ovisnosti.

I jedan i drugi način dodjele specifikacije je prikazan na slici 19.12.



Slika 19.12: Ekvivalentni načini uvezivanja specifikacije raspoređivanja

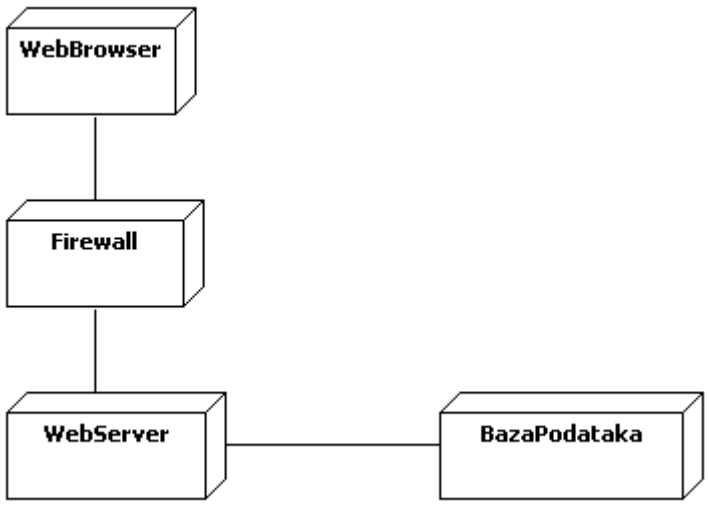
Na slici je specifikacija prikazana u okviru deploy.wsdd koji specificira realizaciju web-servisa, i koji govori koje klase izvršavaju web-servise i koji metodi u tim klasama mogu biti pozvani. Možemo navesti ove osobine u razvojnoj specifikaciji koristeći *ime: tip* notaciju.

Specifikacija raspoređivanja može sadržavati veliki broj osobina koje opisuju prirodu raspoređivanja. Ne trebamo navoditi svaku osobinu u specifikaciji raspoređivanja, samo one koje smatramo važnim.

19.9 Dijagram raspoređivanja kroz faze projekta

Dijagrami raspoređivanja su korisni tokom svih faza procesa dizajna. Kada počinjemo dizajniranje sistema vjerovatno su nam poznate samo osnovne informacije o fizičkom rasporedu komponenti i samoj implementaciji. Na primjer, u slučaju `e-restorana`, znamo da trebamo razviti Web-aplikaciju. Poznato nam je i da arhitektura uključuje Web-server i bazu podataka i da će klijenti pristupati preko Web-preglednika. Još uvijek nismo precizno odlučili koji hardver da koristimo i kako će se rasporediti komponente sistema.

Čak i u ovoj ranoj fazi možemo koristiti dijagram raspoređivanja da modeliramo ove karakteristike. To je prikazano na slici 19.13. Imena čvorova ne moraju biti precizna, i ne moramo specificirati komunikacijski protokol.

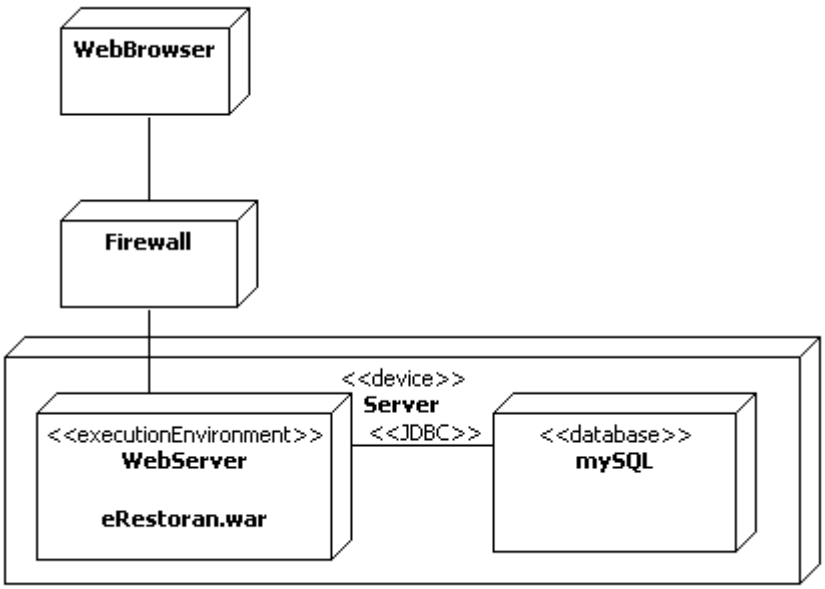


Slika 19.13: Gruba skica Web-aplikacije

Dijagrami raspoređivanja su također korisni u kasnijim fazama razvoja softvera.

Slika 19.14 prikazuje detaljan dijagram raspoređivanja specificirajući J2EE implementaciju sistema, naglašava hardver, komunikacijski protokol i dodjeljivanje softverskih artefakta

čvorovima. Detaljan dijagram raspoređivanja kao na slici 19.14 može biti korišten kao uputa za instaliranje sistema.



Slika 19.14: Možemo prikazati bilo koji obim detaljnosti o fizičkom dizajnu našeg sistema

Možemo revidirati dijagrame raspoređivanja tokom dizajna našeg sistema da popravimo grube inicijalne skice, nakon odluka o tehnologiji, komunikacijskom protokolu, softverskim artefaktima.

Završno razmatranje

Dijagram raspoređivanja je relativno samostalan u odnosu na druge UML dijagrame. Artefakti koji se raspoređuju na čvorove su prikazi komponenti. Da bi se izvršio izbor koje komponente koristiti i gdje ih treba rasporediti potrebno je poznavanje strukture klasa i opisi slučajeva upotrebe.

Pitanja za ponavljanje

1.Koja je osnovna namjena upotrebe dijagrama raspoređivanja?

2.Kojem pogledu pripada dijagram raspoređivanja?

3.Objasniti što predstavlja čvor i opisati UML notaciju za čvor.

4.Objasniti što se podrazumijeva pod pojmom komunikacijski put.

5. Kako se prikazuju instance čvora u dijagramu raspoređivanja?

6. Što se prikazuje sa čvorovima hardvera?

7.Što se prikazuje sa čvorovima izvršnog okruženja?

8. Koje su varijante UML notacije za artefakt?

9.Objasniti načine prikaza kako se artefakti povezuju za čvor.

10.Navedite dva primjera elemenata sa stereotipom na dijagramu raspoređivanja.

11. Što je specifikacija raspoređivanja i koja je UML notacija za njen prikaz?

12.U kojim fazama projekta su korisni dijagrami raspoređivanja?

13.Koja je veza dijagrama raspoređivanja i ostalih UML dijagrama?

POGLAVLJE 20.

MEHANIZMI PROŠIRENJA UML-a

U okviru ove knjige korišteni su primjeri kôda i u Java, C++ programskom jeziku da bi se demonstrirali pojedini UML koncepti. UML koncepti su primjenjivi na većinu objektno orijentiranih sistema, jezika (Java, C++, C# ili Smalltalk), platformi (J2EE or .NET), ili domena (finansije, medicina). Objektno orijentirani sistemi dijele mnoge zajedničke karakteristike u pogledu strukture i ponašanja, ali kada se vežu sa platformama i domenama, tada ima mnogo razlika u terminologiji. Na primjer, J2EE platforma ima EJB, JAR, i JSP, dok .NET platforma ima ADO, assemblies, ASP. Kada se kreira UML model, korisno je označiti elemente modela sa terminologijom specifičnom za izabranu platformu ili okruženje. Na primjer, bolje je specificirati da komponenta, ustvari će biti EJB, umjesto da je samo zovemo komponenta. Pokušavajući da se UML učini mogućim za svaku platformu ili domenu izgubila bi se generalna namjena UML-a i osnovni UML bi postao veoma kompliciran. OMG (Object Management Group) je ovo riješila uvođenjem mehanizama sa kojim UML može biti adaptiran i skrojen da zadovolji specifične potrebe i na taj način omogući da se kreiraju precizniji i jasniji modeli.

U ovom poglavlju će se detaljnije opisati mehanizmi proširenja (ekstenzija) koji postoje u okviru UML-a i koji omogućavaju njegovo proširenje za konkretnu vrstu sistema koji se modelira. UML eksplicitno podržava mehanizme proširenja koji na odgovarajući način dodaju semantiku ili sadržaj na postojeće elemente za modeliranje. Jedan od načina proširenja UML-a je dodavanje stereotipova na postojeće elemente, dodavanje označenih vrijednosti na elemente, ili postavljanje ograničenja na elemente. Neka od spomenutih proširenja smo uveli i prilikom objašnjavanja elemenata pojedinih dijagrama UML-a. Mnogi alati za podršku modeliranje podržavaju ova proširenja. Ova proširenja mogu se zajedno grupirati u profajl (eng. profile). Na taj način može se kreirati novi UML „dijalekt“ za specifično okruženje i specifičnu namjenu.

20.1 Pregled standardnih proširenja

UML se zasniva na svojim mehanizmima proširenja za definiranje dodatnih karakteristika. Evolucija UML je rapidno rasla u pogledu standardnih proširenja. Ovo poglavlje će opisati nekoliko standardnih proširenja zajedno sa nekim instrukcijama kako definirati svoja korisnički definirana proširenja.

Kao što je već spomenuto tri osnovna elementa proširenja su: označene vrijednosti, ograničenja i stereotipovi.

Označene (eng. tagged) vrijednosti su osobine priložene uz UML elemente. Primjer označene vrijednosti je tag (oznaka) autor kojoj je dodijeljeno ime autora klase.

Ograničenja (eng. constraints) su pravila koja ograničavaju semantiku jednog ili više elemenata. Ograničenja se mogu pripojiti uz klase ili objekte, a često i uz relacije.

Stereotipovi (eng. stereotype) predstavljaju glavni mehanizam proširenja. Oni daju metode za uspostavljanje metamodela proširenjem metaklasa i kreiranjem novih elementa koji pripadaju metamodelu. Stereotipovi mogu dodati novu ili proširiti postojeću semantiku za svaki UML element. Pri modeliranju Web orijentiranih sistema može se definirati novi tip klase kao <<JSP>>. Poslije se može prikazati JSP klasa primjenjujući <<JSP>> stereotip, što uzrokuju primjenu svih definiranih pravila za <<JSP>> tip klase.

Mogu se grupirati skupovi ovih proširenja u profajl da bi se upravljalo specifičnim okruženjima.

20.2 Profajl

Profajl (eng. profile) je jedan od načina prilagodbe UML-a posebnoj platformi (J2EE, .NET) ili domenama (financijskom, medicinskom). Profajlovi su sačinjeni od stereotipa, označenih vrijednosti i ograničenja. Oni sadrže rječnik koji se odnosi na domenu sistema ili platformu i dozvoljavaju da se rječnik primijeni na elemente modela, da bi sam model učinili što jasnijim. Mnogi profajlovi su uspostavljeni i održavaju se od strane OMG-a, već više puta spomenute organizacije koja upravlja UML standardima. Postoje također profajlovi koji su proizvedeni od strane individua i organizacija. Neki od standardnih UML profajlova su profajlovi za modeliranje sistema u realnom vremenu, za modeliranje dizajna baza podataka, za .NET

komponente, za J2EE/EJB komponente, za modeliranje poslovnih procesa.

Profajlovi su proširenja paketa. Kako se povezuju sa paketima vidjet ćemo poslije objašnjenja ostalih elemenata profajlova stereotipa, označenih vrijednosti i ograničenja.

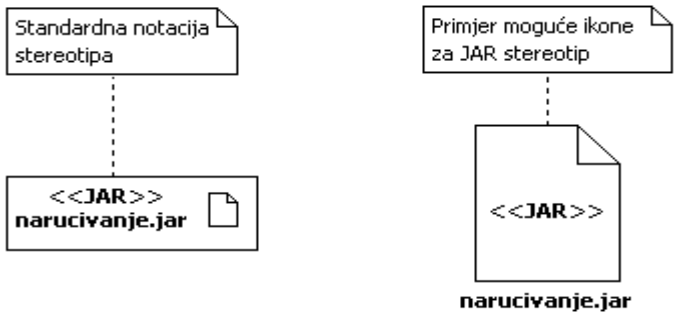
20.3 Stereotip

U ranijim verzijama UML-a, mogli su se stereotipovi deklarirati po potrebi u samoj fazi modeliranja. To je vodilo do konfuzije o tome kada da se koriste stereotipi, kao i koje je njihovo pravo značenje. Od strane korisnika UML-a došlo je do neformalne standardizacije stereotipa, čime se omogućavalo više puta korištenje istih stereotipa. UML 2.0 fiksira ovaj problem deklariranjem stereotipa i označenih vrijednosti u profajlovima.

Stereotipovi označavaju da elementi imaju specijalno korištenje ili namjenu. Podsjetimo se, stereotipi se najčešće prikazuju sa specifikacijom imena stereotipa između << >>. Slika ispod označava artefakt sa stereotipom JAR (ovaj stereotip se nalazi u J2EE profile).

Slika 20.1: Artefakt sa JAR stereotipom

Ako stereotip ima i ikonu kao svoju oznaku, tada se u modelu može prikazati i na taj način. UML alati za modeliranje generalno dozvoljavaju izbor notacija prikaza. Slika ispod pokazuje za JAR stereotip i standardnu notaciju stereotipa i notaciju stereotipa sa ikonom.



Slika 20.2: Notacije stereotipa: standardna i notacija sa ikonom

Ne postoji ograničenje koliko se stereotipa može primijeniti na pojedinačan element, što se vidi i na slici ispod, koja pokazuje primjenu dva stereotipa (JAR i file).

Slika 20.3: Više stereotipa za element

Stereotipovi se najčešće koriste za proširenje klasa i asocijacija, ali mogu se koristiti da prošire i bilo koju metaklasu. Metaklasa je klasa čije instance su klase, a ne objekti. Metaklasa je osnovni element modela koji je proširen sa stereotipom. Metaklasa se može primijeniti na sve vrste UML elemenata. Metaklasa nema standardnu ikonu, ona se samo označava sa stereotipom <<metaclass>>. Na nivou paketa metaklasa je označena sa <<metamodel>>, drugim standardnim stereotipom. Dizajner sistem može koristiti skup metaklasa koje definira metamodel.

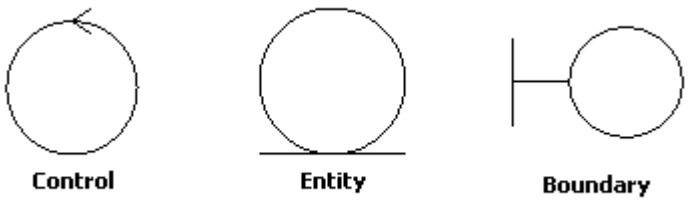
U poglavljima vezanim za dijagrame uveli smo i objasnili značenje jednog broja stereotipa. Za slučajeve upotrebe su korišteni stereotipovi <<extend>> i <<include>>, za artefakte koriste se stereotipi <<executable>>,<<file>>,<<library>>,<<document>>,<<source>> i <<script>>, za komponente <<required interface>> i <<provided interface>>. Veliki broj stereotipa je uveden i pri objašnjavanju ovisnosti, neki od njih su <<call>>,<<refine>>,<<send>>, <<trace>> <<derive>> *i* <<instantiate>>. Uveli smo i stereotipe za kreiranje i uništavanje objekata <<create>> i <<destroy>>.

Sada ćemo predstaviti nekoliko veoma važnih stereotipa koji su ugrađeni u mnoge profajlove, a koje nismo do sada spominjali.

