Univerzitet Singidunum

Specifikacija i modelovanje softvera

Angelina Njeguš



Beograd, 2026.

UNIVERZITET SINGIDUNUM

Angelina Njeguš

SPECIFIKACIJA I MODELOVANJE SOFTVERA

Prvo izdanje

Beograd, 2026.

SPECIFIKACIJA I MODELOVANJE SOFTVERA

Autor:
prof. dr Angelina Njeguš
Redovni profesor Univerziteta Singidunum

Recenzenti:
prof. dr Mladen Veinović
Redovni profesor Univerziteta Singidunum

prof. dr Marko Tanasković
Redovni profesor Univerziteta Singidunum

Izdavač:
Univerzitet Singidunum
Danijelova 32
www.singidunum.ac.rs

Za izdavača:
prof. dr Milovan Stanišić

Priprema za štampu:
Miloš Višnjić

Dizajn korica:
Aleksandar Mihajlović, MA

Godina izdanja:
2026.

Fotografija korica:
Cover designed by Freepik / rawpixel.com

Tiraž:
800

Štampa:
Birograf, Beograd **ISBN:** 978-86-7912-857-7

Copyright © 2026 Univerzitet Singidunum

Izdavač zadržava sva prava.
Reprodukcija pojedinih delova ili celine ove publikacije nije dozvoljena.

Ovu knjigu posvećujem višim silama i veri koja me je naučila smirenju.

Sadržaj

PREDGOVOR **1**

1. UVOD U SOFTVERSKO INŽENJERSTVO **7**
1.1. KRATAK ISTORIJSKI OSVRT RAZVOJA SOFTVERSKOG INŽENJERSTVA 7
1.1.1. Pojam i evolutivni razvoj softverskog inženjerstva 7
1.1.2. Poslovni sistemi 10
1.2. VRSTE APLIKATIVNOG SOFTVERA 11
1.3. INFORMACIONI SISTEMI: POJAM I PODELA 16
1.3.1. Pojam sistema 16
1.3.2. Podatak, informacija, znanje i mudrost 18
1.3.3. Šta je informacioni sistem? 19
1.3.4. Podela informacionih sistema 21
Operacioni informacioni sistemi 23
Analitički informacioni sistemi 23

2. MODELOVANJE SOFTVERA **29**
2.1. UVOD U MODELE I MODELOVANJE 29
2.2. VRSTE MODELOVANJA SOFTVERA 31

3. METODOLOGIJE RAZVOJA SOFTVERA **37**
3.1. UVOD U METODOLOGIJE RAZVOJA SOFTVERA 37
3.1.1. Evolutivni razvoj metodologija 38
3.1.2. Zašto kompanije koriste metodologije? 42
3.2. TRADICIONALNE METODOLOGIJE RAZVOJA SOFTVERA 43
3.2.1. *Microsoft Solution Framework* 43
Upravljanje rizicima 44
Upravljanje timom 45
Upravljanje procesima 46
3.2.2. *Rational Unified Process* metodologija 47
3.3. AGILNE METODOLOGIJE RAZVOJA SOFTVERA 51
3.3.1. Scrum 52
Role prema Scrum okviru 53
Sastanci 54
Artefakti 57
3.3.2. Kanban 64
Nastanak i osnove LEAN principa 66
Koncept i pravila primene Kanban metode 67
Primena Kanban metode u razvoju softvera 68
Kanban vs Scrum 72

3.3.3. Dizajnersko razmišljanje 73
Koncept i faze design thinking okvira 75
Dizajn ili „Outside the Box" razmišljanje 78
DT kao osnova za dizajniranje korisničkog iskustva 81

4. MODELOVANJE PROCESA 87
4.1. UVOD U MODELOVANJE PROCESA 87
4.1.1. Poslovni procesi: pojam i vrste 89
4.1.2. Tehnike za prikupljanje informacija 90
4.2. MODELOVANJE POSLOVNIH PROCESA: POJAM I ZNAČAJ 91
4.3. TEHNIKE MODELOVANJA PROCESA 93
4.3.1. Institucije i organizacije relevantne za standardizaciju modelovanja poslovnih procesa 93
4.3.2. Dijagram toka podataka 96
4.3.3. IDEF0 103
4.3.4. Modelovanje i notacija poslovnih procesa - BPMN 105
Osnovni tipovi BPMN modela 106
Osnovni elementi BPMN notacije 107

5. MODELOVANJE PODATAKA 123
5.1. UVOD U MODELOVANJE PODATAKA 123
5.1.1. Osnovni pojmovi modela podataka 124
Objekat 124
Atribut 125
Primarni ključ 125
Strani ključ 128
Povezivanje objekata 129
Generalizacija i specijalizacija 133
5.1.2. Postupak modelovanja podataka 134
5.2. ANALIZA MODELA PODATAKA PRIMENOM NORMALIZACIJE 135
5.2.1. Prva normalna forma 137
5.2.2. Druga normalna forma 140
5.2.3. Treća normalna forma 143
5.2.4. *Boyce-Coddova* normalna forma 145
5.2.5. Četvrta normalna forma 149
5.2.6. Peta normalna forma 152
5.2.7. Normalna forma ključeva i domena 154
5.3. PRIMER MODELOVANJA PODATAKA 158

6. SPECIFIKACIJA I MODELOVANJE KORISNIČKIH ZAHTEVA 165
6.1. SPECIFIKACIJA ZAHTEVA UPOTREBOM UML-A 165
6.1.1. Funkcionalni zahtevi 166
6.1.2. Nefunkcionalni zahtevi 167
6.2. DIJAGRAM SLUČAJEVA UPOTREBE 168
6.2.1. Osnovni elementi use case dijagrama 169
6.2.2. Kreiranje use case dijagrama 172
Identifikovanje aktera 173
Identifikovanje use case-ova 174
6.3. DIJAGRAM AKTIVNOSTI 186
6.4. DOKUMENT SPECIFIKACIJE SOFTVERSKIH ZAHTEVA 198

7. OBJEKTNO-ORIJENTISANA ANALIZA SOFTVERA 203
7.1. ARHITEKTURA SOFTVERA 203
7.1.1. Stilovi i obrasci arhitekture softvera 205
7.1.2. Popularni obrasci arhitekture softvera 207
Model View Controller obrazac arhitekture 208
7.2. DIJAGRAMI INTERAKCIJA 212
7.2.1. Dijagram sekvenci 212
Koraci kreiranja dijagrama sekvenci u IBM RSA alatu 214
Dodatne opcije na dijagramu sekvenci 226
Značenje poruka za klase 230
Proširenі dijagram sekvenci primenom savremene MVC arhitekture 233
7.2.2. Dijagram komunikacije 235
7.3. DIJAGRAM KLASA 238
7.3.1. Osnovni elementi dijagrama klasa 238
7.3.2. Vrste relacija i zavisnosti između klasa u UML dijagramima 245
Asocijacija 246
Usmerena asocijacija 246
Agregacija 247
Usmerena agregacija 248
Kompozicija 249
Klasa asocijacije 249
Generalizacija 251
Realizacija 253
Interfejs realizacija 254
Substitucija 254
Zavisnost 254
Elementi za rad sa paketima 258
7.3.3. Koraci modelovanja dijagrama klasa 261

8. OBJEKTNO-ORIJENTISANO PROJEKTOVANJE SOFTVERA 273
8.1. UVOD U PROJEKTOVANJE SOFTVERA 273
8.2. DIJAGRAM STANJA 276
8.2.1. Osnovni elementi dijagrama stanja 277
8.2.2. Identifikovanje stanja, događaja i tranzicija 279
8.2.3. Aktivnosti i akcije 280
8.2.4. Vrste događaja 283
8.2.5. Kompozitna i paralelna stanja 286
8.2.6. Povezanost dijagrama stanja i dijagrama sekvenci 288
8.2.7. Tipične greške u kreiranju dijagrama stanja 293
8.3. DIJAGRAM KLASA SA PRIMENJENIM OBRASCIMA PROJEKTOVANJA SOFTVERA 302
8.3.1. Primena obrazaca projektovanja softvera 303
Razumevanje obrazaca projektovanja 307
Primena State obrasca na dijagramu klasa u IBM RSA alatu 308
8.4. DIJAGRAM KOMPONENTI 316
8.4.1. Pojam komponente 316
8.4.2. Relacije između komponenti 319
Relacija zavisnosti 320
Primer relacija između komponenti sistema onlajn prodavnice 321
8.4.3. Osnovni elementi dijagrama komponenti 322
8.4.4. Dijagram kompozitne strukture 325
8.5. DIJAGRAM RASPOREĐIVANJA 338
8.5.1. Osnovni elementi dijagrama raspoređivanja 339
Čvor 339
Izvršno okruženje 340
Artefakt 341
Specifikacija razmeštanja 343
8.5.2. Konekcije i relacije na dijagramu raspoređivanja 346
8.5.3. Koraci u modelovanju distribuiranosti sistema 347

9. SERVISNO-ORIJENTISANO MODELOVANJE SOFTVERA 361
9.1. UVOD U SERVISNO-ORIJENTISANE KONCEPTE I TEHNOLOGIJE 361
9.1.1. Evolucija od monolitne do servisno orijentisane arhitekture 361
9.1.2. Uloge u servisno-orijentisanom modelu 363
9.1.3. SOA životni ciklus razvoja sistema 365
9.1.4. Tehnološki stubovi za implementaciju servisno-orijentisanih sistema 367
Veb servisi: definicija i karakteristike 367
Tehnički standardi i protokoli u SOA arhitekturi 369
Taksonomija i klasifikacija servisa u SOA sistemima 373
Razlozi za primenu i značaj veb servisa u poslovanju 374
9.2. SERVISNO-ORIJENTISANO MODELOVANJE SOFTVERA 376
9.2.1. Standardizacija modelovanja poslovnih procesa 377
9.2.2. Koraci servisno-orijenisanog modelovanja sistema 379

9.3. MODELOVANJE SERVISNO-ORIJENTISANE INTEGRACIJE SOFTVERA 384
9.3.1. Uvod u servisno orijentisanu integraciju 384
9.3.2. Arhitektura i osnovne funkcije magistrale poslovnih servisa - ESB 386
9.3.3. Obrasci posredovanja u ESB arhitekturi 388
9.3.4. Od modela do izvršenja: Uloga orkestracije 390
Jezik za izvršavanje poslovnih procesa 391
Ključni elementi BPEL modela 391
Savremeni alati i platforme za orkestraciju procesa 396
9.4. MIKROSERVISNA ARHITEKTURA SOFTVERA 397
9.4.1. Uvod u mikroservisnu arhitekturu 397
9.4.2. Od klasičnih veb servisa do mikroservisa 398
9.4.3. Evolucija od SOA životnog ciklusa do *DevOps* ciklusa 399
9.4.4. Metodološki pristup modelovanju mikroservisne arhitekture 403
9.4.5. Proces projektovanja mikroservisa 405
Operativne faze projektovanja 406
9.4.6. Studija slučaja: Mikroservisna arhitektura sistema za dostavu dronom 409
Analiza domena 409
Definisanje graničnih konteksta 410
Primena taktičkog DDD-a: Unutrašnji dizajn mikroservisa 411
Modelovanje agregata i koordinacija 412
Identifikovanje granica mikroservisa 413

LITERATURA 418

PRILOG 421

SPISAK VEŽBI 434

SPISAK ZADATAKA 435

Predgovor

Udžbenik „Specifikacija i modelovanje softvera“ je prevashodno namenjen studentima Tehničkog fakulteta i Fakulteta za informatiku i računarstvo Univerziteta Singidunum, ali i svima onima koji svoju karijeru žele da grade u pravcu razvoja softvera, a posebno u ulogama arhitekte softvera, poslovnog analitičara ili projektanta sistema. Cilj ovog udžbenika je da pomogne studentima u razvijanju sistemskog razmišljanja prilikom realizacije projekata razvoja softvera, sa posebnim fokusom na početne faze prikupljanja zahteva i modelovanja.

Specifikacija i modelovanje softvera zavise od izabrane metodologije razvoja, vrste aplikacije i same arhitekture. U ovom udžbeniku detaljno su objašnjene aktuelne metodologije i prikazani praktični pristupi modelovanju različitih vrsta softverskih rešenja.

Nakon uspešnog savladavanja materije, očekuje se da će student steći kompetencije da:

- Kritički odabere i primeni odgovarajuću metodologiju razvoja u zavisnosti od konteksta projekta;
- Prevede kompleksne poslovne procese u precizne modele (BPMN, IDEF0 ili DFD);
- Primeni postupak modelovanja podataka i sprovede proces normalizacije u cilju kreiranja optimalnih struktura podataka;
- Izradi kompletnu UML tehničku dokumentaciju za specifikaciju zahteva, analizu interakcija i projektovanje strukture i ponašanja sistema;
- Razume i primeni obrasce arhitekture softvera;
- Modeluje servisno-orijentisane sisteme i razume principe integracije putem magistrale poslovnih servisa (*Enterprise Service Bus*, ESB);
- Modeluje skalabilne sisteme zasnovane na mikroservisima, primenjujući principe strateškog i taktičkog DDD-a (*Domain-Driven Design*) u savremenom DevOps okruženju.

Ovaj udžbenik predstavlja most između teorijskih postavki softverskog inženjerstva i praktičnih izazova sa kojima se inženjeri susreću u realnom poslovnom okruženju, pripremajući studente za kreiranje softvera koji je pouzdan, održiv i u potpunosti usklađen sa potrebama korisnika.

Zašto izučavati specifikacije i modelovanje softvera?

Dobili ste projekat da razvijete aplikaciju za poslovanje studentske službe, osiguravajućeg društva ili složenog logističkog sistema. Odakle ćete početi?

a) Odmah ću krenuti sa programiranjem.
b) Prvo ću razumeti poslovne procese i način na koji organizacija funkcioniše, kreirajući pritom model procesa.
c) Odmah ću pristupiti analizi softvera izborom adekvatne arhitekture i vizuelnim opisom rada sistema pomoću jezika modelovanja.
d) Prvo ću prikupiti zahteve korisnika i kreirati formalni dokument: Specifikacija korisničkih zahteva (*User Requirement Specification*).
e) Ništa od prethodnog, već ću: _____.

Ukoliko ste zaokružili odgovor pod *a)*, zapitajte se zašto se u svim ostalim granama inženjerstva prvo izrađuju precizni modeli. U automobilskoj industriji, pre serijske proizvodnje kreiraju se digitalni i fizički prototipovi. Arhitekta prvo izrađuje projekat zgrade, a tek potom graditelji pristupaju realizaciji. Softverska industrija nije izuzetak – modeli softvera su planovi prema kojima se gradi digitalni proizvod. Iako se male aplikacije (poput kalkulatora) mogu iskodirati „u hodu", svaki kompleksniji sistem zahteva inženjerski pristup kako bi se izbegao neuspeh.

Ukoliko ste odabrali odgovor pod *b)*, na pravom ste putu. Uspešna realizacija softvera zahteva duboko razumevanje poslovnog domena koji aplikacija treba da podrži. Ovaj put podrazumeva identifikaciju potreba korisnika, kreiranje specifikacije, analizu i projektovanje pomoću vizuelnih jezika poput UML-a, pa tek onda implementaciju i testiranje. Ovi koraci čine okosnicu životnog ciklusa razvoja softvera (SDLC).

Tržište softvera je turbulentnije nego ikada. Kako osigurati da proizvod opstane? Danas nije dovoljno samo napraviti softver koji radi; on mora da izazove oduševljenje kod korisnika. U tom kontekstu, u knjizi obrađujemo Design Thinking okvir koji stavlja akcenat na empatiju sa korisnikom i rano prototipovanje pre ulaska u skupe faze razvoja.

Vreme od ideje do realizacije (*Time-to-Market*) postaje ključni faktor konkurentnosti. Ako menadžer banke želi novu uslugu pre konkurencije, neophodna mu je fleksibilna i skalabilna arhitektura. Tu na scenu stupaju tehnike servisno-orijentisanog razvoja (SOA) i mikroservisa, koje omogućavaju brzu integraciju i modularnost, podržanu algoritmima veštačke inteligencije koji ubrzavaju procese odlučivanja.

Ulazak u petu industrijsku revoluciju donosi eksploziju *IoT* (*Internet Of Things*) uređaja, *Big Data* sistema i *Web* 4.0 generacije (simbiotički veb). Ovi trendovi primoravaju organizacije da se preorijentišu na inteligentne tehnologije koje rade u realnom vremenu. Ova knjiga je vaš vodič kroz te kompleksne sisteme, dajući vam alate da postanete arhitekta softvera koji ne gradi samo za danas, već i za budućnost digitalnog sveta.

POGLAVLJE 1



UVOD U SOFTVERSKO INŽENJERSTVO

Uvod u softversko inženjerstvo

1.1. Kratak istorijski osvrt razvoja softverskog inženjerstva

1.2. Vrste aplikativnog softvera

1.3. Informacioni sistemi: pojam i podela

Cilj poglavlja:

Cilj ovog poglavlja je da pruži kritički uvid u istorijsku evoluciju softverskog inženjerstva i definiše ključne koncepte aplikativnog softvera. Kroz analizu različitih arhitektura, fokus je na razumevanju specifičnosti modelovanja različitih tipova aplikacija, sa posebnim akcentom na savremenu taksonomiju informacionih sistema — od operacionih i analitičkih do specijalizovanih rešenja zasnovanih na naprednim tehnologijama.

1. UVOD U SOFTVERSKO INŽENJERSTVO

1.1. KRATAK ISTORIJSKI OSVRT RAZVOJA SOFTVERSKOG INŽENJERSTVA

Da li ste se nekada zapitali, da li ono što učite danas, važi i sutra, odnosno da li znanje koje stičete sada možete primeniti u budućnosti? Kakav je evolutivni tok razvoja računarstva? Da li je to evolucioni i revolucioni tok, tj. da li će promene biti postepene ili revolutivne - koje će iziskivati promene iz korena?

Pokušajte da odgovorite na sledeća pitanja:

1. Koja su inovativna znanja danas najcenjenija, tj. koja vas čine konkurentnim na tržištu?
2. Kada se očekuje primena kvantnog računara u komercijalne svrhe? Da li će to izazvati revoluciju u računarstvu?

1.1.1. Pojam i evolutivni razvoj softverskog inženjerstva

Naučna oblast u okviru koje se izučava specifikacija i modelovanje softvera je softversko inženjerstvo. **Termin „Softversko inženjerstvo"** se prvi put upotrebio 1968. godine na NATO Konferenciji za softversko inženjerstvo. Formalne definicije uključuju:

- Inženjerska disciplina koja se bavi svim aspektima razvoja softvera.
- Disciplina koja obuhvata znanje, metode i alate za definisanje zahteva, razvoj softvera, rukovanje i održavanje softvera.
- Sistemski pristup u definisanju zahteva, analizi, projektovanju, kodiranju, testiranju, implementaciji, održavanju i reinženjeringu softvera.

Razvoj softverskog inženjerstva je tesno povezan sa razvojem računarstva.

Računarstvo (*Computing*) ili računarske nauke (*Computer Sciences*) razmatraju način konstruisanja (dizajn i arhitektura računara), programiranja, upotrebe i poboljšanja računarskog hardvera i softvera i interakcije čoveka i računara kao i drugih osnovnih principa rada računara. Računarstvo je skup veština i tehnologija koje se odnose na projektovanje i izgradnja hardverskih, softverskih i mrežnih sistema, kao i digitalno pismenih ljudi koji razvijaju i koriste računarske tehnologije.

Sagledavajući evoluciju razvoja računarstva (Slika 1.1), u tabeli 1.1 je dat kratak istorijski pregled razvoja softverskog inženjerstva.



Slika 1.1. Evolucija računarstva

Tabela 1.1. Istorijski osvrt razvoja softverskog inženjerstva i arhitektura

Decenija	Ključni tehnološki trendovi i arhitekturne paradigme
1940-te	**Era pionira:** Ručno pisanje mašinskog koda (binarni sistem). Fokus na hardveru, softver se smatrao delom mašine.
1950-te	**Prvi jezici:** Pojava Asemblera i prvih viših programskih jezika (npr. Fortran). Razvoj prvih kompajlera.
1960-te	**Strukturno programiranje:** Dominacija Mainframe računara. Uvođenje jezika kao što su *COBOL* i *BASIC*. Pojava termina „softversko inženjerstvo" (1968).
1970-te	**Sistemska revolucija:** Razvoj operativnog sistema *Unix* i jezika *C*. Pojava relacionih baza podataka i SQL-a kao standarda za čuvanje podataka.
1980-te	**Monolitna era:** Uspon personalnih računara (PC). Dominacija monolitnih arhitektura. Razvoj objektno-orijentisanih koncepata i informacionih sistema za planiranje resursa (MRP – preteča ERP sistema).
1990-te	**Objektna orijentacija i Web:** Široka primena jezika *Java, C++* i *C#*. Pojava *World Wide Web*-a. Razvoj integrisanih poslovnih rešenja (ERP, CRM, PLM).
2000-te	**Servisna orijentacija (SOA):** Pojava *Cloud Computing*-a i *Web* servisa. Standardizacija Agilnih procesa razvoja. Fokus na interoperabilnosti sistema.
2010-te	**Era mikroservisa:** Prelazak na mikroservisnu arhitekturu. Razvoj kontejnerizacije (*Docker, Kubernetes*). *Big Data* analitika, IoT (*Internet of Things*) i uspon nauke o podacima (*Data Science*).
2020-te	**Inteligentni sistemi:** Dominacija Generativne veštačke inteligencije (LLM modeli). *Cloud*-native razvoj, blokčejn i težnja ka opštoj inteligenciji (*Artificial General Intelligence*, AGI).

Šta mislite, kakva je trenutna situacija u kompanijama, što se tiče instaliranih aplikacija? Šta možete da očekujete da ćete zateći i šta će oni očekivati od vas da održavate i/ili razvijate?

Danas ne postoji kompanija koja nema makar jedan instaliran ERP sistem u svom okruženju, međutim najčešće poseduje niz raznorodnih aplikacija, često nabavljanih od strane različitih vendora. Na primer, koriste SAP ERP za upravljanje internim poslovanjem, Microsoft CRM za direktan kontakt sa potrošačima, neku veb B2C aplikaciju i druge sisteme. Od vas mogu očekivati da znate da koristite ove sisteme, da ih po potrebi prilagodite koristeći njihove specifične programske jezike (npr. SAP koristi ABAP programski jezik), da razvijete nove aplikacije i integrišete sa ovim postojećim sistemima, da upravljate master podacima, da kreirate magistralu poslovnih servisa (*Enterprise Service Bus, ESB*) preko kojih će ovi sistemi međusobno komunicirati i razmenjivati informacije, da ugradite algoritme mašinskog učenja u poslovne procese ovih sistema i učinite ih pametnijim itd.

Ukoliko su vam nepoznati spomenuti pojmovi, u narednom tekstu se daju dodatna pojašnjenja i ujedno prolazi kroz istorijski okvir razvoja softverskog inženjerstva.

1.1.2. Poslovni sistemi

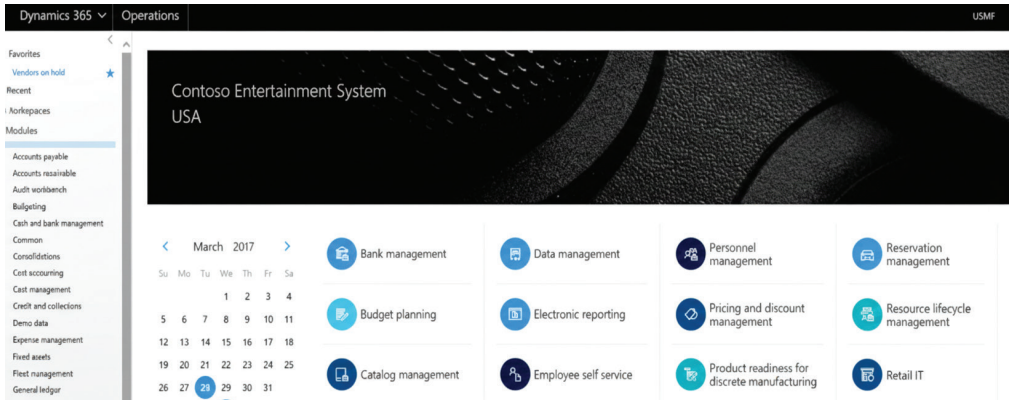
Na tržištu ćete često čuti neke akronime ili skraćenice, na primer kada su u pitanju poslovni informacioni sistemi ili poslovna rešenja najčešće korišćeni akronimi su ERP, CRM, PLM i dr. O čemu se radi?

Šta su ERP, CRM, SCM i ostali poslovni informacioni sistemi?

Poslovni informacioni sistemi (*Business Information Systems*) ili **poslovna rešenja** (*Business Solutions*) se odnose na integrisane informacione sisteme koji podržavaju poslovne funkcije preduzeća, bilo da su orijentisani ka korisnicima (*Customer Relationship Management, CRM*) ili upravljanju životnim ciklusom proizvoda (*Product Lifecycle Management, PLM*) ili upravljanju lancima snabdevanja (*Supply Chain Management, SCM*) ili integrišu interne poslovne procese (*Enterprise Resource Planning, ERP*) ili pomažu donosiocima odluka (*Business Intelligence, BI*) kao i na ostale funkcije koje preduzeće obavlja u svom svakodnevnom poslovanju.

Poslovne funkcije preduzeća su: računovodstvo i finansije, marketing, menadžment, proizvodnja, magacinsko poslovanje, ljudski resursi, logistika i dr.

ERP (*Enterprise Resource Planning*) sistemi ili u bukvalnom prevodu informacioni sistemi za planiranje resursa preduzeća, integrišu ključne poslovne procese kompanije. Ove sisteme koriste svi zaposleni bez obzira u kom sektoru kompanije radili. Cilj integrisanih sistema je da sve informacije budu na jednom mestu, da se transparentno razmenjuju i da menadžer ima jedinstvenu sliku nad poslovanjem kompanije. Primena ERP sistema osigurava da kompanija koristi najbolje prakse, koje su implementirane u procese ERP sistema i da efikasno obavlja svoje poslovanje. Primenom ovakvih sistema znantno se ubrzava proces od poručivanja do isporuke proizvoda na kućnu adresu klijenta.



Slika 1.2. Microsoft Dynamics 365 ERP sistem

Na slici 1.2, dat je prikaz ekrana Microsoft Dynamics 365 ERP rešenja na kome se nalazi lista modula koji reflektuju poslovne procese kompanije, kao što su: upravljanje finansijama (*Financial Management*), prodaja i marketing (*Sales & Marketing*), nabavka (*Purchase*), magacinsko poslovanje (*Warehouse*), proizvodnja (*Manufacturing*), ljudski resursi (*Human Resources*) i dr.

Imajte na umu da su ovi sistemi podložni stalnim nadogradnjama i da u trenutku učenja, izgled ovog rešenja može biti potpuno drugačije.

Koja kompanija je lider u razvoju poslovnih rešenja?

Kompanija koja je svoje poslovanje okrenula jedino razvoju poslovnih rešenja za različite industrijske grane je nemačka kompanija SAP. SAP za razliku od ostalih rešenja, mapira najsloženije poslovne procese u industrijama. Usled toga, SAP je jedini izbor za naftne indsturije, vladine institucije, farmaceutske kompanije i dr. Microsoft-ova, Oracle-ova i poslovna rešenja drugih IT kompanija, uglavnom mapiraju procese koji su zajednički za većinu kompanija i retko kada mogu da kompletno automatizuju, na primer proizvodnju nekog leka u farmaceutskoj industriji.

1.2. VRSTE APLIKATIVNOG SOFTVERA

Uobičajena podela softvera je na sistemski, programski i aplikativni softver. U ovoj knjizi se neće razmatrati sistemski (npr. operativni sistemi) i programski softver (npr. sistemi za razvoj aplikativnog softvera), već je fokus na modelovanju aplikativnog softvera (kraće aplikacije), što i jeste glavni fokus softverskog inženjerstva.

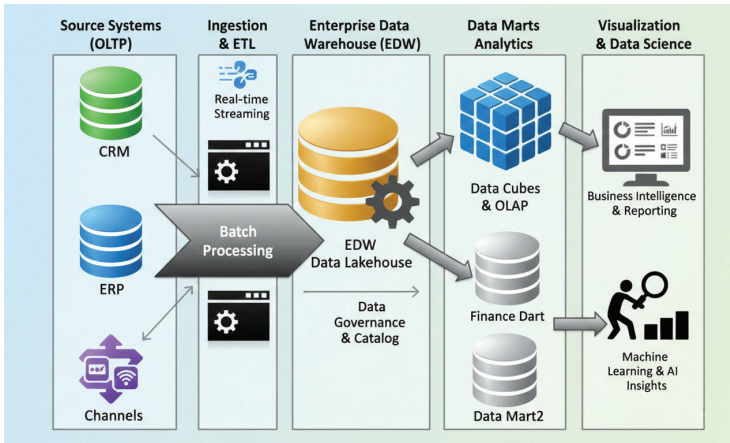
Različite vrste aplikativnog softvera zahtevaju različite pristupe. Softversko inženjerstvo se pre svega odnosi na složene aplikacije i informacione sisteme, pre nego na jednostavne aplikacije (od par linija koda). Neke ključne vrste aplikativnog softvera su (Tabela 1.2):

Samostalne (*Stand-alone*) **aplikacije** su sistemi koji se izvršavaju na lokalnom računaru i ne moraju biti povezani na mrežu. Primeri ovih aplikacija su: CAD programi, softveri za razvoj filmova, softveri za obradu slika, zvuka itd.

Informacioni sistemi su sistemi čija je osnovna svrha upravljanje bazom podataka i poslovnim procesima. Problemi u informacionim sistemima uključuju sigurnost, upotrebljivost, privatnost i održavanje integriteta podataka. Primeri informacionih sistema su: zdravstveni informacioni sistemi, sistemi za upravljanje hotelskim poslovanjem, sistemi za upravljanje ljudskim resursima, kao i već spomenuti ERP, CRM i drugi sistemi.

Interaktivne transakcione aplikacije su aplikacije koje se izvršavaju na udaljenom računaru, a kojima korisnici pristupaju sa sopstvenih računara ili mobilnih telefona (npr. aplikacije za e-trgovinu kao što su Amazon, eBay). Organizacije omogućavaju pristup svojim sistemima putem veb pretraživača klijentskog programa ili *cloud* servisa, kao što su elektronska pošta (*e-mail*) ili sajtovi za deljenje slika (Instagram, Pinterest) i dr. Ovi sistemi zahtevaju pristup i ažuriranje velike memorije podataka pri svakoj transakciji.

Sistemi za grupnu obradu (*Batch processing systems*) su poslovni sistemi koji su projektovani za obradu podataka u velikim serijama (grupna obrada), odnosno podaci se ne obrađuju u trenutku vremena, već se sve promene beleže i skupljaju, kako bi se na kraju dana grupno obradile. Na primer, poslovi u banci na kraju dana (EOD) uključuju obračun kamata, generisanje izveštaja i seta podataka za druge sisteme, štampanje izvoda i obradu plaćanja. Inteligentni poslovni sistemi (*Business Intelligence, BI*) koriste grupnu obradu podataka, tako što sve transakcije u toku dana skupljaju i u određenom trenutku vremena ih prebacuju u skladište podataka (*Data Warehouse, DW*) (Slika 1.3).



Slika 1.3. Grupna obrada podataka u inteligentnim poslovnim sistemima

Ugrađeni kontrolni sistemi (*Embedded control systems*) su sistemi gde softver, koji je ugrađen u sam uređaj, kontroliše rad hardverskog uređaja. Problemi sa ugrađenim sistemima su fizička veličina, upravljanje napajanjem, odzivnosti i dr. Primeri ugrađenih kontrolnih sistema su softverski sistem koji kontroliše medicinski uređaj ili upravlja pametnim kolima i sl. (Slika 1.4).



Slika 1.4. Ugrađeni kontrolni sistemi u različitim uređajima

Sistemi za senzorsko prikupljanje podataka su sistemi čija je primarna svrha prikupljanje i obrada podataka sa senzora. Ključni zahtevi takvih sistema su pouzdanost rada, posebno u lošim uslovima, kao i održivost. Primeri ovih sistema su sistemi meteorološke stanice, IoT aplikacija za parkiranje u realnom vremenu i dr.

Sistemi za zabavu (*Entertainment systems*) su pre svega namenjeni za ličnu upotrebu u cilju zabave korisnika. Većina ovih sistema se odnosi na igrice. Kvalitet korisničke interakcije je najvažnija i prepoznatljiva karakteristika sistema zabave.

Sistemi za modeliranje i simulaciju su sistemi koje su naučnici i inženjeri razvili za modeliranje fizičkih procesa ili scenaria, koji uključuju mnoge odvojene, interaktivne objekte. Ovi sistemi su često računski intenzivni i računarski zahtevni, te su za njihovo izvođenje potrebni paralelni sistemi visokih performansi.

Agentni AI sistemi (*Agentic AI systems*) predstavljaju evoluciju softvera koji koristi veštačku inteligenciju (*Artificial Intelligence, AI*) ne samo za generisanje sadržaja, već za autonomno izvršavanje složenih zadataka i donošenje odluka u realnom vremenu. Za razliku od tradicionalnih aplikacija koje čekaju instrukcije korisnika, AI agenti mogu samostalno planirati korake, koristiti spoljne alate (poput veb pretraživača ili baza podataka) i ispravljati sopstvene greške kako bi postigli definisani cilj. Glavni izazovi u razvoju ovih sistema u okviru softverskog inženjerstva su njihova deterministička nepredvidivost, potreba za robusnim mehanizmima kontrole i etička pitanja autonomije. Primeri uključuju autonomne asistente u korisničkoj podršci koji sami rešavaju reklamacije, softverske agente za automatizovano trgovanje na berzi ili inteligentne sisteme za upravljanje kompleksnim lancima snabdevanja koji samostalno vrše poručivanje resursa.

Sistemi sistema su sistemi koji se sastoje od brojnih drugih softverskih sistema (Slika 1.5).



Slika 1.5. Sistem sistema

Tabela 1.2. Pregled vrsta aplikativnog softvera

Tip softvera	Karakteristike	Primeri
Informacioni sistemi (*Information Systems, IS*)	Fokus na perzistenciju podataka (baze podataka), integritet (ACID svojstva) i podršku poslovnim procesima. Karakteriše ih visok stepen sigurnosti i korisničkih prava.	ERP (npr. SAP), CRM (npr. Salesforce), HIS (*Hospital Information Systems*) i mnogi drugi.
Samostalne aplikacije (*Stand-alone*)	Softverski programi koji se instaliraju lokalno na računaru, rade nezavisno od mrežne veze i ne zahtevaju pristup veb-pretraživaču kako bi funkcionisali. Karakteriše ih visok stepen korišćenja lokalnih resursa (CPU/GPU). Dizajnirane su za zahtevne zadatke visokih performansi, kao što su inženjersko projektovanje, 3D modelovanje i medijska produkcija.	CAD (npr. AutoCAD), Adobe Premiere (za film, video i medije), Photoshop, lokalni plejeri.
Interaktivne transakcione aplikacije	Sistemi na udaljenim serverima (*Cloud/Web*) kojima se pristupa preko klijenta. Zahtevaju obradu ogromnog broja istovremenih upita.	*E-commerce* (npr. Amazon), Instagram, Internet bankarstvo, Gmail.
Sistemi u realnom vremenu (*Real-time Systems*)	Softver koji mora da odgovori na spoljne stimulanse u strogo definisanim vremenskim okvirima (*hard* ili *soft real-time*). Hard ili strogi sistemi ne tolerišu kašnjenje, dok kod je kod soft ili blagih sistema prihvatljivo.	Kontrola leta, sistemi za aktivaciju vazdušnih jastuka, berzanske platforme za trgovanje.
Sistemi sistema (*System of Systems*)	Veliki, kompleksni sistemi nastali integracijom nezavisnih operativnih sistema koji zajedno ostvaruju cilj koji nijedan ne može sam.	*Smart City* platforme (integrišu saobraćaj, energetiku i policiju), vojni odbrambeni sistemi.
Ugrađeni kontrolni sistemi (*Embedded Systems*)	Softver visokog nivoa pouzdanosti koji upravlja hardverskim komponentama. Često ima ograničene resurse (memorija, energija).	Softver u mikrotalasnim pećnicama, ECU u automobilima, kontrolori u pejsmejkerima.