Kontrola, Granica, Model

Klase mogu biti označene sa stereotipom `Control`, `Boundary` i `Entity`. Ovi stereotipi se koriste da povećaju semantičko značenje klasa i njihovo korištenje u situacijama modeliranja. Ovi stereotipi se ne nalaze u osnovnoj UML specifikaciji. Oni su zajednički stereotipi u mnogim UML alatima i koriste se za vrijeme dizajna i analize. Oni su bazirani na model-pogled-kontrolor (eng. MVC model-view-controller) konceptu, gdje je `Entity` model, `Control` je kontrolor, a `Boundary` je pogled.

MVC koncept, prikazan na slici 20.4, je prikladna solucija (patern) za sisteme sa modelom entiteta, koji se predstavljaju i manipuliraju preko grafičkog korisničkog interfejsa (GUI).



Slika 20.4: Tipične ikone za stereotipe `Control`, `Entity`, `Boundary`

`Boundary` stereotip se koristi za predstavljanje klase za prezentaciju i manipulaciju. `Boundary` klase predstavljaju informacije i komuniciraju sa drugim sistemima, kao što je čovjek ili mašina. Ove klase su tipično klase korisničkog interfejsa (prozor, dijalog box) ili komunikacijske klase kao što je TCP/IP.

`Entity` stereotip se koristi za modeliranje trajnih biznis objekata koji predstavljaju neke osnovne koncepte procesa koji se modelira kao što je račun, ugovor i koji se čuvaju u sistemu.

`Control` stereotip se koristi za spajanje granica objekata sa entitetskim objektima i za upravljanjem sekvencom operacija unutar sistema.

20.4 Označene vrijednosti

Označena vrijednost (eng. tagged value) je par oznaka-vrijednost koja daje dodatne informacije o elementu. Označena vrijednost može se pripojiti svakom pojedinačnom elementu. Oznaka je ime neke osobine koju želimo da zabilježimo, a vrijednost je vrijednost osobine za dati element. Označene vrijednosti mogu se koristiti za smještanje proizvoljnih informacija o elementima. One se često koriste za dodavanje informacija o projektu, kao datum kreiranja, status razvoja, status testa, ime osobe odgovorne za neki element modela. i slično. Neki tagovi su unaprijed predefinirani (ordered, composite) i mi smo ih koristili u okviru knjige.

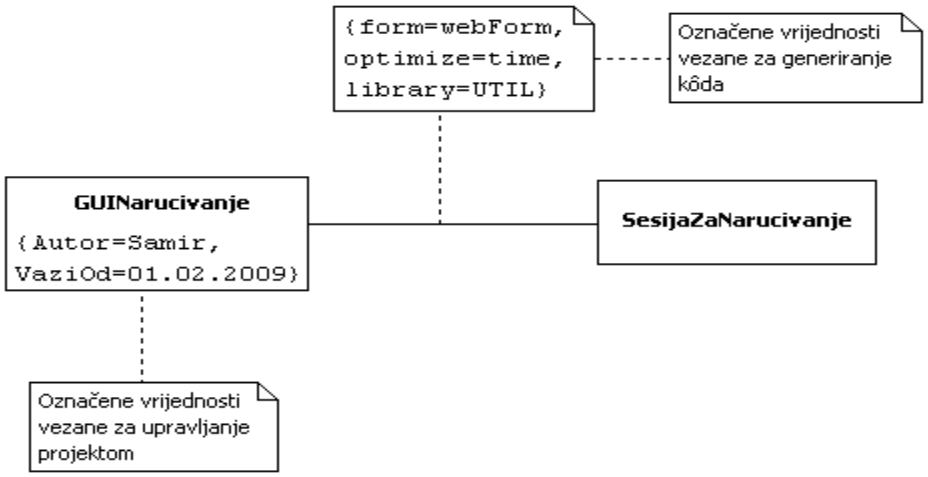
Označene vrijednosti osiguravaju način za dodavanje implementacijski ovisnih informacija o elementima. Na primjer generator kôda treba dodatne informacije o vrsti kôda koji se generira. Često postoji više mogućih načina za korektnu implementaciju modela i dobro je pri modeliranju dati dodatne instrukcije za odabir odgovarajućeg načina.

Da bi definirali označene vrijednosti, potrebno je slijediti sljedeće korake:

- procijeniti namjenu označenih vrijednosti,
- definirati elemente kojima će se dodijeliti,
-imenovati odgovarajuće označene vrijednost,
-definirati tip vrijednosti,
-odrediti ko će je interpretirati čovjek ili mašina, ako je mašina vrijednost mora biti formalnije specificirana,
- dokumentirati uvedenu označenu vrijednost sa jednim ili više primjera da se pokaže kako se koriste.

Dva primjera mogućih korisnički-definiranih označenih vrijednosti (`Autor`, `form`) su prikazana na slici 20.5.

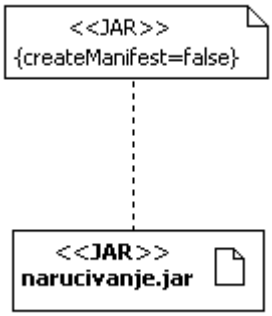
Označene vrijednosti se prikazuju kao stringovi sa imenom taga, znakom jednakosti, i vrijednosti. Smještaju se u vitičastim zagradama. Često se izostavljaju sa dijagrama i pokazuju na pop-up listama i formama.



Slika 20.5: Označene vrijednosti

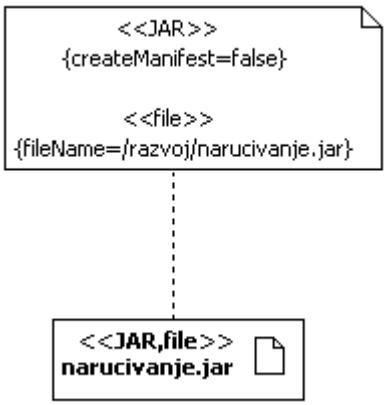
Označene vrijednosti mogu se koristiti i za dodavanje dodatnih vrijednosti za elemente modela koji su već prošireni sa stereotipom.

Stereotipi mogu imati jednu ili više povezanih označenih vrijednosti. Označene vrijednosti daju dodatne informacije koje su povezane sa stereotipom. Označena vrijednost se pokazuje sa simbolom komentara koji je pripojen stereotip elementu kako je pokazano na slici 20.6.



Slika 20.6: Označena vrijednost za stereotip

Označena vrijednost uz stereotip u ovom slučaju označava da li treba ili ne treba kreirati manifest za JAR stereotip. Ako je više stereotipa primijenjeno na element tada se označena vrijednost za svaki od njih dijeli u poseban dio za taj stereotip što ilustrira slika 20.7.



Slika 20.7 : Više stereotipa sa pripadajućim označenim vrijednostima

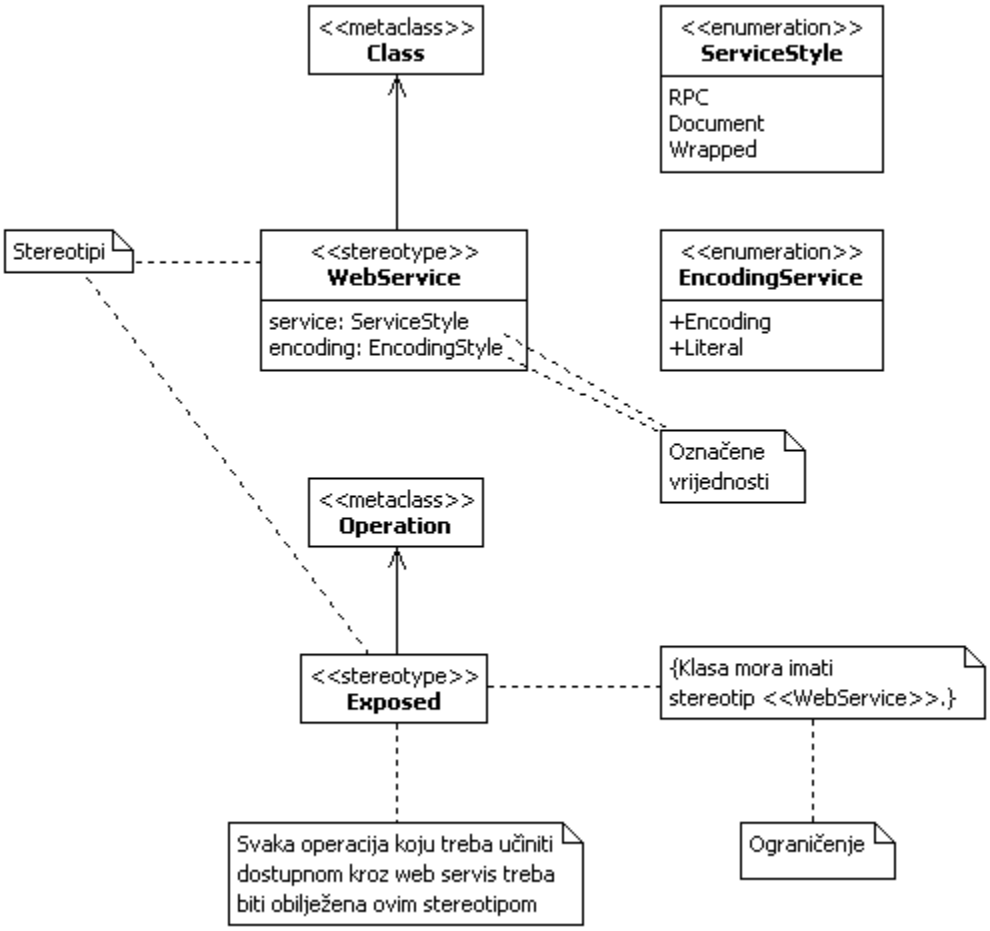
20.5 Ograničenja

Za razliku od stereotipa i označenih vrijednosti, ograničenja ne odgovaraju simbolima koji se koriste u UML modelima. Ograničenja se također specificiraju u definiciju profajla, ali oni nameću pravila ili restrikcije na elemente modela. Ograničenja smo do sada uvodili naročito na dijagramima klase, tako što smo ih pisali u vitičastim zagradama uz element na koji se odnose ili u simbolu komentara. Pravilo koje definira ograničenje može se izraziti u prirodnom jeziku ili u nekom drugom kompjuterskom jeziku. Za ograničenja na sistem modeliran sa UML-om preporučuje se korištenje OCL-a kao specijaliziranog jezika za izražavanje ograničenja. U okviru ovog poglavlja uvodimo OCL detaljnije, pa će o izražavanju ograničenja biti još riječi u nastavku.

20.6 Kreiranje i korištenje profajla

Obično se može jednostavno koristiti neki postojeći profajl koji je ugrađen u UML alat ili neki koji je preporučen od strane OMG-a. Međutim, mnogi UML alati omogućavaju i kreiranje svojih profajlova, pri čemu treba biti pažljiv, jer stvarna snaga profajla dolazi kada se koriste od mnogo pojedinaca ili grupa zainteresiranih za istu platformu ili domenu. UML alat može dozvoliti kreiranje profajla korištenjem grafičkih elemenata prozora i dijaloga.

Grafički model za profajl izgleda kao na slici 20.8 Stereotipi definirani u profajlu imaju standardni stereotip <<stereotype>>. Dva nova stereotipa su deklarirana `WebService` i `Exposed`.



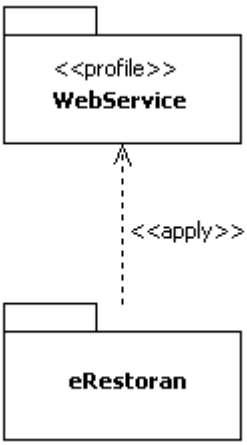
Slika 20.8: Kreiranje novog profila sa dva stereotipa: `Exposed` i `WebService`, pridruženim označenim vrijednostima i ograničenjima

Rad sa meta-modelom

Na slici 20.8 vidimo nazive kao što je `Class` koji pripadaju terminologiji UML meta-modela, a što se u standardnom UML modeliranju ne koristi. Meta-model definira pravila kako se UML elementi koriste i povezuje kao što su da li klasa može imati potklasu ili da li se klasa može povezati sa drugim klasama. Kada se modelira profajl tada se radi sa meta-modelom, i prilagođavaju se regularna UML pravila za konkretni sadržaj.

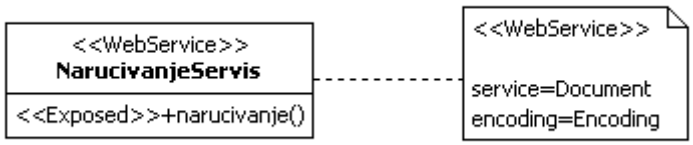
Korištenje profajla

Model na slici 20.8 pokazuje kako se kreira `WebService` profajl. Za njegovo stvarno korištenje, profajl se primjenjuje na paket koji će ga koristiti. Da bi se primijenio na paket, paket i profajl se povezuje isprekidanom linijom, sa <<apply>> stereotipom duž linije, pri čemu je strelica na strani profajla, kao na slici 20.9.



Slika 20.9: Primjena `Web-service` profajla na `eRestoran` paket

Na slici je primijenjen `WebService` profajl na `eRestoran` paket koji se može koristiti u `eRestoran` modelu kao na slici 20.10.



Slika 20.10: Primjena profajla na elemente paketa

Slika 20.10 pokazuje primjenu elemenata `WebService` profajla na `NarucivanjeServis` klasu u `eRestoran` paketu. Potrebno je uočiti da je klasa `NarucivanjeServis` označena sa `WebService` stereotipom a pojedinačan element sa `Exposed` stereotipom. Ograničenje iz

profajla nije eksplicitno pokazano na modelu, ali je primijenjeno na operaciju `narucivanje()`.

20.7 OCL - jezik za izražavanje ograničenja

Ograničenje je pravilo koje dozvoljava specificiranje nekih limita na elemente modela. Ograničenja postoje i u stvarnom životu, na primjer, automobili se ne smije voziti preko ograničenog limita, ili ne smiju ga voziti osobe mlađe od neke starosne dobi. Veoma je važno ova pravila dodati na dijagram, da bi se što bolje definiralo ponašanje samog sistema.

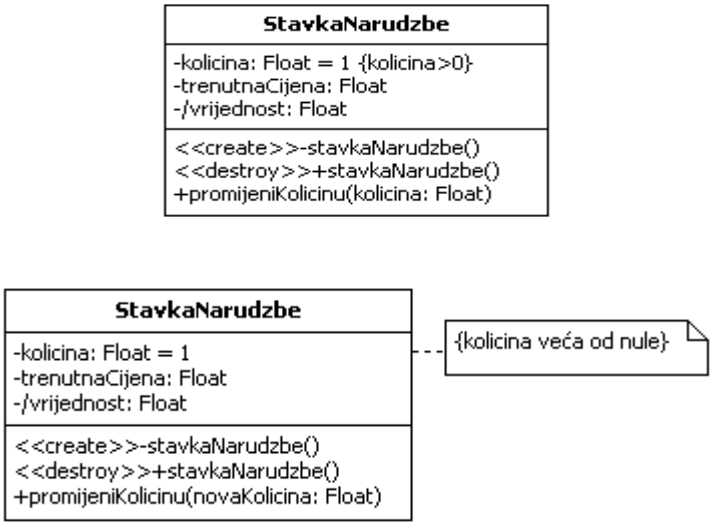
Definiranje vlastitih ograničanja

Modeli su često zasnovani na korisnički-definiranim ograničenjima. Da bi definirali korisnički napredno ograničenje potrebno je slijediti korake:

- odrediti na koji element se primjenjuje ograničenje,
- utvrditi semantički utjecaj na element korisnički definiranog ograničenja,
- jedan ili više primjera primjene korisnički definiranog ograničenje,
- dati objašnjenje kako se može implementirati korisnički definirano ograničenje.