Tip softvera	Karakteristike	Primeri
Sajber-fizički sistemi (*Cyber-Physical Systems*)	Evolucija IoT-a; integracija računarskih algoritama i fizičkih procesa putem povratne sprege. CPS se definiše kroz tri ključne komponente: (1) ***Computation*** (Računanje): Lokalni procesori koji izvršavaju algoritme i AI modele, (2) ***Communication*** (Komunikacija): Sposobnost sistema da komunicira sa drugim sistemima ili *cloud*-om (npr. pametni semafori koji međusobno usklađuju zeleno svetlo) i (3) ***Control*** (Upravljanje): Sposobnost da softver preuzme kontrolu nad fizičkim procesom (npr. sistem koji sam koči auto u opasnosti).	Autonomna vozila (Tesla Autopilot), pametne električne mreže (*Smart Grids*).
Sistemi za senzorsko prikupljanje (*IoT systems*)	Primarna svrha je prikupljanje, obrada i prenos podataka sa udaljenih senzora u realnom vremenu.	Meteorološke stanice, pametni parking, senzori za zagađenje vazduha.
Agentni AI sistemi (*Agentic AI*)	Sistemi sa visokim stepenom autonomije koji koriste rezonovanje za planiranje i izvršavanje zadataka bez stalnog nadzora čoveka.	AutoGPT, napredni botovi za arbitražu, autonomni agenti za sajber-odbranu.
Sistemi za naučno modeliranje i simulaciju	Računarski intenzivni sistemi (*High-Performance Computing, HPC*) koji koriste numeričke algoritme za simulaciju fizičkih fenomena. HPC predstavlja upotrebu "superračunara" ili računarskih klastera za rešavanje ekstremno složenih problema koji bi na običnom računaru trajali godinama.	Modeli za predviđanje klimatskih promena, simulacije sudara čestica (CERN), CFD (*Computational Fluid Dynamics*).
Sistemi za kolaboraciju (*Groupware*)	Softver dizajniran da pomogne ljudima koji rade na zajedničkim zadacima, često distribuirani na različitim lokacijama.	Microsoft Teams, Slack, Google Workspace, GitHub (za inženjere).

Tip softvera	Karakteristike	Primeri
Sveprisutno računarstvo (*Pervasive / Ubiquitous Computing*)	Softver koji je neprimetno utkan u okruženje korisnika, prilagođavajući se kontekstu i lokaciji. Ključne karakteristike su: (1) **Nevidljivost** (*Invisibility*): Tehnologija se "povlači u pozadinu". Više ne sedite za računarom, već je računar u vašem satu, odeći, zidovima ili kućnim aparatima. (2) **Svesnost o okruženju** (*Context-awareness*): Sistem koristi senzore da razume gde se nalazite, ko ste i šta radite, kako bi vam pružio informaciju pre nego što je uopšte zatražite. (3) **Neprimenta i transparentna povezanost** (*seamless*): Uređaji međusobno komuniciraju bez potrebe da ih vi ručno uparujete ili podešavate.	Pametni domovi (*Home Assistant*), nosiva tehnologija (*Apple Watch* zdravstveni senzori).

1.3. INFORMACIONI SISTEMI: POJAM I PODELA

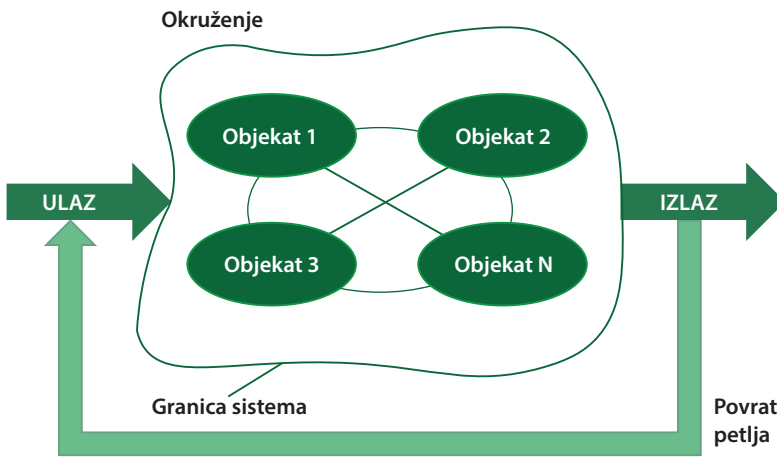
Najčešća vrsta aplikativnog softvera koja se modeluje u praksi su informacioni sistemi. Poslovne organizacije najčešće imaju potrebe za ovom vrstom sistema. U daljem tekstu se pored objašnjenja osnovnih pojmova sistema, informacije i informacionog sistema, navodi i podela informacionog sistema na tri glavne grupe sistema.

1.3.1. Pojam sistema

Opšta teorija sistema predstavlja naučnu oblast koja se bavi izučavanjem sistema i zakonitosti koje u njima nastaju. Jedna od najvažnijih karakteristika teorije sistema jeste u pristupu, u okviru koga se svaka celina posmatra kao deo neke veće celine. Nastajanje opšte teorije sistema dovelo je do stvaranja sistemskog pristupa, kao i do novih tehnika i metoda analize sistema. Kada se upotrebi termin "sistemski pristup", tada se naglašava da se svi predmeti i pojave posmatraju u njihovoj dinamičnosti i celovitosti u odnosu na okruženje.

Cilj opšte teorije sistema je da služi kao jedinstveni metodološki i pojmovno-kategorijalni okvir sporazumevanja ljudi različitih specijalnosti. Takođe, njen cilj je da obuhvati i objedini fundamentalne pojmove koji važe u svim specifičnim sistemima i teorijama koje se njima bave. Imajući u vidu opštu teoriju sistema, u daljem tekstu definisaće se pojam sistema i navesti njegove karakteristike.

Sistem se najopštije definiše kao skup međusobno povezanih objekata (entiteta), **sa jasno definisanom granicom, organizovan za postizanje zajedničkog cilja ili svrhe** (Slika 1.6). Objekti su merljive stvari koje se mogu povezati. Drugi, često korišćen, naziv za objekte je entitet od engleske reči *entity*. Primeri objekata su: zaposleni, kupac, skladište, proizvod, porudžbina, plaćanje itd. Objekti u sistemu se opisuju preko svojih svojstava koje se nazivaju atributima. Ukoliko se posmatra sistem fakulteta, onda su objekti fakulteta: studenti, zaposleni, službe, sale, računari, prijemni ispit, predavanja i dr. Svi ovi objekti su u međusobnoj relaciji, na primer, student prijavljuje ispit, profesor predaje studentima itd. Cilj sistema fakulteta je obrazovanje, te su svi objekti i njihove međusobne veze upravo usmerene ka ostvarenju tog cilja. Cilj sistema je „razlog postojanja" sistema.



Slika 1.6. Pojam sistema

Sistemski pristup razmatra dve osnovne komponente: objekte i procese. Procesi menjaju objekte iz jednog oblika u drugi, odnosno proces na osnovu dobijenih ulaza (podaci, materijali, delovi i dr.) vrši obradu i pretvara stanje objekta u željene izlaze (proizvodi, usluge, informacije i dr.).

Skup objekata koji predstavlja posmatrani sistem definiše granice sistema. Sve izvan granica sistema se naziva okolina ili okruženje sistema. Dejstvo okoline na sistem opisuje se preko ulaza u sistem, a dejstvo sistema na okolinu preko njegovih izlaza, kao što je ilustrovano na slici 1.6. Ulazi mogu biti opipljivi resursi, poput energije ili novca ili nematerijalni, kao što su informacije. Izlazi mogu biti vidljiva ponašanja, rezultati, uputstva, proizvodi ili usluge.

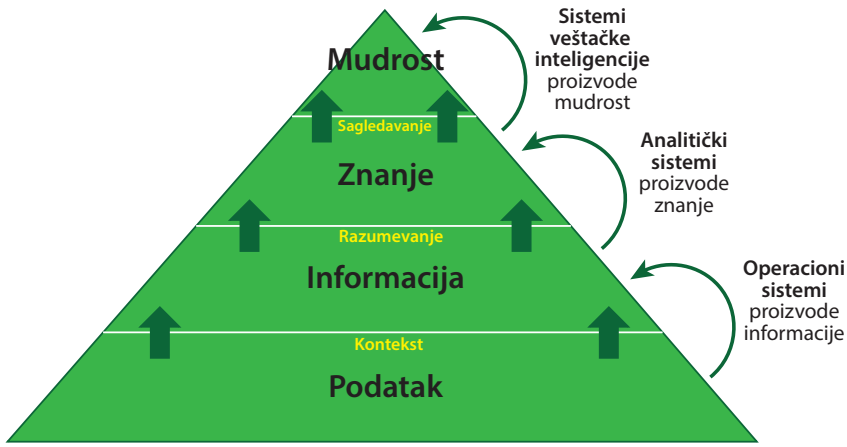
Dinamičko ponašanje realnog sistema standardno se predstavlja na sledeći način: ulazi u sistem menjaju stanja sistema. Stanje sistema se definiše kao skup informacija o prošlosti i sadašnjosti sistema potrebnih da bi se, pod dejstvom budućih poznatih ulaza, mogli odrediti budući izlazi. U stanju sistema koncentrisana je celokupna istorija realnog sistema. Izlazna transformacija definiše neki način merenja ili posmatranja dinamičkog ponašanja realnog sistema i daje, na osnovu stanja sistema, njegove izlaze.

Svaki sistem moguće je dekomponovati na podsisteme. Istovremeno, svaki sistem je deo nekog šireg sistema. Hijerarhičnost se mora uzeti u obzir prilikom istraživanja: ponašanja, funkcionisanja, razvoja i upravljanja sistemima. Sistemi ne egzistiraju izolovani, već teže da budu otvoreni sistemi. Otvorenost predstavlja komunikaciju između objekata sistema i objekata iz njegovog okruženja.

1.3.2. Podatak, informacija, znanje i mudrost

Pre nego što se definiše pojam informacionih sistema, neophodno je pored pojma sistema razumeti i značenje reči informacija. U te svrhe poslužiće tzv. DIKW model (Slika 1.7).

DIKW (*Data, Information, Knowledge, Wisdom*) model je predlog struktuiranja podataka, informacija, znanja i mudrosti u hijerarhiju gde se na svakom nivou dodaju određena svojstva. DIKW model se temelji na pretpostavkama da podatak dolazi u obliku neobrađenog zapažanja i dimenzija, da se informacija oblikuje analiziranjem veza i odnosa između podataka, znanje razumevanjem organizovanih i analiziranih informacija, a mudrost kroz iskustvenu primenu znanja sa razboritošću, praktičnošću, diskretnošću i zdravim razumom, dolazi do novih saznanja.



Slika 1.7. Adaptirani DIKW model

Podatak (*Data*) je kodirana predstava o nekoj činjenici iz realnog sveta. On je nosilac informacija i služi za tehničko uobličavanje informacija, kako bi se one mogle sačuvati ili preneti. Pojedinačni podaci sami za sebe nemaju nikakvo značenje ili ga imaju veoma malo.

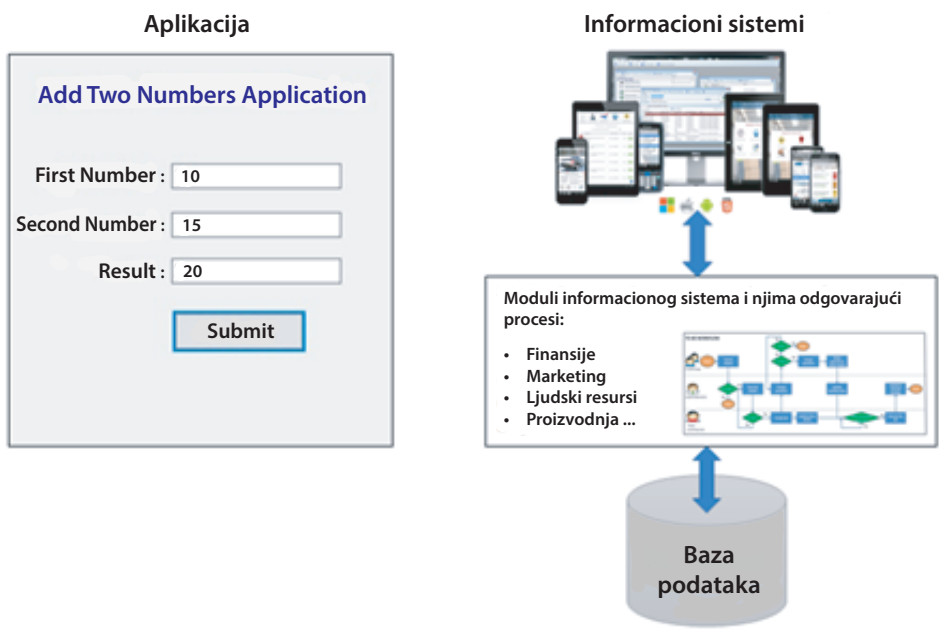
Informacija (*Information*) je protumačeni podatak o pojavi koju podatak prikazuje. Drugim rečima, informacija je prečišćen, organizovan i obrađen podatak u smislenom kontekstu. Informacija je resurs koji je kreiran od podataka.

Znanje (*Knowledge*) se gradi na osnovu razumevanja organizovanih, analiziranih i protumačenih informacija. Puko memorisanje informacija bez njihovog povezivanja, razumevanja i primene (npr. učenje napamet) nije znanje, jer tako stečene informacije nisu upotrebljive. Primena znanja u praksi su veštine. Novi alati i informacioni sistemi ubrzavaju istraživanje podataka, njihovu apsorpciju, stvaraju velike količine informacija i u interakciji sa korisnicima pružaju znanja na osnovu kojih oni donose odluke.

Mudrost (*Wisdom*) podrazumeva sposobnost da se na osnovu sagledavanja postojećeg znanja dođe do novog znanja koje do sada nije postojalo. Mudrost uključuje maštovitost, pronicljivost i kreativnost. Do mudrosti se dolazi uspešnim prolazom kroz sve prethodne kategorije, čiji finalni rezultat je novo znanje. Na primer, student na doktorskim studijama, na osnovu analize postojećeg znanja u određenoj naučnoj oblasti (koja često rezultira u prikazu detaljne analize rezultata istraživanja prethodnih naučnika, tzv. State of the art), sagledavanjem uzročno-posledičnih odnosa i primenom drugih naučnih metoda, dolazi do novih saznanja i rešenja, koja u početku konceptualno postavlja, a kasnije eksperimentom i dokazuje.

1.3.3. Šta je informacioni sistem?

Razumevanje informacionih sistema najbolje započinje odgovorom na pitanje koje se često postavlja u praksi: Zašto se određena vrsta aplikativnog softvera naziva informacionim sistemom i u čemu je ključna razlika između obične aplikacije i informacionog sistema ili kraće sistema? (Slika 1.8)

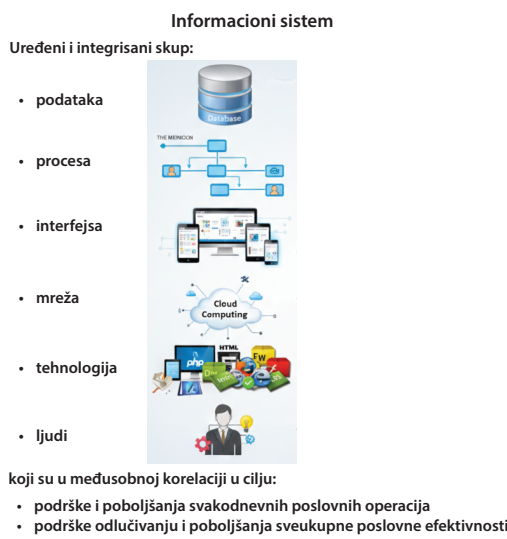


Slika 1.8. Osnovna razlika između informacionog sistema i bilo koje manje složene aplikacije

Na slici 1.8 vizuelno je prikazana osnovna razlika. Dok se jednostavne aplikacije fokusiraju na izvršavanje izolovanih operacija (poput aplikacije za sabiranje dva broja koja nema potrebu za trajnim čuvanjem podataka), informacioni sistem predstavlja daleko složeniju strukturu. On, pored interfejsa i logike, obavezno poseduje sloj podataka (bazu podataka) i sloj poslovne logike kojom se upravlja procesima.

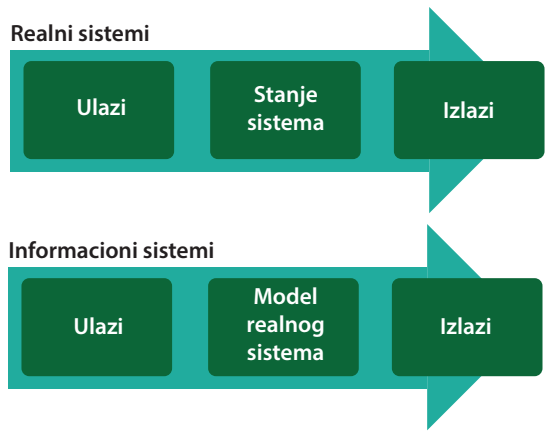
Detaljnije obrazloženje pojma informacionog sistema jeste da **predstavlja uređeni i integrisani skup podataka, procesa, interfejsa, mreža, tehnologija i ljudi koji su u međusobnoj korelaciji u cilju podrške i poboljšanja svakodnevnih poslovnih operacija i podrške menadžmentu u rešavanju poslovnih problema, planiranja, upravljanja, predviđanja, koordinisanja i donošenja odluka** (Slika 1.9). Svaki informacioni sistem ima:

- **Bazu podataka** - služi za strukturirano skladištenje i dugotrajno čuvanje svih relevantnih podataka.
- **Procese** - mapiraju realno poslovanje u digitalnu formu. Na primer, procesi u fakultetskom informacionom sistemu (upis, ocenjivanje) suštinski se razlikuju od procesa u hotelskom sistemu (rezervacije, recepcija), jer reflektuju specifičnu logiku rada te organizacije.
- **Interfejse** - predstavljaju tačke interakcije koje omogućavaju korisnicima da na jednostavan način komuniciraju sa sistemom.
- **Mreže** - obezbeđuju infrastrukturu za prenos podataka. Danas su sistemi gotovo uvek mrežni, bilo da su dostupni samo unutar firme (Intranet), partnerima (Extranet) ili javno putem Interneta.
- **Tehnologije** - obuhvataju hardverske resurse i softverske alate neophodne za razvoj i funkcionisanje sistema.
- **Ljude** - najvažnija komponenta sistema. To su stručnjaci odgovorni za razvoj i održavanje, ali i krajnji korisnici kojima je sistem namenjen.



Slika 1.9. Pojam informacionog sistema

Informacioni sistem treba da bude veran model realnog sistema u kome deluje (Slika 1.10). U ovom kontekstu, ulazi u sistem menjaju njegovo stanje, a ta promena se direktno reflektuje na izlazu. Preslikavanje realnog sveta u digitalno okruženje izvodi se postupkom modelovanja. Većina realnih sistema je suviše kompleksna da bi se shvatila u celini. Zbog toga se pribegava kreiranju modela koji omogućavaju simulaciju, testiranje i stalno poboljšanje. Suštinski, informacioni sistemi se ne mogu pravilno razumeti ni projektovati bez preciznih modela koji verno predstavljaju njihovu realnost.



Slika 1.10. Modelovanje sistema

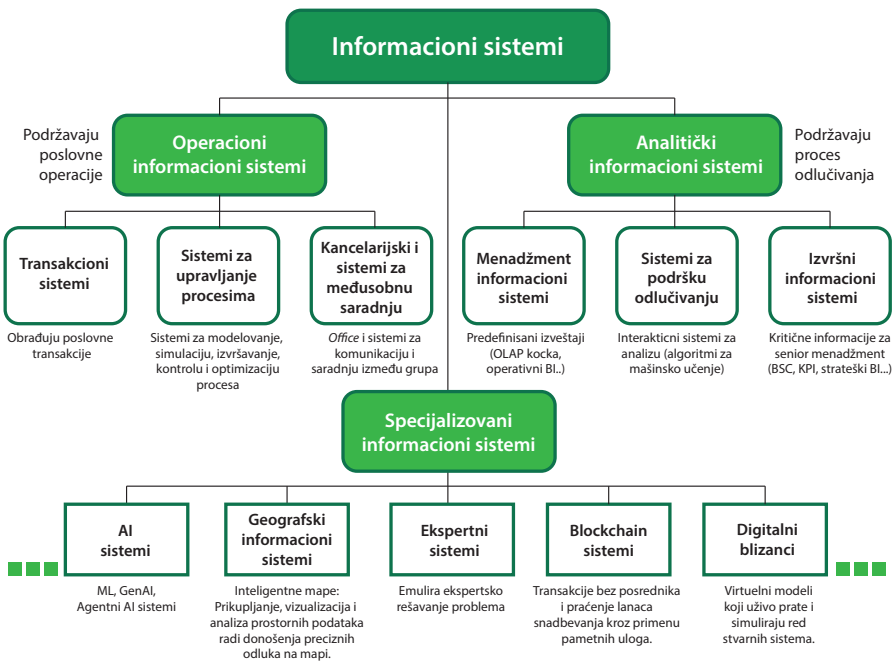
1.3.4. Podela informacionih sistema

Na tržištu postoji puno razvijenih informacionih sistema, međutim oni se svi mogu svrstati u tri grupe (Slika 1.11):

1. **Operacioni informacioni sistemi** – Automatizuju svakodnevne poslovne procese i obezbeđuju integritet podataka putem relacionih baza (*Relational Database Management Systems, RDBMS*). Njihovo jezgro čine sistemi za obradu transakcija (*Transaction Processing Systems, TPS*), čiji su najobuhvatniji primer ERP sistemi koji integrišu ključne poslovne funkcije u jedinstvenu bazu. Savremeni operacioni sistemi evoluiraju kroz integraciju robotske automatizacije procesa (*Robotic Process Automation, RPA*) i tehnologija veštačke inteligencije (*Artificial Intelligence, AI*), čijom sinergijom (tzv. *Intelligent Automation*) prevazilaze puku transakcionu obradu i transformišu se u inteligentne platforme koje omogućavaju naprednu analitiku i automatizovano donošenje odluka u realnom vremenu.
2. **Analitički informacioni sistemi** – Dizajnirani su kao podrška procesima odlučivanja na svim nivoima organizacione hijerarhije (operativni, taktički i strateški nivo menadžmenta). Tradicionalno se oslanjaju na skladišta podataka (*Data Warehouse, DW*), dok savremene arhitekture, radi obrade podataka u realnom vremenu, koriste jezera podataka (*Data Lake*) ili hibridna rešenja poput *Data Lakehouse* koncepta, koji objedinjuje performanse DW-a sa fleksibilnošću jezera

podataka. Iako su u jednom periodu bili označavani kao inteligentni sistemi (prevashodno zbog rane implementacije AI rešenja i tehnika rudarenja podataka – *Data Mining*), danas se naglasak ponovo stavlja na njihovu analitičku funkciju, s obzirom na to da se ML algoritmi sada integrišu i u operacione (transakcione) sisteme. Savremeni portfolio analitičkih sistema, pored poslovne inteligencije (BI), obuhvata i Big Data analitiku. Ovi sistemi obrađuju ogromne količine podataka različitih formata (strukturirani, polustrukturirani i nestrukturirani podaci poput slike, glasa, videa i teksta) koji se generišu velikom brzinom u realnom vremenu.

3. **Specijalizovani informacioni sistemi** – Obuhvataju sisteme dizajnirane za specifične domene primene koji se po svojoj arhitekturi i nameni ne mogu primarno svrstati u operacione ili analitičke sisteme. Ovi sistemi koriste napredne tehnologije za rešavanje kompleksnih problema u usko definisanim oblastima. U ovu kategoriju spadaju:
   - Ekspertni sistemi (zasnovani na bazi znanja i pravilima zaključivanja),
   - Geografski informacioni sistemi (GIS) (za upravljanje prostornim podacima),
   - Blockchain sistemi (za decentralizovano poverenje i nepromenljivost transakcija),
   - Sistemi zasnovani na veštačkoj inteligenciji (AI sistemi), kao što su sistemi za prepoznavanje govora ili slike, generativni AI sitemi, Agentni (*Agentic*) AI sistemi i dr.
   - Digitalni blizanci (*Digital Twins*) i sistemi za simulaciju kompleksnih fizičkih procesa i dr.



Slika 1.11. Vrste informacionih sistema

Operacioni informacioni sistemi

Operacioni informacioni sistemi su uređeni skup alata i sistema koji služe za prikupljanje, skladištenje, obradu i transformaciju podataka u informacije, a koje podržavaju svakodnevne poslovne operacije. U ovu grupu spadaju sistemi za obradu transakcija (kraće transakcioni sistemi), sistemi za upravljanje poslovnim procesima i drugi.

Najveću grupu operacionih sistema čine transakcioni sistemi koji prikupljaju, ažuriraju, skladište i obrađuju svakodnevne poslovne transakcije. Postoje dva načina obrade transakcija i to:

1. **Trenutna obrada** (*Online processing*) – Sistemi koji trenutno obrađuju transakcije se često nazivaju OLTP (*Online Transaction Processing*) sistemi. Na primer, kada podižete novac sa bankomata, iznos koji ste podigli će automatski biti skinut sa stanja na vašem računu. Promena stanja će biti odmah realizovana.
2. **Grupna obrada** (*Batch processing*) – Podrazumeva da se transakcije sakupljaju i onda periodično obrađuju. Na primer, inteligentni poslovni sistem banke će u 3 ujutru, kada se očekuje najmanji promet, prebaciti sve dnevne transakcije u skladište podataka (*Data Warehouse, DW*). Ukoliko bi zamislili da se na bankomatu realizuje grupna obrada transakcija, to bi značilo da korisnik može podizati novac u toku dana, a da mu stanje na računu ostane nepromenjeno, sve do trenutka kada bi se izvršila grupna obrada, odnosno skinuo ukupan iznos svih realizovanih transakcija sa stanja računa klijenta.

Kod transakcionih sistema najčešće se misli na OLTP sisteme čija baza podataka je potpuno normalizovana u cilju obavljanja CRUD (*Create, Read, Update, Delete*) operacija.

Analitički informacioni sistemi

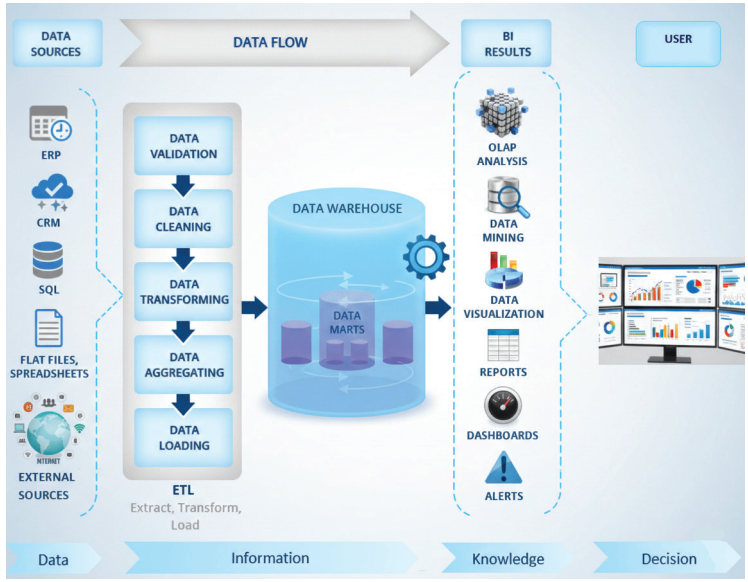
Analitički informacioni sistemi su uređeni skup alata i sistema koji omogućavaju pronalaženje, prikupljanje, ekstrakovanje, transformisanje, organizovanje, skladištenje, analiziranje i prevođenje poslovnih informacija u znanje u cilju podrške odlučivanju i poboljšanja sveukupne poslovne efektivnosti.

Kada je tržište bilo preplavljeno raznim transakcionim sistemima, menadžeri su se sa pravom zapitali, čemu ovoliko sistema koji svakodnevno generišu sve veći broj podataka, koje je teško pratiti i analizirati u cilju donošenja odluka. U te svrhe razvijaju se analitički informacioni sistemi koji automatizuju proces analize podataka i pružaju podršku odlučivanju.

Analitički sistemi se potpuno drugačije razvijaju u odnosu na operacione sisteme. Dok operacioni sistemi mapiraju svakodnevne poslovne operacije, koje obavljaju zaposleni u kompaniji, analitički sistemi mapiraju proces donošenja odluka od strane menadžmenta. Na primer, ukoliko treba da razvijete analitički sistem za menadžera u banci koji donosi odluke o odobravanju kredita klijentima, postavlja se pitanje da li je njemu dovoljan podatak iz baze podataka transakcionog bankarskog sistema o trenutnom stanju na računu klijenta ili o trenutnoj vrednosti plate i sl. Najverovatnije da će menadžer tražiti istoriju u prethodnih pet godina o prosečnom stanju na računu klijenta, broju podizanja kredita i dinamici njihovih otplata, prosečnoj plati klijenta, izveštaje sa spoljnih izvora, na primer, berze, Kreditnog biroa i dr. Na slici 1.12, prikazana je okvirna arhitektura inteligentnih poslovnih sistema (*Business Intelligence, BI*).

Prvi korak u razvoju BI sistema je identifikovanje **izvora podataka**. Iz prethodnog primera, uočili ste da su menadžeru bili potrebni podaci iz raznih izvora, kao što su interna baza podataka, Excel ili drugi fajlovi koji nisu deo baze podataka kompanije, izveštaji od poslovnih partnera, razni izvori sa Interneta i sl. Kako ćete znati koje izvore podataka treba identifikovati u prvom koraku? Upravo menadžer za koga pravite aplikaciju, će vam reći na primer, sa kojih sajtova prikuplja određene informacije, koji su to tačno podaci iz baze podataka i sl.

Drugi korak, nakon identifikovanja izvora podataka, jeste primena alata za **ekstrakciju, transformaciju i punjenje** (*Extraction, Transformation, Load - ETL*) u skladište podataka. Ovaj korak čini 80% razvoja BI sistema. Neki od zadataka ETL procesa su: izvlačenje (ekstrakovanje) podataka iz izvornih sistema; transformacija podataka koja podrazumeva čišćenje (npr., mapiranje "Muški pol" u M i "Ženski pol" u Ž), filtriranje (učitavanje samo određenih kolona), razdvajanje kolona u više i obrnuto, spajanje podataka iz više izvora (npr. *lookup, merge*), validaciju podataka, primenu poslovnih pravila i dr.; punjenje podataka u skladište. Ovi koraci su često SQL zasnovani, odnosno koriste komande SQL jezika.



Slika 1.12. Opšti okvir arhitekture inteligentnih poslovnih sistema (adaptirano)

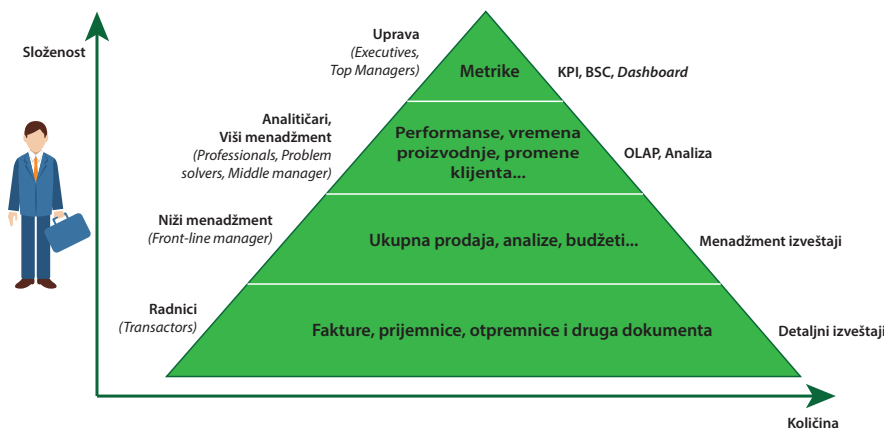
Treći korak je razvoj **skladišta podataka** (*Data Warehouse, DW*). Ovo skladište je namenjeno samo za čitanje podataka, odnosno brzi odziv na upite donosioca odluka. Podaci u skladištu, jednom upisani, ne mogu se menjati, niti brisati. Na primer, nije izvodljivo da neko izmeni podatak, na primer stanje na računu klijenta, koji je unet pre tri godine. S obzirom da se DW koristi samo za čitanje, nema potrebe za normalizacijom podataka, tako da je šema DW potpuno denormalizovana. Arhitektura DW je u obliku tzv. šeme zvezde. To podrazumeva da postoji tabela činjenica (*Fact table, FT*) u okviru koje se nalaze podaci (mere) koje donosioc odluka analizira i oko FT se nalaze dimenzione tabele iz kojih se izvlače vrednosti tih podataka. Strani ključevi se nalaze u FT.

Četvrti korak u razvoju BI sistema je kreiranje **OLAP kocki** (višedimenzionalnih upita) **i primena algoritama mašinskog učenja**. OLAP kocke su upiti koji se prave nad skladištem podataka i koje omogućavaju vršenje raznih operacija, kao što su *drill down* (kopanje u dubinu), *slice* (sečenje tj. analiza po jednoj dimenziji) i dr. Za razliku od obične baze podataka, koja se sastoji od dvodimenzionih tabela koje su povezane stranim ključevima, DW se sastoji od višedimenzionih tabela, te se upiti koji se postavljaju višedimenzionalni. Reč kocka ne znači da se eksplicitno koristi trodimenzionalna struktura, već se samo naglašava da se radi o višedimenzionalnoj strukturi podataka. Kada se za neke zahteve donosioca odluka, ne može postaviti upit, tada se pribegava primeni određenih algoritama mašinskog učenja, koji će izvući neophodno znanje iz DW i prikazati ga korisniku.

Peti korak u razvoju BI sistema je **kreiranje prezentacionog sloja**. Ovaj sloj je interfejs kako donosioc odluka vidi rezultate i obavlja analize i druge operacije nad BI sistemom. Prezentacioni sloj je obično u obliku kontrolnih tabli (*dashboard*) koji koristi jednostavne grafičke prikaze, kako bi korisniku jasno bili vidljivi rezultati i kako bi mu se olakšalo vršenje analiza i donošenje odluka. Međutim, strateški menadžment je više zainteresovan za *scorecard* tip izveštaja gde se prikazuju ključni indikatori performansi (*Key Performance Indicators, KPI*) prema BSC (*Balanced ScoreCard*) metrici. Danas postoji puno *front-end* alata koji u zavisnosti od tipa podataka, asocijacije između njih i dr. automatski generišu željeni izgled izveštaja. Jedan od najkorišćenijih alata u 2020. godini, prema Gartner istraživanjima su Microsoft PowerBI, Tableau i Qlik.

Prilikom razvoja BI sistema veoma je važno znati za koji nivo menadžmenta se razvija, jer svako od njih ima različiti pogled i potrebe za podacima. Postoje tri nivoa menadžmenta (Slika 1.13):

- Najniži nivo menadžmenta ili operativni menadžment ili *font-line* menadžeri (npr. šef recepcije)
- Srednji nivo menadžmenta ili taktički menadžment (npr. marketing menadžer)
- Najviši nivo menadžmenta ili strateški menadžment (npr. generalni menadžer).



Slika 1.13. Nivoi izveštavanja

POGLAVLJE 2



MODELOVANJE SOFTVERA

Modelovanje softvera

2.1. Uvod u modele i modelovanje

2.2. Vrste modelovanja softvera

Cilj poglavlja:

Cilj ovog poglavlja je da postavi konceptualni okvir za razumevanje uloge modela u razvoju softvera. Fokus je na savladavanju osnovnih principa modelovanja kao ključnog alata za apstrakciju i specifikaciju softverskih rešenja, uz upoznavanje sa različitim pristupima koji omogućavaju preciznu komunikaciju između inženjera i ostalih učesnika u projektu.

2. MODELOVANJE SOFTVERA

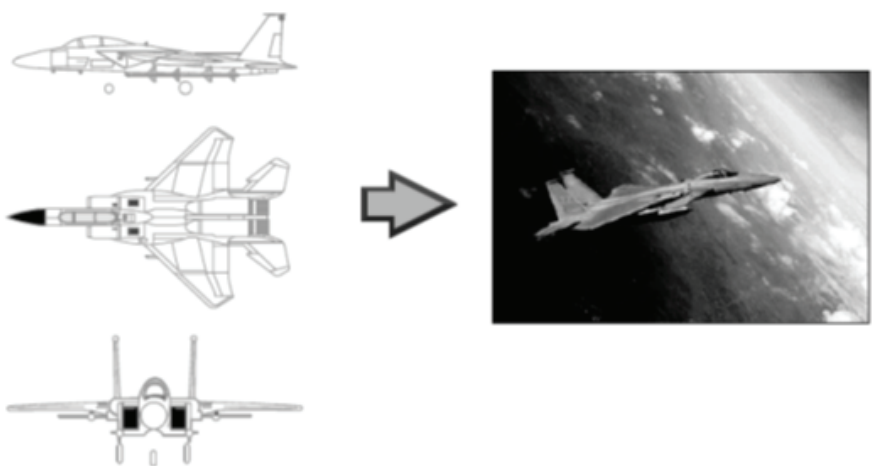
2.1. UVOD U MODELE I MODELOVANJE

U skoro svakoj oblasti inženjerstva, na primer u automobilskoj industriji, prvo se prave modeli kola, nad kojima se sprovode razna testiranja od kvaliteta, funkcionalnosti, performansi itd, a zatim se ulazi u fazu proizvodnje, tako i u softverskom inženjerstvu, prvo se kreiraju modeli budućeg softvera, vrše razna testiranja, a zatim se proizvodi, odnosno razvija pomoću razvojnih okruženja.

U softverskom inženjerstvu, modeli predstavljaju sheme, dijagrame, prototipe i sl., koji opisuju softver iz različitih perspektiva, isto kao što arhitekta zgrade na različitim arhitektonskim crtežima odnosno planovima prikazuje različite aspekte građevinskog projekta.

Model je slikovita predstava realnosti gde se uglavnom koriste grafički elementi (najčešće pravougaonici i linije) kako bi se pojednostavio prikaz i osiguralo jednoznačno razumevanje složenog sistema od strane svih zainteresovanih stana (menadžera, korisnika, projektanata, developera i dr.). Generalno, modeli predstavljaju apstraktne prikaze sistema koji opisuju ključne elemente kompleksnog problema ili strukture, eliminišući manje bitne detalje i na taj način problem čine daleko lakšim za razumevanje (Slika 2.1). Modeli su korisni za razumevanje trenutnog stanja sistema, novih korisničkih zahteva, služe za komunikaciju među ljudima koji su uključeni u projekat, za pripremu dokumentacije i projektovanje sistema koji se razvija.

Modelovanje softvera je proces kreiranja modela softvera koji pomaže u razumevanju, analizi, projektovanju i dokumentovanju softverskog sistema pre nego što se razvije. Dakle, rezultat modelovanja je model.



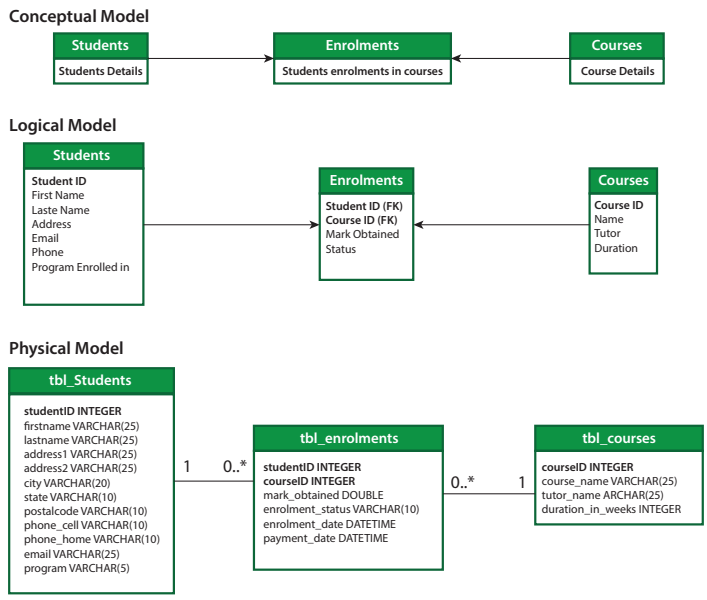
Slika 2.1. Modelovanje aviona

Modeli mogu biti **neformalni** i **formalni**. Neformalni modeli se mogu koristiti samo za dokumentaciju, dok se formalni mogu izvršavati, kompajlirati ili prevesti u rezultate koji se mogu kompajlirati.

Prilikom modelovanja, posebno složenijih informacionih sistema, prvo se prave konceptualni modeli, zatim logički modeli koji se kasnije prevode u fizičke modele. **Konceptualni modeli** pokazuju veoma grubo šta je sistem, to su apstraktni modeli visokog nivoa koji se fokusiraju na konceptualne strukture sistema, a ne na specifične tehničke detalje. Na primer, ukoliko bi se modelovala baza podataka onda bi model sadržao samo nazive entiteta (objekata) i njihove veze, bez ulaženja u detalje o atributima i sl. (Slika 2.2)

Logički modeli su detaljniji od konceptualnih, opisuju podatke, algoritme i logiku sistema bez vezivanja za specifične tehnologije. Ovi modeli pokazuju šta je sistem i šta on radi, odnosno opisuju sistem nezavisno od bilo koje tehničke implementacije. Na primeru baze podataka, logički model bi bio u vidu modela objekti-veze (*Entity Relationship, ER*), na kome se pored naziva entiteta vide i atributi i ključevi.

Fizički modeli pokazuju ne samo šta je sistem i šta on radi, već i kako je sistem fizički i tehnički implementiran na specifičnoj platformi. U kontekstu baze podataka, fizički model bi uključivao i tipove podataka, indekse, particije podataka itd. Sinonimi za fizički model su implementacioni model, tehnički model i sl.



Slika 2.2. Poređenje konceptualnog, logičkog i fizičkog modela

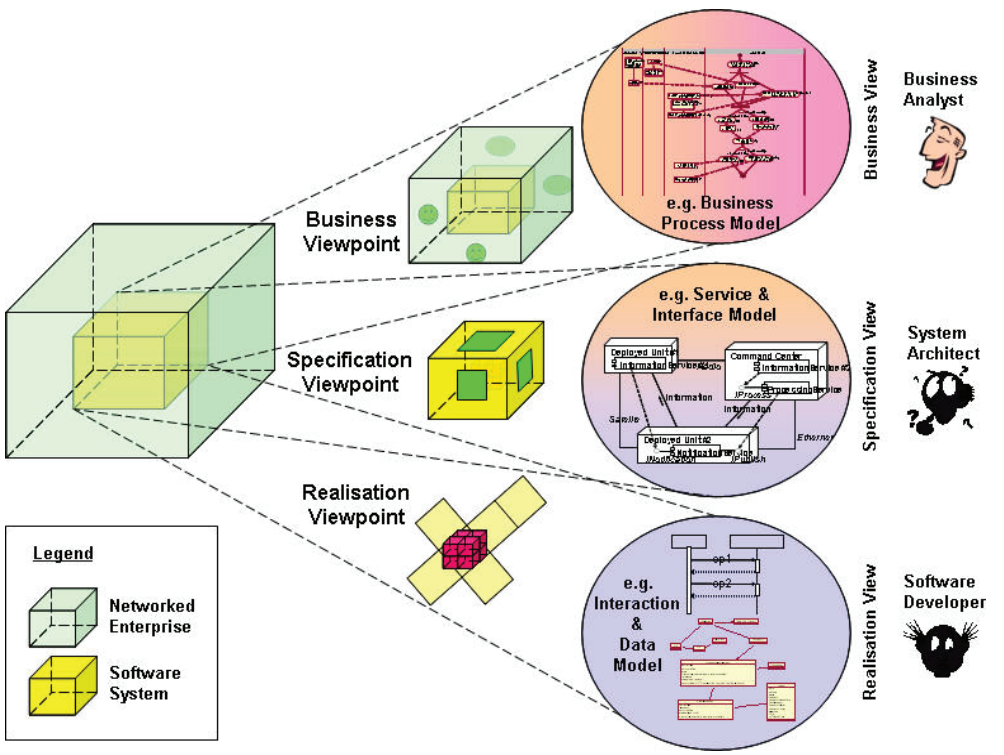
Cilj u razvoju modela jeste da se smanji vreme kodiranja, odnosno da se samo iscrtavanjem modela generiše programski kod. Ukoliko je projektant softvera dovoljno iskusan i dobar u modelovanju softvera, dobijeni generisani programski kôd će biti i odmah upotrebljiv, bez dodatnih modifikacija.

2.2. VRSTE MODELOVANJA SOFTVERA

Pod modelovanjem se podrazumeva proces izrade pojednostavljenog prototipa softverskog sistema, odnosno projektovanje softverskih aplikacija pre njihovog kodiranja. Trenutno i buduće stanje sistema se modeluje kako bi se identifikovali procesi, korisnički zahtevi, problemi, rizici i informacije koje nedostaju ili nisu jasne. Istraživanja pokazuju da veliki softverski projekti koji nisu dobro modelovani imaju veću verovatnoću na neuspeh.

Modelovanjem se dostižu četiri cilja (Slika 2.3):

- Vizuelizacija sistema – model pomaže timu da lako sagleda sistem sa svih perspektiva
- Specifikacija strukture i ponašanja sistema – modeli dokumentuju ponašanje i strukturu sistema pre njegovog kodiranja
- Model daje smernice za razvoj sistema - služi kao vodič za developere
- Modeli dokumentuju diskusije tj. odluke donešene tokom projektovanja sistema.



Slika 2.3. Različite perspektive modela

Postoji nekoliko vrsta modelovanja softvera, od kojih su neki:

- Strukturalno modelovanje: Fokusira se na strukturu sistema i kako su njegove komponente povezane i organizovane.
- Funkcionalno modelovanje: Opisuje funkcionalnost sistema, tj. šta sistem radi.
- Modelovanje ponašanja sistema: Odnosi se na ponašanje sistema u različitim uslovima i interakcijama.
- Modelovanje arhitekture sistema: Odnosi se na organizaciju komponenti sistema i njihove međusobne veze.

Kako je modelovanje opšti pristup u skoro svim inženjerskim disciplinama, u svakoj oblasti postoji više jezika modelovanja i alata. U oblasti razvoja softvera, modelovanje se može izvoditi tradicionalnim pristupom, odnosno definisanjem modela podataka i modela procesa ili pak objektno-orijentisanim pristupom korišćenjem UML (*Unified Modeling Language*) jezika ili servisno-orijentisanim upotrebom BPMN (*Business Process Model and Notation*) notacije.

Za postupak modelovanja razvijeni su razni CASE (*Computer-Aided Software Engineering*) alati koji pomažu inženjerima u automatizaciji mnogih zadataka koji se javljaju tokom softverskog inženjerskog procesa, čineći ga efikasnijim i manje sklonim greškama.

CASE alati su softverski alati u koje je ugrađena određena metodologija razvoja softvera u cilju podrške različitih faza razvoja softvera, kao što su:

- Planiranje: Alati za projektno upravljanje koji pomažu u planiranju resursa, praćenju napretka i upravljanju vremenskim rokovima.
- Analiza: Alati za analizu zahteva koji pomažu u prikupljanju, dokumentovanju i validaciji zahteva korisnika.
- Projektovanje: Alati za modelovanje sistema koji omogućavaju kreiranje dijagrama i modela koji opisuju strukturu i ponašanje softverskog sistema.
- Implementacija: Alati koji podržavaju kodiranje, generisanje koda iz modela, upravljanje verzijama koda i integraciju.
- Testiranje: Alati za automatizaciju testiranja koji omogućavaju kreiranje i izvršavanje test slučajeva, praćenje grešaka i analizu rezultata testiranja.
- Održavanje: Alati za praćenje problema i upravljanje promenama koji pomažu u održavanju i unapređenju softverskog sistema nakon njegovog puštanja u rad.

CASE alati omogućavaju:

- izvršenje sistemske analize i projektovanja informacionih sistema na svim nivoima;
- stvaranje dokumentacije kao osnova za integraciju informacionog sistema i ISO 9000 standarda;
- bolju komunikaciju između menadžera, korisnika, projektanta i programera;
- efikasno upravljanje velikim i složenim projektima.

CASE alati su ključni za moderni softverski inženjering jer omogućavaju efikasniji, konzistentniji i kvalitetniji razvoj softverskih sistema. Danas je na raspolaganju mnogo CASE alata koji su popularni među razvojnim timovima zbog njihove sposobnosti da automatizuju, poboljšaju kvalitet i efikasnost razvoja softvera. Neki CASE alati su: IBM *Rational Software Architect, Enterprise Architect* i mnogi drugi. Ovi alati nude razne funkcionalnosti kao što su automatizacija dokumentacije, vizuelno modelovanje, generisanje programskog koda, integracija sa drugim alatima za razvoj i testiranje, kao i podrška za različite razvojne metodologije.

POGLAVLJE 3



METODOLOGIJE RAZVOJA SOFTVERA

Metodologije razvoja softvera

3.1. Uvod u metodologije razvoja softvera
3.2. Tradicionalne metodologije razvoja softvera
3.3. Agilne metodologije razvoja softvera

Cilj poglavlja:

Cilj ovog poglavlja je da objasni uticaj izabrane metodologije razvoja na proces specifikacije i modelovanja softverskih sistema. Kroz komparativnu analizu tradicionalnih (prediktivnih) i agilnih (adaptivnih) metodologija, fokus je na razumevanju načina na koji različiti okviri diktiraju upotrebu specifičnih tehnika — od formalnog modelovanja putem dijagrama do narativnih tehnika poput korisničkih priča (*User Stories*). Poglavlje teži da osposobi čitaoca za odabir adekvatnog metodološkog pristupa u zavisnosti od dinamike projekta i zahteva tržišta.

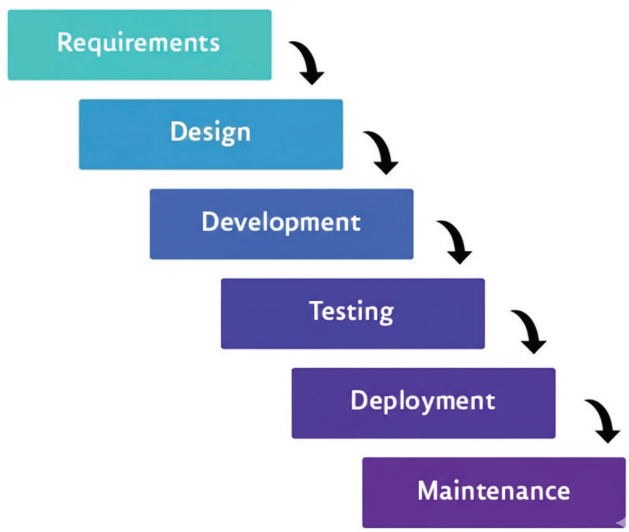
3. METODOLOGIJE RAZVOJA SOFTVERA

3.1. UVOD U METODOLOGIJE RAZVOJA SOFTVERA

Metodologije razvoja softvera su provbitno nastale kao potreba za organizovanjem tima, grupisanjem aktivnosti i procesa, sistematskim i efikasnim pristupom kroz primenu dobrih praksi razvoja softvera.

Termin „metodologija“ u kontekstu razvoja softvera, prvi put je upotrebio Herbert D. Benington, 1956. godine, tokom prezentacije svog rada "*Production of Large Computer Programs*" na simpozijumu o naprednim programskim metodama. U ovom radu, Benington je predstavio sistematičan pristup razvoju velikih softverskih sistema, uključujući faze planiranja, analize, dizajna, implementacije i testiranja, koristeći iskustva iz projekta SAGE (*Semi-Automatic Ground Environment*) - jedan od najranijih i najkompleksnijih softverskih projekata tog vremena. Inače, SAGE je bio softverski sistem za vazdušnu odbranu koji je integrisao radare, računare i komunikacione sisteme za praćenje i presretanje neprijateljskih aviona.

Iako Benington nije koristio termin "Vodopad" (*Waterfall*), njegovi radovi su kasnije interpretirani i dalje razvijeni u ono što je postalo poznato kao Vodopad model, koji je formalno predstavljen 1970. godine od strane Winstona W. Roycea (Slika 3.1).



Slika 3.1. Vodopad model

3.1.1. Evolutivni razvoj metodologija

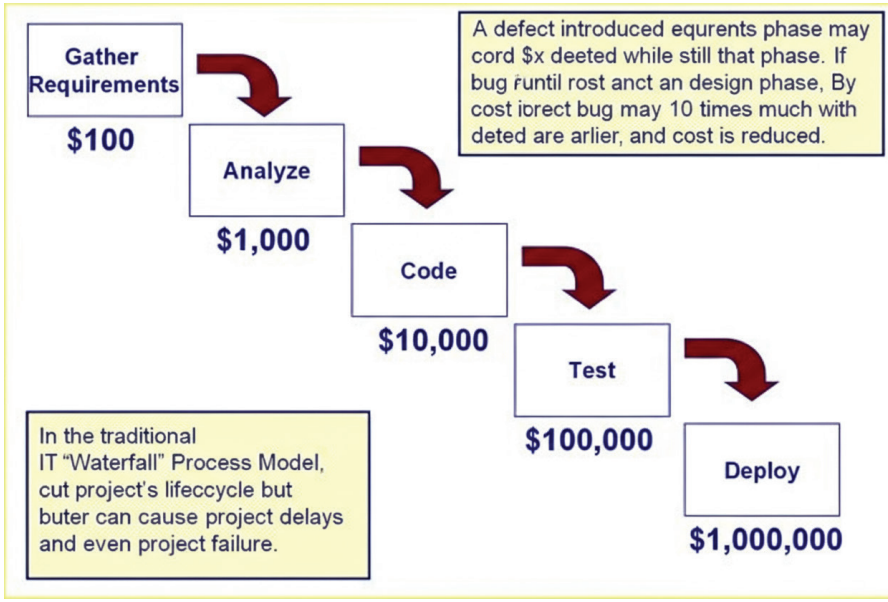
Evolucija metodologija može se pratiti kroz nekoliko ključnih faza i događaja:

1. Rani razvoj softvera (1950-1960-ih)

U ranim danima razvoja softvera, nije bilo jasno definisanih metodologija. Softver je razvijan ad hoc pristupima, što je često dovodilo do problema sa kvalitetom, održavanjem i vremenskim rokovima.

2. Vodopad (*Waterfall*) **model** (1970-ih)

Vodopad model je jedan od prvih formalnih modela za razvoj softvera koji je uveo sekvencijalni pristup razvoju softvera, gde se aktivnosti odvijaju u jasno definisanim fazama: analiza zahteva, dizajn, implementacija, testiranje i održavanje. Iako je pružio strukturu, nedostatak fleksibilnosti i problemi sa kasnim otkrivanjem grešaka (Slika 3.2) doveli su do potrebe za novim pristupima.

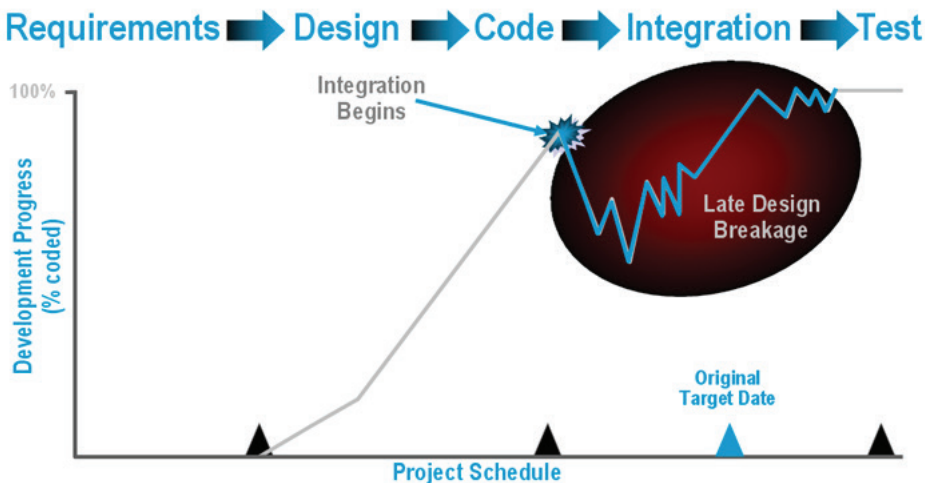


Slika 3.2. Troškovi prepoznavanja grešaka po fazama vodopad modela

Softver razvijan prema vodopad modelu nije mogao adekvatno da se izbori sa sve većom kompleksnošću koja nastaje:

- produženim trajanjem projekta
- sve većom veličinom aplikacije
- velikim ili distribuiranim timovima
- povećanom tehničkom kompleksnošću
- novinama u tehnologijama.

Glavni problem ovog pristupa je što ne omogućava identifikovanje i umanjenje rizika u ranim fazama projekta, te nakon završetka projekta i njegovog uvođenja u realno, tzv. produkciono okruženje, doživljava krah u smislu pojave bagova, nezadovoljstva korisnika što generalno dovodi do sporije integracije (Slika 3.3). Period stabilizacije takvog rešenja duže traje.



Slika 3.3. Problemi uvođenja softvera u produkciono okruženje

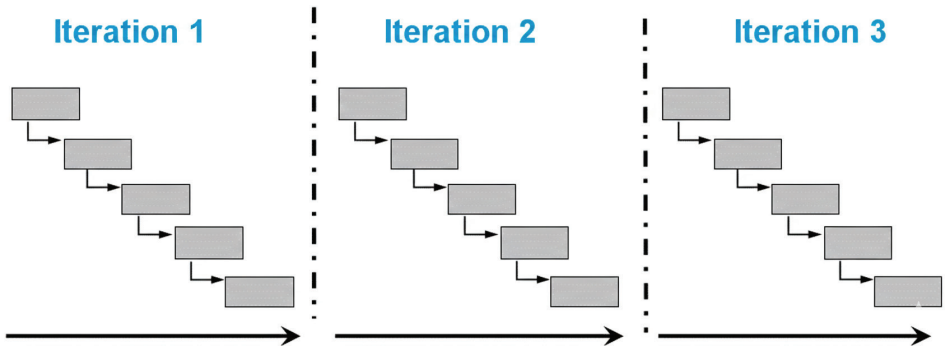
3. Iterativni i inkrementalni modeli (1980-ih)

Kako su projekti postajali sve složeniji, pojavila se potreba za fleksibilnijim pristupima. Iterativni modeli, poput spiralnog modela koji je predložio Barry Boehm 1986. godine, omogućili su razvoj softvera kroz ponovljene cikluse (iteracije), gde se svaki ciklus sastoji od planiranja, dizajna, implementacije i evaluacije. Ovaj pristup omogućava ranu identifikaciju problema i prilagođavanje novonastalim korisničkim zahtevima tokom razvoja.

Iterativno inkrementalni razvoj softvera je pristup u razvoju softvera koji se fokusira na postepeno poboljšavanje i proširivanje softverskog proizvoda kroz ponovljene cikluse (iteracije). Ključni aspekti ovog pristupa su (Slika 3.4):

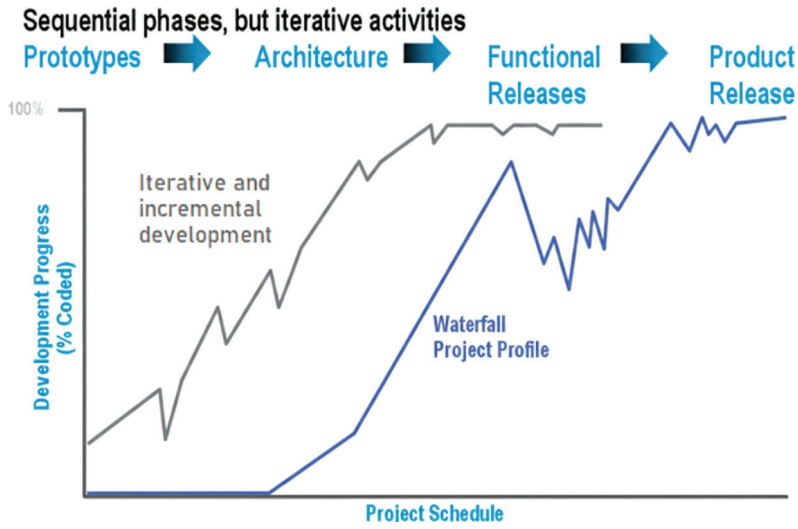
- **Iteracije**: Svaka iteracija je poseban mini-projekat koji se sastoji od aktivnosti kao što su analiza zahteva, dizajn, programiranje i testiranje. Iteracije u prvim fazama ukazuju na najveće rizike. Svaka iteracija proizvodi izvršne verzije i uključuje integraciju i testiranje. Cilj završetka iteracije je stabilan, integrisan, testiran deo celokupnog softverskog sistema koji se gradi.
- **Inkrementalni razvoj:** Umesto da se softver razvija u celini odjednom, funkcionalnosti se dodaju postepeno kroz iteracije. Svaka nova verzija softvera (inkrement) sadrži dodatne funkcionalnosti i poboljšanja u odnosu na prethodnu.
- **Fleksibilnost**: Ovaj pristup omogućava timovima da brzo reaguju na promene zahteva i povratne informacije korisnika. Promene i poboljšanja mogu se lako integrisati u narednim iteracijama.

- **Kontinuirana evaluacija**: Na kraju svake iteracije, tim evaluira napredak, identifikuje probleme i planira naredne korake. Ovo omogućava stalno poboljšavanje kvaliteta i funkcionalnosti softvera.
- **Povećana saradnja**: Iterativno inkrementalni pristup podstiče bolju komunikaciju i saradnju među članovima tima i sa korisnicima, jer se povratne informacije redovno prikupljaju i implementiraju. Na primer, u prvoj iteraciji može biti razvijena osnovna funkcionalnost proizvoda. U narednim iteracijama, na osnovu povratnih informacija korisnika, dodaju se nove funkcionalnosti i poboljšanja, sve dok se ne postigne željeni nivo kvaliteta i kompletiranosti softverskog proizvoda.



Slika 3.4. Iterativno inkrementalni proces razvoja softvera

Na slici 3.5 se može uporediti vodopad i iterativno-inkrementalni razvoja softvera i brzina njihove integracije u realno okruženje.



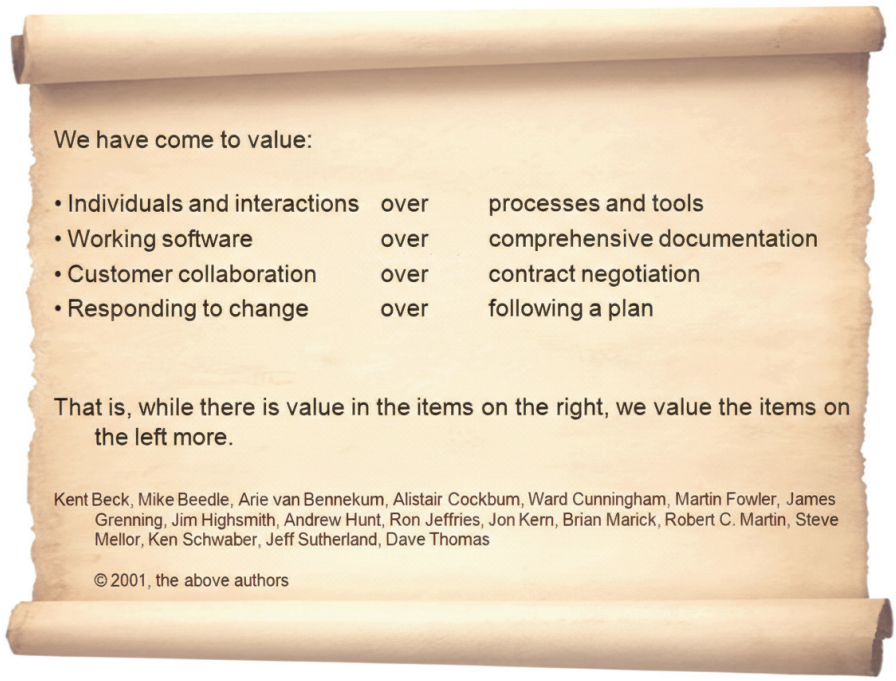
Slika 3.5. Poređenje vodopad i iterativno-inkrementalnog modela

4. Agilne metodologije (2000-ih)

Pored različitih pristupa razvoju softvera i razvijenih metodologija, projekti i dalje bivaju neuspešni bilo da ne ispunjavaju dogovorene korisničke zahteve ili probijaju rokove isporuke ili zahtevaju dodatna ulaganja. Grupa eksperata iz industrije softvera okupila se 2001. godine u Utahu, SAD, i razgovarala o merama poboljšanja procesa razvoja softvera. Rezultat skupa je bio formiranje Agilne Alijanse - neprofitne organizacija koja promoviše koncepte agilnog razvoja i pomaže organizacijama u njihovom usvajanju. Kreira se Agile Manifesto koji definiše osnovne principe i vrednosti agilnog razvoja softvera.

Agilni manifest sadrži četiri osnovne vrednosti (Slika 3.6):

- Pojedinci i interakcije između njih su važnije od procesa i alata
- Softver koji radi je važniji od dokumentacije koja raste
- Saradnja sa korisnicima je važnija od procesa ugovaranja
- Odziv na promene je važniji od striktnog pridržavanja i praćenja plana.



Slika 3.6. Agilni Manifesto

Agilne metodologije, kao što su Scrum, Extreme Programming (XP), Kanban i druge, fokusiraju se na fleksibilnost, brze iteracije, stalnu komunikaciju i saradnju između članova tima i korisnika. Agilne metodologije omogućavaju timovima da se brzo prilagode promenama i isporuče funkcionalne delove softvera u kratkim vremenskim intervalima.

Na narednoj slici 3.7 su prikazan evolutivni razvoj metodologija.



Slika 3.7. Evolutivni razvoj metodologija

3.1.2. Zašto kompanije koriste metodologije?

U današnje vreme, bez primene adekvatne metodologije razvoja softvera, rukovodiocima softverskih projekata bi bilo vrlo teško da organizuju tim, prate izvršavanje dnevnih zadataka po članovima tima, upravljaju verzijama softverskih projekata, kontinuirano prate korisničke zahteve i kontrolišu celokupan razvoj rešenja.

Kompanije koriste metodologije razvoja softvera iz više razloga, koji im pomažu da povećaju efikasnost, kvalitet i uspešnost svojih projekata. Neki od ključnih razloga su:

- **Poboljšana organizacija i planiranje**: Metodologije razvoja softvera pružaju strukturiran okvir za organizaciju i planiranje projekata. Ovo pomaže timovima da jasno definišu ciljeve, zadatke i rokove, što vodi ka boljoj koordinaciji i upravljanju resursima.
- **Efikasna komunikacija i saradnja**: Metodologije, posebno agilne, promovišu redovne sastanke i transparentnu komunikaciju među članovima tima i sa klijentima. Ovo omogućava brzo razjašnjavanje nesporazuma, bržu identifikaciju problema i zajedničko rešavanje izazova.
- **Fleksibilnost i prilagodljivost promenama zahteva i prioriteta**: Metodologije posebno agilne omogućavaju timovima da se brzo prilagode promenama zahteva i prioriteta.
- **Kontinuirana isporuka i poboljšanje**: Iterativni pristup omogućava redovno isporučivanje funkcionalnog softvera, što omogućava ranu detekciju i ispravku grešaka, kao i kontinuirano poboljšanje proizvoda na osnovu povratnih informacija korisnika.
- **Bolje upravljanje rizicima**: Metodologije razvoja softvera uključuju alate i prakse za identifikaciju i upravljanje rizicima tokom celog životnog ciklusa projekta. Ovo pomaže u smanjenju neizvesnosti i povećanju verovatnoće uspeha projekta.

- **Povećana produktivnost i efikasnost**: Standardizovani procesi i najbolje prakse koje su deo metodologija razvoja softvera mogu povećati produktivnost tima, smanjiti nepotrebne zadatke i optimizovati upotrebu resursa.
- **Kontrola kvaliteta**: Metodologije uključuju rigorozne prakse testiranja i integracije koje osiguravaju visok kvalitet softverskih proizvoda.
- **Transparentnost i praćenje napretka**: Metodologije razvoja softvera omogućavaju praćenje napretka kroz metrike, alate za praćenje zadataka i redovne revizije. Ovo omogućava menadžerima projekata da imaju jasan uvid u status projekta i donose informisane odluke.
- **Smanjenje troškova**: Efikasnije planiranje, brža detekcija grešaka i bolja kontrola kvaliteta mogu smanjiti ukupne troškove razvoja softvera, što je ključno za konkurentnost kompanije.
- **Zadovoljstvo korisnika**: Fleksibilnost i sposobnost da se brzo odgovori na korisničke zahteve i povratne informacije povećava zadovoljstvo korisnika, što može dovesti do dugoročne lojalnosti i uspeha proizvoda na tržištu.

Korišćenje metodologija razvoja softvera omogućava kompanijama da sistematski pristupe izazovima u razvoju softvera, smanje rizike, poboljšaju efikasnost, brže i lakše se prilagode promenama tokom trajanja projekta i da isporuče kvalitetne proizvode u skladu sa zahtevima i očekivanjima korisnika.

3.2. TRADICIONALNE METODOLOGIJE RAZVOJA SOFTVERA

U nekim organizacijama se i dalje koriste tradicionalne metodologije, međutim često kombinuju dobre prakse tradicionalnih sa agilnim pristupom. U ovom delu će se ukratko predstaviti dve primenjivane tradicionalne metodologije i to MSF (*Microsoft Solution Framework*) i RUP (*Rational Unified Process*) kako bi se predstavili osnovni pojmovi, aktivnosti i generalno uočile jasne razlike između tradicionalnih i agilnih metodologija.

3.2.1. Microsoft Solution Framework

Kako je na tržištu više bilo neuspelih projekata od uspešnih, Microsoft je analizirao ključne faktore uspešnih projekata koje je ugradio u MSF (*Microsoft Solution Framework*). Prva verzija je objavljena 1993. godine.

MSF je skup principa, modela, disciplina, koncepata i smernica za isporuku IT rešenja od strane Microsofta. Za razliku od konketnih metodologija (npr. CMMI) koje imaju jasno definisane aktivnosti po fazama, gde se ne može krenuti u naredni korak, ako se prethodni nije završio, kod okvira (*framework*) ne postoje striktni koraci koje je potrebno uraditi, već se daju smernice i tim navodi ka uspešnom rešenju.

Prema MSF okviru da bi projekat bio uspešan, neophodno je da upravlja rizicima, timom i procesima razvoja softvera (Slika 3.8).



Slika 3.8. MSF discipline uspešnih projekata

Upravljanje rizicima

Upravljanje rizicima podrazumeva preventivno (proaktivno) upravljanje rizicima, kontinualnu ocenu rizika i odlučivanje tokom životnog ciklusa projekta. Projektni tim upravlja rizicima tako što kreira dokument procene rizika, identifikuje i dokumentuje sve moguće rizike i ocenjuje rizike prema verovatnoći pojave i uticaja na projekat (Tabela 3.1). Primer jednog dokumenta rizika je prikazan u tabeli 3.2.

Tabela 3.1. Osnovni elementi dokumenta procene rizika

Sadržaj	Opis
Opis rizika	Priroda svakog rizika. Brejnstorming metodom se popisuju svi mogući rizici. Na primer, popisan je rizik DoS (*Denial Of Service*).
Verovatnoća rizika	Verovatnoća da se rizik dogodi (npr. u skali od 1 do 10). Za prethodno popisane rizike, prema dogovorenoj skali, označava se verovatnoća da će rizik da se dogodi. Na primer, verovatnoća da će se DoS dogoditi je 65%.
Mera rizika	Uticaj rizika na sistem. Za prethodno popisane rizike razmatra se ukoliko bi se rizik desio, koliki je uticaj na sistem. Na primer, ukoliko bi se DoS dogodio, uticaj na sistem bi bio 100%.
Izloženost riziku	Sveukupna pretnja rizika (verovatnoća * uticaj = izloženost riziku). Izloženost riziku se dobija kada se pomnože dve prethodne kolone i rizici sortiraju od najrizičnijeg do najmanje rizičnog. Na primer, izloženost rizika za DoS je 65 x 100 = 6500.
Planovi umanjenja	Planovi zaštite ili minimizovanja rizika. Obično se odmah rešava prvih pet rizika. Na primer, ukoliko je DoS na vrhu najrizičnijih, potrebno je da se napravi plan zaštite, na primer nabaviti još jedan server.
Planovi akcija	Koraci koji se preduzimaju kada se rizik dogodi. Za one nešto manje rizične, prave se planovi akcija – koraci koji se preduzimaju ako se rizik desi.
Odgovorni za rizik	Naziv osobe koja je zadužena i odgovorna za upravljanje rizikom. Za svaki popisani rizik na kraju se navodi ime osobe koja je odgovorna ukoliko se desi rizik.