Specificiranje ograničenja

Podsjetimo se da se ograničenja mogu pisati u okviru vitičastih zagrada i postaviti uz element za koji se postavlja ograničenje, ili se može napisati u okviru simbola komentara i povezati sa elementom. Slijedi prikaz različitih načina specificiranja atributa i ograničenja za atribut `kolicina` koji treba biti veći od nule.



Slika 20.11: Različiti načina prikazivanja ograničenja

Alternativna notacija je da se ograničenja evidentiraju u odvojenom dokumentu, što će biti objašnjeno u nastavku ove sekcije.

Ograničenja se mogu pisati u prirodnom jeziku kao na primjer: količina veća od nule.

S obzirom na to da prirodni jezik može biti dvosmislen, mnogi moderatori koriste sintaksu sličnu programskim jezicima: `kolicina > 0` izgleda kao da je Java ili C++ izraz.

To znači da ima mnogo opcija za izražavanje ograničenja. I kako odlučiti koju notaciju koristiti? Neki izrazi mogu izgledati različito u različitim programskim jezicima. Ako su ograničenja izražena na standardni i predikativni način, ne samo da mi lakše razumijemo ograničenja, već i automatski alati mogu razumjeti ta ista ograničenja. To omogućava automatsku provjeru ograničenja na dijagramima i generiranje kôda na osnovu dijagrama. Zbog toga, Object Management Group (OMG) je zaključila da je potreban jedan formalni jezik za izražavanje ograničenja koji će biti dovoljno jednostavan za ljude koji ga koriste, koji će biti generalan i sa kojim će se moći izraziti ograničenja bez obzira na jezik implementacije. Taj jezik je upravo OCL što je skraćenica od Object Constraint Language.

OCL je razvijen od strane IBM-a za izražavanje ograničenja pri poslovnom modeliranju, pošto je kao takav zadovoljio, izabran je i za rad sa UML-om. U okviru UML dijagrama, OCL se primarno koristi za pisanje ograničenja na dijagramima klase i uvjeta na dijagramima stanja i aktivnosti.

Mi ne moramo, kako smo naveli, koristiti OCL da bi izrazili ograničenja, ali namjena ove sekcije je da prikažemo prednosti upotrebe OCL-a i da damo više detalja o samom jeziku.

OCL je jezik izraza, pri čemu izrazi ne mijenjaju vrijednost bilo kojeg elementa modela. Izrazi jednostavno vraćaju vrijednosti. Programeri interpretiraju OCL izraze i konvertiraju ih u akcije programskih jezika. Posljedično, OCL nema notaciju za kontrolne strukture.

Tipovi podataka

OCL ima 4 ugrađena tipa: Boolean, Integer, Real i String. Tabela 20.1 pokazuje primjere tipičnih konstanti za ova 4 tipa.

Tabela 20.1: Osnovni OCL tipovi

Tip	**Primjer**
Boolean	true; false
Integer	1; 6,664; -200
Real	2.7181828; 10.5
String	"Hello, World."

Operatori

OCL ima osnovne aritmetičke, logičke i operatore poređenja. OCL sadrži i napredne funkcije kao maksimum dvije vrijednosti, spajanje stringova, i slično. Tabela 20.2 prikazuje operatore i standardne operacije koje se primjenjuju u OCL izrazima.

Tabela 20.2: OCL operatori

Grupa operatora	**Simboli**	**Tip podataka**
Aritmetički	+, -, *, /	Integer, Real
Dodatni aritmetički operatori	abs(), max(), min( )	Integer, Real

Grupa operatora	Simboli	Tip podataka
Poređenje	<, <=, >, >=	Integer, Real
Jednakost	=, <>	Svi
Logički	and, or, xor, not	Boolean
String	concat(), size(), substring( ), toInteger(), toReal( )	String

OCL izrazi

OCL izrazi sadrže elemente modela, konstante i operatore. Elementi modela uključuju atribute klasa, operacije i članove asocijacije. OCL izrazi na slici 20.12 koriste elemente modela klase `Narudzba` i `StavkaNarudzbe`.

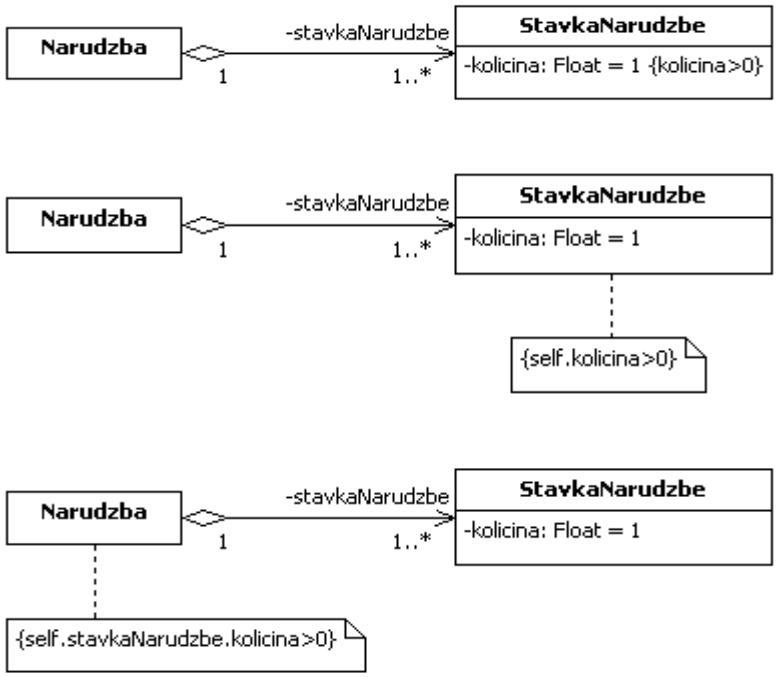
Konstante su nepromjenjive vrijednosti nekog od definiranih tipova u okviru OCL-a. Na slici 20.12 konstante su 0 tipa Real i 'Odustao gost' tipa String. Operatori kombiniraju model elemente i konstante u izraz. Na slici 20.12 uočavaju se operatori su <>, +, i =. Kombiniranje operatora i zagrada formiraju se izrazi: `kolicina > 0`, `trenutniStatus <> 'Odustao gost'`. Jednostavnije izraze možemo kombinirati u cilju izražavanja kompleksnijih izraza, pri čemu možemo koristiti i zagrade. Prioritet računanja je isti kao u matematici. `vrijednost=kolicina*trenutnaCijena*(1+vratStopuPDV()/100)` je primjer jednog takvog izraza.

Slika 20.12: Primjer OCL ograničenja različite kompleksnosti

OCL ograničenja na slici 20.12 napisani su uz atribute klase, međutim OCL izrazi se mogu pisati na različitim mjestima na dijagramima. Kako pisati OCL izraze ovisi od konteksta (eng. context) odnosno od mjesta gdje smo na dijagramu.

Kontekst

Kontekst (eng. contex) može biti klasa, asocijativna klasa ili slučaj upotrebe. Slika 20.13 pokazuje kako da provjerimo da je `kolicina > 0` na različitim tačkama na dijagramu. Prvi dijagram pokazuje ograničenje u kontekstu atributa `kolicina` klase `StavkaNarudzbe`, drugi dijagram pokazuje ograničenje u kontekstu klase `StavkaNarudzbe`, a treći pokazuje ograničenje na klasu `Narudzba` u odnosu na njenu asocijaciju sa klasom `StavkaNarudzbe`.



Slika 20.13: Načini pisanja ograničenja u ovisnosti od elementa primjene

Mogu se pisati i OCL ograničenja koja nisu fizički spojena za neki od elemenata modela. UML alat može sadržavati tekst editor za unos ograničenja i eksplicitno pisanje ograničenja, ili se ograničenja mogu smještati u posebnom tekstualnom dokumentu. Sam kontekst se uvodi sa ključnom riječi context.

Ako je kontekst `StavkaNarudzbe` klasa, tada se ograničenje piše:

context StavkaNarudzbe
inv: self.kolicina > 0

inv ključna riječ koja indicira tip ograničenja o kojima slijedi u nastavku.

Tipovi ograničenja

Kada se specificira contex, može se specificirati i tip ograničenja. Postoje 3 tipa ograničenja:

Invarijantna ograničenja

Invarijantno ograničenje je osobina koja mora biti tačna sve vrijeme života elementa modela. Ako osobina nije tačna sistem je u invalidnom stanju. Definira se na

atributima klase. Na slici 4 `kolicina` atribut klase `StavkaNarudzbe` mora uvijek biti veći od nule.

Razlog uvođenja invarijanti je specificiranje ograničenja u skladu sa kojima sistem može raditi. Kada sistem izvodi zadatak, invarijante trebaju biti tačne na početku zadatka, i ostati takve do kraja njegovog izvršavanja. Odgovornost je dizajnera i implementatora osigurati da invarijante ostanu tačne za legalno izvršavanje sistema. Invarijante se postavljaju u kontekstu sa ključnom riječju inv.

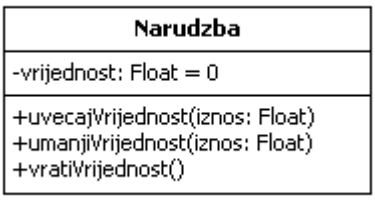
Preduvjeti

Preduvjeti je ograničenje koje mora biti tačno prije nego što se izvrši pojedini dio sistema. Mogu se primijeniti na operacije za vrijeme dizajna i implementacije, i na slučajeve upotrebe za vrijeme analize. Dizajneri i implementatori mogu koristi preduvjete da izvrše provjere prije nego što se operacija izvrši, kao što je validacija ulaznih parametara operacije. Mogu se koristiti da osiguraju da se invarijante nikada ne povrijede odnosno postanu netačne.

Postuvjeti

Postuvjet je ograničenje koje mora biti tačno nakon što se sistem izvrši. Postuvjeti se također kao i preduvjeti definiraju na metodi i provjeravaju se nakon što se metoda izvrši.

Isto kao invarijante, preduvjeti i postuvjeti se postavljaju u kontekstu i zahtijevaju navođenje pre i post ključne riječi respektivno. Slijedi primjer koji pokazuju kako osigurati preduvjete i postuvjete za operaciju `uvecajVrijednost` za referentni dijagram klase na slici 20.14.



Slika 20.14: Klasa sa preduvjetima i postuvjetima za **`uvecajVrijednost`** metod

Pretpostavimo da će `uvecajVrijednost` uvećati atribut vrijednost za vrijednost iznos. Prvo želimo specificirati preduvjet da je legalan samo iznos manji od 100, i postuvjet koji osigurava da će vrijednost biti uvećana za iznos. Da bi napisali preduvjete i postuvjete specificira se contex i poslije ograničenja.

context Narudzba::uvecajVrijednost(iznos:Float)
pre: iznos < 100
post: vrijednost = vrijednost@pre + iznos

Treba uočiti i način korištenja @pre direktiva. `vrijednost`@pre je vrijednost atributa `vrijednost` prije nego što se metoda izvrši.

Notacija @pre se može primijeniti i na metode:

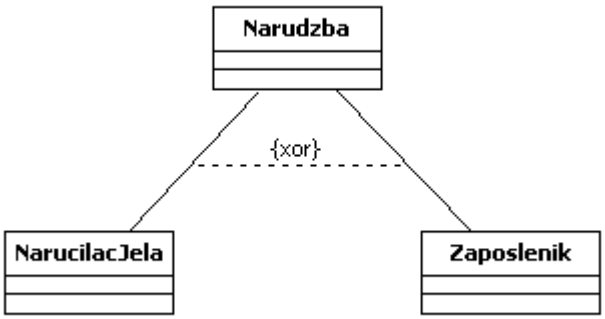
context Narudzba::uvecajVrijednost(iznos:Float)
pre: iznos < 100
post: vratiVrijednost( ) = **vratiVrijednost@pre( )** + iznos

Tipovi ograničenja invarijanti, preduvjet, postuvjet su dio pristupa poznatog kao dizajniranje po ugovoru ("Design by Contract"), razvijenog od Bertrand Meyer. Ovaj pristup pokušava učiniti pouzdanijim sam kôd uspostavljanjem ugovora između klasa i njegovih klijenata. Ugovor garantira njenim klijentima da će, ako oni pozivaju metodu sa valjanim vrijednostima, oni primiti valjanu vrijednost u odgovoru.

Primjena ograničenja na više elemenata

Ograničenja se primjenjuju na elemente korištenjem izraza, jedno ograničenje se primjenjuje na jednu vrstu elementa. Pošto ograničenja mogu uključiti više elemenata, oni moraju biti iste vrste. Na slici 20.15 su prikazane asocijacije klase `Narudzba` sa klasom `NarucilacJela` i klasom `Zaposlenik`, i xor ograničenje koje specificira da skup asocijacija ima vezano ograničenja na njihovim linkovima. Na slici xor ograničenje daje dodatnu restrikciju da klasa

`Narudzba` može biti vezana u jednom trenutku samo sa jednom od klasa: `NarucilaJela` ili `Zaposlenik`.



Slika 20.15: xor ograničenje vezano uz asocijacije klase

Pitanja za ponavljanje

1.Što se dobiva sa UML mehanizmima proširenja?
2.Koja su tri osnovna načina proširenja UML-a?
3.Što se podrazumijeva pod pojmom stereotip?
3.Kako se stereotipi koriste da bi se proširio UML?
4.Što je profajl?
5.Kako se stereotipi prikazuju u profajlu?
6.Navedite neki primjer upotrebe stereotipa na UML model elemente?
7.Koje je značenje označene vrijednosti?
8.Kako se označene vrijednosti prikazuju na dijagramima?
9.Što je ograničenje?
10.Kako se ograničenje prikazuje na dijagramima?
11.Navedite neki primjer ograničenja?
12.Kako se naziva jezik kojim se najčešće prikazuju ograničenja?
13.Kako UML alati omogućavaju kreiranje profajlova?
14.O čemu treba voditi računa prilikom kreiranja vlastitih profajlova?
15. Objasniti primjenu profajla na elemente paketa.
16.Što je OCL?
17.Koji je problem upotrebe prirodnog jezika za definiranje ograničenja?

18.Što je preduvjet?

19.Što je postuvjet?

20.Što je invarijanta?

21.Što je kontekst za ograničenja?

22.Koji su osnovni tipovi za OCL izraze?

23.Koji su osnovni OCL operatori?

POGLAVLJE 21.

DIZAJN PATERNI

Paterni (eng. patterns) nisu dio osnovnog UML, ali se široko primjenjuju prilikom dizajna objektno orijentiranih sistema. UML sadrži mehanizme koji se mogu koristiti za predstavljanje paterna u grafičkoj formi.

Dizajn patern je solucija za uobičajene probleme u dizajnu softverskih sistema. To je solucija koja je prepoznata kao vrijedna dokumentacija koju drugi razvojni inženjeri mogu primijeniti pri rješavanju problema. Na sličan način kako objektno orijentirana analiza i dizajn promovira ponovno korištenje biblioteka klasa i komponenti, korištenje paterna se promovira za dizajn sistema.

Dizajn patern je dakle, ponovo iskoristivo rješenje za uobičajene probleme koji se javljaju prilikom dizajna softvera. Dizajn patern nije gotovo rješenje koje se može direktno pretvoriti u kôd. To je ustvari, opis ili templejt (šablon) za rješavanje problema, koji se mogu primijeniti u mnogo različitih situacija. Objektno orijentirani dizajn paterni pokazuju odnose i djelovanja među klasama i objektima, bez definiranja krajnjeg izgleda klasa ili objekata koji se koriste u aplikaciji.

Algoritmi se ne ubrajaju u dizajn paterne, zbog toga što oni rješavaju računske probleme, a ne probleme dizajna. Ne ubrajaju se ni svi softverski paterni u dizajn paterne. Dizajn paterni se isključivo bave problemima na nivou softverskog dizajna. Ostale vrste paterna, kao što su na primjer, paterni arhitekture, opisuju probleme i rješenja koja imaju alternativna područja primjena.