Smatra se da upravljanje rizicima štiti tim za razvoj softvera, jer se naručiocu posla ukazuje na potencijalne rizike koji mogu dovesti do produžetka roka isporuke ili nekih dodatnih ulaganja i sl. Na primer, u fazi specifikacije korisničkih zahteva, neophodno je da korisnici budu dostupni, ali ako oni nemaju vremena da timu posvete određeni broj sati tokom dana, faza specifikacije zahteva će se prolongirati što može direktno uticati na rok završetka celokupnog projekta.

Tabela 3.2. Primer dokumenta procene rizika

Rbr	Opis rizika		(Skala 1-100%)	(Skala 1-10)		Umanjenje	Kontigencija	Trigeri	Odgovorni
	Stanje	Posledice	Verovatnoća	Uticaj	Izloženost				
1	Korisnici neće biti dostupni 50% od zahtevanog vremena.	Može doći do odlaganja projekta. Mogu se doneti nekorektne odluke.	80%	9	7,2	Nabaviti pismeni zahtev za angažovanje ugovorenog korisničkog vremena.	Sortirati pitanja prema prioritetu, koja će biti postavljena korisnicima. Odrediti druge izvore informacija.	Početak projekta	Program menadžer
2	Mogući napad radi blokiranja usluga (*denial of service attack* - DoS).	Sistem je nedostupan klijentima i krajnjim korisnicima.	70%	10	7	Ispravljanje sigurnosnih propusta, korišćenje *firewall* tehnologije i procedura za ažuriranje (*update*) tehnologija.	Upotreba softvera za detekciju radi zaštite od DoS-a. Razvijati, testirati i redovno pregledati procedure oporavka.	Početak projekta	IT menadžer, menadžer mreže
3	Ukoliko prodaja opadne, budžet projekta će biti umanjen za 10,000e ili će biti otkazan.	Otkazivanje projekta, odlaganje ili redukcija granica projekta.	50%	10	5	Nisu moguće akcije unutar granica projekta.	Razmotriti odlaganje nekih funkcionalnosti za narednu verziju rešenja.	Kada su poznate izmene budžeta projekta.	Proizvodni menadžer
4	Nepotpuni podaci o klijentima dovode do otežane analize.	Nemogućnost identifikovanja najboljih klijenata.	80%	6	4,8	Utvrditi opseg problema i razmotriti načine prikupljanja podataka koji nedostaju.	Iščistiti podatke izvlačenjem iz živog sistema.	Kada je donešena odluka o procentu nekompletnih podataka i njihovog uticaja na sistem.	Proizvodni menadžer
5	Može se dogoditi da administrator baze podataka ne bude na raspolaganju 50% od ugovorenog vremena.	Projekat bi mogao da se odloži. Mogu biti donešene nekorektne odluke.	30%	6	1,8	Obavezati administratora na ugovoreno vreme.	Sortirati pitanja prema prioritetu i ukoliko je moguće odrediti druge izvore informacija.	Početak projekta	Program menadžer

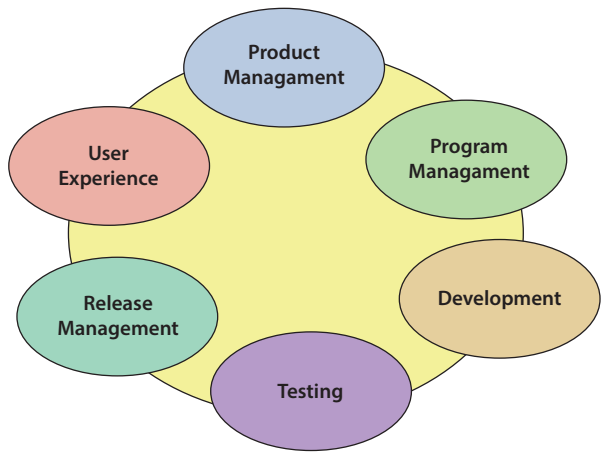
Upravljanje timom

Prema MSF okviru, članovi tima za razvoj softvera su (Slika 3.9):

- **Mendžment proizvoda** (*Product Management*) koji vodi početne razgovore sa naručiocima posla, prikuplja zahteve, dogovara šta će se raditi, uspostavlja rokove, ograničenja i sl. Često se ova rola prema drugim metodologijama zove rukovodioc projekta ili product owner.
- **Menadžment programa** (*Program Management*) koji sastavlja funkcionalnu specifikaciju i odgovoran je za razvoj i isporuku rešenja. Prema drugim metodologijama ova rola se naziva projektant softvera ili softver arhitekta.
- **Razvoj** (*Development*) je odgovoran za razvoj rešenja prema specifikacijama dobijenih od strane menadžera programa. Ova rola se često u praksi naziva developer, programer i sl.
- **Testiranje** (*Testing*) je jedna od važnijih rola, jer testira softver kroz razna scenarija, npr. analizira performanse softvera ukoliko mu istovremeno pristupi 1000 korisnika.

Tester testira sve module kako bi finalno rešenje blo što stabilnije, upoređuje funkcionalnost programa sa ciljevima i vizijom projekta itd.

- **Menadžment otpuštanja projekta** (*Release Management*) bavi se uvođenjem rešenja u realno okruženje, nabavkom potrebnih licenci, hardvera, instalira sistem i generalno dgovoran je za rukovanje i održavanje sistema, ali i za obuku korisnika.
- **Iskustveni korisnik** (*User Experience*) je osoba koja ima dosta iskustva za poslove za koje se razvija softver i može da pomogne u ranim fazama, kako bi softver bio što lakše prihvaćen od strane krajnjih korisnika. On može da unosi podatke, analizira performanse, koriguje ekrane kako bi rešenje bilo što upotrebljivije i prihvaćenije od strane korisnika sistema.



Slika 3.9. Članovi tima prema MSF okviru

Upravljanje procesima

Prema MSF okviru, životni ciklus razvoja softvera prolazi kroz nekoliko faza:

- **Definisanje vizije** (*Envisioning*) u okviru koje menadment proizvoda u saglasnosti sa ostalim članovima tima kreira dokument Vision/Scope koji treba da bude potpisan između obe strane.
- **Faza planiranja** (*Planning*) u okviru koje menadžment programa kreira funkcionalnu specifikaciju rešenja, odnosno projektuje željeni modul sistema.
- **Razvoj** (*Developing*) u okviru koga developeri razvijaju rešenje prema specifikaciji i modelima dostavljenih od strane projektanata.
- **Stabilizacija** (*Stabilizing*) u okviru kojeg testeri rade zajedno sa iskustvenim korisnicima kako bi stabilizovali rešenje.
- **Uvođenje** (*Deploying*) u okviru koda se rešenje uvodi u realno okruženje i ovde glavnu ulogu ima rola release management.



Slika 3.10. MSF upravljanje softverskim prcesima

MSF obezbeđuje skup modela, pravila i smernica za planiranje, projektovanje, razvoj i uvođenje poslovnih rešenja na način koji osigurava da se svim elementima projekta (ljudi, procesi, alati) uspešno upravlja.

3.2.2. *Rational Unified Process* metodologija

Ova metodologija je ugrađena u CASE alat IBM *Rational Software Architect* koji će se koristiti u ovom udžbeniku za objektno-orijentisano modelovanje softvera. RUP metodologija je prvobitno dobila naziv *Unified process*, ali kako ju je sponzorisala tadašnja kompanija *Rational*, dodaje joj se i taj naziv.

Ova metodologija se naziva objedinjeni proces (*Unified Process*) jer nastaje ujedinjavanjem pristupa projektovanja softverskih sistema koji su formulisani od strane tri softverskih eksperata Grady Booch, Ivar Jacobson i James Rumbaugh (Slika 3.11).



Dr Ivar Jacobson je otac komponenata i komponentne arhitekture, slučajeva korišćenja, UML-a i RUP-a



Grady Booch je lider u objektno-orijentisanoj analizi i razvoju UMLa



James Rumbaugh je kreator tehnike objektnog modelovanja (OMT) i UML-a

Slika 3.11. Tvorci RUP metodologije

RUP se oslanja na UML (*Unified Modelling Language*) jezik koji služi za specifikaciju, vizuelizaciju, konstrukciju i dokumentaciju razvoja softvera. Dakle, UML jezik nastaje tokom 1990-ih godina kao rezultat spajanja više različitih metodologija i notacija za modelovanje softvera. Formalno, razvoj UML jezika je započeo 1994. godine kada su ova tri ključna inženjera, Booch, Jacobson i Rumbaugh, koji su predstavljali različite metodologije (*Booch metodologija, Object Modeling Technique - OMT i Object-Oriented Software Engineering*), počeli da rade zajedno kako bi stvorili jedinstveni standardizovani jezik za modelovanje softvera. Ovaj proces je krunisan formalnom specifikacijom UML 1.0 verzije 1997. godine od strane *Object Management Group* (OMG), međunarodne organizacije koja je nadgledala razvoj UML-a i koja ga i danas održava i upravlja njegovim standardima.

RUP je pristup razvoja softvera koji je iterativan, centriran oko arhitekture i vođen slučajevima korišćenja.

Iterativnost podrazumeva da u prvoj iteraciji razvijamo ključne funkcionalnosti softvera koje korisnici često koriste ili od koje imaju koristi odnosno donosi im neku tržišnu vrednost. U današnje vreme ovaj koncept se naziva **MVP** (*Minimal Viable Product*) i posebno je prisutan u agilnom pristupu razvoja softvera.

Međutim, postavlja se pitanje, kako da znamo koje su to prioritetne funkcionalnosti na koje treba se fokusiramo u prvoj iteraciji. Smatra se da korisnici koriste 20% funkcionalnosti, a da 80% vrlo retko koriste. Cilj je da se fokusiramo na ovih 20%. Međutim, često u razgovoru sa korisnicima, u želji da uvidimo tih 20%, dobijamo potpuno pogrešne informacije ili korisnici ističu da im je sve skoro podjednako bitno. Tada iskusan rukovodilac softverskog projekta u intervjuu ispituje koje su to svakodnevne aktivnosti koje korisnik obavlja, kako se odvijaju neki konkretni procesi, koji od njih direktno utiče na profit i sl. Na osnovu te simulacije obavljanja svakodnevnih aktivnosti mogu se izvući one učestale, na koje se fokusira u prvoj iteraciji razvoja.

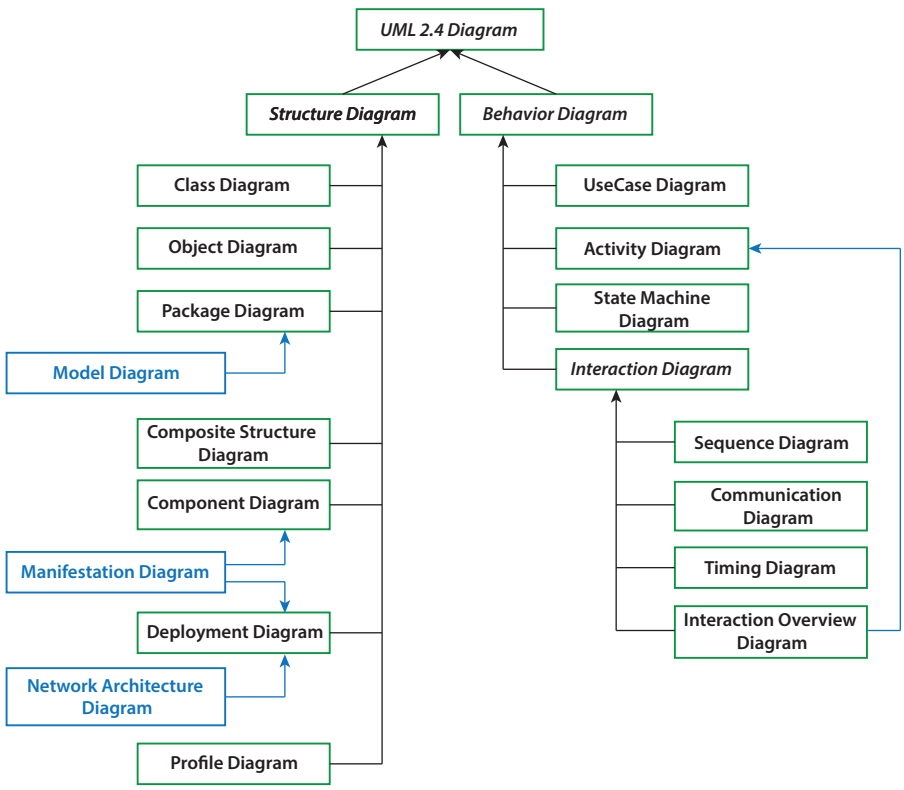
Centriranost oko arhitekture softvera podrazumeva, da pre nego što se upustimo u razvoj softvera, a nakon prikupljanja korisničkih zahteva i specifikacije osnovnih funkcionalnosti softvera, softver arhitekta donosi odluku, a u zavisnosti i od vrste aplikacije koja se razvija, koja arhitektura softvera će se koristiti. Na tržištu postoji dosta obrazaca (*template*) arhitekture softvera i oni mogu biti početni okvir za uspostavljanje konkretne arhitekture aplikacije. Na primer, ako je odlučeno da će se koristiti MVC (*Model-View-Controller*) arhitektura, tada ceo tim definiše i raspoređuje komponente softvera prema toj arhitekturi.

Slučaj korišćenja (*use case*) podrazumeva jednu funkcionalnost aplikacije (npr. plaćanje ili kreiranje šoping korpe i sl.) koja daje merljive rezultate od vrednosti za korisnika. Softver koji je vođen slučajevima korišćenja podrazumeva da se u početnim fazama prikupljanja korisničkih zahteva kreira dijagram slučajeva korišćenja (*use case diagram*), koji sadrži dogovorene *use case*-ove koje će se razvijati. Svaki naredni korak u razvoju softvera je elaboracija tih *use case*-ova. Ukoliko se omaši u tim početnim fazama, svaki dalji korak razvoja će biti pogrešan.

Dakle osnovni koncepti RUP metodologije su:

- **Iterativno inkrementalni** proces
- ***Milestone*** (kontrolna/ključna/kritična tačka) - Svaka faza u razvoju projekta se završava sa nekim *milestone*-om koji sumiraju rezultate svih prethodnih iteracija i u njoj se donose značajne odluke za ceo projekat u celini.
- **Role** - definišu ponašanja i odgovornosti pojedinca ili tima
- **Artefakti**:
  - **Dokumenta**: beleže zahteve sistema, kao i upotrebljivost, pouzdanost, performanse i podršku zahtevima
  - **Modeli**: pojednostavljeni pogled sistema koji prikazuje osnovni sistem bez nepotrebnih detalja
  - **Elementi modela**: pomaže timu da vizuelizuju, konstruišu i dokumentuju strukturu i ponašanja sistema

Modeli su uglavnom opisani UML jezikom. UML sadrži skup dijagrama za opis strukture (npr. dijagram klasa) i dinamike svih vrsta sistema (npr. dijagrami slučajeva korišćenja, aktivnosti, stanja, sekvenci).

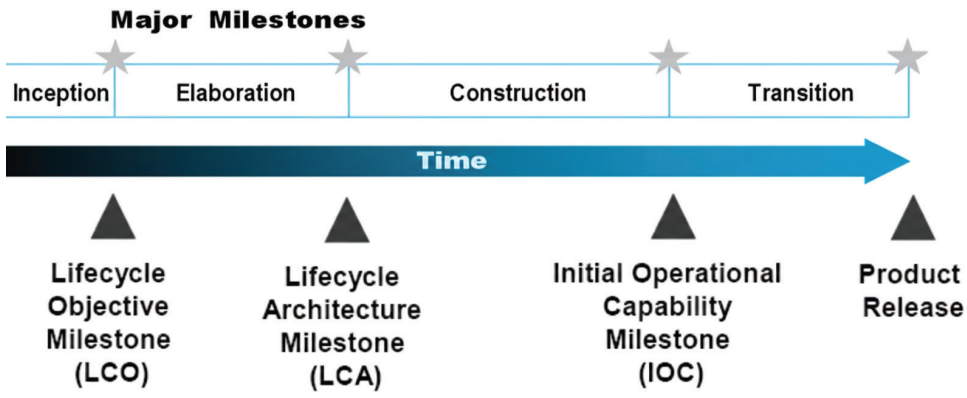


Slika 3.12. Osnovni UML dijagrami podržani verzijom 2.4

UML je standardni vizuelni jezik za specifikaciju zahteva, analizu, projektovanje i implementaciju softverskih sistema. Koriste ga poslovni analitičari, softver arhitekte i developeri za opisivanje, specifikaciju, razvoj i dokumentovanje postojećih ili novih poslovnih procesa, strukture i ponašanja softverskih sistema

Razvoj softvera prema RUP metodologiji prolaze kroz četiri faze (Slika 3.13):

- **Početna faza** (*Inception*) - razumeti **šta** treba graditi:
  - Vizija, zahtevi visokog nivoa, poslovni slučajevi korišćenja
  - Identifikovanje rizika
  - Procena troškova, vremena, plana i kvaliteta proizvoda
  - Inicira se kreiranje poslovne studije opravdanosti ulaženja u projekat i sl.
- **Faza elaboracije** (*Elaboration*) - razumeti **kako** treba graditi:
  - Postavljena osnovna arhitektura
  - Detaljno opisani zahtevi (dijagrami: aktivnosti, sekvenci i klasa)
  - Faza projektovanja softvera nije detaljno urađena
  - Precizna procena resursa i vremena
- **Konstrukcija** (*Construction*) - Izgradnja proizvoda:
  - Faza projektovanja softvera je detaljno urađena (fizički dijagram klasa sa uključenim obrascima projektovanja softvera, dijagram stanja, dijagrami komponenti i raspoređivanja)
  - Faza implementacije urađena
  - Završeno testiranje sistema
- **Tranzicija** (*Transition*) - Validacija rešenja: Prihvatljivost od strane stejkholdera.



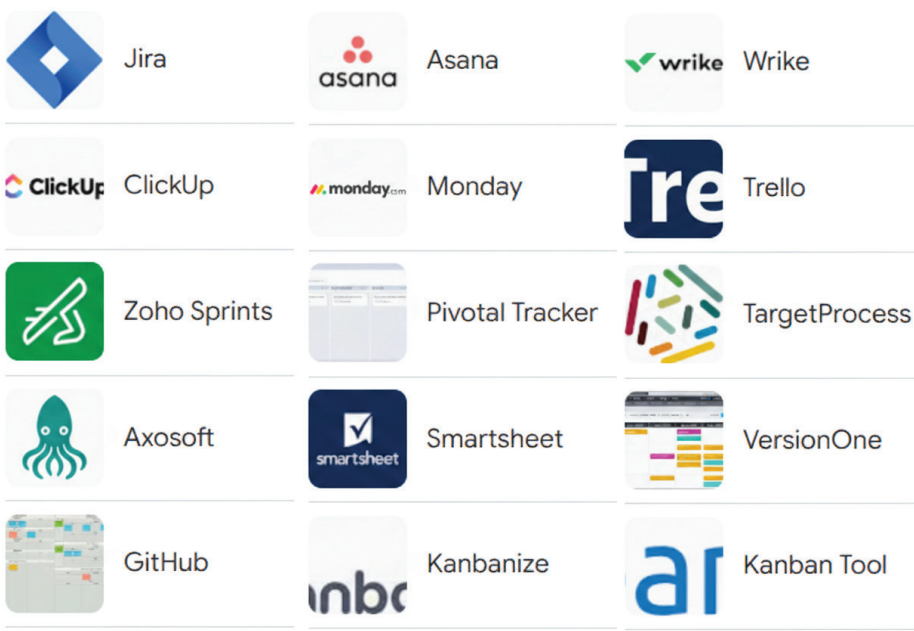
Slika 3.13. Faze razvoja softvera prema RUP metodologiji

Glavne kontrolne tačke su (Slika 3.13):

- LCO (*Lifecycle Objective*): dogovoreno je oko domena i rizici su razumljivi i prihvatljivi
- LCA (*Lifecycle Architecture*): visoki rizici su rešeni i arhitektura je stabilna
- IOC (*Initial Operational Capability*): proizvod je završen i kvalitet je prihvatljiv.

3.3. AGILNE METODOLOGIJE RAZVOJA SOFTVERA

U današnje vreme razvojni timovi uglavnom koriste agilne metodologije za upravljanje softverskim projektima. Najčešće korišćeni okviri su Scrum, Kanban, njihova kombinacija Scrumban, Design Thniknig, DevOps ili njihova kombinacija. Na tržištu su razvijeni razni alati koji podržavaju ove okvire, kao što su Trello, Jira, Asana, Azure DevOps i dr. (Slika 3.14)



Slika 3.14. Alati koji podržavaju agilni pristup razvoju softvera

3.3.1. Scrum

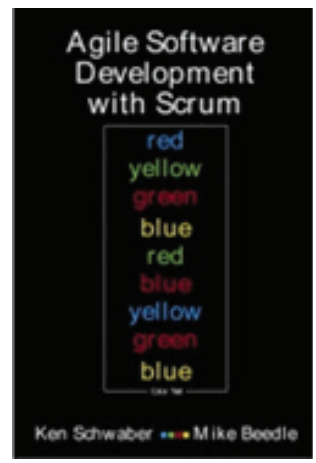
Scrum nije akronim, već termin koji se koristi u ragbiju i označava ponovno pokretanje igre nakon manjeg prekršaja (Slika 3.15).



Slika 3.15. Scrum u ragbiju

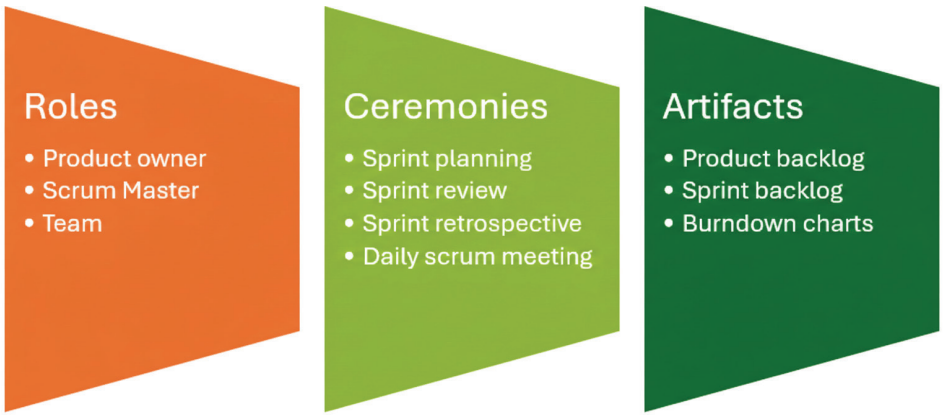
Prva javna prezentacija o Scrum-u, održana je od strane Ken Schwaber i Jeff Sutherland (Slika 3.16). Schwaber zajedno sa Mike Beedle, 2001. godine, opisuje Scrum u knjizi *Agile Software Development with Scrum*.





Slika 3.16. Autori Scrum okvira i prva knjiga o Scrumu

Scrum je okvir (*framework*) za brzi agilni proces **razvoja i održavanja** složenih softverskih proizvoda. Osnovni elementi Scrum okvira su prikazani na slici 3.17.

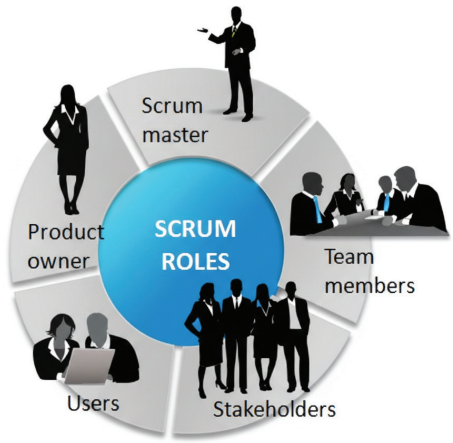


Slika 3.17. Osnovni elementi Scrum okvira

Role prema Scrum okviru

Osnovne role prema Scrum-u su (Slika 3.18):

- **Vlasnik proizvoda** (*Product Owner*)
  - Zastupa klijenta
  - Definiše i postavlja prioritetne funkcionalnosti
  - Odlučuje o datumu i sadržaju verzije
  - Odgovoran za profitabilnost, dinamici povraćaja investicije (*Return on Investment, ROI*) i sl.
  - Prihvata ili odbacuje rezultate sprinta.
- **Scrum Master**
  - Menadžer projekta
  - Osigurava da je tim potpuno funkcionalan i produktivan
  - Osigurava blisku saradnju između svih rola
  - Štiti tim od spoljnjih ometanja
  - Rešava sve probleme tokom sprinta.
- **Razvojni tim** (*Development Team*)
  - Tim čini od pet do devet osoba, različitih uloga (programer, tester, dizajner i sl), svako je stručan u svom domenu.
  - Zahteva se puno radno vreme
  - Timovi se samostalno organizuju, a međuljudski odnosi su bazirani na prijateljstvu, osećanju zajedništva i dr.
  - Timovi se mogu menjati između sprintova.



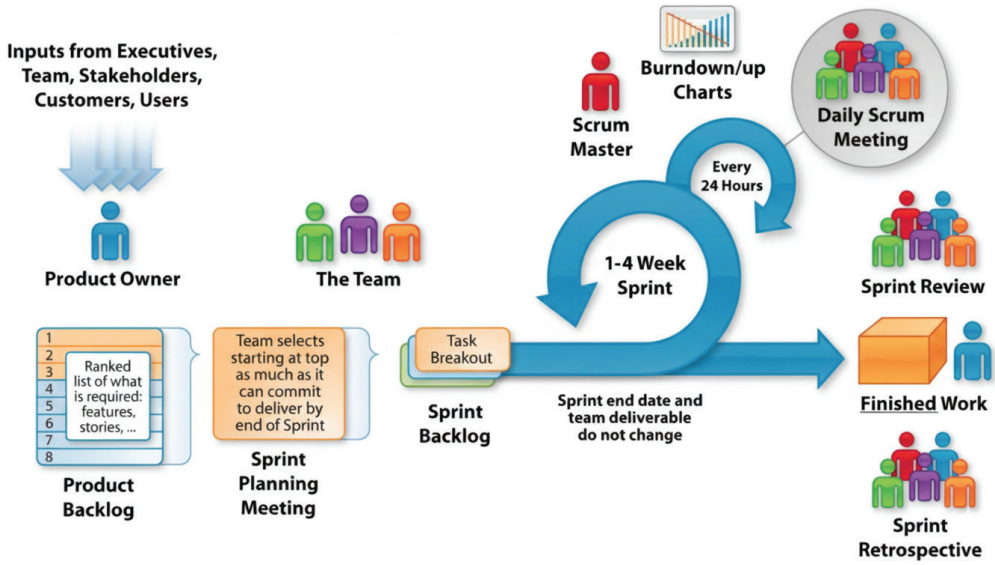
Slika 3.18. Osnovne role prema Scrum okviru

Sastanci

Osnovni koraci Scrum model procesa su (Slika 3.19):

1. *Product owner* kreira *Product backlog* koji čini spisak funkcionalnosti celokupnog projekta, opisanih u vidu korisničkih priča i sortiranih prema prioritetu.
2. Na **sastanku planiranja sprinta** (*Sprint Planning Meeting*), Scrum master izdvaja onoliko prioritetnih funkcionanosti koliko može da realizuje u toku jednog sprinta (obično u trajanju od dve do četiri nedelje). Korisničke priče sa *product backlog*-a se dele na konkretne zadatke, gde se svakom zadatku dodeljuju težinski poeni. Lista konkretnih zadataka tokom jednog sprinta naziva se *Sprint backlog*. Zadaci se raspodeljuju po članovima tima.
3. Tokom sprinta se organizuju dnevni kratki stojeći sastanci (*daily standup meetings*) u trajanju oko 15-ak minuta, gde se informišu šta je neko radio prethodnog dana i šta namerava da radi i da li ima nekih problema.
4. Tokom sprinta se potpuno razvijaju izabrane funkcionalnosti koje se mogu na kraju sprinta pokazati *product owner*-u.
5. Sprint se ne prekida. Jedina osoba koja može prekinuti sprint je *Sprint master*. Na primer, ukoliko je tim u kašnjenju, zakazuje se sastanak na kome treba da se sagledaju razlozi neuspeha i izvedu zaključci.
6. Sprint je uspeo ako su spaljeni svi težinski poeni zadataka koje je tim predvideo za konkretni sprint i za predviđeno vreme.
7. Posle svakog sprinta, tim organizuje sastanak pregleda sprinta (*Sprint review*) gde se prikazuje šta je urađeno.
8. Ukoliko je tim zadovoljan urađenim, organizuje se naredni sastanak tzv. retrospektiva sprinta (*Sprint Retrospective*) sa *product owner*-om i drugim zainteresovanim stranama.

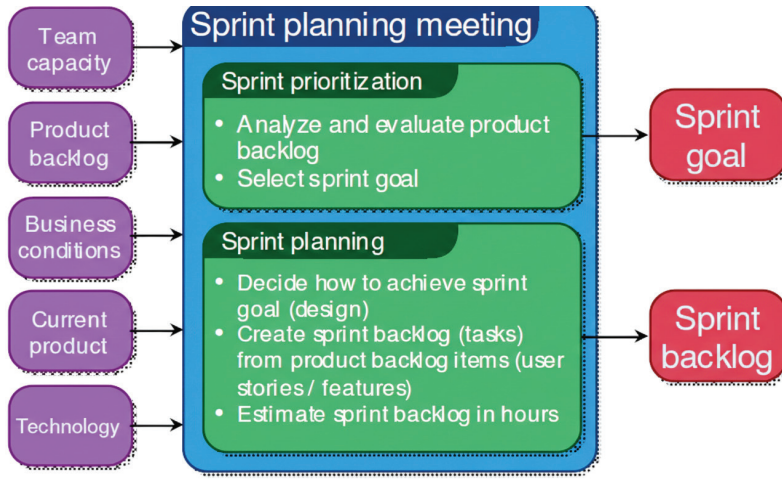
9. Ako klijent prihvati rezultate, onda se funkcionalnosti mogu postaviti u produkciono okruženje. Međutim, često zbog turbulentnih promena u okruženju, korisnik može odlučiti i da prekine projekat ili promeni dalje zahteve na projektu.



Slika 3.19. Scrum model procesa

Detaljnije aktivnosti tokom **sastanka planiranja sprinta** (*Sprint Planning Meeting/ Release Planing*) su:

- Tim bira zadatke za koje se obavezuje da će završiti:
  - Sa *Product backlog*-a identifikuju korisničke priče (*user stories*) koje žele da realizuju u narednom sprintu.
  - Sortiraju izabrane korisničke priče prema prioritetu i određuju njihovo vremensko trajanje, tj. kreira se *Release backlog*.
  - Dogovaraju se cilju sprinta.
- Odgovara se na pitanja:
  - Šta će biti urađeno u sprintu?
  - Kako će se izabrani posao uraditi?
- Rezultat sastanka je kreiran *Sprint backlog* gde su identifikovani zadaci, procenjena njihova vremena (1-16 sati) i dodeljeni težinski poeni.



Slika 3.20. Zadaci tokom sastanka planiranja sprinta

Karakteristike **dnevnih Scrum sastanaka** (*Daily Scrum meeting*) su:

- 15. minutni, stojeći sastanak kako bi se izbegli dugi nepotrebni sastanci.
- Članovi tima odgovaraju na pitanja:
  - Šta sam uradio juče?
  - Da li je nešto ostalo nezavršeno?
  - Šta radim danas?
  - Da li me nešto usporava i da li mi nešto stoji na putu?
- Ne rešavaju se problemi.
- Nije merilo kontrole, već poveravanje i obavezivanje drugim članovima tima.



Slika 3.21. Dnevni Scrum sastanak

Nakon završenog sprinta, organizuje se sastanak pregleda sprinta (*Sprint Review*). Osnovne karakteristike ovog sastanka su:

- Tim prezentuje šta je urađeno tokom sprinta
- Obično je u formi demo novih funkcionalnosti
- Neformalan je, nema slajdova, dva sata pripreme
- Učestvuju svi članovi tima
- Svi su pozvani.


Demo

Ukoliko je tim zadovoljan urađenim, naredni sastanak koji se organizuje je sa *product owner*-om i eventualno sponzorima ili naručiocima posla. Karakteristike sastanaka retrospektive sprinta (*Sprint retrospective*) su:

- Organizuje se nakon svakog sprinta u trajanju od 15 do 30 min.
- Svi timovi učestvuju, a mogu prisustvovati i klijenti.
- Može se periodično organizovani kako bi se sagledalo koje funkcionalnosti su razvijene i lepo rade, a koje nisu, kako bi se identifikovale mere poboljšanja.
- Nakon prezentovanja urađenog i diskusije o potencijalnim problemima, ceo tim odlučuje o tome da li nastaviti ili započeti sa novim funkcionalnostima ili prekinuti ceo projekat.


Retrospective

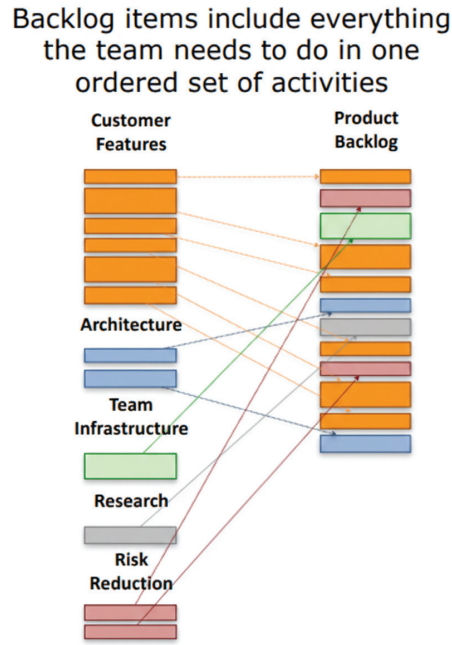
Artefakti

Product backlog

Prvi korak je kreiranje plana projekta tzv. *Product backlog*-a, koji sadrži listu svih korisničkih zahteva, funkcionalnosti, poboljšanja i popravki koje je potrebno implementirati u budućim sprintovima, opisanih u vidu korisničkih priča (*user stories*). Dakle, product backlog su:

- zahtevi koji imaju konkretnu vrednost za klijente
- lista svih željenih poslova na projektu napisanih u vidu korisničkih priča (*user stories*).

Product owner sortira korisničke priče prema prioritetu. Pre početka svakog sprinta ti prioriteti se ponovo razmatraju i mogu se dodavati nove funkcionalnosti ili ukloniti postojeće.