Ne postoji tačno određena interpretacija što jeste, a što nije patern. U okviru problematike izložene u ovoj knjizi paterni se mogu tretirati kao opisi klasa i objekata koji međusobno komuniciraju i koji su prilagođeni da rješavaju uopćene probleme dizajna u određenom kontekstu. Dizajn patern imenuje, izvodi i identificira ključne aspekte uobičajne strukture dizajna koja je čini korisnom za kreiranje ponovno iskoristivog objektno orijentiranog

dizajna. Dizajn patern identificira klase i instance, njihove uloge, sudjelovanje, kao i raspored odgovornosti. Svaki dizajn patern se fokusira na određeni problem objektno orijentiranog dizajna.

21.1 Historija dizajn paterna

Paterni vode porijeklo od arhitektonskih koncepata Christopher Alexander-a (1977/79), koji je definirao "pattern" jezik za opisivanje uspješne arhitekture u zgradama i gradovima.

Christopher Alexander je uočio često ponavljajuće situacije u oblasti arhitekture te ih definirao kao paterne i za njih dao rješenje. Njegov rad je uticao i na dizajniranje paterna u drugim oblastima, pa i u oblasti objektno orijentirane analize i dizajna.

1987. godine Kent Beck i Ward Cunningham su počeli eksperimentirati sa idejom primjene paterna za programiranje i prezentirali su svoje rezultate na OOPSLA (Object-Oriented Programming, Systems, Languages & Applications) konferenciji te godine. Sljedećih godina Beck, Cunningham i drugi nastavili su sa ovim radom.

U augustu 1994. je održana prva „Pattern Languages of Programming" konferencija (PLoP), koja je dala veliki napredak pri uspostavi paterna za softverska rješenja.

Dizajn paterni su stekli popularnost u računarskoj nauci nakon objavljivanja knjige „Design Patterns: Elements of Reusable Object-Oriented Software" (Design Patterns: Elementi ponovo iskoristivog objektno orijentiranog softvera) početkom 1995. godine, koja sadrži osnovni katalog dizajn paterna i uspostavlja paterne kao novo područje istraživanja za softversku disciplinu. Ova knjiga je poznata i kao „Gang of Four" a paterni uvedeni kroz tu knjigu se nazivaju „Gang of Four" paterni.

Prvi paterni bili su isključivo dizajn paterni, sa težnjom da riješe probleme koji se tiču dizajna. Poslije toga su uvedene i druge vrsta paterna, paterni koji su se koriste u fazi analize i koji ustvari predstavljaju koncepte koji se koriste pri modeliranju domena; organizacijski paterni, koji predlažu rješenja za uobičajene probleme upravljanja poslovnim procesima, uključujući razvoj softvera.

21.2 Format opisivanja dizajn paterna

Kako opisati dizajn patern? Jedan od načina je grafički prikaz pomoću UML-a o čemu ćemo više diskutirati u nastavku. Ovi prikazi iako važni i korisni, nisu uvijek dovoljni jer oni prikazuju krajnji proizvod procesa dizajna kao što su odnosi među klasama i objektima. Da bi ponovo iskoristili patern u procesu dizajna, moramo zabilježiti o njemu i mnoge druge podatke. Paterni se dokumentiraju koristeći predloške, koji daju detalje o paternima i koji pomažu da se utvrdi da li se patern može primijeniti za problem. Postoji raspoloživo više predložaka. Dva najčešće korištena su Coplien&Schmidt, 1995 i Gamma, 1995. Zaglavlje većine predložaka sadrži sljedeće podatke:

1. **Naziv paterna i klasifikacija** je nešto što možemo koristiti da opišemo problem dizajna, njegova rješenja i konsekvence pomoću jedne ili dvije riječi. Imenovanje paterna povećava dizajnerski rječnik i omogućava da dizajniramo apstrakcije na višem nivou. Posjedovanje rječnika za paterne olakšava komunikaciju unutar dizajnerskog tima, pojednostavljuje dokumentaciju i ubrzava razvoj. O samoj klasifikaciji paterna će biti više u nastavku.

2. **Problem** i svrha paterna

- opisuje situacije kada treba primijeniti patern,
- objašnjava sam problem i njegov kontekst,
- opisuje specifične probleme dizajna,
- može opisati klase ili strukture objekata koje su simptomatične za nefleksibilan dizajn.

Problem će ponekad uključiti i nekoliko uvjeta koji moraju biti ispunjeni prije nego što se primjeni patern.

3. **Rješenje** opisuje elemente koji sačinjavaju dizajn, njihove odnose, odgovornosti i suradnju. Rješenje ne opisuje neki određeni dizajn ili implementaciju, jer je patern kao šablon koji se može primijeniti u mnogo različitih situacija. Umjesto toga, patern pruža apstraktni opis problema dizajna i uopćeni raspored elemenata (klasa i objekata) koji rješavaju problem.

4. **Konsekvence** su rezultati i balansiranja prilikom primjene paterna. Iako konsekvence često nisu spomenute tokom opisivanja odluka o dizajnu, one su veoma važne za procjenu alternativa dizajna i za razumijevanje cijene i koristi primjene dizajna. Softverske konsekvence se često tiču balansa između vremena i prostora. Također, mogu se ticati pitanja jezika i implementacije. Pošto je ponovna iskoristivost važan faktor u objektno orijentiranom

dizajnu, konsekvence paterna uključuju njegov utjecaj na fleksibilnost sistema, proširivost ili prenosivost. Eksplicitno navođenje ovih konsekvenci pomaže njihovom razumijevanju i procjeni.

Pored gore navedenih opisa za patern često se navode i druga poznata imena za patern, motivacija koja je dovela do uvođenja paterna, situacije u kojima je primjenjiv patern, detaljnija struktura paterna, način implementacije paterna, primjer kôda, poznate upotrebe paterna, povezani paterni.

Dokumentacija o paternima može uključiti primjere kôda i dijagrame. UML dijagrami se mogu koristiti da ilustriraju način kako patern radi. Izbor dijagrama ovisi od prirode paterna.

Dizajn paterni se mogu klasificirati kao kreacijski, paterni strukture i paterni ponašanja. Kreacijski paterni se bave procesom kreiranja objekata. Paterni strukture se bave strukturnim relacijama između instanci, posebno koristeći generalizaciju, agregaciju i kompoziciju. Paterni ponašanja se bave načinom na koji klase ili objekti međusobno djeluju i dijele odgovornost.

Za paterne strukture, dijagrami klase mogu date informacije o paternu, za paterne ponašanja dijagrami interakcije ili dijagrami stanja, za kreacijske paterne izbor dijagrama će ovisiti od prirode paterna.

U kreacijske paterne se ubrajaju: Factory, Abstract Factory, Builder, Prototype, Singleton.

U paterne strukture se ubrajaju: Adapter , Bridge, Composite, Decorator, Façade, Proxy.

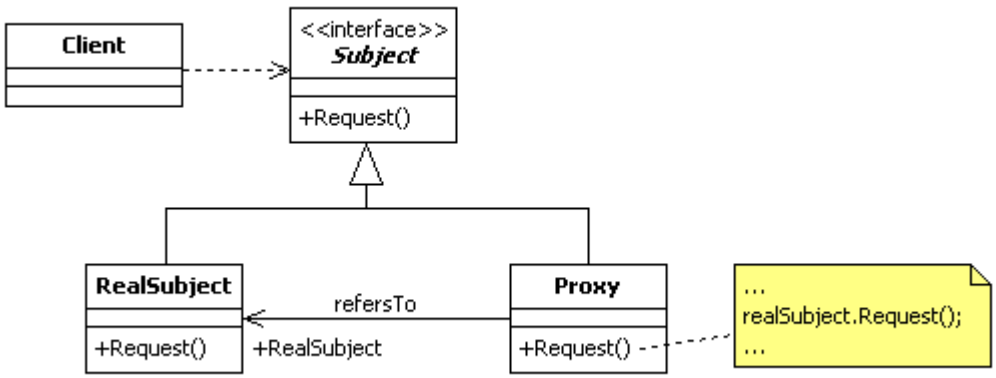
U paterne ponašanja ubrajaju se: Interpreter, Template Method, Chain of Responsibility, Command, Flyweight, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor.

Slijedi opis jednog od paterna, a to je Proxy patern. Ovaj patern je veoma jednostavan, ali zadovoljava namjenu opisivanja i demonstriranja paterna u UML-u. Ostale paterne nećemo detaljnije opisivati.

21.3 Proxy patern

Proxy patern je jedan od danih u knjizi Gang " Four's *Design Patterns".* Proxy je strukturalni patern koji odvaja interfejs od implementacije. Ideja paterna je da proxy objekt radi kao surogat objekt za stvarne objekte. On rješava probleme kada objekt ne može uvijek biti

instanciran direktno, što se može desiti u slučaju restrikcije pristupa. Slika 21.1 prikazuje dijagram klase za Proxy patern u UML-u.



Slika 21.1: Proxy patern

Sa slike je vidljivo postojanje dvije klase i jednog interfejsa Proxy paterna: `Proxy`, `RealSubject`, i `Subject`. `Client` klasa u dijagramu pokazuje korištenje paterna u ovom slučaju, ona uvijek operira sa interfejsom `Subject`. Operacija deklarirana u `Subject` klasi je implementirana i u `RealSubject` klasi i `Proxy` klasi. `RealSubject` klasa implementira operacije u interfejsu, dok `Proxy` klasa samo delegira svaki poziv koji primi `RealSubject` klasi.

Preko interfejsa `Client` klasa će uvijek raditi sa `Proxy` objektom, koji nadalje kontrolira pristup `RealSubject` objektu.

Kôd paterna je najčešće veoma jednostavan. Java i C++ kôd za `Proxy` klasu koja instancira `RealSubject` na zahtjev izgleda kao:

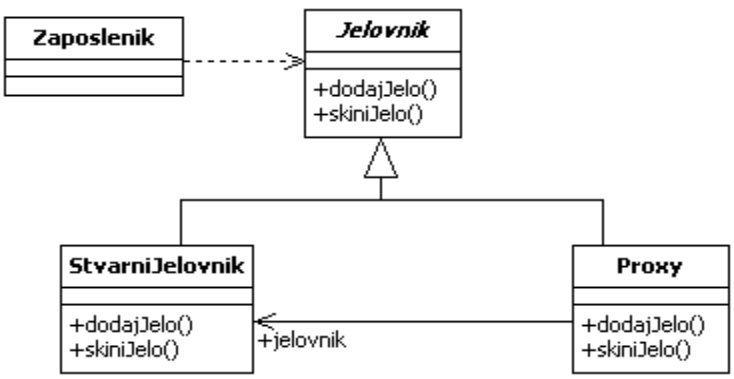
Java :
```
public class Proxy implements Subject
{
        private RealSubject realSubject;
        public void Request()
        {
                if (realSubject == null)
                        realSubject = new RealSubject();
                realSubject.Request ();
        }
}
```

C++ :
```
// Proxy.h
// ---------------------------------------
class Proxy : public Subject
{
private:
        RealSubject *realSubject;
public:
        void Request();
};

// Proxy.cpp
// ---------------------------------------
void Proxy::Request
{
        if (realSubject == null)
                realSubject = new RealSubject;
        realSubject->Request();
}
```

Implementacija Proxy paterna

Primjer primjene paterna:

Neka je u `e-restoranu` potrebno realizirati sljedeću situaciju: Objekt klase `Jelovnik` može mijenjati samo vlasnik restorana, dok ostali korisnici ne mogu. U ovom slučaju bi se mogao upotrijebiti Proxy patern da bi se napravila restrikcija pristupa.



Slika 21.2: Proxy za restrikciju pristupa - primjer

Generirani kôd za ovaj patern prikazan na slici u Javi, C++ je:

Java :
```
public class Proxy implements Jelovnik
{
	private StvarniJelovnik jelovnik;
	public void dodajJelo()
	{
		// Provjera prava pristupa
		if (imaPravo)
			jelovnik.dodajJelo();
	}
	public void skiniJelo()
	{
		// Provjera prava pristupa
		if (imaPravo)
			jelovnik.skiniJelo();
	}
}
```

C++:
```
// Proxy.h
// ------------------------------------
class Proxy : public Jelovnik
{
private:
	StvarniJelovnik *jelovnik;
public:
	void dodajJelo();
	void skiniJelo();
};

// Proxy.cpp
// ------------------------------------
void Proxy::dodajJelo
{
	if (jelovnik == null)
		jelovnik = new StvarniJelovnik;
	jelovnik->dodajJelo();
};

void Proxy::skiniJelo
{
	if (jelovnik == null)
		jelovnik = new StvarniJelovnik;
	jelovnik->skiniJelo();
};
```

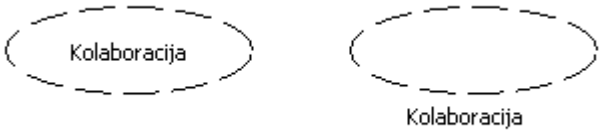
Implementacija Proxy paterna sa slike 21.2

21.4 Predstavljanje paterna UML notacijom

Notacija za kolaboraciju (suradnju)

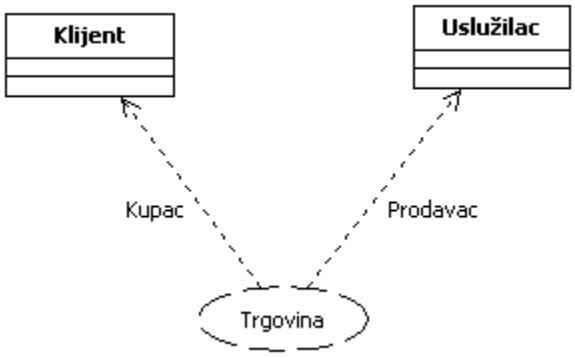
Logička arhitektura nije samo sastavljana od statičkih elemenata. Dinamička kolaboracija (suradnja) je ključni dio arhitekture sistema koja prikazuje kooperaciju pojedinih dijelova sistema.

Kolaboracija omogućava opis relevantnih aspekata suradnje i identificiranje specifičnih uloga instance. Kolaboracija se predstavlja kao isprekidana elipsa sa imenom kolaboracije unutar ili ispod elipse kao na slici 21.3.



Slika 21.3: Notacija kolaboracije

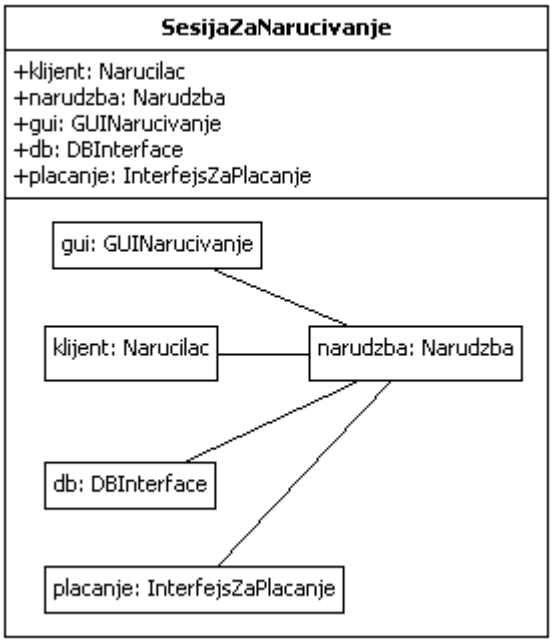
Pošto se kolaboracija ne može instancirati, kolaboracijska ikona može pokazati korištenje kolaboracije zajedno sa aktualnim klasifikatorima koji se pojavljuju u pojedinačnim primjenama kolaboracije. Slika 21.4 pokazuje kolaboraciju sa klasama i njihovim ulogama.



Slika 21.4: Uloge kolaboracije

Dijagram složene strukture omogućava mehanizam za opis veza između elemenata koji rade skupa.

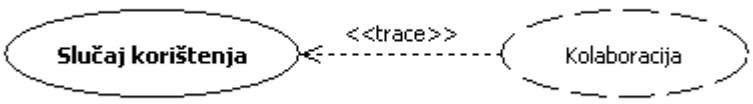
Slika ispod predstavlja jedan patern prikazan pomoću dijagrama složene strukture. To pokazuje povezanost klasa koje se koriste u klasi `SesijaZaNarucivanje`. Same za sebe, klase `Narudzba`, `Narucilac` i ostale ne znaju da predstavljaju dio strukture `SesijaZaNarucivanje`. Samo `SesijaZaNarucivanje` zna da te klase čine njenu strukturu.



Slika 21.5: Patern pomoću dijagrama kompozitne strukture

Kolaboracija i slučaj korištenja

Sljedeća notacija se koristi da pokaže relaciju između kolaboracija i slučaja korištenja koji je realizira.

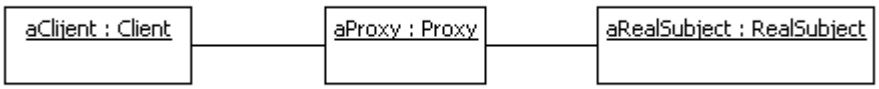


Slika 21.6: Kolaboracijska <<trace>> relacija

Notacija za paterne

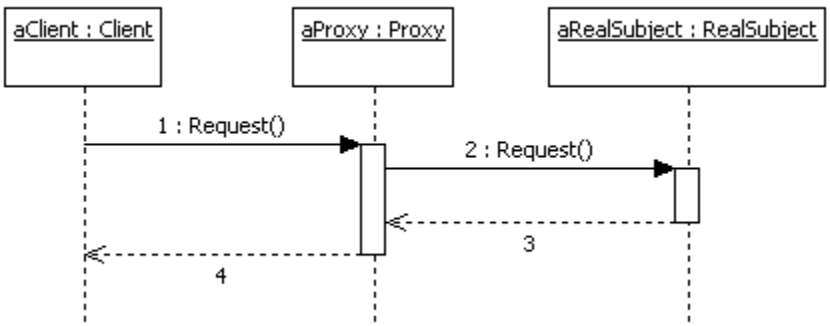
Moguće je predstaviti dizajn paterne korištenjem notacije za kolaboraciju. Kolaboracija opisuje i kontekst i interakciju. Kontekst je opis uključenih objekata u kolaboraciju, kako su povezani jedan sa drugim, i od kojih klasa su instancirani. Interakcija pokazuje komunikaciju (poruke) koju objekti izvršavaju u kolaboraciji. Patern ima obadvoje, i kontekst i interakciju i može se zadovoljavajuće opisati pomoću kolaboracije. Slika 21.6 pokazuje simbol isprekidana elipsa za patern koji je u komunikaciji. Kada se koristi tool za crtanje dijagrama, simbol se obično proširuje sa dijelovima u kojima se pokazuje i kontekst i interakcija paterna.

Dijagram objekata za Proxy patern je pokazan na slici 21.7. Da bi razumjeli u potpunosti dijagram objekata, referenca na dijagram klase mora biti raspoloživa (slika 21.1). Dijagram objekata za Proxy patern pokazuje kako `Client` objekt ima link na Proxy objekt, koji ima link na `RealSubject` objekt.



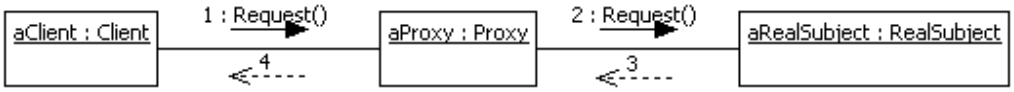
Slika 21.7: Dijagram objekata za Proxy patern

Kada se opisuje Proxy patern, dijagram interakcije pokazuje interakciju objekata koja se inicira na osnovu zahtjeva klijenta. Dijagram sekvenci na slici 21.8 pokazuje kako se zahtjev delegira `RealSubject` objektu i kako se rezultat vraća klijentu.



Slika 21.8: Dijagram sekvence za Proxy patern

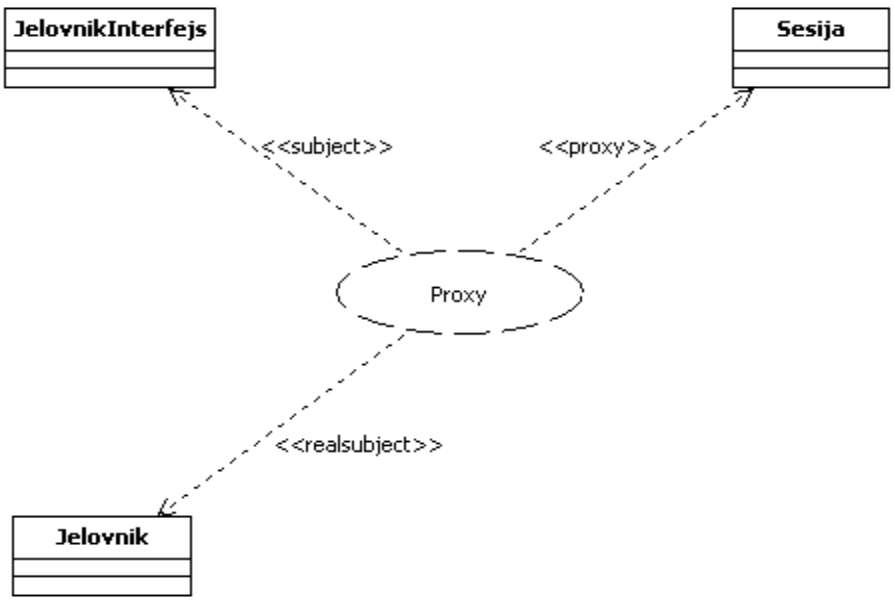
Dijagram komunikacija može pokazati obadvoje i kontekst i interakciju. Kontekst iz dijagrama objekata sa slike 21.7 i dijagrama sekvence sa slike 21.8 se kondenzira u jedan dijagram komunikacije koji je predstavljen na slici 21.9.



Slika 21.9: Dijagram komunikacija

Kompleksniji patern zahtijeva više interakcija, pa samim tim i više dijagrama interakcije, da pokaže različito ponašanje paterna.

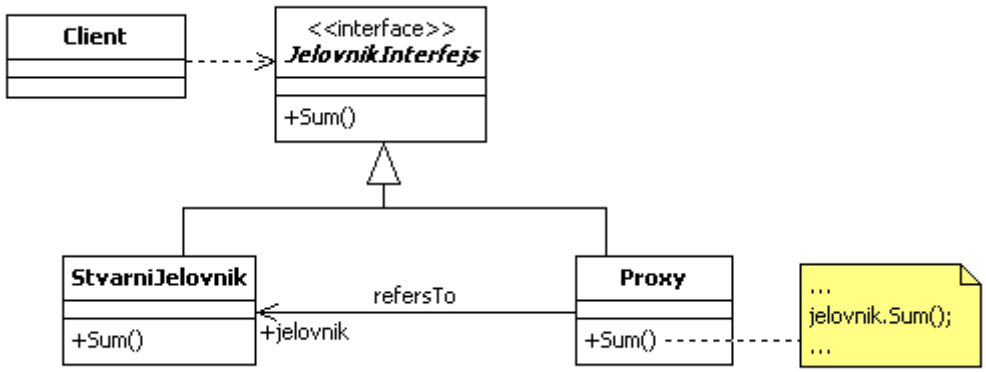
Jedan od dijagrama interakcije može pokazati kako je patern primijenjen na aktualne učesnike unutar nekog dizajna. Učesnici u dizajn paternima su aplikacijski elementi koji imaju uloge definirane u paternima; npr. za Proxy patern, to su klase koje imaju ulogu `Subject`, `Proxy`, ili `RealSubject` klase. Patern simbol i spomenuti elementi koji imaju različitu ulogu u sistemu su povezani linijom koja je označena sa imenom uloge koja opisuje ulogu koju klasa ima u dizajn paternu.



Slika 21.10: Kolaboracija Proxy patern i klasa

Slika 21.10 iznad predstavlja Proxy patern upotrijebljen na dijagramu klase, gdje klasa `JelovnikInterfejs` kao učesnik ima ulogu `Subject`, klasa `Sesija` ima ulogu `Proxy`, a klasa `Jelovnik` ima ulogu `RealSubject`.

Ako se kolaboracija proširi, tada se učesnici i njihove uloge prikazuju kao na slici 21.11.

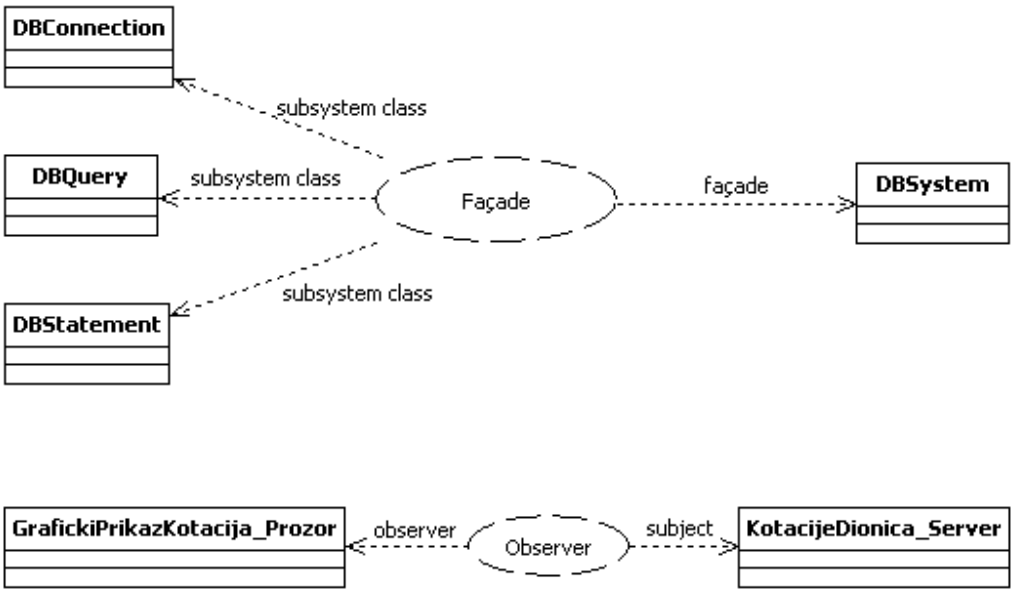


Slika 21.11: Složenija kolaboracija Proxy paterna i klasa

21.5 Prikazivanje paterna na dijagramima

Paterni definirani kao kolaboracija mogu se koristiti kao konstruktori dizajna sa većim nivoom apstrakcije nego osnovni elementi. Na taj način paterni mogu pojednostaviti model jer nisu svi dijelovi dizajna prikazani. S obzirom na to da je kontekst i interakcija paterna implicitna, patern je, ustvari, generator dizajna. Slika 21.12 pokazuje klase čiji su dijelovi konteksta i kolaboracije opisani korištenjem paterna.

U mnogim alatima za UML modeliranje moguće je pristupiti detaljima o paternu proširenjem kolaboracije. Patern se posmatra u sklopu participirajućih klasa koje imaju uloge u paternu. Alati za modeliranje se mogu koristiti i za uspostavljane novih paterna u repozitorij, koji se mogu koristiti i za druge projekte.



Slika 21.12: Facade i Observer patern

U okviru gornjeg prikaza paterna korištena su dva paterna Facade patern i Observer patern koji imaju sljedeću strukturu:

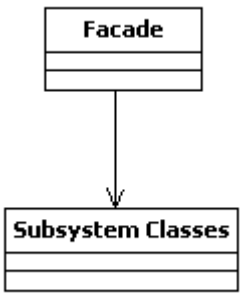
Facade patern

Svrha

Pruža jedinstven interfejs na skup interfejsa u podsistemu. Facade definira interfejs na višem nivou koji olakšava upotrebu podsistema.

Motivacija

Organiziranje sistema u podsisteme pomaže u smanjenju kompleksnosti. Uobičajeni cilj dizajna je da se minimizira komunikacija i ovisnosti između podsistema. Jedan način da se ostvari ovaj cilj je da se uvede facade (fasadni) objekt koji pruža jedinstven, pojednostavljen interfejs na ostale dijelove podsistema.



Slika 21.13: Facade patern

Primjenjivost

Facade patern koristiti kad se želi pružiti jednostavan interfejs na kompleksan podsistem. Podsistemi često postaju složeniji kako se razvijaju. Većina paterna, kada se primjeni, rezultiraju većim brojem manjih klasa. Ovo čini podsistem ponovno iskoristivijim i lakšim za prilagođavanje. Ovaj patern omogućava da se razdvoji podsistem od klijenata i drugih podsistema čime se postiže neovisnost i portabilnost podsistema.

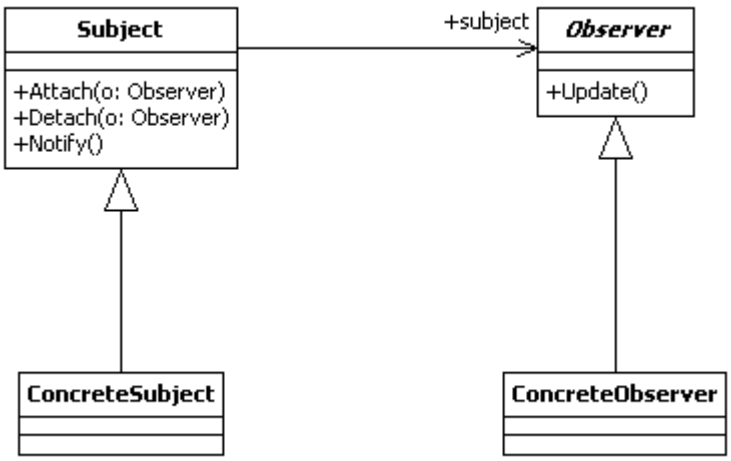
Observer

Svrha

Definiranje jedan-na-mnogo ovisnosti između objekata je tako da, kada se promijeni stanje jednog objekta, svi objekti koji ovise o njemu budu obaviješteni i ažurirani automatski.

Motivacija

Uobičajena nuspojava dijeljenja sistema u kolekciju surađujućih klasa je potreba za održavanjem konzistentnosti između povezanih objekata. Nije poželjno da se konzistentnost postigne strogim uparivanjem klasa, jer to smanjuje njihovu ponovnu iskoristivost.



Slika 21.14: Oberver patern

Primjenjivost

Koristiti Observer patern u bilo kojoj od sljedećih situacija:

- kada apstrakcija ima dva aspekta, jedan ovisan o drugom. Enkapsuliranje ovih aspekata u različite objekte omogućava njihovu zasebnu promjenu i ponovno iskorištavanje,
- kada promjena jednog objekta zahtijeva mijenjanje drugih, a ne zna se koliko objekata treba biti promijenjeno,
- kada bi objekt trebao biti u stanju da obavijesti druge objekte bez znanja ko su ti objekti.

21.6 Kako koristiti paterne

Dizajn paterni rješavaju mnoge svakodnevne probleme koje objektno orijentirani dizajneri susreću na mnogo različitih načina. Neki od problema za koje nam dizajn paterni pomažu pri pronalaženju odgovarajućih objekata, pri određivanju strukture objekta, specificiranju interfejsa i implementacije objekta.