Slika 3.22. Product backlog

Korisničke priče (*User Stories*) predstavljaju mali deo funkcionalnosti kojoj obično treba od jednog do tri dana da se završi. Pomažu da se određena funkcionalnost razbije na manje i jasne delove. Korisničke priče treba da budu napisane sa perspektive korisnika. Format korisničkih priča koji bi trebao da se prati je (Slika 3.23):

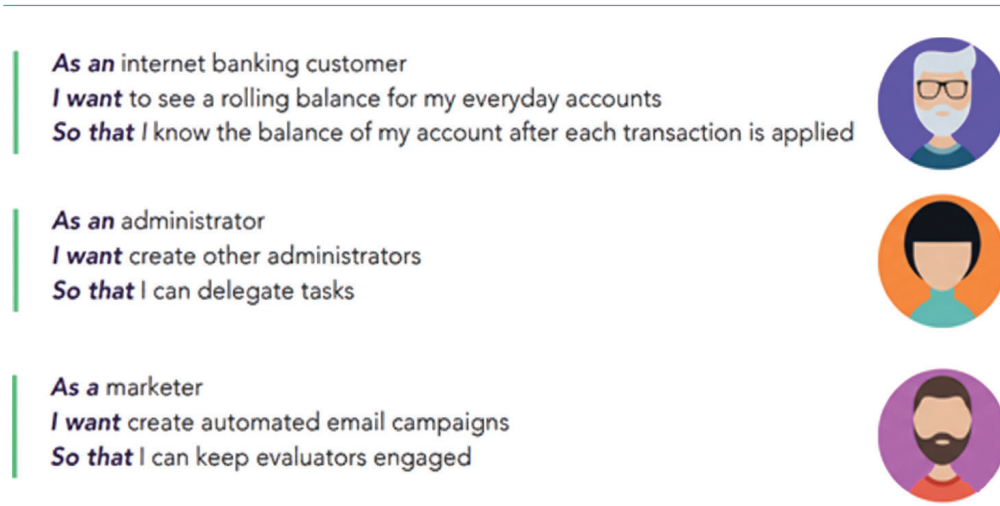
As a (role/user)	Kao (tip korisnika)
I want to (functionality)	želim da (uradim)
So that (benefit)	kako bih mogao (ostvariti određeni cilj)

WHO are we building it for? Who is the user?	**As a <type of user>**
WHAT are we building? What is the intention?	**I want <some goal or objective>**
WHY are we building it? What is the value for the customer?	**So that <benefit/value>**

Slika 3.23. Format korisničke priče

Primeri korisničkih priča (*User stories*) su (Slika 3.24):

As an *online* shopper,
I want to *add an item to my cart*,
so that *I can purchase it*



Slika 3.24. Primeri korisničkih priča

Prilikom kreiranja korisničkih priča treba imati na umu sledeće:

- Neka budu kratke (*Keep them short*)
- Neka budu jednostavne (*Keep them simple*)
- Pišite iz perspektive korisnika (*Write from the perspective of the user*)
- Jasno istaknite vrednost ili korist priče – šta je razlog za njeno postojanje? (*Make the value/benefit of the story clear - what is the reason for the story?*)
- Opišite samo jednu funkcionalnost. Ako morate da upotrebite veznik „i", podelite je na dve priče (*Describe one piece of functionality. If you have to write "and", break it into 2 stories*)
- Pišite priče timski (*Write stories as a team*)
- Koristite kriterijume prihvatanja da biste definisali MVP (*Use acceptance criteria to show a MVP*)

Korisnik može da prihvati ili odbaci korisničke priče. Kriterijumi prihvatljivost pomažu da:

- se razjasni šta tačno tim treba da radi
- osigura da svi imaju isto razumevanje problema
- članovi tima znaju kada je priča završena
- pomogne u verfikovanju priča kroz automatske testove.

Na primer, za korisničku priču:

> **"As an online banking customer, I want a strong password so that my credit card information is secure."**

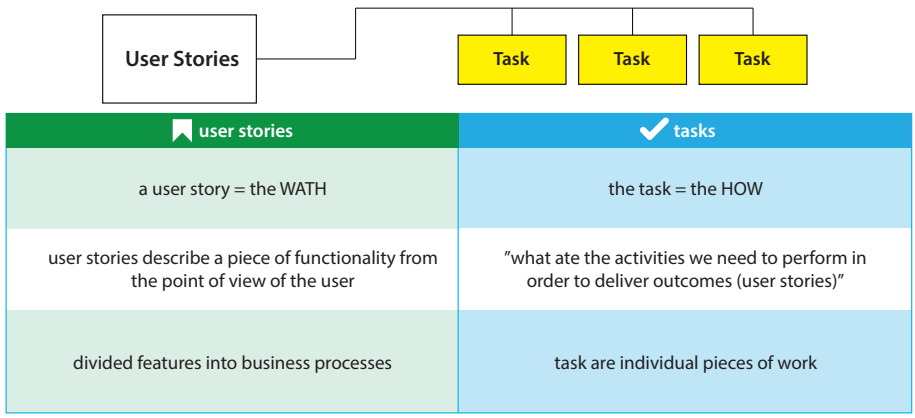
Kriterijumi prihvatljivosti su:

Acceptance Criteria:
- The password must be at least 8 characters.
- The password must contain at least 1 character from each of the following groups: lower case alphabet, upper case alphabet, numeric, special characters (!, @, #, $, %, ^, &, *).

Scrum backlog

Scrum master na osnovu product backlog-a, kreira sprint backlog. Svaki sprint se posebno planira. Koraci kreiranja sprint su:

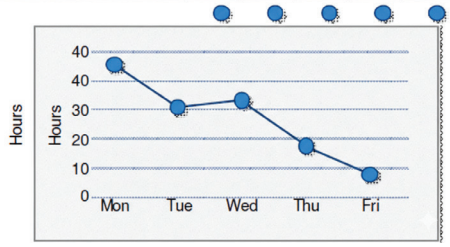
- Identifikovati cilj sprinta - kratak iskaz o tome šta je fokus u narednom sprintu.
- Tim bira zahteve iz *product backlog*-a i kreira *sprint backlog*.
- Jedna stavka iz *product backloga* se obično opisuje sa više zadataka (*tasks*) u *sprint backlog*-u (Slika 3.26).
- Tim licitacijom daje težinske poene (*Story points*) zadacima (težina izvođenja zadataka)
- Za praćenje napredovanja sprinta može da se kreira *sprint burndown* grafikon u koji se po danima unosi koliko poena je ostalo timu na kraju svakog radnog dana, tzv. spaljivanje poena (*burndown*).



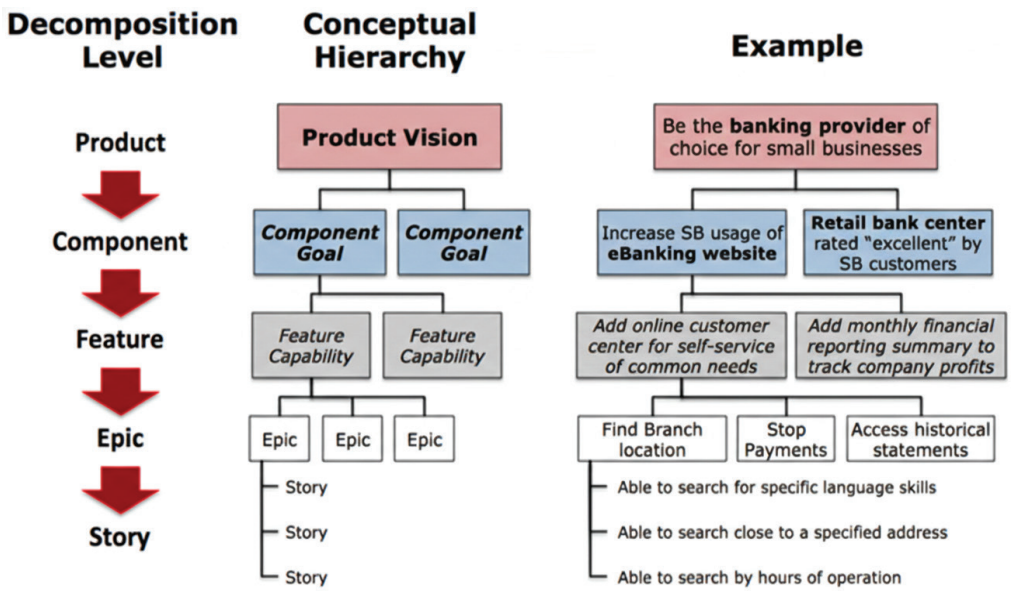
Slika 3.25. Postupak izvlačenja zadataka iz korisničke priče

Primer zadataka i spaljivanja težinskih poena:

Tasks	Mon	Tues	Wed	Thur	Fri
Code the user interface	8	4	8		
Code the middle tier	16	12	10	7	
Test the middle tier	8	16	16	11	8
Write online help	12				



U zavisnosti od kompleksnosti zadataka, hijerarhija definisanja zadataka se može razložiti, na konkrente funkcionalnosti (*features*), opšte zadatke (*epic*) i detaljne zadatke (*story*). Ovakav pristup je podržan u alatu JIRA.



Slika 3.26. Nivo dekompozicije sprint backlog-a (*Feature, Epic* i *Story*)

Vežba 3.1. Kreirati *Product backlog* i *Sprint backlog* za Amazon *e-commerce* aplikaciju.

Korisničke priče prema funkcionalnostima aplikacije bi bile:

1. Registracija i autentifikacija korisnika

- Kao novi korisnik, želim da se registrujem sa svojim Gmail nalogom kako bih mogao/la da počnem sa kupovinom.
- Kao registrovani korisnik, želim bezbedno da se prijavim koristeći svoje podatke kako bih pristupio/la svom nalogu.

2. Pretraživanje proizvoda

- Kao kupac, želim da pretražujem proizvode po kategorijama (npr. elektronika, odeća) kako bih lako pronašao/la ono što mi je potrebno.
- Kao kupac, želim da vidim detalje proizvoda (cena, opis, recenzije) kako bih donosio/la informisane odluke o kupovini.

3. Pretraga i filteri

- Kao kupac, želim da pretražujem proizvode koristeći ključne reči kako bih brzo pronašao/la specifične artikle.
- Kao kupac, želim da filtriram rezultate pretrage po ceni, brendu, ocenama, itd., kako bih suzio/la izbor.

4. Dodavanje u korpu i listu želja

- Kao kupac, želim da dodam proizvode u korpu i da ih vidim u realnom vremenu.
- Kao kupac, želim da sačuvam proizvode na listi želja za kasniju kupovinu.

5. Upravljanje korpom

- Kao kupac, želim da menjam količinu proizvoda u korpi i da ažuriram ukupnu cenu.
- Kao kupac, želim da uklonim proizvode iz korpe ako promenim mišljenje.

6. Proces naplate

- Kao kupac, želim da prolazim kroz glatki proces naplate sa opcijama za dostavu i plaćanje.
- Kao kupac, želim da dobijem potvrdu mog/narudžbine putem emaila i na sajtu.

7. Praćenje porudžbine i Istorija

- Kao kupac, želim da pratim status svojih narudžbina (obrada, isporuka, dostava).
- Kao kupac, želim da vidim istoriju svojih narudžbina i detalje prethodnih kupovina.

8. Upravljanje nalogom

- Kao registrovani korisnik, želim da ažuriram svoje lične informacije (adresu, načine plaćanja) po potrebi.

9. Korisnička podrška

- Kao kupac, želim pristup korisničkoj podršci (*chat* uživo, često postavljana pitanja) kako bih dobio/la pomoć oko svojih narudžbina.
- Kao kupac, želim da lako kontaktiram službu za korisnike za povrate, refundacije ili bilo koje probleme sa mojim kupovinama.

10. Promocije i popusti

- Kao kupac, želim da primenim kodove za popust ili promotivne ponude tokom naplate.
- Kao kupac, želim da vidim personalizovane preporuke ili specijalne ponude na osnovu moje istorije pregleda i kupovina.

11. Integracija mobilne aplikacije (ako je primenjivo)

- Kao kupac, želim da koristim mobilnu aplikaciju kako bih pregledao/la i kupovao/a na praktičan način putem svog pametnog telefona.
- Kao kupac, želim besprekornu sinhronizaciju između sajta i mobilne aplikacije za svoj nalog i korpu.

Ove korisničke priče pokrivaju širok spektar funkcionalnosti koje bi bile potrebne za e-trgovinsku platformu kao što je Amazon. Svaka korisnička priča predstavlja korisnu funkcionalnu celinu ili mogućnost iz perspektive krajnjeg korisnika, omogućavajući da se tim fokusira na isporuku inkrementalne vrednosti svakim sprintom.

Nakon izdvajanja prioritetnih korsničkih priča, kreirani su zadaci u okviru Sprint Backlog-a:

1. Pretraživanje proizvoda

Zadatak 1: Implementirati backend funkcionalnost za pretraživanje proizvoda po ključnim rečima.

Zadatak 2: Razviti *frontend* komponente za prikaz rezultata pretrage sa filtrima (cena, brend, ocene).

Zadatak 3: Testirati integritet pretraživanja proizvoda i ispraviti eventualne greške.

2. Dodavanje u korpu i listu želja

Zadatak 1: Kreirati korpu za korisnika sa opcijama za dodavanje, uklanjanje i ažuriranje proizvoda.

Zadatak 2: Omogućiti čuvanje proizvoda na listi želja korisnika sa opcijom deljenja.

Zadatak 3: Implementirati backend API-je za upravljanje korpe i liste želja.

3. Proces naplate

Zadatak 1: Razviti korak po korak proces naplate sa opcijama za unos adrese dostave i izbor metoda plaćanja.

Zadatak 2: Implementirati backend API-je za komunikaciju sa spoljnim sistemima naplate (PayPall, Bankarski sistem i sl.)

Zadatak 3: Testirati sigurnost i performanse procesa naplate, optimizovati vreme odziva.

4. Korisnička podrška

Zadatak 1: Implementirati *live chat* podršku sa agentima i automatskim odgovorima na česta pitanja.

Zadatak 2: Omogućiti korisnicima slanje zahteva za podrškom putem emaila ili direktnog kontakta.

Zadatak 3: Poboljšati UI/UX za korisničku podršku radi bolje korisničke interakcije.

5. Integracija mobilne aplikacije (ako je primenjivo)

Zadatak 1: Razviti mobilnu aplikaciju sa osnovnim funkcionalnostima pretrage, dodavanja u korpu i naplate.

Zadatak 2: Sinhronizovati podatke između veb sajta i mobilne aplikacije za korisničke naloge i korpe za kupovinu.

Zadatak 3: Testirati mobilnu aplikaciju na različitim uređajima i platformama radi osiguranja stabilnosti i performansi.

Ovi zadaci su konkretne aktivnosti koje bi se mogle sprovesti tokom sprinta kako bi se implementirale funkcionalnosti koje podržavaju prioritetne korisničke priče. Svaki zadatak bi trebalo da bude dovoljno jasan i definisan da tim može efikasno raditi na njegovom izvršenju tokom sprinta.

Zadatak 3-1. Kreirajte *Product Backlog* i *Sprint Backlog* za Instagram aplikaciju.

3.3.2. Kanban

Ukoliko danas pitate programere kako im izgleda jedan tipičan radni dan, većina će odgovoriti:

- da radi na nekoliko projekata paralelno
- da radi neplanirane zadatke, često paralelno
- da ne zna koji su prioritetni zadaci, jer nisu vizuelizovani
- da ne zna ni šta tačno treba da radi, jer ni korisnik ne zna šta hoće!

Jedno od rešenja na ove probleme je primena Kanban metode.



Slika 3.27. Osnovna Kanban tabla

Kanban metoda je razvijena tokom 1950-ih godina u Toyoti, a jedan od razloga nastanka je tadašnja nedovoljna produktivnost kompanije u odnosu na američke konkurente. Kanban potiče od japanske reči „kan" što znači vizuelno i „ban" što znači kartica ili tabla. Kanban bi se mogao prevesti kao "vizuelna kartica",„tabla za označavanje" ili„oglasna tabla/bilbord". Toyota je prvobitno koristila Kanban kartice kako bi ograničila količinu zaliha vezane za „nedovršenu proizvodnju" na proizvodnom mestu.

Kanban se oslanja na sledeće principe lean proizvodnje:

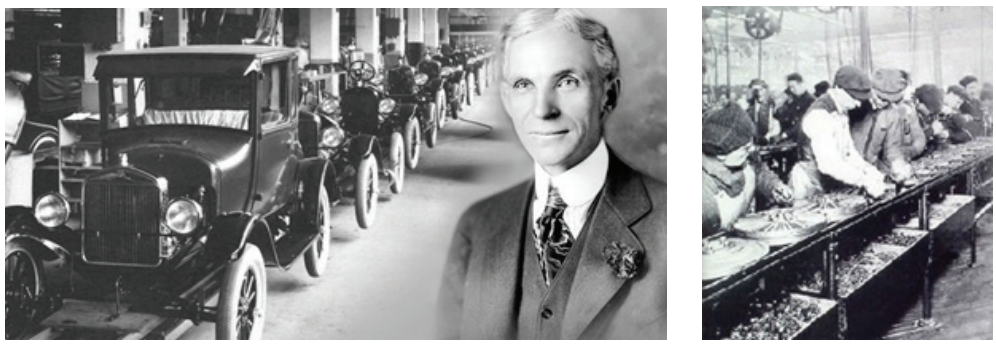
- Sistem vučenja ili povlačenja predmeta proizvodnje (Pull)
- Kontinuirani tok
- Vrednost za potrošača
- Eliminacija gubitaka
- Kontinualno poboljšanje.



Slika 3.28. Sistem guranja (*push*) u odnosu na sistem povlačenja (*pull*) predmeta proizvodnje

Reč „*lean*“ je engleskog porekla i znači mršav-vitak. Ranije se ova reč odnosila na sportiste koji su vitki, žilavi, vrhunski i efektni u sportu kojim se bave. Pored toga, ova reč ne označava samo fizičko stanje, već i posebnu mentalnu disciplinu (psihičku snagu). Primenjeno na poslovne organizacije, *lean* podrazumeva da je organizacija maksimalno efektna u onome što radi, da troši minimum potrebnih resursa, a pruža najbolji mogući kvalitet proizvoda ili usluga. *Lean* ne znači samo postizanje ovakvog stanja, već i posvećenost nizu principa i praktičnih vežbi koji ovo željeno stanje kontinuirano održavaju.

Lean koncept ima svoje korene u proizvodnji, a posebno u automobilskom sektoru. Henry Ford je uneo revoluciju u proizvodnji automobila pokrenuvši proizvodnju baziranu na principu kontinuirane pokretne trake. Njegov koncept masovne proizvodnje je bio osnovni industrijski koncept toga vremena.



Slika 3.29. Henry Ford i pokretna traka u Ford kompaniji

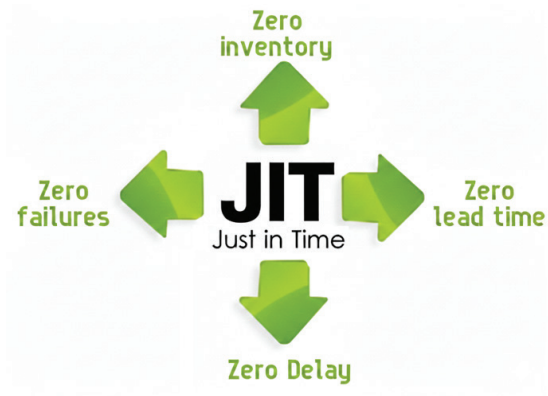
Nastanak i osnove LEAN principa

Toyotini menadžeri su 1950-ih godine posetili Fordove fabrike u SAD-u. Prilikom te posete posmatrali su kako rade i njihovi supermarketi. Primetili su da oni nadomešćuju samo one proizvode koji se prodaju, te stoga samo skladište onu robu za koju postoji realna potražnja. Nakon te posete kreiran je Toyota proizvodni sistem (*Toyota Production System, TPS*) koji se zasniva na LEAN principima. Ideja LEAN principa je eliminacija ili smanjenje svih vrsta gubitaka u proizvodnji.



Slika 3.30. Sedam vrsta gubitaka u proizvodnji

Sa Kanban metodom postignuta je pravovremena (*Just in time*) proizvodnja, odnosno proizvodnja bez zaliha. Kanban se često i naziva „*Just in time production*" metoda za upravljanje i kontrolu proizvodnog procesa. Ova filozofija je najzaslužnija za poslovni uzlet japanske industrije. Krajem 70-ih godina Kanban se proširio i na tržište SAD-a.

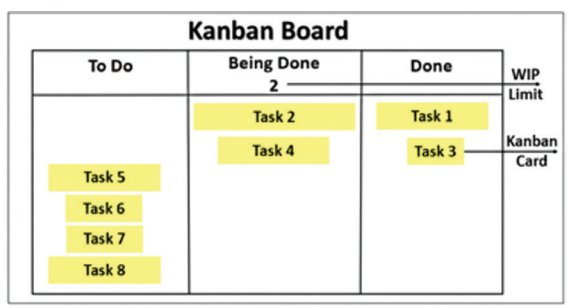


Slika 3.31. Principi pravovremene proizvodnje

Koncept i pravila primene Kanban metode

Ključni koncepti Kanban metode uključuju:

- **Vizuelizaciju toka poslova kroz Kanban tablu** - Celokupan posao se deli na definisane segmente ili stanja koji su vizuelno predstavljeni kao nazivi kolona na tabli. Svaki artikl (predmet proizvodnje) se ispisuje na karticu i postavlja u okviru odgovarajuće kolone, kako bi se naznačilo gde se artikal trenutno nalazi u toku proizvodnje.
- **Ograničenja poslova tzv. WIP** (*Work in Progress*) – označava limit broja artikala koji mogu da se rade u toku jednog segmenta (kolone) ili stanja radnog procesa.
- **Vreme izvođenja** (*Lead Time*) - vreme trajanja ili izvođenja, poznato i kao vreme isporuke (*lead time*) jednog artikla.



Slika 3.32. Osnovni elementi Kanban table

Tvorac TPS sistema, Taiichi Ohno, definisao je šest pravila za efektivnu primenu Kanban metode, nazvanih *Toyota six rules*:

1. Ne šaljite neispravne proizvode u naredni proces
2. Sledeći proces povlači za sobom samo ono što je neophodno
3. Proizvesti samo onu količinu koja se povlači narednim procesom
4. Izravnati/nivelisati proizvodnju
5. Kanban je sistem za fino podešavanje
6. Stabilizovati i racionalizovati proces

Primena Kanban metode u razvoju softvera

Gubitak ili trošak u razvoju softvera predstavlja:

- Delimično urađen posao
- Konstantno prelaženje sa jednog na drugi zadatak
- Dodatni korisnički zahtevi
- Neiskorišćavanje talenata i veština ljudi
- Ponovno učenje
- Defekti
- Odlaganja i sl.

Ukoliko bi se Toyotinih šest pravila primenilo na proces razvoja softvera, pravila bi glasila:

- Nemojte razvijati funkcionalnosti aplikacije koje nikome trenutno ne trebaju!
- Nemojte pisati više specifikacija i modela nego što možete da iskodirate!
- Nemojte pisati više koda nego što možete da testirate!
- Nemojte testirati mnogo koda nego što ih možete uvesti u realno okruženje!

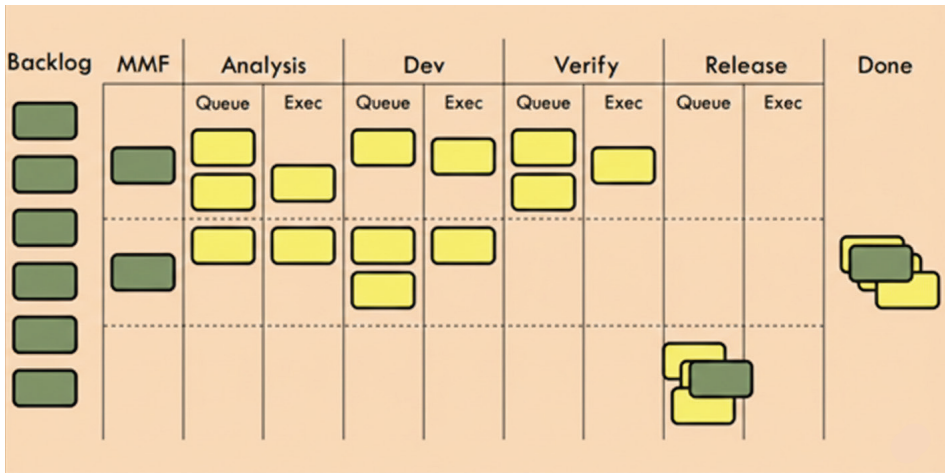
Don't build features that nobody needs right now
Don't write more specs than you can code
Don't write more code than you can test
Don't test more code than you can deploy

- Corey Ladas

Slika 3.33. Kanban pravila na proces razvoj softvera

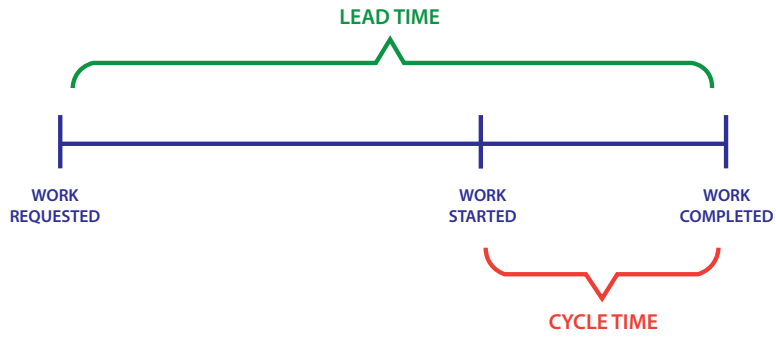
Principi Kanbana u razvoju softvera podrazumevaju:

- **Povlačenje** (*pull*) **zadataka** (*tasks*) **kroz Value Stream** (tok vrednosti). *Value stream* može biti podeljen, na primer, na sledeće faze: *Request -> Analysis -> Development -> Verify -> Release*. Tim bira kako će formirati faze toka vrednosti.
- **Ograničenje rada u toku** (*work in progres, WIP*). Potrebno je početi od najkritičnije faze, na primer, ukoliko je to *Development* onda pomnožiti broj developera u timu sa 1.5. Ako vaš development tim čine dva developera, onda u toj fazi možete imati maksimalno 3 zadatka koja se razvijaju.
- **Vizuelizaciju toka vrednosti.** Potrebno je sve zadatke učiniti vidljivim čitavom timu. Usled toga se koristi Kanban tabla na kojoj mogu da se lepe stikeri ili elektronski Kanban *board* podržan raznim alatima, kao što su Jira, Trello i slični.



Slika 3.34. Primer Kanban table sa vizuelizovanim zadacima tokom faza toka vrednosti

- **Procenjivanje vremena izvođenja** (*Lead time*). *Lead time* je vreme koje prođe od trenutka kada se zahteva novi zadatak do njegovog završetka, odnosno, vreme kad se zadatak postavi na Kanban tablu do njegovog odlaska, bez obzira da li se trenutno radi na tom zadatku ili ne. Pored ovog termina koristi se i termin vreme ciklusa (*Cycle time*) koji podrazumeva realno vreme koje se provede u radu na konkretnom zadatku.



Slika 3.35. *Lead time* (vreme izvođenja) u odnosu na *cycle time* (vreme ciklusa obrade)

Na Kanban tabli, kao i kod Scrum okvira, prvo se postavljaju zadaci koji imaju minimalnu tržišnu vrednost (*Minimal Marketable Feature, MMF*) za klijenta. Ta vrednost se odnosi bilo na stvaranje prihoda, konkurentskoj diferencijaciji, uštedi troškova ili poboljšanoj lojalnosti kupaca. Dakle, minimalna tržišna funkcionalnost (*Minimal Marketable Feature*) je segment funkcionalnosti koji ispunjava deo korisničkog zahteva, a koji kada se objavi kao nezavisna celina, donosi konkretnu vrednost korisniku.

Na primer, uočeno je da u sistemu postoje dupli korisnički zapisi, odnosno za jednog istog klijenta postoji više različitih zapisa u sistemu, sa različitim vrednostima pojedinih atributa. Na primer, klijent je promenio prezime i zaposleni ga ponovo upisuje u sistem kao novog klijenta sa novim prezimenom ili novom adresom i slično. MMF funkcionalnost bi glasila da sistem treba da omogući korisnicima da rešavaju duple korisničke zapise. Na osnovu ovog MMF-a kreira se korisnička prča (*user story*) koja glasi: Kao korisnik, želim da se spoje dupli zapisi u jedan, kako bi se eliminisali dupli procesi u sistemu. Na osnovu ove korisničke priče, kreiraju se konkretni zadaci (tzv. taskovi) i to:

- Proveriti da li korisnik ima pravo da spaja „*merge*" zapise u sistemu.
- Tokom spajanja, verifikovati da li bar jedan zapis ima validne demografske informacije.
- Nakon uspešnog spajanja, relaciju spajanja upisati u log merge tabele.

Početni koraci kreiranja Kanban table u razvoju softvera su:

1. **Definišite tok procesa rada** – Prema tipu aplikacije koja se razvija, zahtevima korisnika, bzrini isporuke, timova za razvoj i drugih činilaca, tim definiše faze toka vrednosti, odnosno kako će imenovati faze kroz koje će pratiti tok izvršavanja zadataka.
2. **Postavite vizuelnu Kanban tablu** – Bilo da je u pitanju obična ili pametna tabla, neophodno ju je kreirati.
3. **Odlučite o maksimalnom broju stavki** (*Work In Progress,* WIP) koji mogu da se istovremeno izvršavaju po pojedinačnim fazama, odnosno kolonama na Kanban tabli. Dobra granica je kada se broj ljudi koji mogu da rade na jednoj stavci u datom koraku procesa pomnoži sa 1.5.

4. **Postavite prioritetne ciljeve u levoj koloni table**. Dobar cilj opisuje ishod za koji se nadamo da će biti postignut nakon isporuke softvera. Ciljevi pomažu u fokusu na veći ishod.
5. **Pokrenite Kanban tablu** tako što ćete postaviti konkretne zadatke tj. kartice na tablu. Svakoj kartici označite datum kada je ušla na red, kako bi merili *lead time*. Koristite datume na karticama da biste izračunali i vreme ciklusa. Prosečno vreme ciklusa koristite za podešavanje vremena čekanja iz različitih tačaka na tabli. Obratite pažnju na protok i uska grla.

Primer Kanban table za aplikaciju mobilnog bankarstva, gde su u timu dva delevopera, izgledao bi kao na sledećoj tabeli.

Tabela 3.3. Primer Kanban table za aplikaciju mobilnog bankarstva

Backlog	Razvoj u toku WIP = 3	Testiranje WIP = 2	Završeno
• Implementacija procesa transfera i menjačnice	• Razvoj funkcije IPS skeniraj i prikaži	• Razvoj funkcije prenosa sredstava	• Implementacija procesa autentikacije korisnika
• Dodati prečice na login strani: IPS scan, Prenos, Kontakti i Lokacije	• Kreiranje šablona za plaćanje	• Razvoj funkcionalnosti plaćanja računa "Novo plaćanje"	• Prikaz prve strane: stanja na računu i poslednjih pet transakcija
• Dodati pregled personalizovanog kataloga proizvoda banke	• Razvoj opcije za devizno plaćanje		• Pregled stanja po svim računima klijenta
• Razvoj funkcije obaveštenja o transakcijama			• Razvoj korisnčkog profila

Kanban vs Scrum

Scrum i Kanban dele neke principe i prakse, ali su dva različita pristupa, što se može videti na sledećoj tabeli.

Tabela 3.4. Poređenje Scrum i Kanban metode

	Scrum	**Kanban**
Ritam	Fiksirane dužine iteracija (npr. 2 nedelje)	Neprekidan tok
Princip isporuke	Na kraju sprinta	Neprekidna isporuka
Uloge	Scrum Master. Produst Owner, razvojni tim	Nema novih uloga. Tim i proces poštuju nove uloge koje su zatečene u datom trenutku
Ključne metrike	Brzina (eng. velocity)	Dužina ciklusa (eng. Cycle time)
Filozofija promene	Tokom sprinta tim ne prihvata promenu.	Promena se može desiti u bilo kom trenutku.

Neki timovi pokušavaju da kombinuju prakse Scrum-a i Kanbana stvarajući "Scrum-ban". Oni uzimaju uloge i sprintove iz Scrum-a, a fokusiraju su na limitiran broj zadatka u procesu rada i dužinu ciklusa iz Kanbana.

3.3.3. Dizajnersko razmišljanje

Dizajnersko razmišljanje (*Design Thinking*) je pristup koji se utemeljio još 1940-ih godina u oblasti kreativnog inženjerstva koji se zasniva na rešavanju problema u dizajnu i unapređenju inventivnosti. Sa sve većom primenom različitih domena veštačke inteligencije i personalizacijom usluga i proizvoda, *design thinking* se proširio i na ostale grane, kao što su softversko inženjerstvo, marketing, ljudski resursi i mnoge druge oblasti.



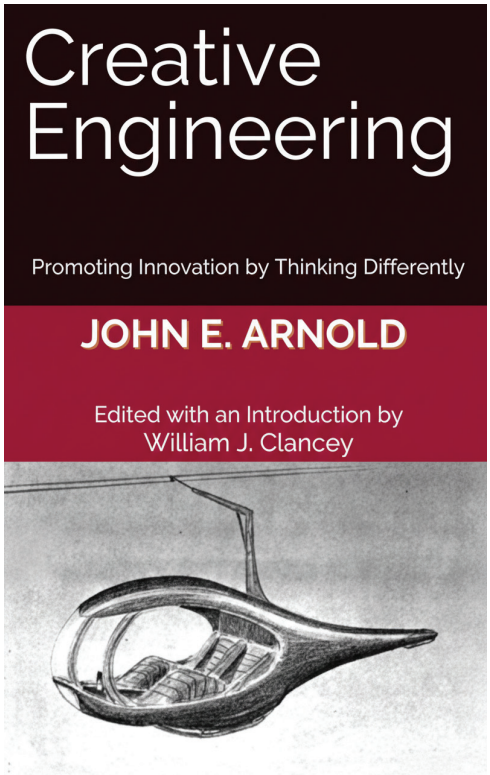
Slika 3.36. Potreba za primenom desing thinking okvira u softverskom inženjerstvu

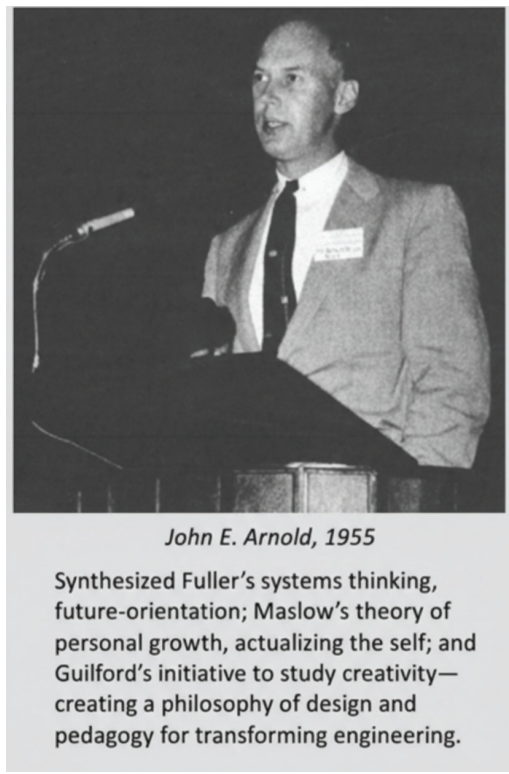
Design thinking je okvir (*framework*) koji nam pomaže da, pre nego što se upustimo u razvoj aplikacije, duboko razumemo korisnika kroz proces empatije, izazovemo sopstvene pretpostavke (naša ubeđenja šta je korisniku potrebno), redefinišemo probleme, identifikujemo alternativna strateška rešenja, koja čak možda i neće biti u skladu sa našim početnim nivoom razumevanja problema, kreiramo i testiramo prototipe rešenja i na kraju definišemo najkreativnije rešenje za konkretne potrebe korisnika.



Slika 3.37. Koraci design thinking okvira

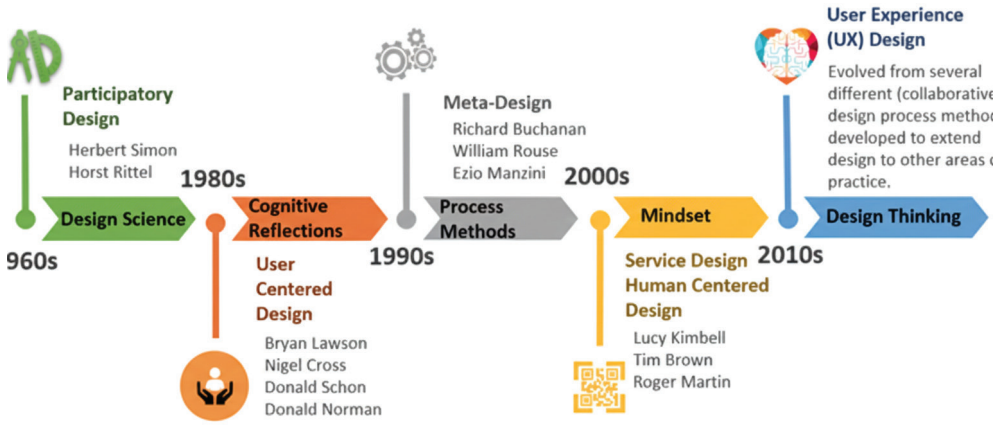
Utemeljivač *Design Thinking* metode je John E. Arnold (1913-1963), koji je diplomirao psihologiju, 1934. godine, na Univerzitetu u Minesoti i magistrirao u oblasti mašinskog inženjerstva (1940) na MIT-u. Kasnije je radio kao profesor mašinstva i poslovne administracije na Stanford univerzitetu. Arnold je pionir u naučnom definisanju i unapređivanju inventivnosti, zasnovan na psihologiji kreativnog mišljenja i mašte. Bio je osnivač odeljenja za dizajn za mašinstvo, nastavljajući da formuliše i podučava o kreativnosti u inženjerstvu.





Slika 3.38. Korice knjige John Arnolda "Kreativno inženjerstvo"

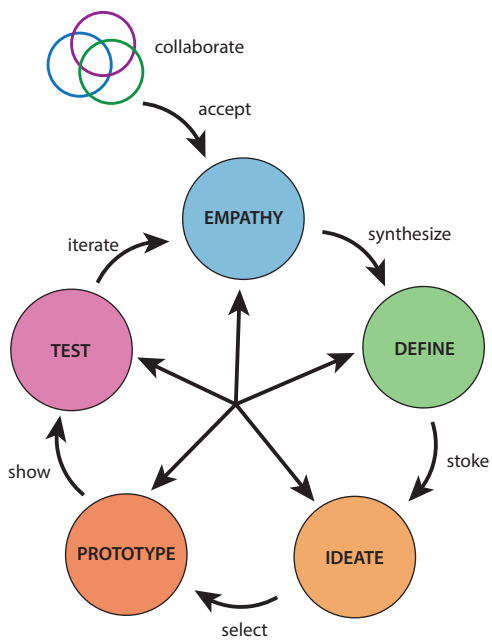
Design Thinking nema tačno određeno vreme nastanka, već se razvio iz različitih disciplina tokom 20. veka, uključujući inženjering, psihologiju, umetnost i menadžment. Njegova formalizacija kao pristupa rešavanju problema vezuje se za rad dizajnera, akademika i kompanija tokom 1960-ih i kasnije. Ključne oblasti i njihovi tvorci su prikazani na slici 3.27.



Slika 3.39. Istorijski osvrt razvoja *design thinking*-a

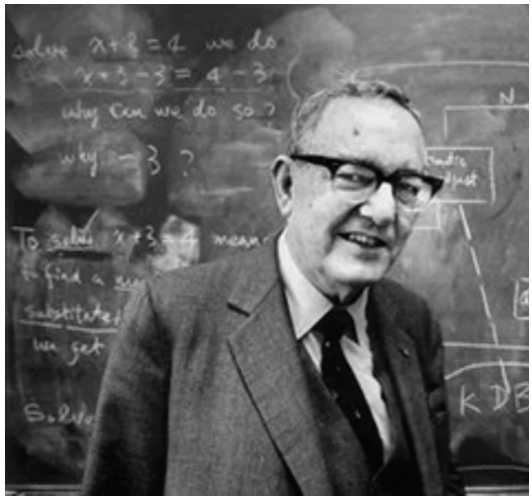
Koncept i faze design thinking okvira

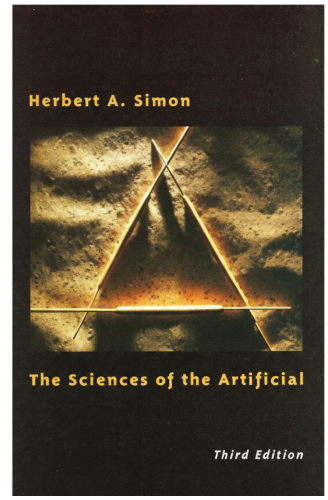
Da bi ste razumeli koncept *design thinking* okvira, pokušajte prvo da definišete šta je za vas dizajn. Neko će prvo pomisliti na estetski izgled nekog proizvoda, dok će drugi pomisliti na inovativnu funkcionalnost proizvoda ili na iskustvo koje neko dizajnersko rešenje proizvodi kod korisnika. Dizajn je pre svega proces!



Slika 3.40. Dizajn kao proces

Postoji nekoliko varijanti *Design Thinking* (DT) procesa koje se danas koriste i ono mogu varirati od tri do sedam faza. Bilo koja DT varijanta sadrži iste principe koje je prvobitno opisao dobitnik Nobelove nagrade Herbert Simon u časopisu "*The Science of the Artificial*", 1969. god. Ovde ćemo se fokusirati na pet faza DT modela predloženih od strane Hasso-Plattner Instituta za dizajn na Stanfordu, a koji je takođe poznat i kao *d.school*.

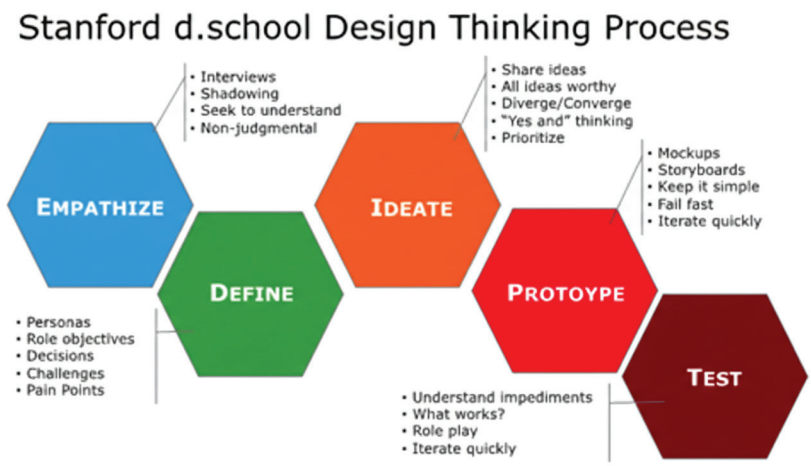




Slika 3.41. Nobelovac Herbert Simon i korice časopisa "*The Science of the Artificial*" izdatog 1969. godine

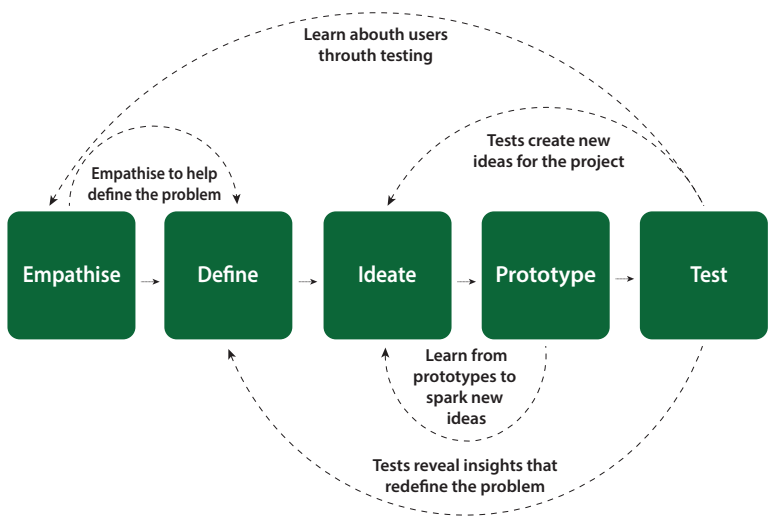
Pet faza DT prema *d.school* su:

1. **Empatija** ili saosećanje (*Empathise*) – cilj ove faze je da se duboko razumeju realne potrebe ili problemi korisnika.
2. **Definisanje** (*Define*) – u ovoj fazi se konkretizuju odnosno jasno definišu konkretne potrebe korisnika, problemi i naši uvidi u njihove probleme.
3. **Generisanje ideja** (*Ideate*) – na osnovu prethodno definisanih jasnih potreba u ovoj fazi se generišu inovativne ideje, razmenjuju se sa timom, prevazilaze se sopstvene tvrdnje i pretpostavke i odabiraju se one za koje će se kreirati prototipi u narednom koraku.
4. **Kreiranje prototipa** (*Prototype*) – koristeći neki od *mockup* alata kreira se prototip za prethodno odlučeno inovativno rešenje.
5. **Testirajte** (*Test*) – u ovom koraku se demonstrira prototip korisniku i kreiraju mape empatije, odnosno zapisuje se koje komentare je korisnik dao tokom interakcije sa prototipom rešenja, ali i one reakcije koje neverbalno pokazao.



Slika 3.42. Faze *Design Thinking* okvira prema *d.school*

DT je iterativni, nelinearni proces, tj. faze se ne moraju sekvencionalno odvijajati, niti treba da prate određeni redosled, već se često mogu odvijati paralelno ili ponavljati kroz iteracije. Dizajn tim kontinualno koristi svoje rezultate radi pregleda, preispitivanja i poboljšanja svojih početnih pretpostavki, shvatanja i rezultata. Rezultati finalne faze pomažu da se odrede parametri problema, omogućava redefinisanje problema i ono što je najvažnije, pruža nam nove uvide kako bismo videli bilo kakvu alternativu rešenja koja možda nije bila vidljiva sa našim prethodnim nivoom razumevanja problema.

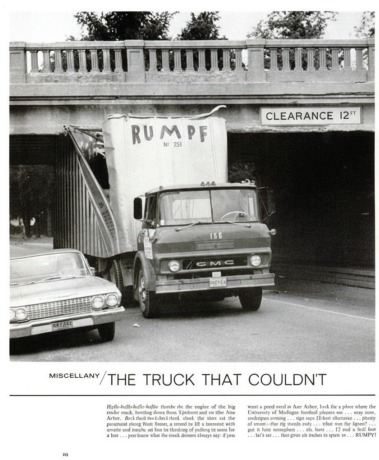


Slika 3.43. Iterativni i nelinearni DT proces

Dizajn ili „*Outside the Box*“ razmišljanje

Ljudi prirodno razvijaju obrasce razmišljanja zasnovanih na aktivnostima koje se ponavljaju i opšte dostupnog znanja. Ovi obrasci nam pomažu da u sličnim situacijama brzo primenimo iste akcije i znanja, ali nas takođe sprečavaju u kreativnom razmišljanju i da brzo i lako pristupimo ili razvijemo nove načine gledanja, razumevanja i rešavanja problema. Obrasci razmišljanja se često nazivaju šemama, koje predstavljaju organizovane skupove informacija i odnosa između stvari, radnji i misli, podstaknute i inicirane u ljudskom umu kada god se naiđe na podsticaje iz okruženja. Kako se ove šeme automatski podstiču, to može ometati u proceni realne situacije ili mogu da spreče u sagledavanju problema kako bi se kreirala nova strategija za rešavanje problema. Inovativno rešavanje problema je takođe poznato kao „razmišljanje izvan kutije“ (*thinking outside of the box*).

U cilju razumevanja *out of box* razmišljanja, navešćemo jednu studiju slučaja iz realnog života. Vozač kamiona, u pokušaju da prođe ispod nižeg mosta, biva zaglavljen, što je prouzrokovalo velike probleme u saobraćaju. U pomoć su pristigli inženjeri, vatrogasci, radnici za vanredne situacije i vozači kamiona kako bi osmislili i pregovarali o raznim rešenjima. Radnici službe za vanredne situacije raspravljali su o tome da li treba demontirati delove kamiona ili da odseku deo mosta. Svako je govorio o rešenju koje je bilo u domenu njihove stručnosti. Eksperti se posebno oslanjaju na svoje čvrste misaone obrasce i njima je posebno teško da preispitaju svoje znanje. Šta bi vi predložili u ovakvoj situaciji?



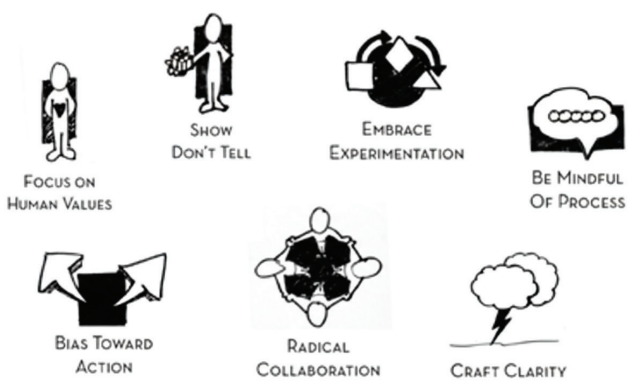
Slika 3.44. Studija slučaja razmišljanja “*out of box*”

Dečak koji je prolazio i bio svedok intenzivne rasprave pogledao je kamion, most, a zatim put i nonšalantno izgovorio: „Zašto jednostavno ne ispustite vazduh iz guma?". Ovo je izazvalo apsolutno čuđenje svih specijalista i stručnjaka koji su pokušavali da otklone problem. Ovo dete nam pokazuje zašto je važno osporiti naše pretpostavke i pronaći nove načine za rešavanje naših problema.

Dizajneri se opiru iskušenju da odmah uskoče u rešenje navedenog problema. Umesto toga, oni prvo treba da utvrde šta je osnovni izvor problema, a zatim da ponude više potencijalnih rešenja i tek tada će se približiti krajnjem rešenju.

Osnovni cilj DT-a je poboljšanje proizvoda kontinuiranom analizom i razumevanjem načina kako korisnici komuniciraju sa proizvodima i istraživanjem uslova u kojima koriste proizvod.

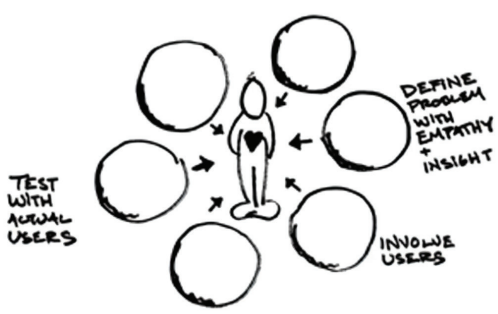
DT je u suštini skup načina razmišljanja (*mindsets*) u rešavanju problema (Slika 3.45).



Slika 3.45. Načini dizajnerskog razmišljanja

Primena DT podrazumeva usvajanje sledećih glavnih principa načina razmišljanja:

Fokus na ljudske vrednosti (*Focus on human values, Human centered*) podrazumeva definsanje problema sa empatijom i dubljim uvidom u potrebe korisnika, kontinuirano posmatranje korisnika radi inspiracije i novih ideja, uključivanje korisnika u fazi generisanja inovativnih ideja, testiranje idejnih rešenja sa aktuelnim korisnicima i dr.



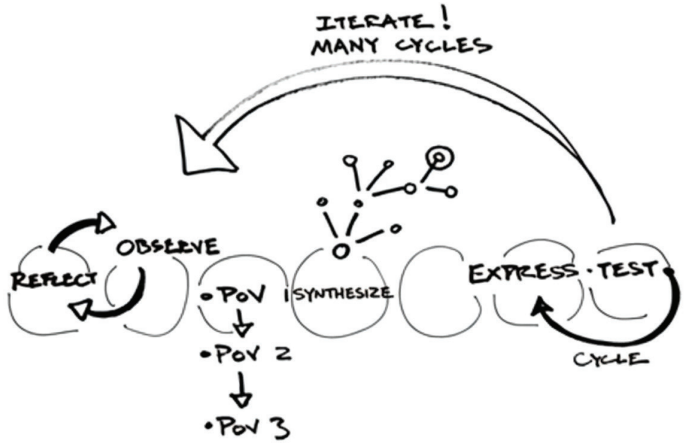
Slika 3.46. Razmišljanje centrirano oko korisnika

Radikalna saradnja (*Radical collaboration*) podrazumeva timsko posmatranje i analiza problema ili rešenja, uključivanje autsajedra u brejnstorming ideja i testiranje sa korisnicima. Ovakva saradnja osigurava da proizvod bude poželjan za korisnika (dizajn tim brine o ovome), održiv i i da ima vrednost za poslovanje (biznis tim brine o ovome) i tehnički izvodljiv (posao razvojnog tima).



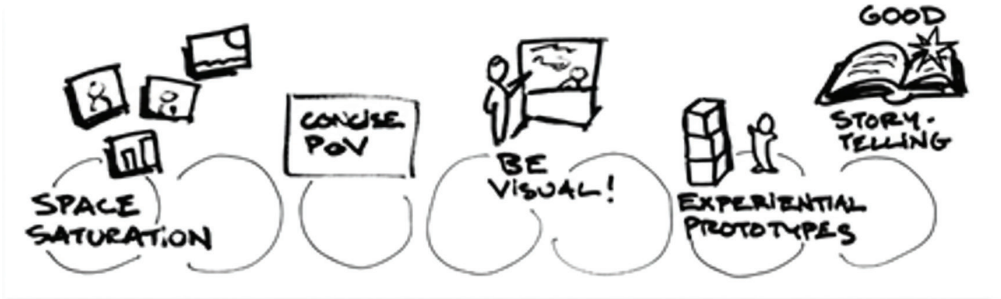
Slika 3.47. Radikalna saradnja

- **Kultura prototipovanja** (*Culture of prototyping, Embrace the experimentation*) podrazumeva vizuelizaciju ideja i ubrzano testiranje predloga rešenja sa korisnicima.



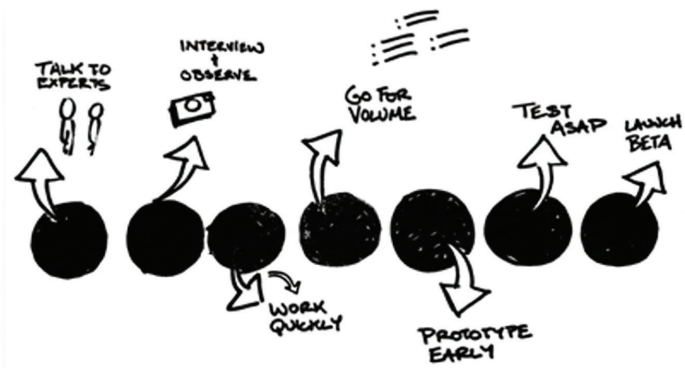
Slika 3.48. Kreiranje prototipa rešenja

- **Pokaži, ne pričaj** (*Show don't tell*) podrazumeva fokus na demonstraciju prototipa rešenja, pre nego da se vreme troši na duge razgovore.



Slika 3.49. Način razmišljanja "*Show don't tell.*"

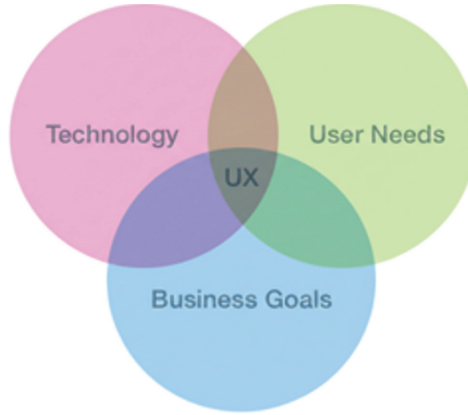
- **Pristrasnost prema akciji** (*Bias toward action*) podrazumeva razgovore sa stručnjacima, posmatranje i brzo delovanje, rano prototipovanje, brzo testiranje i lansiranje beta verzije.



Slika 3.50. Pristrasnost prema akciji

DT kao osnova za dizajniranje korisničkog iskustva

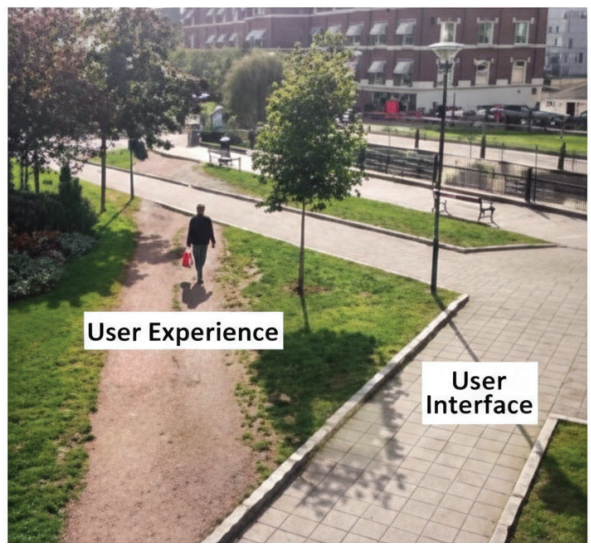
Design thinking je osnovna metodologija za dizajniranje korisničkog iskustva (*User Experience*, UX). Termin UX je prvi put upotrebljen 1990. godine od strane dr Donalda Normana, kognitivnog inženjera. UX dizajn je interdisciplinarna oblast koja uključuje psihologiju, dizajn i tehnologije u cilju konstantnog usklađivanja potreba korisnika i poslovnih ciljeva.



Slika 3.51. Položaj UX dizajna

Dizajniranje korisničkog iskustva (*User Experience Design* ili *UX design*) je proces kreiranja proizvoda, usluga ili sistema koji pružaju korisnicima optimalno iskustvo tokom njihove interakcije s tim proizvodima ili uslugama. Cilj je učiniti to iskustvo intuitivnim, efikasnim, prijatnim, relevantnim i što jednostavnijim. UX dizajneri koriste DT proces kako bi pomirili želje korisnika sa tehničkom izvodljivošću i poslovnom održivošću.

Dizajn korisničkog interfejsa (*User Interface*, UI) nije isto što i UX dizajn. Dizajn korisničkog interfejsa se bavi svim vizuelnim i interaktivnim elementima interfejsa proizvoda, pokrivajući sve, od tipografije i paleta boja do animacija i navigacije. Svakako da UI dizajn ima ogroman uticaj na celokupno korisničko iskustvo.



Slika 3.52. Poređenje UX sa UI dizajnom u realnosti

Ključni elementi dizajniranja korisničkog iskustva:

- **Istraživanje korisnika** - Razumevanje potreba, očekivanja, problema i ponašanja ciljne publike ili ključnih persona putem intervjua, posmatranja i analize podataka.
- **Organizacija i struktura informacija** kako bi korisnici lako pronašli ono što traže.
- Dizajn korisničkog interfejsa - Vizuelni i interaktivni elementi s kojima korisnici direktno komuniciraju, kao što su dugmad, meniji, boje, tipografija i raspored elemenata.
- **Prototipiranje i testiranje** - Stvaranje prototipova (skica, žičanih okvira ili interaktivnih modela) kako bi se testiralo kako korisnici reaguju na dizajn i pronašli potencijalni problemi pre finalne implementacije.
- **Upotrebljivost** - Ocenjivanje kako lako korisnici mogu da postignu svoje ciljeve koristeći proizvod ili uslugu.
- **Interakcijski dizajn** (IxD) - Planiranje načina na koji korisnici stupaju u interakciju sa sistemom, uključujući animacije, prelaze i povratne informacije koje dobijaju tokom korišćenja.
- **Emocionalni dizajn** - Kreiranje iskustava koja izazivaju pozitivne emocije i povezanost korisnika s proizvodom.

Dobro osmišljen UX dizajn može povećati zadovoljstvo korisnika, smanjiti frustracije i greške tokom upotrebe, povećati efikasnost i produktivnost, povećati lojalnost i angažovanje korisnika i poboljšati poslovne rezultate.

POGLAVLJE 4



MODELOVANJE PROCESA

Modelovanje procesa

4.1. Uvod u modelovanje procesa
4.2. Modelovanje poslovnih procesa: pojam i značaj
4.3. Tehnike modelovanja procesa

Cilj poglavlja:

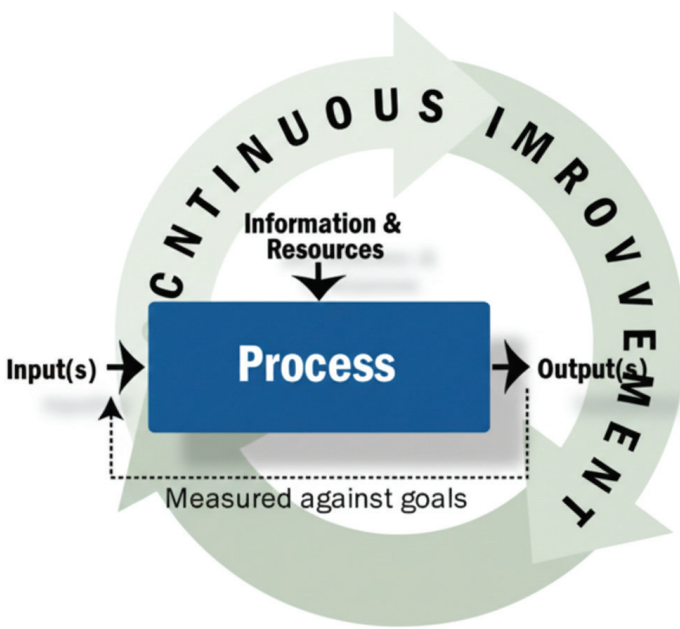
Razumevanje i organizacija poslovnih procesa predstavljaju osnovu za postizanje organizacione efikasnosti i efektivnosti. Ključni izazov softverskog inženjerstva je uspostavljanje „zajedničkog jezika" između poslovnih korisnika i inženjera, što se postiže kroz precizno modelovanje AS-IS (postojećih) i TO-BE (budućih/unapređenih) procesa. Cilj ovog poglavlja je da osposobi čitaoca za identifikaciju, analizu i formalizaciju poslovnih aktivnosti korišćenjem savremenih tehnika i notacija, čime se obezbeđuje da buduće softversko rešenje bude u potpunosti usklađeno sa realnim poslovnim potrebama i ciljevima sistema.

4. MODELOVANJE PROCESA

4.1. UVOD U MODELOVANJE PROCESA

Ukoliko treba da razvijete aplikaciju za finansijsko odelenje neke kompanije, odakle ćete početi? Pretpostavljamo da vam nisu poznati termini kao što su konta, bilansi, procesi knjiženja i slično, pa stoga ne znate ni kako da iskodirate sve te finansijske procese. Pre nego što se i upustite u proces razvoja aplikacije, neophodno je razumeti kako zaposleni u finansijskom odelenju obavljaju posao, kako se odvija tok poslova (kako rezultat rada jednog zaposlenog se prenosi kao ulazni zadatak drugom zaposlenom), sa kojim problemima ili izazovima se suočavaju, šta ih frustira, ali i kakve navike imaju (npr. obično koriste velika slova za pretragu podataka) i drugo. Postoje razne tehnike za upoznavanje načina rada, snimanje postojećeg stanja, prikupljanja informacija i analize zahteva korisnika. Međutim, pre nego što se upustimo u tehnike prikupljanja informacija i razumevanja i modelovanja poslovnih procesa, neophodno je razumeti šta je to proces.

Sama reč proces, označava da se tu nešto obrađuje (engl. *to process*). U svaki proces ulazi podatak ili podaci (tzv. *Input*), koje proces obrađuje i kao rezultat obrade dobija se izlaz (tzv. *Output*). Taj izlaz može predstavljati ulaz u neki drugi proces i na taj način se sagledava tok poslova (*workflow*) radi ispunjenja određenog cilja (Slika 4.1).



Slika 4.1. Pojam procesa

Proces može biti prost, ukoliko se obavlja samo jedan zadatak (jedna aktivnost), na primer kreiranje profakture, a može biti i složen, na primer proces izdavanja kredita klijentu. Složen proces može unutar sebe da sadrži više manje složenijih procesa i više prostih procesa ili tzv. zadataka.

Kako ćete identifikovati procese, dok razgovarate sa zaposlenima? Ukoliko ulazite u finansijsko odelenje, njega možete sagledati, a i nacrtati kao jedan veliki složeni proces. Ukoliko u finansijskom odelenju zatičete nekoliko kancelarija sa posebnim natpisima na vratima, kao što su Računovodstvo (*Accounting*), Kontroling (*Controlling*), Blagajna i upravljanje gotovinom, Interna revizija i slično, to znači da u okviru tog odelenja postoji više pododelenja. Ta pododelenja su takođe složeni procesi. Zamislite da ulazite u kancelariju Računovodstva i vidite da tamo sede neki zaposleni, koji su možda grupisani po sličnosti posla koje obavljaju, pa ih možete pitati, šta je deo njihovog poslovanja. Na primer, jedni će vam reći da rade knjiženje poslovnih transakcija, drugi da rade finansijske izveštaje (npr. bilanse stanja, uspeha, tokove novca i sl.). Ta grupa zaposlenih je takođe jedan složen proces u okviru procesa Računovodstvo, kao na primer proces "Priprema osnovnih finansijskih izveštaja". Kada bi sada išli od zaposlenog do zaposlenog i pitali ih koji su tačno njihovi zadaci, došli bi do prostih procesa (zadataka ili aktivnosti) i tu bi se dekompozicija crtanja procesa završila.

Na slici 4.2 je prikazan primer organizacione strukture nekog finansijskog odelenja, na čelu sa direktorom finansija ili CFO (*Chief Financial Officer*).



Slika 4.2. Primer organizacione strukture finansijskog odelenja

4.1.1. Poslovni procesi: pojam i vrste

Poslovni proces je skup povezanih aktivnosti ili zadataka koji se obavljaju u određenom redosledu s ciljem stvaranja vrednosti za organizaciju, njene korisnike ili druge interesne strane. Poslovni procesi su osnova funkcionisanja svake organizacije i mogu biti formalni (precizno definisani i dokumentovani) ili neformalni (implicitno izvedeni iz svakodnevnih aktivnosti).

Osnovne karakteristike poslovnog procesa su:

- **Cilj** - Svaki proces ima jasno definisan cilj ili ishod, na primer, proces kreiranja fakture ili proces izdavanja kredita i sl.
- **Ulazi i izlazi** - Proces obrađuje ono što dobije preko ulaza (*input*). Ulazi mogu biti informacije, proizvodi, resursi i sl. Rezultat obrade procesa su izlazi (*output*), koji isto tako mogu biti neke informacije, proizvodi, usluge, odluke i sl. Na primer, izlazi mogu biti kreirana faktura ili izdat kredit korisniku.
- **Aktivnosti** - Složeniji procesi se sastoje od niza koraka koji se izvršavaju određenim redosledom.
- **Učesnici** - U procesima učestvuju ljudi, tehnologije, resursi ili sistemi, odnosno konkretne aktivnosti ili čitavi procesi mogu obavljati ljudi ili mogu biti već impelementirani u okviru nekog sistema.
- **Vrednost** – Na osnovu povratnih informacija sa izlaza, dodaju se vrednosti ulazima kako bi proces stvorio korisne izlaze.

Na osnovu opisanih karakteristika, poslovni proces se može definisati kao skup povezanih aktivnosti koje stvaraju određenu vrednost preko transformacije ulaza u izlaze, ispunjavajući pritom jedan poslovni cilj. Aktivnosti unutar procesa, koji transformišu ulaze u izlaze, obavljaju se bilo od strane ljudi ili mašina.

Celokupno poslovanje nekog preduzeća se odvija kroz poslovne procese. Postoje različite vrste poslovnih procesa i to:

- **Osnovni ili operativni procesi** - čine jezgro poslovanja i osnovnu delatnost kompanije. Direktno doprinose stvaranju vrednosti za krajnjeg korisnika. Opšti operativni procesi su: proizvodnja, marketing i dr.
- **Procesi podrške** - podržavaju operativne procese i omogućavaju njihovo funkcionisanje, na primer IT podrška.
- **Upravljački procesi** - obezbeđuju planiranje, praćenje i kontrolu aktivnosti organizacije. Na primer: upravljanje projektima, strateško planiranje.

Dobro definisani poslovni procesi omogućavaju efikasno funkcionisanje organizacije, smanjenje troškova, veću produktivnost i bolje iskustvo korisnika.

4.1.2. Tehnike za prikupljanje informacija

Efikasno prikupljanje informacija predstavlja ključnu fazu u analizi poslovnih procesa i razvoju informacionih sistema. Prema Lau (2000) i Sommerville (2016), šest najčešće korišćenih tehnika uključuju: praćenje, intervjue, fokus grupe, istraživanja, instrukcije korisnika, prototipovanje i instrumentovane verzije softvera.

Praćenje (*Shadowing*) metoda podrazumeva direktno posmatranje zaposlenog u njegovom radnom okruženju, sa ciljem uvida u svakodnevne aktivnosti, tokove rada i potencijalne probleme. Postoje dve varijante i to pasivno praćenje (samo posmatranje) i aktivno praćenje (uz postavljanje pitanja i eventualno uključivanje u zadatke). Praćenje je naročito korisno za rutinske zadatke, ali može biti ograničeno vremenskim faktorom, jer se svi relevantni procesi možda ne odvijaju tokom sesije (zbog sezonalnosti ili periodičnosti, npr. izrada mesečnih izveštaja).

Intervju omogućava prikupljanje kvalitativnih podataka o procesima koji nisu lako posmatrani, poput automatizovanih radnji ili prošlih događaja. Dobro strukturiran intervju treba da koristi jasna i fokusirana pitanja, bez sugestija ili procena, kako bi se obezbedila nepristrasna i relevantna perspektiva sagovornika (Avison & Fitzgerald, 2006). Svrha intervjua je da se istraži, a ne da se ocenjuje i kritikuje. Koristite jasne i koncizne rečenice. Nemojte davati vaša mišljenja u okviru pitanja. Izbegavajte duga i složena pitanja. Trudite se da ne postavljate pitanja koja iziskuju više od jednog tipa informacija.

Fokus grupe uključuju vođene diskusije sa više korisnika, radi identifikacije zajedničkih iskustava i uvida. Ova metoda omogućava razmenu stavova i doprinosi otkrivanju različitih pogleda na poslovne procese. Ključno je obezbediti prijatnu atmosferu za iskrenu komunikaciju.

Istraživanje (*Survey/Questionnaire*). Korišćenjem strukturisanih upitnika moguće je prikupiti statističke i deskriptivne podatke o organizacionim strukturama, tehničkoj podršci, potrebama za obukom, kao i povratnim informacijama korisnika. Istraživanja su posebno korisna kada je potrebno analizirati širok uzorak korisnika.

Instrukcije korisnika (*User Instruction*) uključuje obavljanje zadataka prema uputstvima krajnjeg korisnika, čime se analitičar stavlja u poziciju korisnika. Time se mogu otkriti znanja koja korisnici imaju, ali nisu dokumentovana. Ipak, metoda zavisi od sposobnosti korisnika da jasno prenesu znanje, kao i od njihove raspoloživosti.

Prototipovi omogućavaju simulaciju rada budućeg sistema i korisničku validaciju zahteva. Ova tehnika je posebno pogodna za složene softverske sisteme (npr. u medicini), gde nije moguće neposredno testiranje u realnom okruženju. Iako korisna, izrada prototipa može biti skupa.

Instrumentovane verzije (*Instrumented Versions*) podrazumeva ugradnju mehanizama za praćenje u aplikacije, radi automatskog prikupljanja podataka o korišćenju sistema. Omogućava odgovore na pitanja poput: Koje funkcije se najviše koriste? Koliko traje izvršavanje zadatka? Koji elementi ostaju neiskorišćeni? (Rosson & Carroll, 2002).

4.2. MODELOVANJE POSLOVNIH PROCESA: POJAM I ZNAČAJ

Modelovanje poslovnih procesa je tehnika kojom se analiziraju, organizuju i dokumentuju poslovni procesi u organizaciji, na način da se sagleda i razume kako organizacija funkcioniše, koje politike i procedure primenjuje, identifikuju problemi i optimizuju radni tokovi (*workflow*). S obzirom da procesi organizacije mogu biti vrlo složeni za opisivanje i razumevanje, kod modelovanja procesa koristi se sistemski pristup, odnosno početni dijagrami predstavljaju visok nivo apstrakcije sistema, tj. izvlače se zajedničke karakteristike kako bi sistem bio jasnije prikazan, a zatim se na dijagramima nižeg nivoa, sistem prikazuje sa više detalja. Na primer, ukoliko treba da razvijate aplikaciju za neku tipičnu kompaniju koja se bavi proizvodnjom nekog proizvoda, nećete hteti da se upustite odmah u opis detaljnih procesa te proizvodnje, već ćete želeti da sagledate prvo glavne procese te kompanije. Kako tipična kompanija ima sledeće departmane ili odelenja, kao što su finansije, marketing, proizvodnja, ljudski resursi i sl. onda ćete na početnim dijagramima apstrakcije prikazati samo te glavne procese. Svaki sledeći dijagram predstavlja neku vrstu dekompozicije nekog složenog procesa. Na primer, ukoliko želite da pokažete detaljnije procese sektora za ljudske resurse, onda ćete na dijagramu dekompozicije tog procesa prikazati njemu glavne procese, kao što su na primer, selekcija, regrutacija, zapošljavanje, obučavanje i dr. Zatim ćete na sledećem dijagramu opisati detaljnije procesa samo na primer procesa zapošljavanja itd.

Sistemskim pristupom, modelovanje procesa može bolje da pojasni složene procese, buduće zahteve, postojeće probleme i rizike, informacije koje su manje poznate i samim tim da olakša komunikaciju između razvojnog tima i korisnika.