Kako izabrati dizajn patern

Sa više od dvadeset dizajn paterna u katalogu, pronalaženje onoga koji se odnosi na određeni problem dizajna može biti teško. Ovo su neki od načina kako izabrati dizajn patern koji odgovara nekom problemu:

1. Razmotriti kako dizajn patern rješava probleme dizajna.
2. Pregledati svrhu paterna.
3. Proučiti kako su paterni povezani.
4. Proučiti paterne sa sličnom svrhom.
5.Ispitati razlog ponovnog dizajniranja i pogledati paterne koji pomažu u izbjegavanju tih razloga.
6. Razmotriti što treba biti promjenjivo u dizajnu.

Kada se izabere dizajn patern, kako se on koristi?

Slijedi pristup, korak po korak, za efektivnu primjenu dizajn paterna:

1. Analizirati patern temeljito radi općeg uvida u isti. Obratiti posebnu pažnju na sekcije o primjenjivosti i konsekvencama radi osiguranja da je patern odgovarajući za problem.
2. Vratiti se na analizu oblasti strukture, sastavnih dijelova i suradnji. Osigurati dobro razumijevanje klasa i objekata u paternu i kako se odnose jedni prema drugim.
3. Obratiti pažnju na primjer kôda da se vidi konkretan primjer paterna u kôdu. Analiziranje kôda pomaže u razumijevanju kako implementirati patern.
4. Izabrati nazive za sastavne dijelove paterna koja imaju smisla u kontekstu aplikacije. Nazivi za sastavne dijelove dizajn paterna su obično previše apstraktni da bi se pojavili direktno u aplikaciji. Međutim, korisno je uključiti te nazive u imena koja se pojavljuju u operaciji. Ovo pomaže da patern bude eksplicitniji u implementaciji. Na primjer, ako se koristi Strategy patern za algoritam kompozicije teksta, tada se klase mogu nazvati SimpleLayoutStrategy ili TextLayoutStrategy.
5. Definirati klase. Deklarirati njihove instance, uspostaviti odnose nasljeđivanja i definirati instance varijabli koji predstavljaju podatke i reference objekta. Identificirati postojeće klase u aplikaciji na koje će uticati patern i modificirati ih prikladno.

6. Definirati specifična imena za operacije u paternu u ovisnosti od same aplikacije. Koristiti odgovornosti i suradnje asocirane sa svakom operacijom kao vodič. Implementirati operacije da bi se izvršile odgovornosti i suradnje u paternu.

Pitanja za ponavljanje

1.Objasniti što se podrazumijeva pod pojmom dizajn patern?
2.Navedite historiju dizajn paterna?
3.Kako se paterni dokumentiraju?
4.Koja su tri različita tipa dizajn paterna?
5. Koja je notacija za kolaboraciju?
6.Kako se prikazuju paterni korištenjem kolaboarcije?
7.Koja je uloga dijagrama složene strukture u prikazivanju paterna?
8.Opisati Proxy patern?
9.Koji UML dijagrami se mogu koristiti za dizajn paterne?
10. Koji su kriteriji za izbor dizajn patern?
11.Kada se paterni izaberu kako se koriste?

Literatura

Ambler, W. Scott., *The Elements* of UMLTM*2.0 Style,* Cambridge University Press, 2005

Ambler W. Scott., *The Object Primer*, Cambridge University Pres, 2004

Arlow J., Neustadt I., *UML2 and Unified Process: Practical Object-Oriented Analysis and Design*, Addison-Wesley Object Technology Series

Barclay K., Savage J., *Object-Oriented Design with UML and Java,* Elsevier, Ltd, 2004

Bennett, S., McRobb, S&Farmer,R., *Object-Orientied Systems Analysis and Desing using UML,* McGraw-Hill, 2002

Booch, G. *Object-Oriented Analysis and Design with Applications* Benjamin Cummings, 1994.

Booch, G., Rumbaugh J., Jacobson,I.. *The Unified Modeling Language User's Guide*. Addison-Wesley, 1999

Coplien, J.O., Advanced C++:Programming Style and Idioms, Addison-Wesley, 1992

Eriksson, H., Penker M., Lyons B., Fado D., *UML™ 2 Toolkit*, Wiley Publishing 2003
Fowler,M., *UML Distilled - A Brief Guide to the Standard Object Modeling Language*, Addison Wesley, 2003

Fowler,M., Scott ,K., *UML Distilled Second Edition A Brief Guide to the Standard Object Modeling Language*, Addison Wesley ,1999

Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*,Reading, MA: Addison-Wesley,1995

Graessle P., Baumann H.,Baumann P. ,*UML 2.0 in Action: A project-based tutorial: A detailed and practical walk-through showing how to apply UML to real world development projects*, Packt Publishing, 2005

Hamilton K., Miles R., *Learning UML 2.0,* , O'Reily 2006

Larman C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, Addison Wesley Professional,2004

McLaughlin, Pollice G., West D., *Head First Object-Oriented Analysis and Design*, O'Reilly, 2007

O'Docherty M. ,*Object-Oriented Analysis and Design Understanding System Development with UML 2.0*, John Wiley & Sons Ltd, 2005

Object Management Group (2003a), *OMG UML 2.0 OCL Specification*, Technical Report, Object Management Group

Object Management Group (2004a), *OMG Unified Modeling Language 2.0 Specification*, Technical Report, Object Management Group

Pilone D.,Pitman N., *UML 2.0 in a Nut*shell, O'Reilly, 2005

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and F. Lorenson. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

Shoval, P, *Functional and object oriented analysis and design : an integrated methodology*, Idea Group Publishing, 2007Van Vliet H.,*Software Engineering - Principles and Practice*, 2-nd Edition. John Wiley and Sons, Chichester, England, 2000

Wixom, D., Wixom B., Tegarden D., *Systems Analysis and Design with UML Version 2.0: An Object-Oriented Approach* , , John Wiley & Sons Ltd, 2005

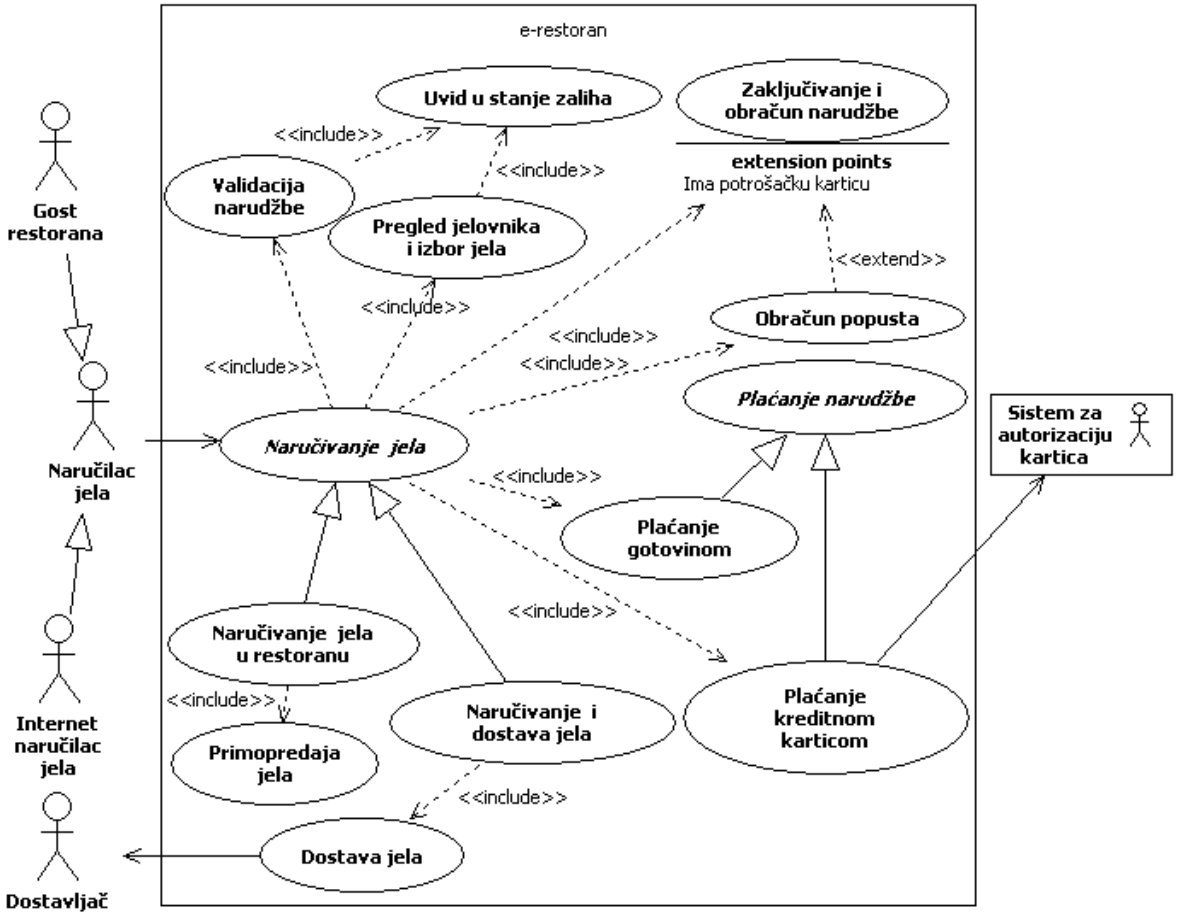
On-line izvori:

http://www.omg.org/

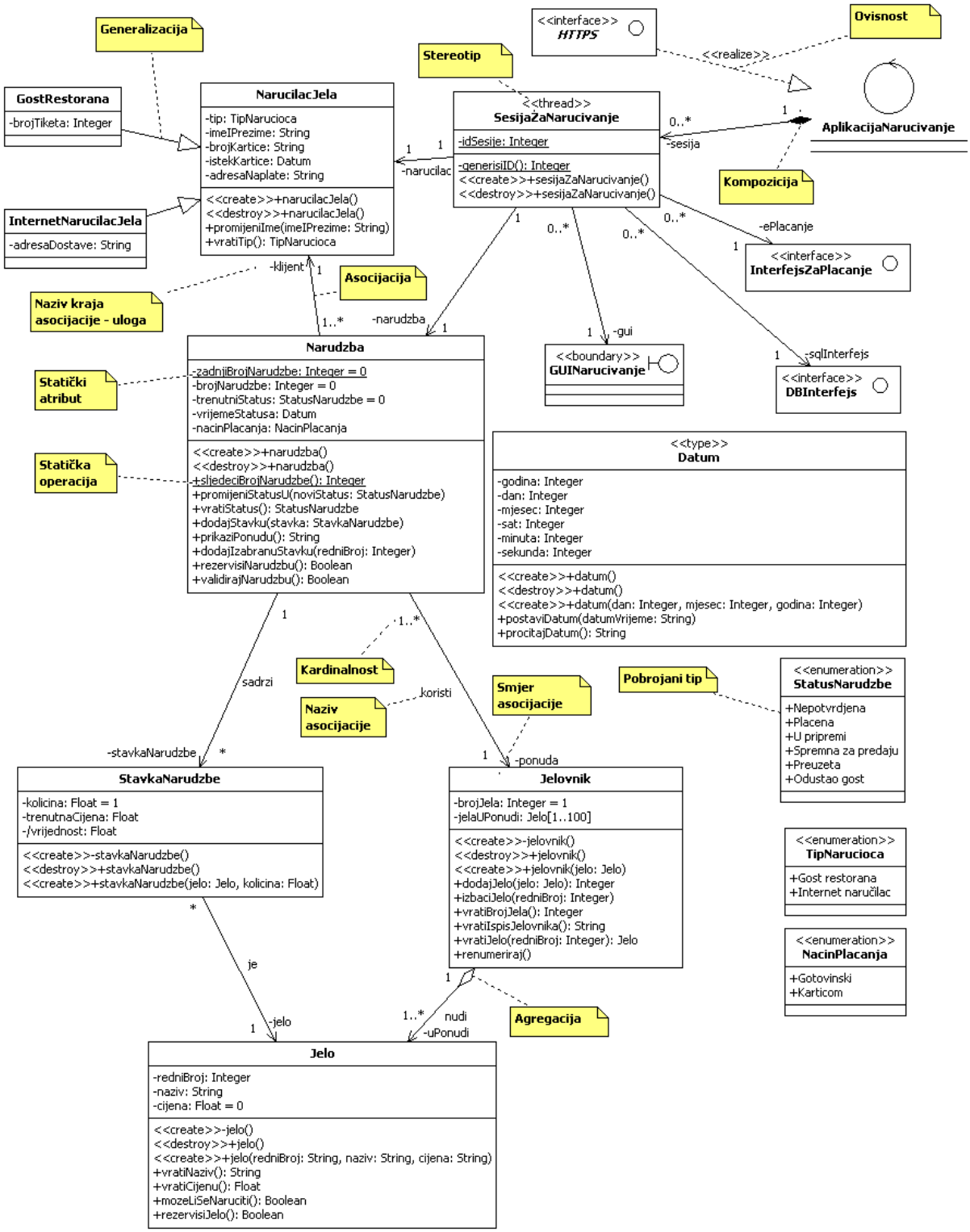
http://www.uml.org

http://www.ooad.org

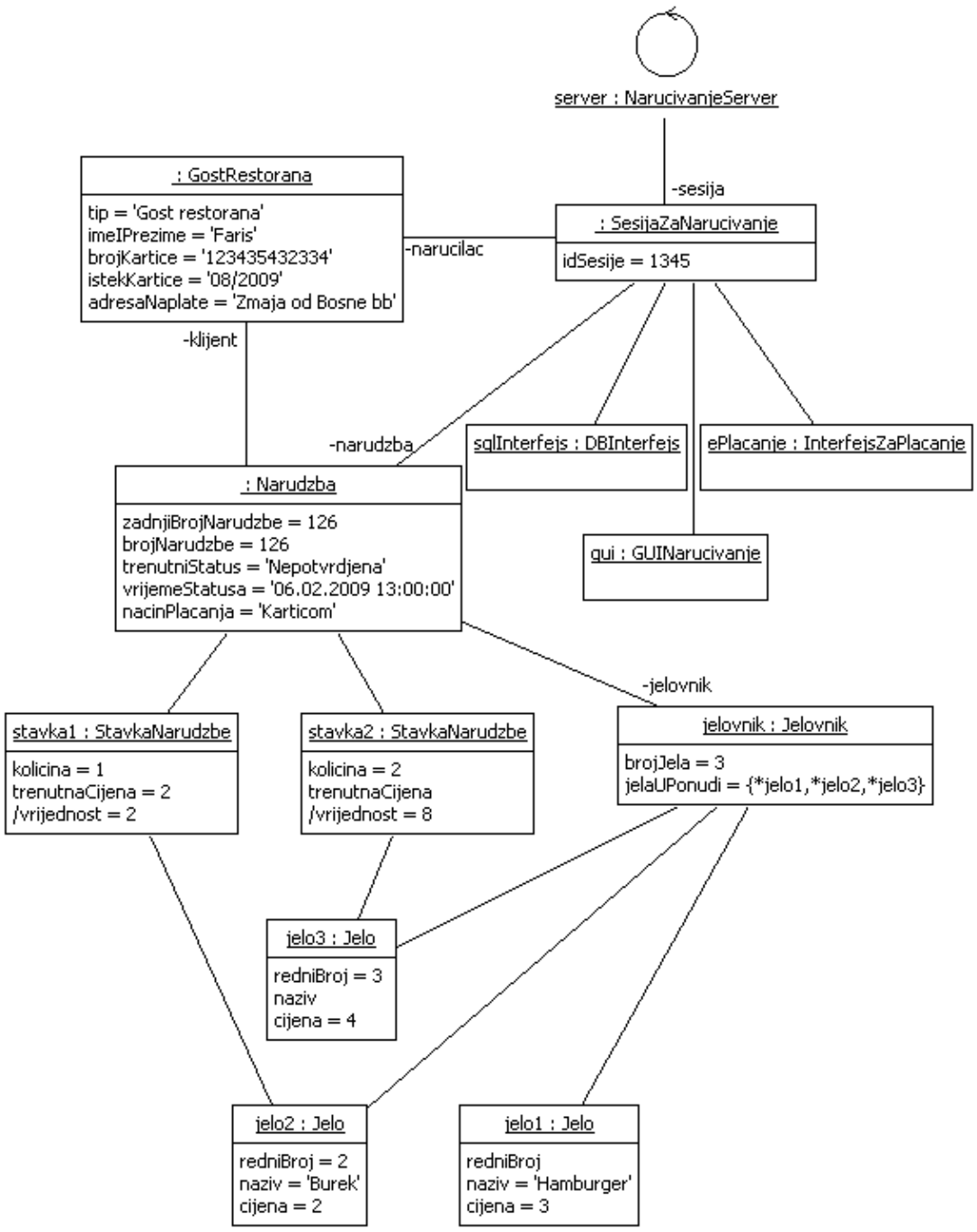
PRILOG – `e-restoran` UML dijagrami



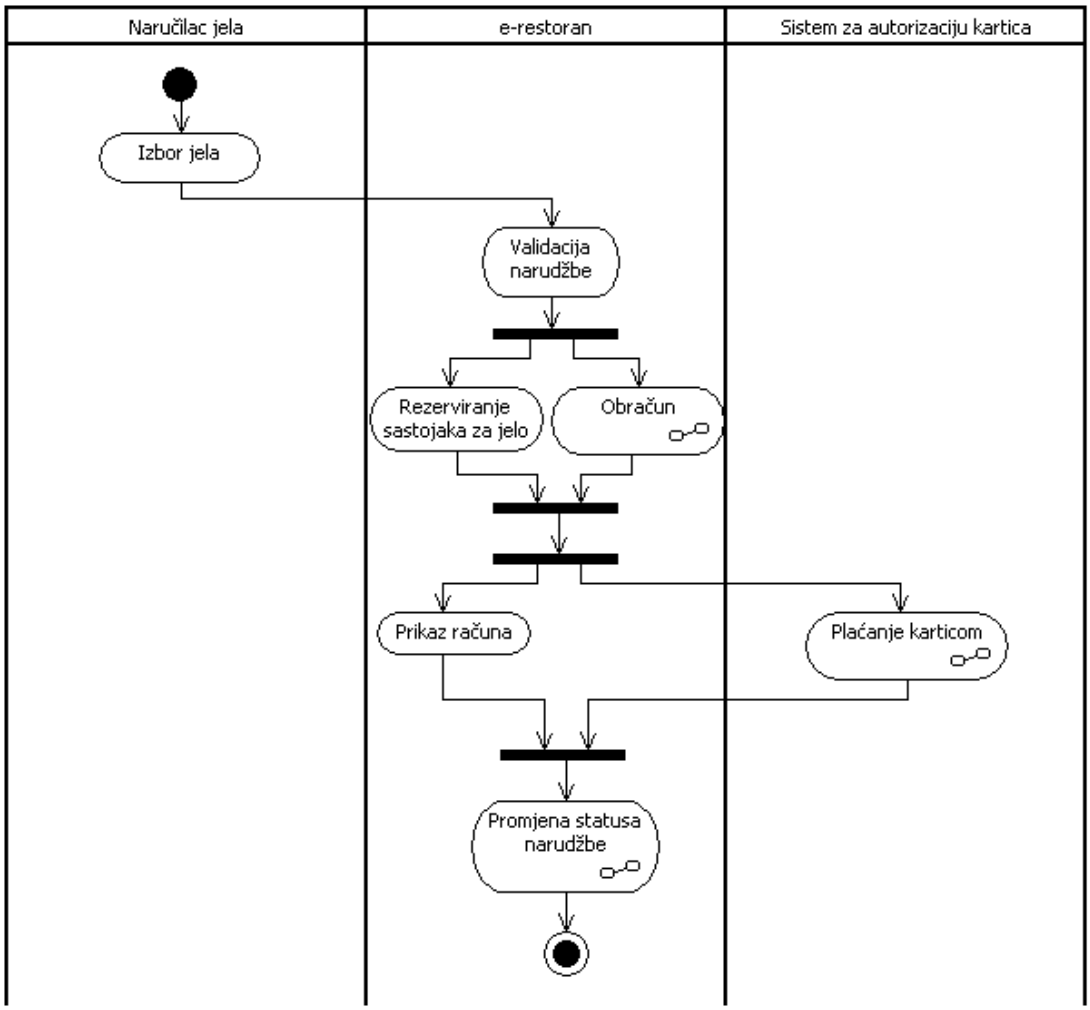
Dijagram slučajeva upotrebe



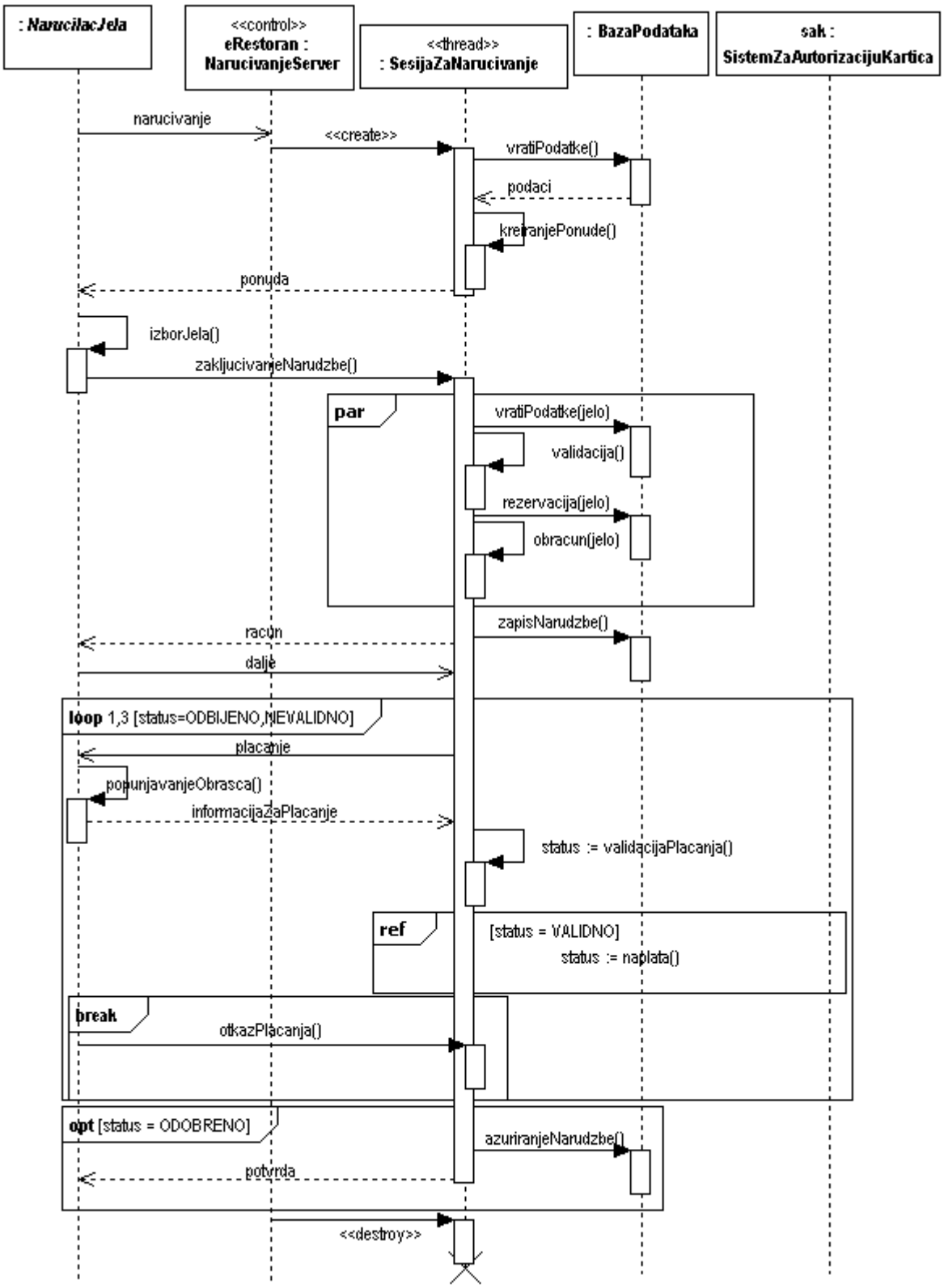
Dijagram klasa



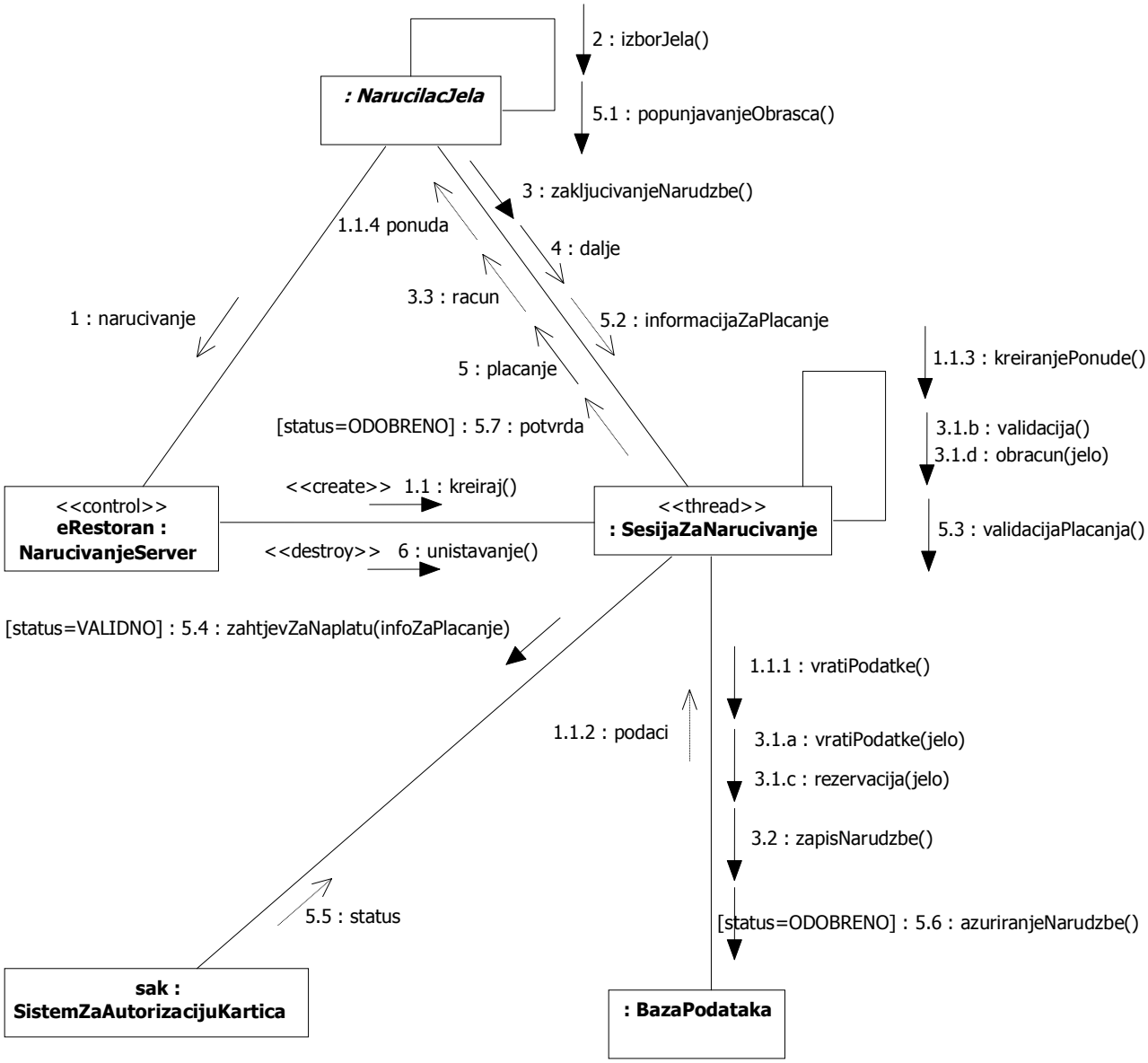
Dijagram objekata



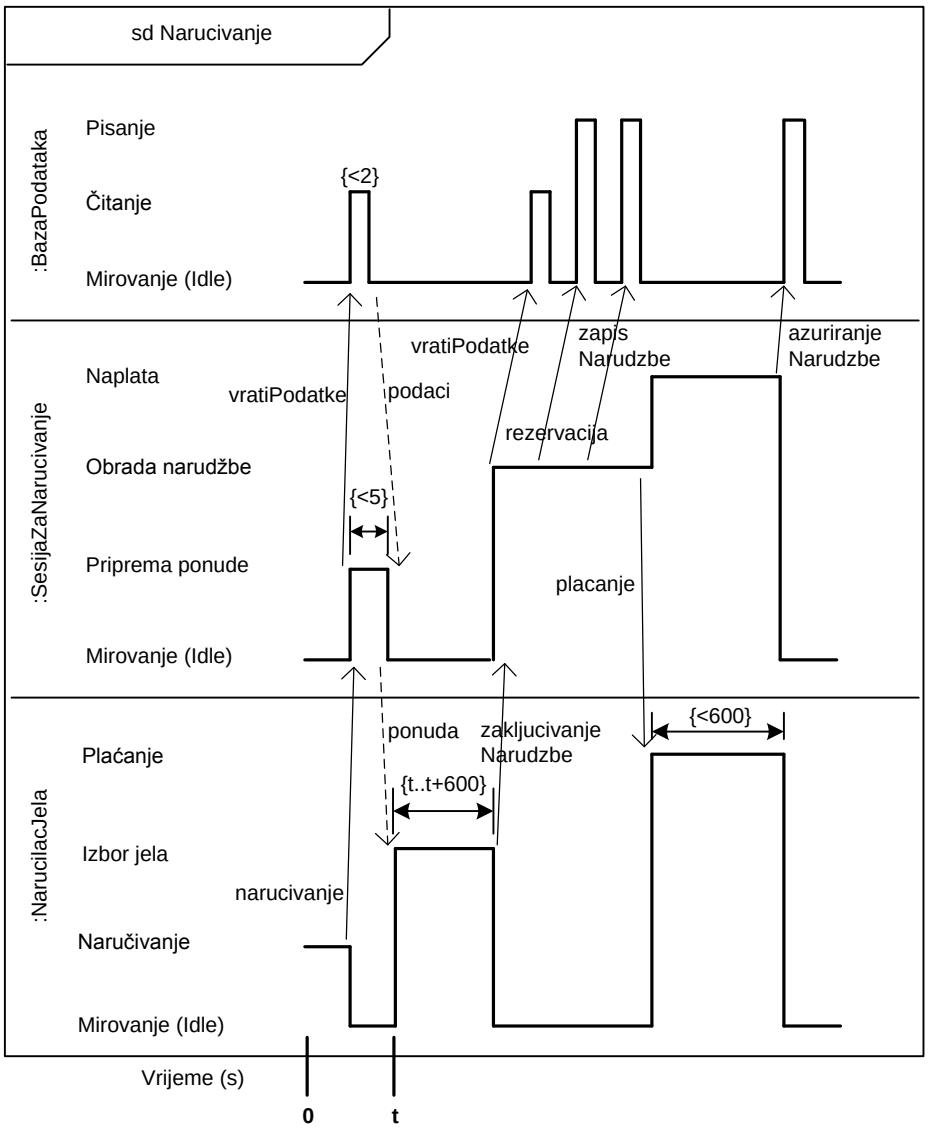
Dijagram aktivnosti



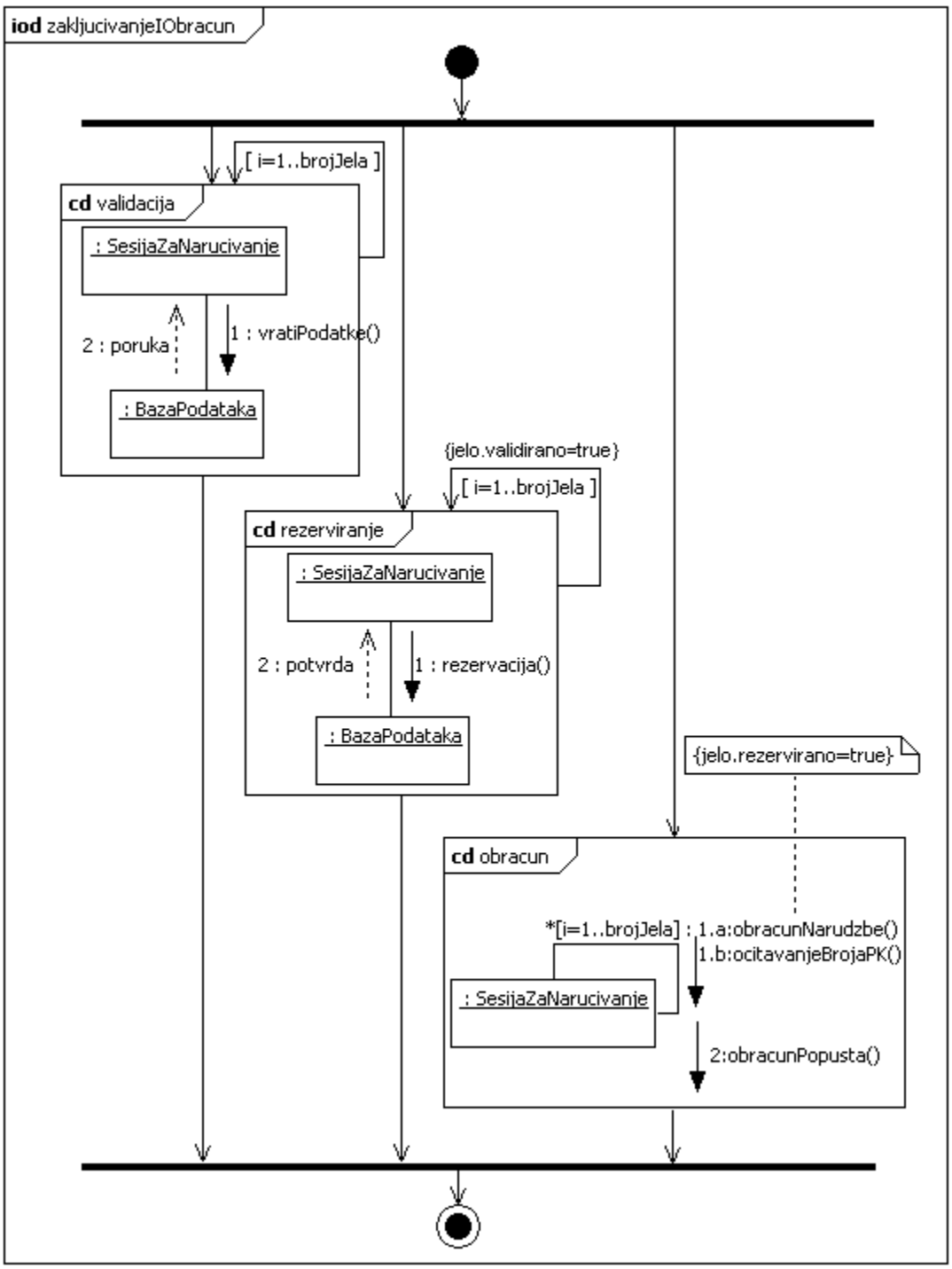
Dijagram sekvence



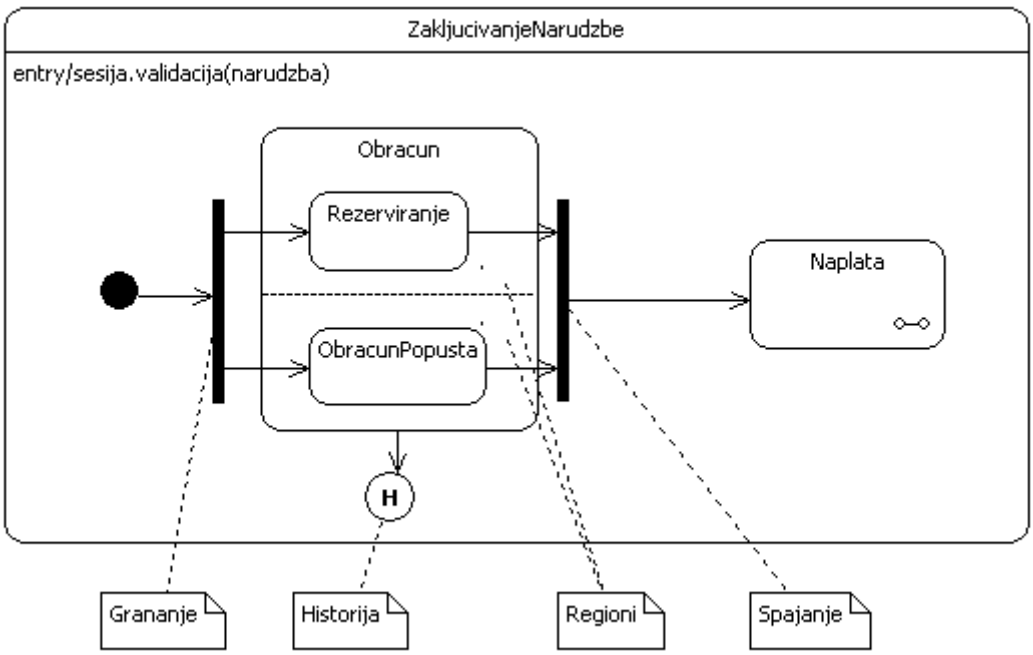
Dijagram komunikacije



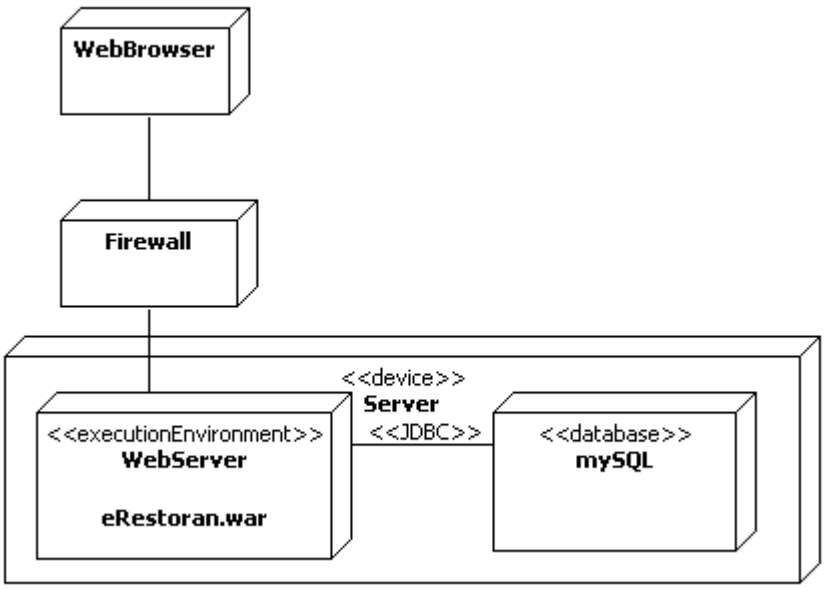
Dijagram toka vremena



Dijagram pregleda interakcije



Dijagram stanja



Dijagram raspoređivanja

INDEX

A
agregacija, 78
akcija, 125, 134, 164
akter, 19, 33, 257
aktivnost, 125
apstrakcija, 4, 117
apstraktna klasa, 87, 90
artefakt, 235, 263, 265
<<artifact>> , 233, 263
asinhrona poruka, 151, 153
asocijacija, 11, 32, 66
atributi, 50, 54, 57, 72, 103, 120, 287

C
<<component>>, 239
<<concurrent>>, 141
C++, 1,5,9,52,58,76,83,89,90,152,244,246,254,271, 282,295, 297

D
dijagram aktivnosti, 13,124
- aktivnost i akcija, 125
- čvor grananja, 130
- čvor odluke, 128
- čvor spajanje, 130
- čvor stapanja, 128
- objekti u aktivnosti , 132
- oblast primjene, 141
- particija, 135
- početni, krajnji čvor aktivnosti , 126
- podaktivnost, 134
- područje prekida, 139
- signal, 137
- specifikacija spajanja, 143
- tokovi i ivice,127

dijagrami interakcije, 14, 145
dijagram klase, 12, 48
- agregacija,78
- apstraktna klasa, 87
- asocijacija, 66
- atributi klase, 50
- generalizacija, 80, 82

interfejsi, 90
<<interface>>, 90
kardinalnost, 53
klasa, 50
klasa asocijacije, 75
kompozicija, 80
operacije klase, 55
osobine, 59
ovisnost, 84
parametirizirane klase, 93
statički atributi i operacije, 57
dijagram komunikacije, 15,170
istovremene poruke,
linkovi , 171
učesnici, 171
poruke, 172
poruke koje se ponavljaju, 174
poruke u isto vrijeme,173
poruke sa uvjetom,174
poruka samom sebi,175
dijagram komponenti, 13, 229
delegacijski konektori, 238
interna struktura, 238
komponenta, 230
konektori, 238
konektori spajanja, 239
krug i polukrug notacija za interfejs, 231
listing notacija za stereotip, 233
notacija stereotipa za interfejs, 232
pogled na komponente-bijela kutija, 240
pogled na komponente-crna kutija,240
ponuđeni interfejs, 231
port, 237
realizirajuće klase, 235
zahtijevani interfejs,231
dijagram objekata, 13, 102
anonimni objekt , 103
atributi objekta, 103
instance objekta, 102
link, 104
ograničenja, 105
dijagram paketa,13,244
<<access>>, 253
<<import>>, 252
klase u paketu, 248
ovisnost paketa, 254, 255
paket, 245
uvoz paketa, 252
varijacije prikaza paket, 248

dijagram pregleda interakcija, 15, 203
dijagram sekvence, 14, 146
alt, 164
asinhrone poruke, 153
brisanje učesnika, 154
<<create>>, 154
<<destroy>>, 154
događaji, 149
fragmenti, 162, 165
kreiranje učesnika, 154
loop, 163
opt, 163
par, 164
poruke, 149
povratne poruke, 154
refleksivne poruke, 157
sd, 164
signali, 149
sinhrone poruke, 151
trake aktivnosti, 150
učesnici, 147
ugniježđene poruke, 156
vrijeme, 148
dijagram složene strukture, 13,222
interna struktura, 223
konektori, 225
osobine, 224
port, 225
dijagram slučajeva upotrebe, 14, 31
akter, 33
asocijacija, 38
<<extend>> , 39
generalizacija, 40
granice sistema, 36