Dakle, ključne karakteristike modelovanja poslovnih procesa su:

- **Vizualizacija**: Procesi se prikazuju u obliku dijagrama kako bi se bolje razumeli odnosi između koraka, aktivnosti i učesnika.
- **Analiza**: Modelovanje omogućava identifikaciju problema, nedostataka, uskih grla ili širokih mesta u postojećim procesima.
- **Optimizacija**: Pomoću modela, može se izvršiti reinženjering postojećih procesa radi povećanja efikasnosti, smanjenja troškova ili poboljšanja kvaliteta.
- **Dokumentacija**: Modeli služe kao dokumentacija procesa za obuku zaposlenih, usklađivanje sa propisima ili kao osnova za automatizaciju ili standardizaciju.

Modeli procesa prikazuju:

- ko su korisnici (spoljni akteri)
- šta su ulazi i izlazi
- koje se aktivnosti odvijaju unutar procesa
- način odvijanja poslova (*workflow*)
- ko ih obavlja (unutrašnji akteri) i dr.

Koristi od modelovanja poslovnih procesa su:

- jasno, precizno i nedvosmisleno definisani poslovni procesi
- jasno identifikovani poslovi preduzeća
- jasnije razumevanje uloga i odgovornosti u organizaciji
- služi kao podrška u donošenju strateških odluka
- smanjenje operativnih rizika
- formalno zapisano znanje o načinu obavljanja poslova u preduzeću
- mogućnost transfera znanja (obuka radnika za obavljanje poslova, transfer znanja između pojedinih organizacionih entiteta i dr.)
- standardizacija obavljanja poslova
- mogućnost analize poslovnih procesa (otkrivanje nekonzistentnosti i redundansi, poređenje različitih alternativa i dr.)
- mogućnost poboljašanja poslovnih procesa (povećavanje efikasnosti poslovanja, povećanje kvaliteta proizvoda ili usluga i sl.)
- mogućnost automatizacije (formalni modeli se mogu prevesti u izvršne).

Modelovanje koristi različite tehnike i alate za grafičko ili tekstualno predstavljanje aktivnosti, resursa, informacija i tokova rada unutar procesa. Neke od tehnika modellovanja procesa su:

- Dijagram toka podataka (*Data Flow Diagram, DFD*) - jednostavan način za predstavljanje procesa.
- IDEF0 (*Integration Definition*) – funkcionalno modelovanje procesa.
- Jedinstveni jezik modelovanja (*Unified Modeling Language, UML*) - koristi se u softverskom inženjeringu, ali može biti primenjen i na poslovne procese.
- Notacija i modelovanje poslovnih procesa (*Business Process Model and Notation, BPMN*) - standardizovan jezik za modelovanje procesa.

Za postupak modelovanja razvijeni su razni CASE (*Computer Added System Engineering*) alati kao što su IBM Rational Software Architect, BPWin, ARIS i mnogi drugi.

Modelovanje poslovnih procesa kao aktivnost u kojoj se specificiraju poslovni procesi nekog preduzeća na način da se mogu analizirati, poboljšavati i automatizovati, postaje ključni alat za organizacije koje žele da se prilagode promenama, unaprede svoje operacije i ostvare strateške ciljeve.

4.3. TEHNIKE MODELOVANJA PROCESA

Pokušajte da odgovorite na sledeća pitanja:

- U kojoj fazi razvoja softvera ćete imati potrebu za modelovanjem procesa?
- Da li je potrebno uvek modelovati procese?
- Kako započeti modelovanje procesa, odnosno kako identifikovati i opisati procese?
- Od čega zavisi, koju ćemo tehniku modelovanja softvera primeniti?

Ukoliko ste dobili projekat razvoja aplikacije za bankarsko poslovanje, da li ćete se upustiti odmah u razvoj te aplikacije ili ćete se prvobitno upoznati sa trenutnim načinom poslovanja, odnosno kako zaposleni obavljaju svoje svakodnevne aktivnosti? Svakako da bi znali šta treba da razvijete, neophodno je prvo jasno razumeti procese. U početnim fazama razvoja softvera (faza analize trenutnog stanja sistema), ukoliko razvijate informacioni sistem za neku kompaniju sa složenijim procesima poslovanja i ukoliko ta organizacija nema već modelovane procese, neophodno je da odvojite vreme za upoznavanje sa radom te organizacije, a rezultat te faze je model procesa te organizacije. Dešava se da organizacija koja je već implementirala neki od ISO (*International Organization for Standardization*) standarda ima već jasno definisane modele procesa, u tom slučaju nema potrebe da radite dupli posao, već ćete od njih zatražiti te modele radi upoznavanja sa celokupnim poslovanjem. U situacijama kada treba da razvijete neku manju aplikaciju npr. neki kalkulator cena, u tom slučaju vam nije potrebno da modelujete procese, jer aplikacija sama po sebi nije složena.

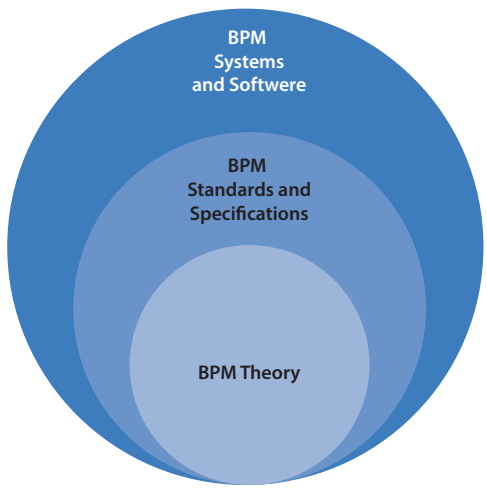
Izbor tehnike modelovanja procesa zavisiće od vrste softvera koji se razvija i izabrane arhitekture. Ukoliko razvijate aplikaciju zasnovanu na servisno orijentisanoj arhitekturi (SOA), tada ćete najverovatnije želeti da primenite BPMN jezik za modelovanje procesa. Ukoliko je organizacija već implementirala neke ISO standarde onda je najverovatnije koristila neke dosadašnje tehnike funkcionalnog modelovanja procesa. U narednom tekstu se ukratko objašnjavaju dve najkorišćenije tehnike funkcionalnog modelovanja procesa i to Dijagram toka podataka (*Data Flow Diagram*) i IDEF0 standard za funkcionalno modelovanje procesa.

4.3.1. Institucije i organizacije relevantne za standardizaciju modelovanja poslovnih procesa

U savremenoj praksi modelovanja poslovnih procesa (*Business Process Modeling, BPM*), posebno u kontekstu njihove standardizacije, značajnu ulogu igraju međunarodne organizacije i institucije koje razvijaju, održavaju i promovišu standarde i metodologije. Sledeće organizacije su među najrelevantnijima u ovoj oblasti (Dumas et al., 2018; Weske, 2012):

- **OMG** (*Object Management Group*) - Organizacija odgovorna za razvoj i održavanje najpoznatijeg standarda za modelovanje poslovnih procesa — BPMN (*Business Process Model and Notation*). Nakon preuzimanja inicijative BPMI (*Business Process Management Initiative*) 2005. godine, OMG je postao centralno telo za standarde u ovoj oblasti (OMG, 2024).

- **OASIS** (*Organization for the Advancement of Structured Information Standards*) - Aktivna u definisanju XML-baziranih standarda, uključujući BPEL (*Business Process Execution Language*), koji omogućava izvršavanje modelovanih procesa u servisno orijentisanim arhitekturama (SOA) (OASIS, 2024).
- **UN/CEFACT** (*United Nations Centre for Trade Facilitation and Electronic Business*) - Fokusirana na standardizaciju poslovnih procesa i dokumenata u međunarodnoj trgovini. Nudi preporuke i specifikacije kao što je *Core Component Technical Specification* (CCTS) (UN/CEFACT, 2024).
- **W3C** (*World Wide Web Consortium*) - Iako nije primarno orijentisan ka poslovnim procesima, W3C razvija ključne web standarde (npr. XML, RDF, OWL) koji omogućavaju interoperabilnost i formalizaciju modela (W3C, 2024).
- **WfMC** (*Workflow Management Coalition*) (istorijski značaj) - Organizacija koja je razvila standarde poput XPDL (*XML Process Definition Language*), korišćene za razmenu procesa između alata. Prestala sa radom oko 2019. godine, ali se njeni standardi i dalje koriste u određenim alatima (Weske, 2012).
- **Akademske ustanove** - Fakulteti i istraživački centri širom sveta aktivno doprinose teorijskom i metodološkom razvoju oblasti kroz modele, tehnike analize i alate. Među vodećima su Humboldt-Universität zu Berlin, Queensland University of Technology, TU Wien i HPI – Hasso Plattner Institute (Dumas et al., 2018).



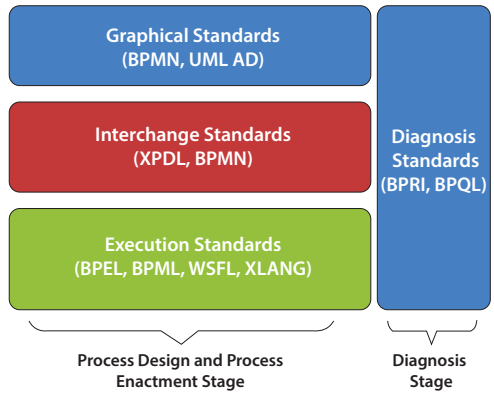
Slika 4.3. Razvoj modelovanja poslovnih procesa kroz teoriju, standarde do alata

Pored organizacija koje razvijaju standarde za modelovanje poslovnih procesa, važno je pomenuti i one koje su doprinele razvoju metoda funkcionalnog modelovanja, kao što su IDEF0 i Dijagram toka podataka (*Data Flow Diagram,* DFD). Ove metode su i dalje prisutne u industriji i obrazovanju, posebno u sistemskom inženjeringu, analizi zahteva i modelovanju informacionih sistema.

- **DFD** (*Data Flow Diagram*) – Ova tehnika je razvijena kao deo metodologije strukturalne analize sistema, najpoznatije kroz Yourdon–DeMarco metodu (Yourdon & DeMarco, 1979). Iako nije formalizovana kroz međunarodni standard, DFD je široko prihvaćena u obrazovanju i praksi, a metodološke smernice su nastale kroz rad konsultantskih firmi i autora.
- **IDEF0** (*Integration DEFinition for Function Modeling*) – Ova metodologija potiče iz američkog Ministarstva odbrane i razvijena je kroz rad organizacije ICAM (*Integrated Computer Aided Manufacturing*) u okviru U.S. Air Force tokom 1980-ih. Kasnije je IDEF0 standardizovan u okviru ANSI (*American National Standards Institute*) kao IDEF0 Function Modeling Method (ANSI/ICAM DEF0) (Marca & McGowan, 1988).

U savremenom upravljanju poslovnim procesima, standardi se mogu klasifikovati prema svojoj funkciji unutar životnog ciklusa procesa. Akademska literatura i industrijska praksa identifikuju četiri osnovne kategorije BPM standarda (van der Aalst, 2004; Weske, 2012; Dumas et al., 2018) (Slika 4.4):

- **Grafički standardi** (*Graphical Standards*) - Omogućavaju korisnicima da vizuelno modeluju poslovne procese, tokove, aktivnosti i interakcije između učesnika. Ovi standardi se koriste u fazi analize i dokumentovanja procesa. Najčešći predstavnici su:
  - BPMN (*Business Process Model and Notation*)
  - UML Activity Diagrams (*UML AD*)
- **Standardi razmene** (*Interchange Standards*) - Omogućavaju prenos modela poslovnih procesa između različitih softverskih alata i okruženja bez gubitka semantike. Tipični primeri uključuju:
  - XPDL (*XML Process Definition Language*)
  - BPDM (*Business Process Definition Metamodel*)
- **Izvršni standardi** (*Execution Standards*) - Ovi standardi definišu kako se modeli procesa mogu izvršavati u okviru informacionih sistema i okruženja za automatizaciju procesa. Koriste se u fazi izvođenja procesa (*Process Enactment*). Najpoznatiji su:
  - BPEL (*Business Process Execution Language*)
  - BPML, WSFL, XLANG
- **Dijagnostički standardi** (*Diagnosis Standards*) - Namenjeni su za nadzor, analizu performansi i evaluaciju poslovnih procesa. Ovi standardi omogućavaju praćenje procesa u stvarnom vremenu, identifikaciju uskih grla i optimizaciju. Uključuju standarde kao što su:
  - BPRI (*Business Process Runtime Interface*)
  - BPQL (*Business Process Query Language*)

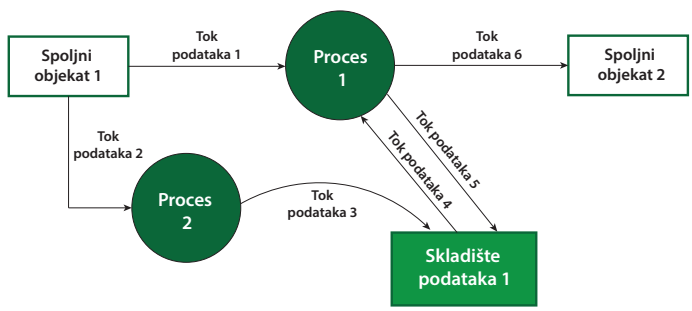


Slika 4.4. Klasifikacija BPM standarda

4.3.2. Dijagram toka podataka

Dijagram toka podataka (*Data Flow Diagram, DFD*) predstavlja tehniku dekompozicije složenog sistema na manje podsisteme, sve do nivoa zadataka (tzv. primitivnih procesa, odnosno procesa koji se dalje ne mogu dekomponovati) vizuelno predstavljajući protok informacija kroz sistem. DFD pomaže da se razumeju procesi unutar sistema, identifikuju ulazi, izlazi, skladišta podataka i interakcije između različitih komponenti. Dakle, osnovni elementi DFD-a su (Slika 4.5):

- **Procesi** – prikazuju se u obliku kruga i mogu biti složeni (dekomponuju se na druge manje procese) ili primitivni (procesi koji se ne mogu dalje dekomponovati, a obično se još nazivaju zadaci ili aktivnosti).
- **Spoljni objekti ili akteri** – prikazuju se u obliku pravougaonika i predstavljaju spoljne entitete ili objekte koji komuniciraju sa sistemom.
- **Skladišta podataka** – predstavljene su sa dve paralelne linije i obično predstavljaju mesta gde se trajno čuvaju podaci.
- **Tokovi podataka** – povezuju spoljne objekte sa procesima i procese sa skladištima. Grafički se predstavljaju kao usmerene linije.



Slika 4.5. Osnovni elementi DFD dijagrama

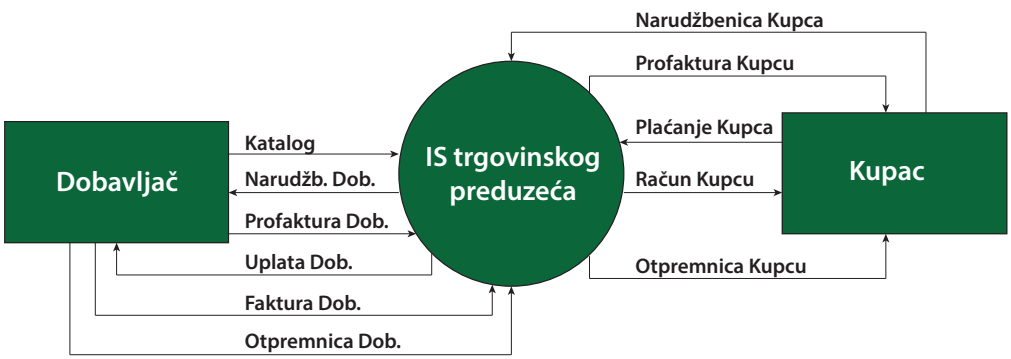
Suština ove tehnike je **hijerarhijska dekompozicija** gde se jedan proces višeg nivoa apstrakcije dekomponuje na njegove podprocese na nižem nivou apstrakcije. Jedan od načina savladavanja složenosti poslovnih sistema je funkcionalna dekompozicija. Tradicionalna struktuirana sistemska analiza (*Structured System Analysis, SSA*) je jedna od metoda koja se zasniva na funkcionalnoj dekompoziciji.

Na primer, pretpostavimo da treba da modelujete procese jednog manjeg trgovinskog preduzeća. Prvo što crtate je dijagram konteksta ili dijagram nultog nivoa. Na njemu se nalazi samo jedan proces, a to je proces tog trgovinskog preduzeća, za koga želite da napravite aplikaciju. Fokus na ovom prvom dijagramu nije toliko na procese koliko na spoljne objekte (entitete) koji će komunicirati (razmenjivati informacije) sa trgovinskim preduzećem i na sve tokove podataka koje će se odvijati između njih.

Na slici 4.6 se može videti da će trgovinsko preduzeće razmenjivati podatke sa dva spoljna entiteta i to Dobavljači i Kupci. Tokovi podataka između Dobavljača i trgovinskog preduzeća su:

- Dobavljač šalje katalog svojih proizvoda trgovinskom preduzeću
- Preduzeće šalje porudžbinu dobavljaču.
- Dobavljač na osnovu pordudžbine šalje profakturu preduzeću.
- Preduzeće šalje uplatnicu dobavljaču o izvršenoj uplati.
- Dobavljač šalje fakturu preduzeću.
- Dobavljač šalje robu sa otpremnicom preduzeću.

Tokovi podataka između trgovinskog preduzeća i njegovih kupaca su veoma slični. Tok počinje kada kupac šalje porudžbenicu, na osnovu koga preduzeće šalje profakturu kupcu i redom kako je i prikazano na slici 4.6.



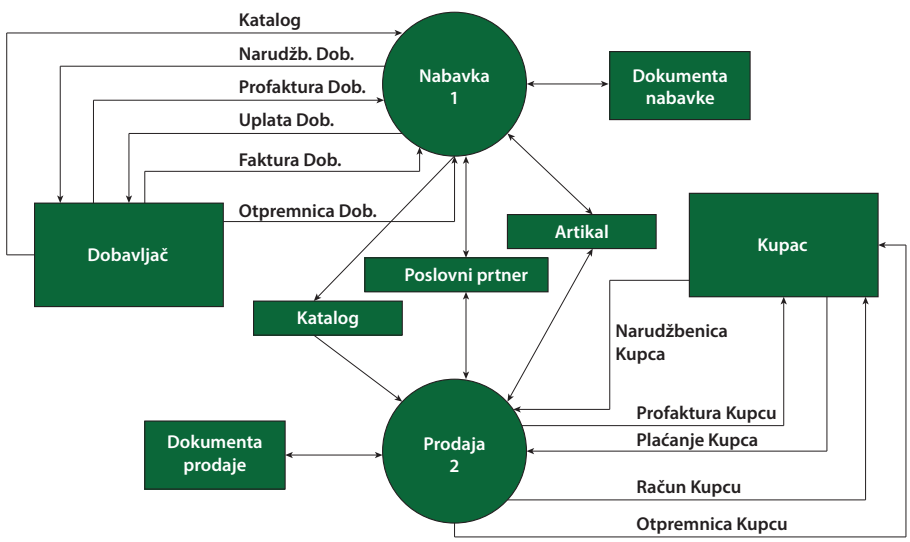
Slika 4.6. Dijagram konteksta

Vrlo je važno prilikom crtanja dijagrama konteksta uočiti sve moguće objekte sa kojima će naš budući informacioni sistem preduzeća razmenjivati podatke, kao i sve moguće tokove podataka. Ukoliko niste sigruni da li ste obuhvatili sve tokove podataka, možete se pitati kako teče kompletan životni ciklus posmatranog entiteta, od njegovih procesa rađanja, razvoja, opadanja i nestanka. Na primer, ukoliko treba da modelujete sve tokove između službe za ljudske resursa i zaposlenih, onda bi trebalo da se pitate kako teče životni ciklus zaposlenih u preduzeću, a to su obično koraci regrutacije, selekcije, zapošljavanja, praćenja performansi, obučavanje, razvoj, tranzicije, prestanak radnog odnosa i sl.

Kada ste sagledali sve spoljne entitete i tokove podataka, sledeći dijagram koji crtate je dijagram prvog nivoa. Fokus na tom dijagramu su ključni podprocesi, u ovom primeru, posmatranog trgovinskog preduzeća. Na dijagramu prvog nivoa ste limitirani po broju podprocesa, neki alati vam ne dozvoljavaju da imate vše od osam podprocesa. Podprocesi su obično odelenja (departmani) u nekoj složenijoj organizaciji, koji grupišu određena poslovanja. U našem primeru, malo trgovinsko preduzeće ima samo dva odelenja i to Nabavku i Prodaju, tako da ima samo dva podprocesa na dijagramu prvog nivoa (Slika 4.7).

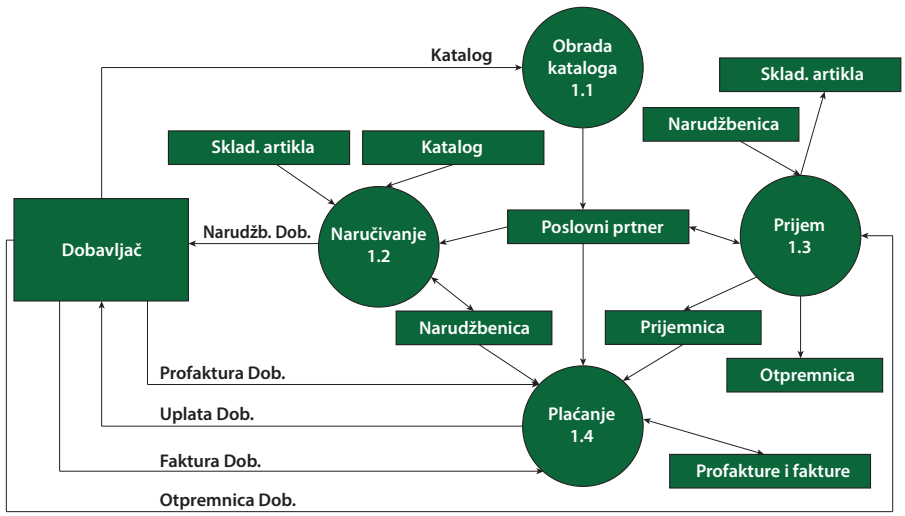
Fokus na dijagramu prvog nivoa su:

- Pridružiti tokove podataka odgovarajućem podprocesu.
- Uočiti skladišta podataka, npr. da li u odelenju nabavke zaposleni skladište neke podatke. Na našem primeru to su baze artikala, poslovnih partnera, kataloga i dokumenta nabavke.
- Strelice tokova podataka su ključne, jer pokazuju da li proces Nabavke samo unosi podatke ili ih i uzima radi obrade. U većini slučajeva strelice su dvosmerne.



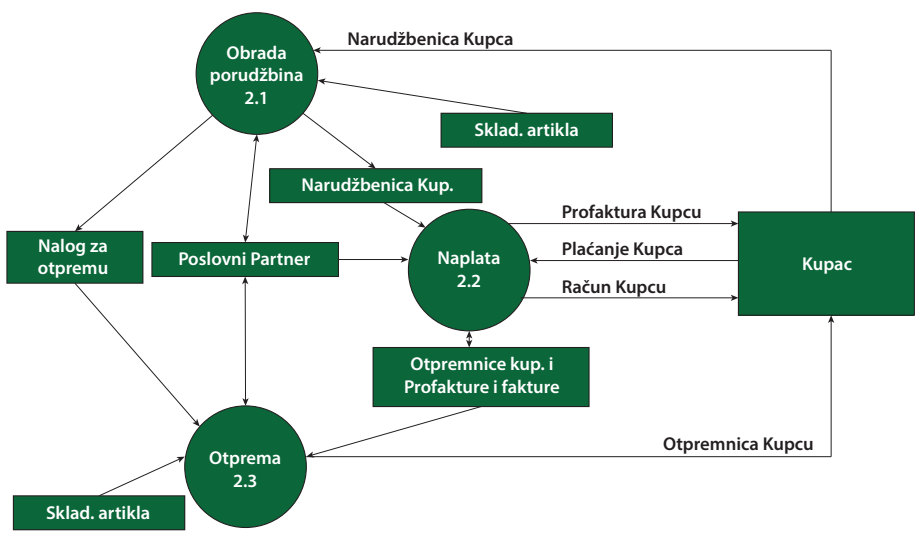
Slika 4.7. Dijagram prvog nivoa

Naredni dijagrami koji se crtaju su dijagrami dekompozicije podprocesa. Na slici 4.8 se može videti da se proces Nabavke deli na četiri podprocesa i to: Obrada kataloga, Naručivanje, Prijem i Plaćanje. Tokovi sa prethodnog dijagrama se pridružuju odgovarajućim podprocesima i uočavaju se neka nova skladišta, ukoliko se pojave.



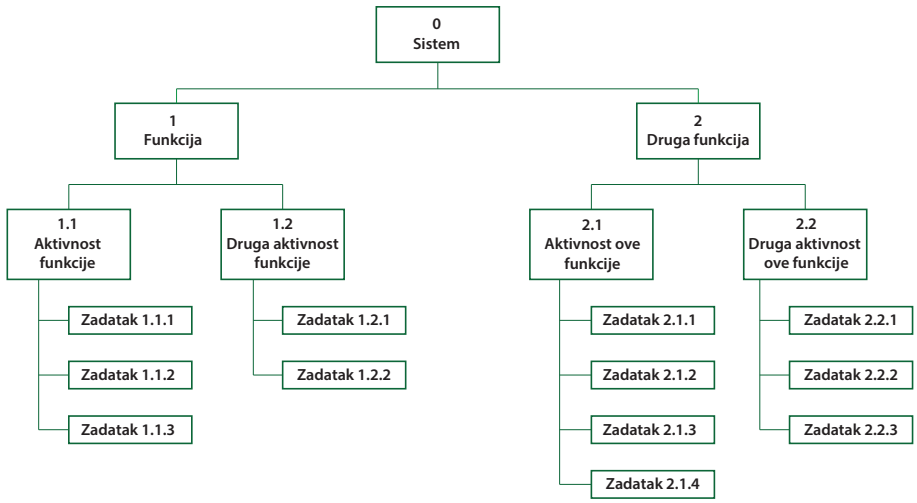
Slika 4.8. Dekompozicija procesa nabavke

Pokušajte samostalno da protumačite šta je prikazano na slici 4.9:



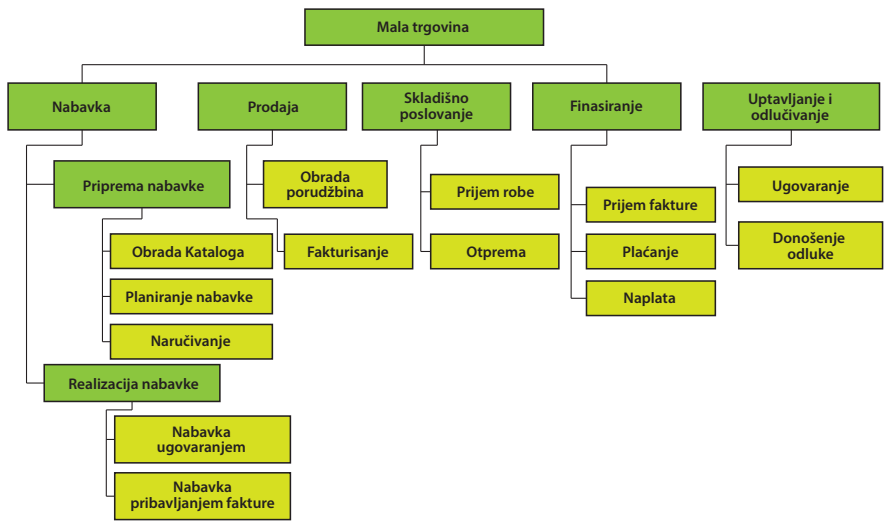
Slika 4.9. Dekompozicija procesa prodaje

Dekompozicija se vrši sve do nivoa zadataka koji se više ne mogu dekomponovati. Dijagram dekompozicije prikazuje *top-down* (sa vrha na dole) funkcionalnu dekompoziciju i strukturu sistema. Na slici 4.10 su prikazani samo procesi pomoću stabla aktivnosti sistema.



Slika 4.10. Funkcionalna dekompozicija (stablo aktivnosti sistema)

Primer klasifikacione dekompozicije jednog manjeg trgovinskog preduzeća je prikazan na slici 4.11.



Slika 4.11. Stablo aktivnosti manjeg trgovinskog preduzeća

Najvažnije pravilo koje se mora poštovati pri dekompoziciji procesa je pravilo balansa tokova. Na nižim dijagramima dekompozicije, ne mogu se pojavljivati novi tokovi koji se nisu prikazali na dijagramu konteksta. Međutim, ono što je dopustivo, pre svega zbog preglednosti, da se dijagrami tokova mogu kasnije dekomponovati na više prostijih tokova podataka.

Pored procesa, mogu se dekomponovati i tokovi i skladišta - njihov opis se detaljno daje u rečniku podataka.

Rečnik podataka (*Data Dictionary*) predstavlja struktuirano skladište meta podataka, tj. podataka o podacima. Prvobitno se pojavio kao proširenje dijagrama toka podataka, za opisivanje sadržaja i strukture svih tokova i skladišta podataka. Naime, tokom snimanja stanja sistema i crtanja DFD dijagrama, za svaki tok podataka i skladište se prikazuju podaci koje sadrže. Na primer, ukoliko je tok podataka faktura, onda se sagledava koja polja čine tu fakturu (broj fakture, datum, naziv kupca, adresa kupca). Primer BNF (*Backus-Naur Form*) notacije rečnika podataka je prikazan na slici 4.12.

Znak	Opis
=	Struktura sa leve strane se sastoji od delova sa desne (sastavljeno od“)
+	Agregacija elemenata
( )	Opcionalnost elemenata u zagradi (0 ili 1)
{ }	Ponavljanje (iteracija) elemenata u zagradi do konačnog broja puta
[ ]	Altemativa (selekcija) elemenata u zagradi
I	Odvajanje altemativnih elemenata u [ ] izrazu
-	Početna i završna vrednost raspona defimsanog [ ] izrazom
**	Komentar
@	Oznaka ključa

Slika 4.12. Elementi Backus-Naur notacije rečnika podataka

Jedan primer rečnika podataka za Račun prema BNF notaciji je:

```
Racun      = @BrRac + DatRac + BrKupca
           + { SifArt, NazArt, Cena, Kol, Vrednost }
           + (IznosRac)
```

Prilikom sagledavanja tokova podataka, prikupićete i dokumenta koje kompanije skladište. Uvidom u dokument račun, videćete da on sadrži polja, kao što su broj računa, datum računa, broj kupca, šifra artikla, naziv artikla, cena artikla, količina artikla, vrednost i iznos računa. Oznake pored ovih polja označavaju:

- Broj računa je jedinstven te stoga može biti deo ključa pomoću koga će se pretraživati računi – zato se stavlja oznaka @.
- Za polja šifra artikla, naziv artikla, cena, količina i vrednost postoji više vrednosti podataka, naime jedan račun sadrži više kupljenih artikala – zato je stavljena uglasta zagrada.
- Iznos računa se može izračunati sabiranjem prethodnih vrednosti artikla – izvedena polja se mogu staviti u obične zagrade.

Da zaključimo, nivoi dekompozicije DFD-a su:

- Nivo 0 (kontekstni dijagram): Prikazuje ceo sistem kao jedan proces i njegovo povezivanje s spoljnim entitetima.
- Nivo 1: Razlaže glavni proces iz nivoa 0 na manje podprocese (obično maksimalno 8), pružajući više detalja o protoku podataka.
- Nivo 2 (i više): Dijagrami dekompozicije koji prikazuju dalje razlaganje podprocesa, radi detaljnijeg razumevanja.

Prednosti DFD-a:

- Jasna vizualizacija: Omogućava lakše razumevanje kako podaci teku kroz sistem.
- Identifikacija problema: Pomaže u otkrivanju slabosti ili problema u dizajnu sistema.
- Komunikacija: Olakšava komunikaciju između tima za razvoj, klijenata i drugih zainteresovanih strana.
- Planiranje: Koristi se za planiranje razvoja i implementacije sistema.

DFD je moćan alat u fazama analize i projektovanja aplikacije jer omogućava kreiranje osnovnog modela za razumevanje i dokumentovanje složenih sistema.

4.3.3. IDEF0

Ranih 90-tih IDEF (*Integration DEFinition*) Users Group usvojila je sledeće glavne standarde:

- IDEF0 za funkcionalno modelovanje procesa
- IDEF1X za informaciono modelovanje, odnosno modelovanje podataka.

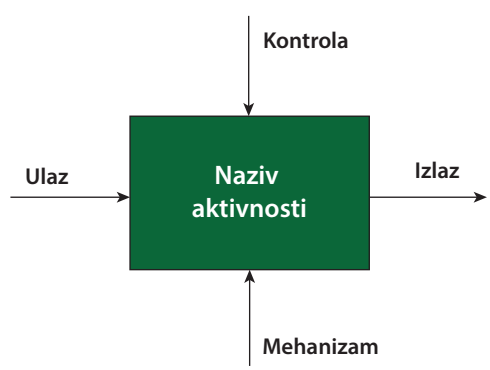
Softverski alati koji su podržavali ove IDEF standarde su:

- za IDEF0 standard je alat BPwin (*Business Process Windows*) firme LogicWorks
- za IDEF1X standart je alat ERwin (*Entity Relationships for Windows*)

Za razliku od DFD dijagrama, IDEF0 standard nema spoljne entitete, ali zato ima pored ulazni i izlaznih podataka i strelice kontrole (koje pokazuju da li taj proces mora da implementira neko pravilo, regulativu, zakone) i strelice mehanizama (koje označavaju ko obavlja taj proces ili u okviru kog sistem je taj proces već implementiran). Kod ovog standarda, veoma je važno da tokovi podataka koji ulaze u naredni proces budu prikazani na adekvatnom mestu, odnosno, ako je izlazni rezultat prethodnog procesa neko novo pravilo koje je menadžment doneo, da se taj tok podataka prikaže kao kontrola (da ulazi odozgo) u naredni proces.

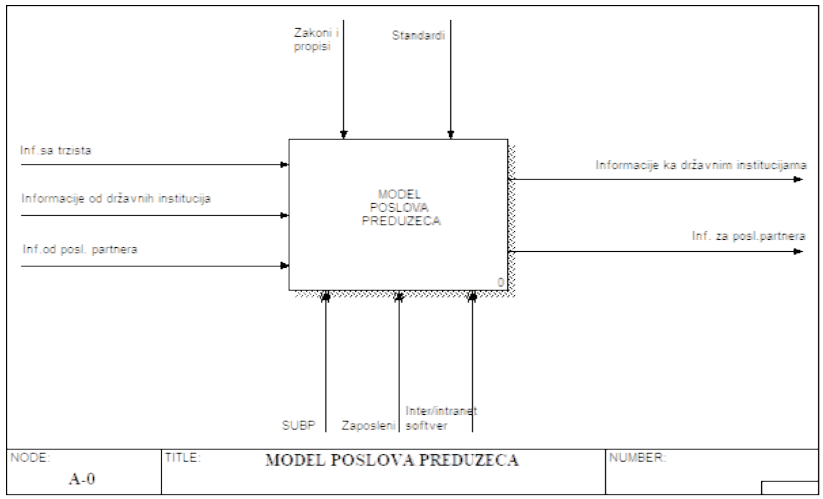
Osnovni elementi IDEF0 standarda su (Slika 4.13):

- **Ulaz** – ulazni podaci mogu biti materijali, informacije i drugi resursi koji se koriste dalje za obradu.
- **Kontrola** – uključuju pravila, politike, procedure ili standarde koji utiču na izvođenje procesa.
- **Izlaz** – izlazni podaci koji su rezultat obrade procesa, a mogu biti materijali, informacije i drugi resursi.
- **Mehanizam** – nazivi ljudi, mašina, opreme, tehnologija i sve ono što obezbeđuje energiju potrebnu za izvođenje procesa, a koji pokazuju ko izvodi ili gde se izvršava proces.