<<include>>, 38
proširene veze, 39
relacije, 38
sadržajne veze, 38
scenarij, 43
slučaj upotrebe, 35
dijagram stanja, 15, 209
do , 216
događaj, 213
entry, 216
exit, 216
grananje, 218
inicijalno stanje, 211
interni prijelazi, 216
interno ponašanje, 215

izraz, 214
kompozitna stanja, 217
krajnje stanje, 211
napredna pseudostanja, 218
operacija, 214
prijelaz, 212
pseudostanje, 211
regioni, 217
signali, 219
spajanje, 218
stanje,211
stanje historije,220
dijagram toka vremena,15, 185
alternativna notacija,199
događaji, 189
linija stanja, 188
poruke, 189
stanja, 186
učesnici, 186
vremenska ograničenja, 190
vrijeme, 187
dijagram raspoređivanja, 13, 269
artefakt, 264
<<artifact>>, 264
čvor, 261
<<device>>, 262
instance čvora, 261
komunikacijski put, 261
specifikacija raspoređivanja, 266
dizajn patern, 170, 223, 291
događaj 15, 20, 43, 137, 140, 149, 186,189, 198, 213

E
enkapsulacija, 4,111
<<enumeration>>, 97
<<extend>>, 32,39

G
generalizacija, 11, 15, 80

H
historija stanja 120
hijerarhija, 5, 119, 248

I

<<include>>, 32, 38, 274
instance, 58, 66,74,79,91,102,103
interakcija, 5, 14, 31, 43, 110, 145

<<interface>>, 91, 232
interfejs, 1, 3, 20, 89, 90, 91, 112
invarijante, 287

J

Java, 1,5,9,52,58,76,83,89,90,152,244,246,254,271, 282,295, 297

K
kardinalnost,51, 53, 70
klasa, 3, 12, 22, 49, 50, 60, 64, 112,224,229
kolaboracija, 22, 168, 298
komentar, 10, 47, 54
komponenta, 24, 229, 230
kohezivnost, 112
Krutchten 4+1 pogled, 19
kvalitet dizajna, 111

L
linija života, 147
link, 104, 170
loop, 163

M
mašina stanja, 210
metaklasa, 274
metamodel, 272
metod, 3, 55
metoda razvoja softvera, 8, 9, 25
metrika objektno orijentiranog dizajna, 119
broj djece, 119
dubina drveta nasljeđivanja, 119
faktor međusobne ovisnosti, 121
faktor nasljeđivanja atributa, 120
faktor nasljeđivanja metoda, 120
faktor polimorfizma, 121
faktor skrivanja atributa, 120
faktor skrivanja metoda, 120
međusobna ovisnost između objekata klasa, 119
model, 2,17
modularnost, 4

N
namespace, 246
navigabilnost, 68

O
Objekt, 13, 103, 112, 120, 132, 146, 211, 221

objektno dizajnirani sistem, 5
objektno orijentirana analiza i dizajn, 2, 22, 60
objektno orijentirano programiranje, 2
OCL Object Constraint Language,96, 106, 129, 281
ograničenja, 15, 17, 95, 106, 129, 281
OMG Object Management Group, 8,10,271
operacija,5, 55, 62, 64, 87, 112, 125, 146, 166, 184, 214, 221

P
paket, 10, 245, 247
paralelne aktivnosti,
parametizirane klase, 130, 143
patern, 7, 170, 233, 291
pattern, 291
particija, 136
podsistem,35, 49, 146, 148, 185, 210, 230
pogled , 19
implementacijski,20
logički, 20
procesni, 20
razvojni, 21
slučaja upotrebe, 19
poruka, 2, 35, 150, 172, 189
potklasa, 92, 119
povratna poruka,154
prijelaz, 186, 200, 216
principi dobrog dizajna, 113
Liskov princip zamjene, 116
princip inverzije ovisnosti, 117
princip izoliranja interfejsa, 118
princip pojedinačne odgovornosti, 113
otvoreno zatvoren princip,115
profajl,21, 271, 272
<<profile>>, 21, 271
<<provided interface>>, 231,274
pseudostanje, 211

R
region, 141, 165, 217
<<required interface>>, 274

S
spajanje, 130, 218
stanje , 211
stapanje, 129
stereotip, 15, 33, 38, 57, 91, 96, 154, 230, 252, 261, 273
<<subsystem>>, 231

T
templejt, 93
<<type>>,98

U
ugniježđene poruke, 156
UML Unified Modeling Language
UP Unified Process, 23

V
vrijeme, 148, 187