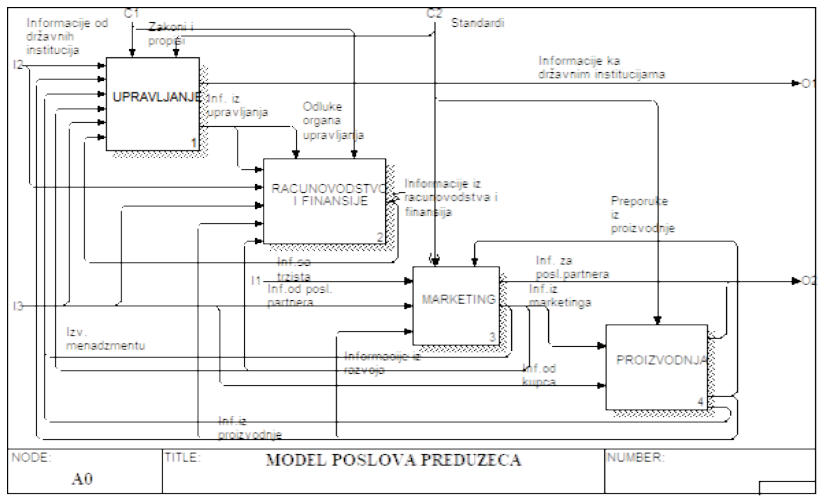
Slika 4.13. Osnovni elementi procesa prema IDEF0 standardu

Postupak modelovanja procesa pomoću IDEF0, je sličan kao kod DFD dijagrama. Prvi dijagram, tzv. dijagram nultog nivoa (Slika 4.14) sadrži samo jedan proces i sve tokove podataka, jedino što su ulazni tokovi prikazani sa leve strane, izlazni sa desne itd.



Slika 4.14. Dijagram nultog nivoa opšteg modela preduzeća

Dijagram prvog nivoa sadrži ključne podprocese i njima pridružene odgovarajuće tokove podataka (Slika 4.15).

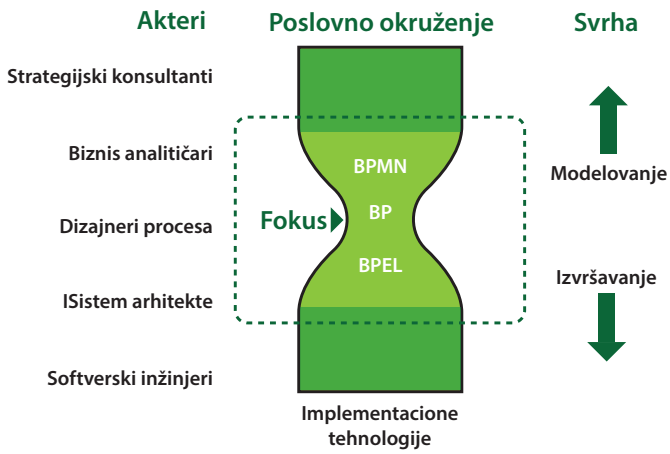


Slika 4.15. Dijagram prvog nivoa opšteg modela preduzeća

Ostali dijagrami su dijagrami dekompozicije određenih složenih procesa.

4.3.4. Modelovanje i notacija poslovnih procesa - BPMN

U savremenom poslovnom okruženju, pojavila se potreba za jedinstvenom, standardizovanom notacijom kojom bi poslovni korisnici i IT stručnjaci mogli jasno i jednoznačno da komuniciraju procese. Kao odgovor na tu potrebu razvijen je BPMN (*Business Process Model and Notation*).



Slika 4.16. BPMN kao zajednički jezik komunikacije između biznis i IT stručnjaka

BPMN je nastao kao inicijativa organizacije *Business Process Management Initiative* (BPMI), a kasnije je njegovo održavanje preuzela Object Management Group (OMG), koja i danas razvija i objavljuje aktuelne verzije standarda. Današnja najrašireniija verzija je BPMN 2.0, koja pored grafičkog prikaza obuhvata i semantičku i izvršnu specifikaciju procesa.

Za razliku od notacija poput DFD i IDEF0, koje su fokusirane na protok podataka i funkcionalnu dekompoziciju, BPMN pruža detaljan, ali intuitivan način za prikaz toka aktivnosti, uloga, događaja i pravila odlučivanja. Zahvaljujući jasno definisanim elementima i bogatoj simbolici, BPMN je postao dominantni standard za modelovanje poslovnih procesa, kako u poslovnoj analizi, tako i u implementaciji rešenja (npr. u okviru sistema za upravljanje poslovnim procesima – BPMS).

Glavni cilj BPMN notacije jeste povezivanje poslovnog i tehničkog sveta, omogućavajući:

- poslovnim korisnicima da razumeju i dizajniraju procese bez tehničkih znanja,
- IT timu da modele implementiraju i automatizuju u informacionim sistemima.

Osnovni tipovi BPMN modela

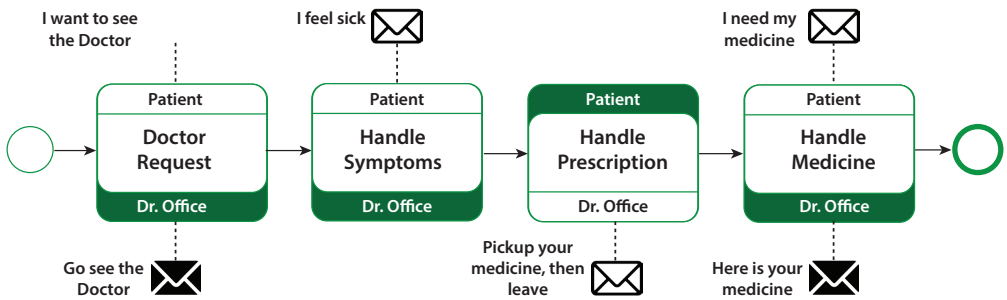
BPMN podržava modelovanje različitih tipova poslovnih procesa u zavisnosti od njihove funkcionalne i komunikacione uloge. Standardna klasifikacija uključuje sledeće tri glavne kategorije:

1. **Orkestracija** (*Process Orchestration*) - predstavlja modelovanje internih tokova aktivnosti unutar jedne organizacije ili poslovne jedinice. Najčešće se prikazuje u jednoj particiji.



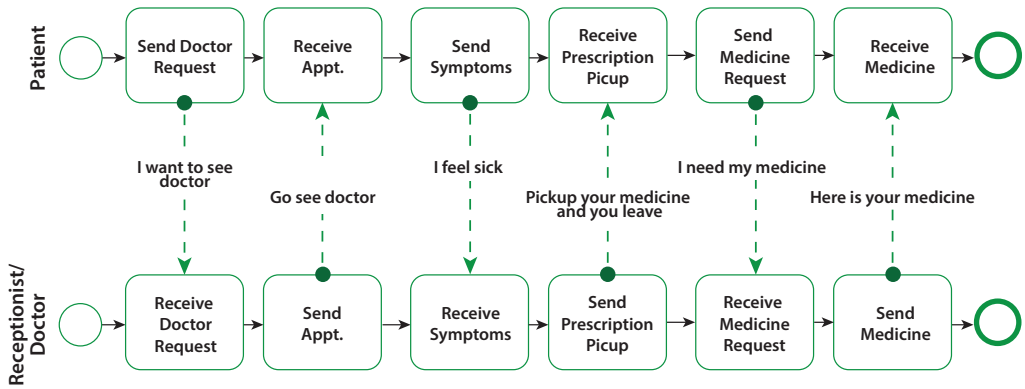
Slika 4.17. Primer orkestracije

2. **Koreografija** (*Process Choreography*) - Koreografija modeluje međusobnu razmenu poruka između nezavisnih učesnika (organizacija ili sistema), bez centralne kontrole. Fokus je na pravilima interakcije i redosledu komunikacije —„ko komunicira s kim, kada i kako". Koreografija ne uključuje detalje unutrašnjih procesa učesnika, već samo međusobne interakcije.



Slika 4.18. Primer koreografije

3. **Kolaboracija** (*Collaboration*) - Kolaboracijski modeli proširuju koreografiju time što obuhvataju i tokove procesa unutar učesnika, zajedno sa njihovim međusobnim porukama. Ovi modeli su pogodni za prikaz: B2B komunikacije (*peer-to-peer*) između partnera i ugovornih i semantičkih odnosa između partnera (uključujući komunikaciju preko Interneta, usaglašavanje standarda razmene i poruka i dogovor o značenju i formatu poslovnih podataka).



Slika 4.19. Primer kolaboracije

Zahvaljujući podršci za ove modele, BPMN omogućava sveobuhvatno modelovanje kako internih, tako i međusobno koordinisanih procesa između više aktera, što je posebno važno u kontekstu savremenih digitalnih lanaca vrednosti.

Osnovni elementi BPMN notacije

BPMN modeli se sastoje od pet osnovnih kategorija elemenata:

- **Tok objekata** (*Flow Objects*) – osnovni grafički elementi za definisanje ponašanja/semantike poslovnih procesa: *Event, Activity* i *Gateway*.
- **Povezivanje objekata** (*Connecting objects*) – opisuju interakciju između tokova objekata: *Sequence Flow, Message Flow* i *Association*.
- **Podaci** (*Data*): Ulazni i izlazni podaci, Podaci objekta i kolekcije, Skladište i Poruke.
- **Staze** (*Swimlines*) – grupišu elemente u *Pools* i *Lanes*.
- **Artefakti** (*Artifacts*) – obezbeđuju dodatne informacije o procesima i mogu biti: *Group* i *Annotation*.

Events su događaji koji se dešavaju u toku poslovnog procesa i utiče na njegov tok izvršavanja. BPMN dešiniše tri vrste događaja i to (Slika 4.20):

- ***Start Events*** - započinje tok procesa i može se koristiti da ukaže na to kako će prijem poruke pokrenuti instancu procesa, npr., Prijem porudžbine.



- ***Intermediate Events*** - ukazuje na to gde se nešto dešava tokom izvršenja procesa i utiče na sam tok procesa. Može da ukaže gde bi se mogla primiti poruka ili gde se očekuju kašnjenja ili pak da prekinu normalan tok usled pojave izuzetaka.



- ***End Events*** (opcioni) - završava tok procesa i može se koristiti za slanje poruke ili da ukaže na grešku i sl.

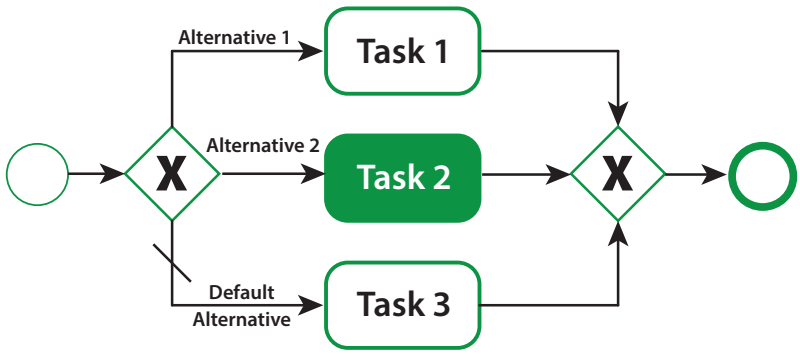


Events	Start			Intermediate				End
	Top-Level	Event Sub-Process Interruping	Event Sub-Process Non-Interruping	Catching	Bounday Interruping	Bounday Non-Interruping	Throwing	
None: Untyped events, indicate start point, state changes or final states.								
Message: Receiving and sending messages.								
Timer: Cyclic timer events, points in time, time spans or timeouts.								
Escalation: Escalating to an higher level of responsibility.								
Conditional: Reacting to changed business conditions or integrating business rules.								
Link: Off-page connectors. Tv/o corresponding link events equal a sequence flow.								
Error: Catching or throwing named errors.								
Cancel: Reacting to cancelled transactions or triggering cancellation.								
Compensation: Handling or triggering compensation.								
Signal: SignaUing across different processes. A signal throv/n can be caught multiple times								
Multiple: Catching one out of a set of events. Throv/ing all events defined.								
Parallel Multiple: Catching all out of a set of paraUel events.								
Terminate: Triggering the immediate termination of a process.								

Slika 4.20. Različite oznake događaja (*events*)

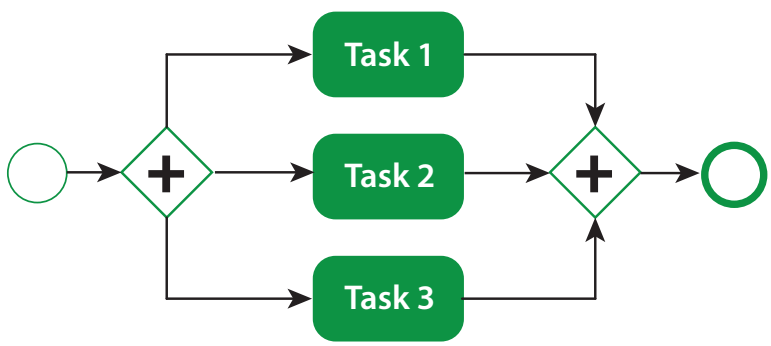
Gateways ili kontrolni čvorovi se koriste pri definisanju kako se tokovi aktivnosti razdvajaju ili spajaju u okviru procesa. BPMN definiše četiri vrste kontrolnih čvorova:

- **Exclusive Data-Based** (XOR) - Uslov ispunjava samo jedan tok događaja (npr. kao kod *if-then* strukture). Ukoliko služi za razdvajanje, izvršava se tačno jedna grana, a ukoliko služi za spajanje, čeka samo jednu granu da se izvrši pre nego što nastavi dalje.

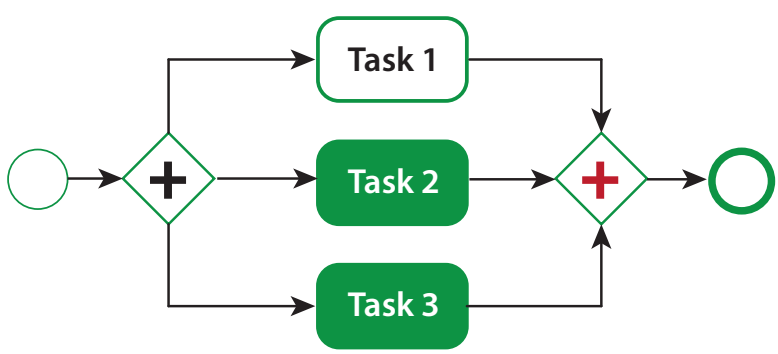


Slika 4.21. Primeri XOR *gateway*-a

- **Parallel Gateways** (AND) ukazuju na paralelno izvršavanje tokova događaja. Ako služi kao element spajanja, tada sSve paralelne aktivnosti moraju biti završene da bi se nastavilo dalje.



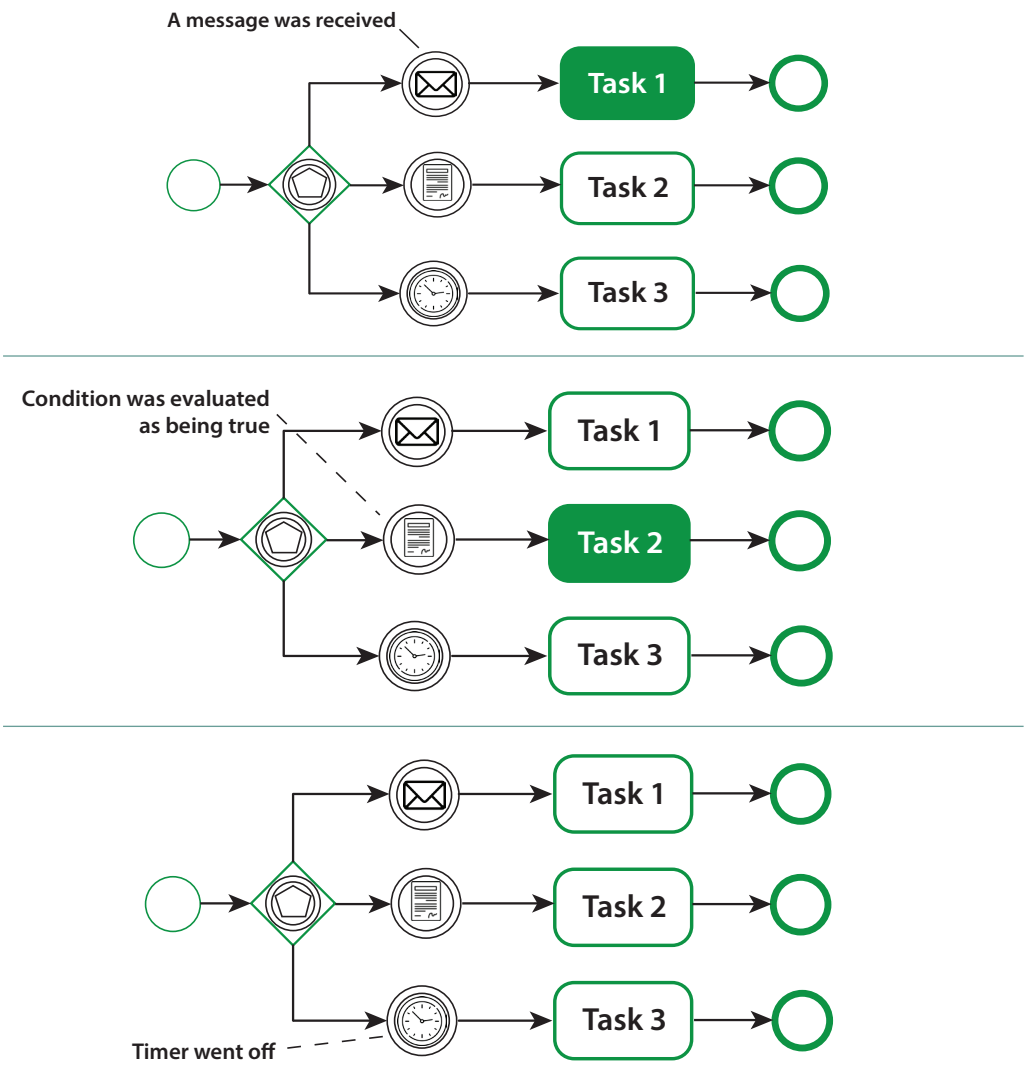
Paralelno se izvršavaju aktivnosti, ali ne moraju istovremeno da budu obrađene



Join čeka na rezultate svih aktivnosti da bi se nastavilo dalje

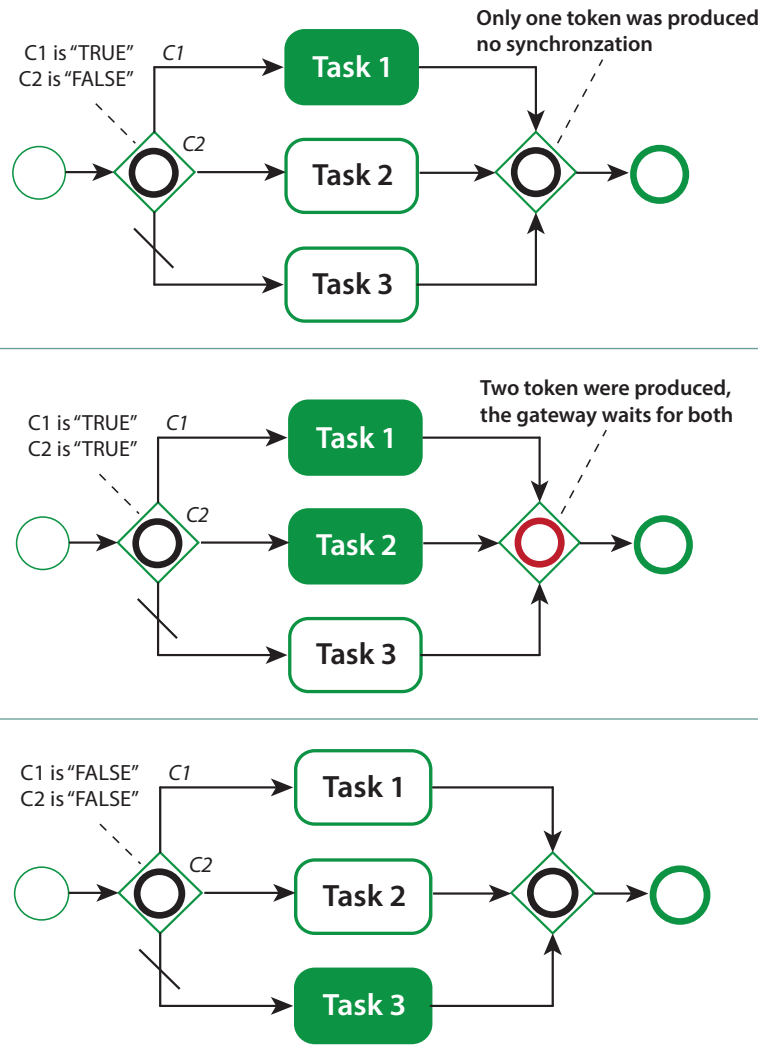
Slika 4.22. Primeri paralelnih kontrolnih čvorova

- **Exclusive Event-Based Gateway** (XOR) je sličan Exclusive Data-Based Gateway gde umesto utvrđivanja skupa alternativa tj. uslova koga može da ispuni samo jedan tok, ovde zavisi od rezultata događaja



Slika 4.23. Primer XOR kontrolnih čvorova

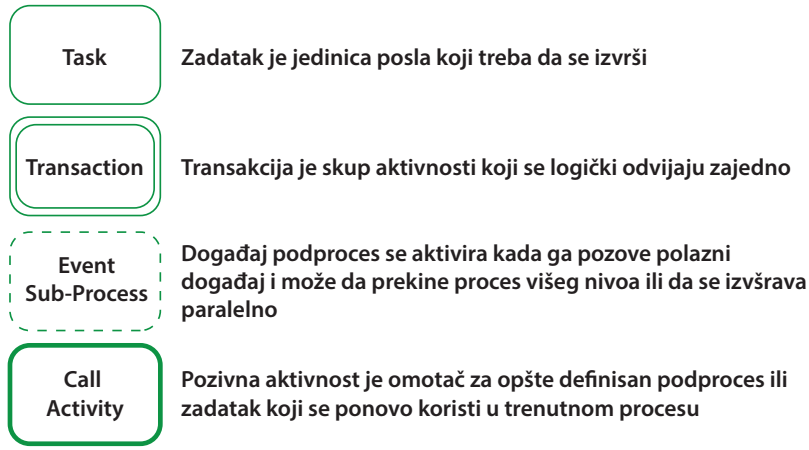
- **Inclusive Gateway** (OR) se može koristiti kao tačka odlučivanja gde je moguća realizacija više tokova događaja. Svi tokovi su ograničeni sa uslovom i svaki tok koji ispunjava uslov biće izvršen. Ukoliko više tokova ispunjava uslov, neće se krenuti dalje dok se sve aktivnosti ne završe.



Slika 4.24. Primeri OR kontrolnih čvorova

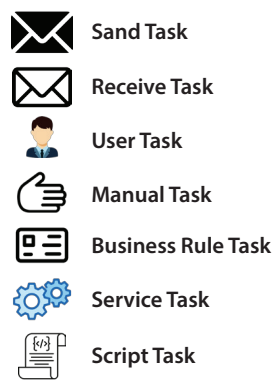
Aktivnost je jedinica posla koja treba da se izvrši - može biti zadatak, proces ili podproces. BPMN definiše dve glavne vrste aktivnosti:

- **Zadatak** (*Task*) – atomska aktivnost koja je uključena u proces
- **Podproces** (*Sub-process*) je proces koji je uključen u drugi proces.



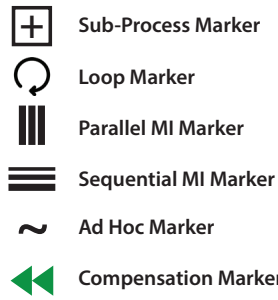
Slika 4.25. Notacija različitih aktivnosti

Tipovi ukazuju na prirodu akcije koja treba da se izvrši (Slika 4.26).



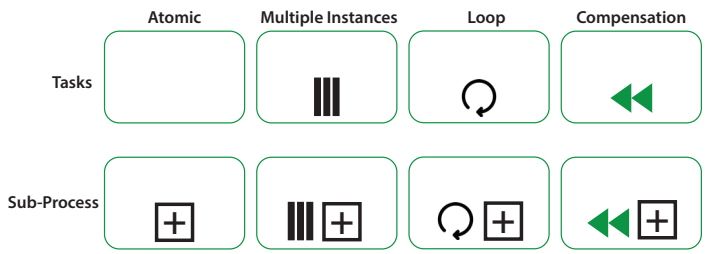
Slika 4.26. Tipovi aktivnosti

- **Markeri aktivnosti** (*Activity Markers*) - Markeri (oznake) ukazuju na izvršno ponašanje aktivnosti. Markeri se definišu radi specifikacije dodatne semantike, npr. petlje.



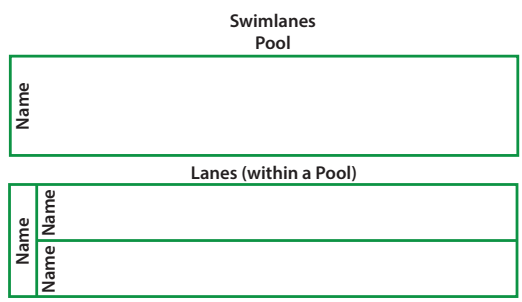
Slika 4.27. Primeri oznaka aktivnosti

Primeri označavanja aktivnosti su prikazani na slici 4.28.



Slika 4.28. Primeri označavanja aktivnosti

Plivačke staze (*Swimlanes*) – važnu ulogu igraju particije ili plivačke staze (*swimlanes*), koje vizuelno odvajaju različite uloge ili organizacione entitete (npr. *Pool* i *Lane*). *Pool* (bazen) je grafički kontejner ili particija za razdvajanje skupa aktivnosti. Plivačke staze (*Lanes*) su podelementi (particije) u okviru pool elementa, prikazani vertikalno ili horizontalno. Određena staza će prikazivati zadatke koji su relevantni za određenu poslovnu jedinicu (aktera).



Slika 4.29. Primeri particija

Podaci mogu biti:

- Ulazni podatak (*Data Input*) – spoljni input za čitav proces.
- Izlazni podatak (*Data Output*) – promenljiva nastala kao rezultat procesa.
- Podatak objekta (*Data Object*) – podaci koji se razmenjuju između procesa, kao što su poslovna dokumenta, emailovi ili pisma.
- Kolekcija podataka objekta (*Collection Data Object*) - kolekcija informacija, npr. lista poručenih proizvoda.
- Skladište podataka (*Data Store*) – mesto odakle proces može čitati ili upisivati podatke.
- Poruka (*Message*) – opisuje sadržaj komunikacije između učesnika.

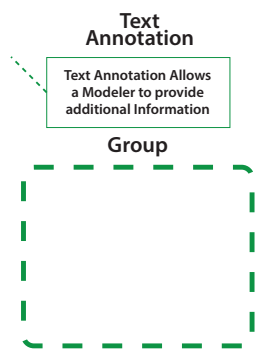
Tokovi podataka prikazuju međusoban odnos između elemenata procesa i mogu biti:

- ***Sequence Flow*** (sekvencijalni tok) – povezuje aktivnosti i definiše redosled izvršavanja. Oznaka:
- ***Default Flow*** (Difoltni tok) – prikazuje difoltnu granu koja će biti izabzrana ukoliko niko ne ispunjava zadate uslove. Oznaka:
- ***Conditional Flow*** (Uslovni tok) – sadrži uslov koji ukazuje da li će tok biti realizovan. Oznaka:
- ***Message Flow*** (tok poruke) – prikazuje razmenu informacija između različitih učesnika (npr. između Pool-ova). Oznaka:

Korišćenjem ovih tokova model postaje čitljiv i funkcionalan, pogodan za analizu, simulaciju i implementaciju.

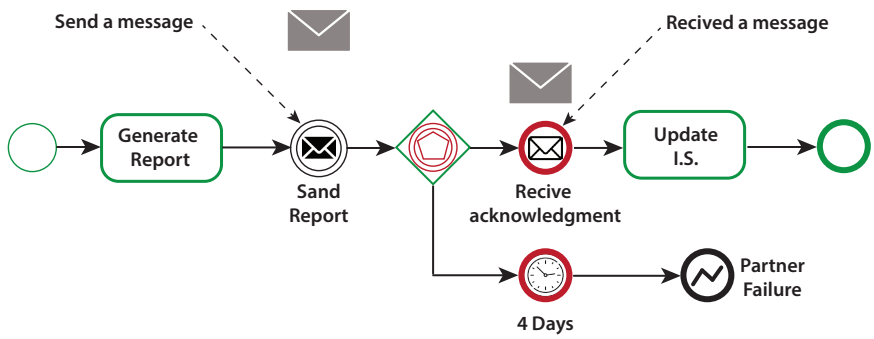
Pomoćni elemeti (*Artifacts*) mogu biti:

- ***Text Annotations*** je mehanizam za specificiranje dodatnih informacija za "čitaoce" BPMN diagrama
- **Grupa** obezbeđuje mehanizam za vizuelno grupisanje elemenata

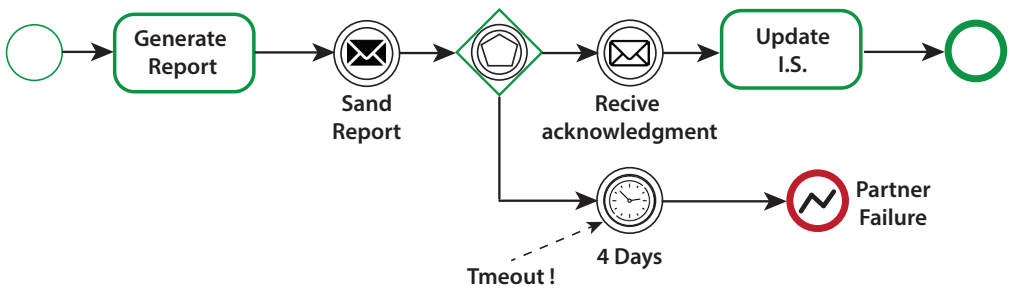


Slika 4.30. Primeri pomoćnih elemenata

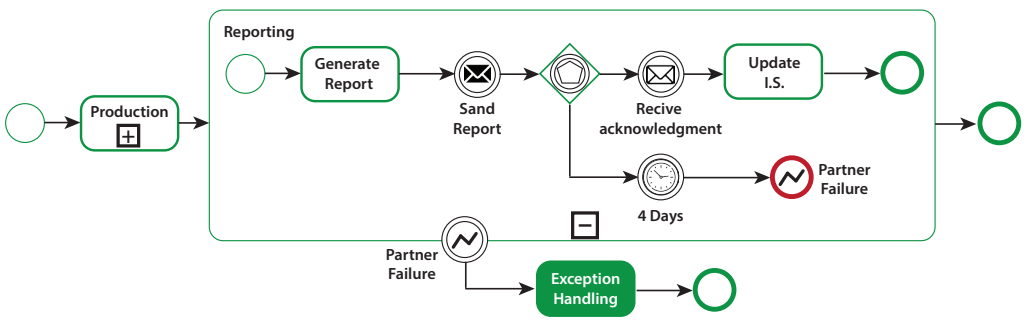
Protumačite sledeće primere scenarija:



Slika 4.31. Scenario 1



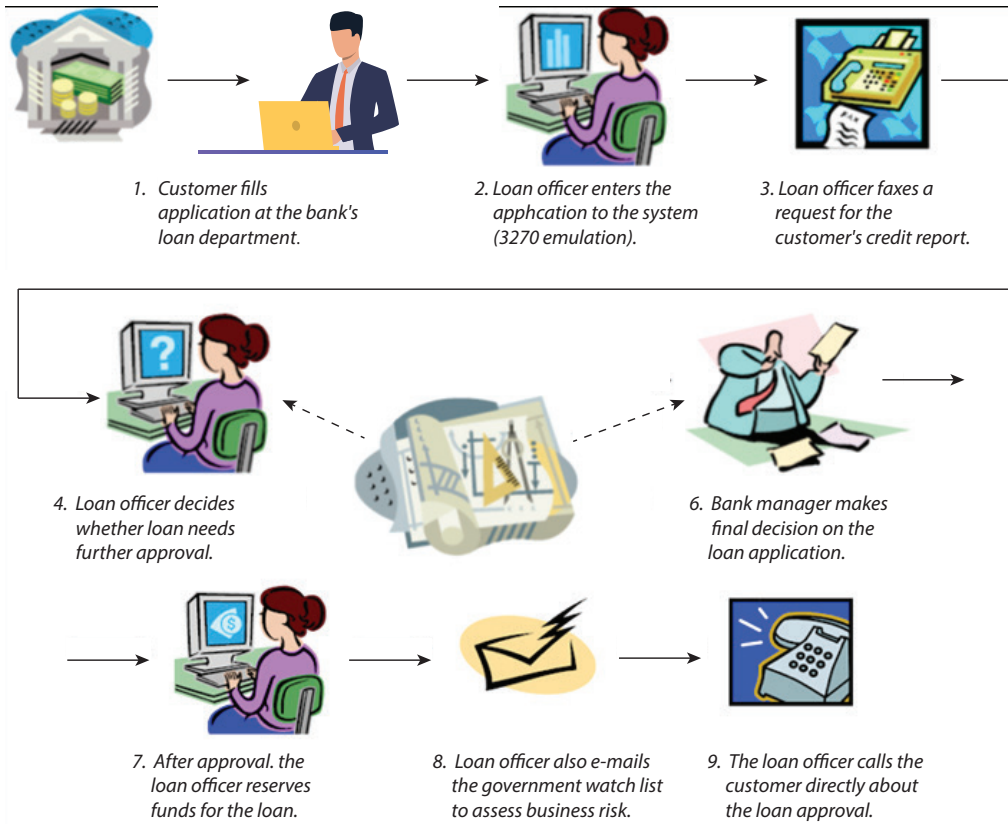
Slika 4.32. Scenario 2



Slika 4.33. Scenario 3

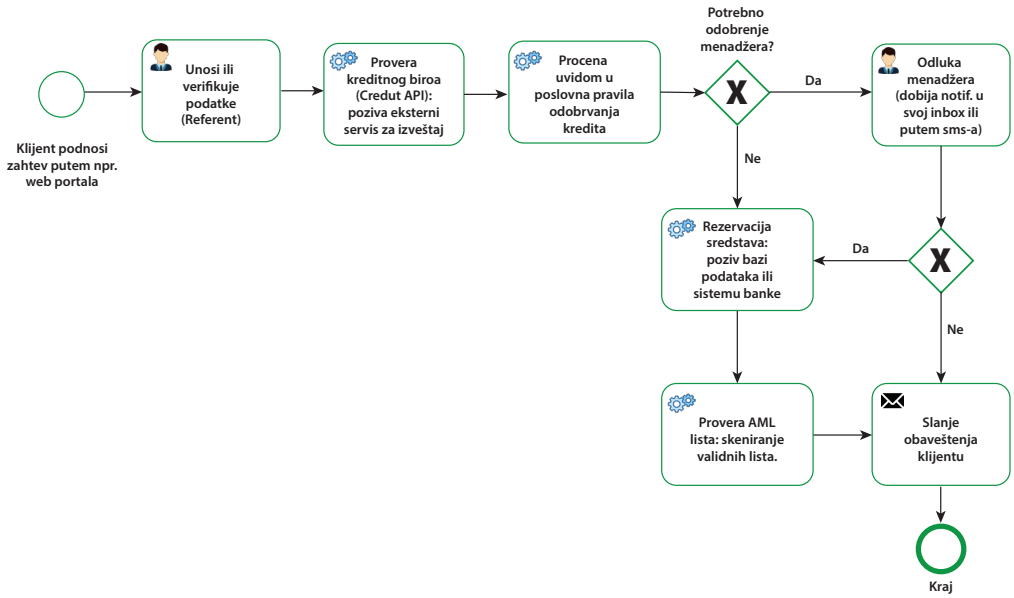
Vežba 4.1. Nacrtati BPMN za proces "Odobravanje kredita"

Na osnovu realnog scenarija izdavanja kredita u jednoj banci, prikazanog na slici 4.34, potrebno je nacrtati BPMN model. Preporučuje se korišćenje softvera kao što su Camunda Modeler, Bizagi Modeler, Lucidchart ili bpmn.io za crtanje BPMN dijagrama.



Slika 4.34. Opis jednog realnog scenarija za izdavanja kredita u banci

Na osnovu opisa koraka sa slike 4.34, transformisaćemo ovaj zastareli, manuelni proces (faksiranje, kucanje u terminal, telefonski pozivi) u moderan servisno-orijentisani model koristeći BPMN notaciju. Kod servisno orijentisanih aplikacija, cilj je da što više koraka bude automatizovano putem servisa, dok se ljudska intervencija zadržava samo tamo gde je neophodna odluka. Koristeći onlajn alat https://bpmn.io/, BPMN model ovog realnog scenarija bi izgledao kao na sledećoj slici 4.35.



Slika 4.35. BPMN model za proces izdavanja kredita jedne banke

Vežba 4.2. Kreirati BPMN dijagram za proces "Podnošenje i obrada zahteva za godišnji odmor"

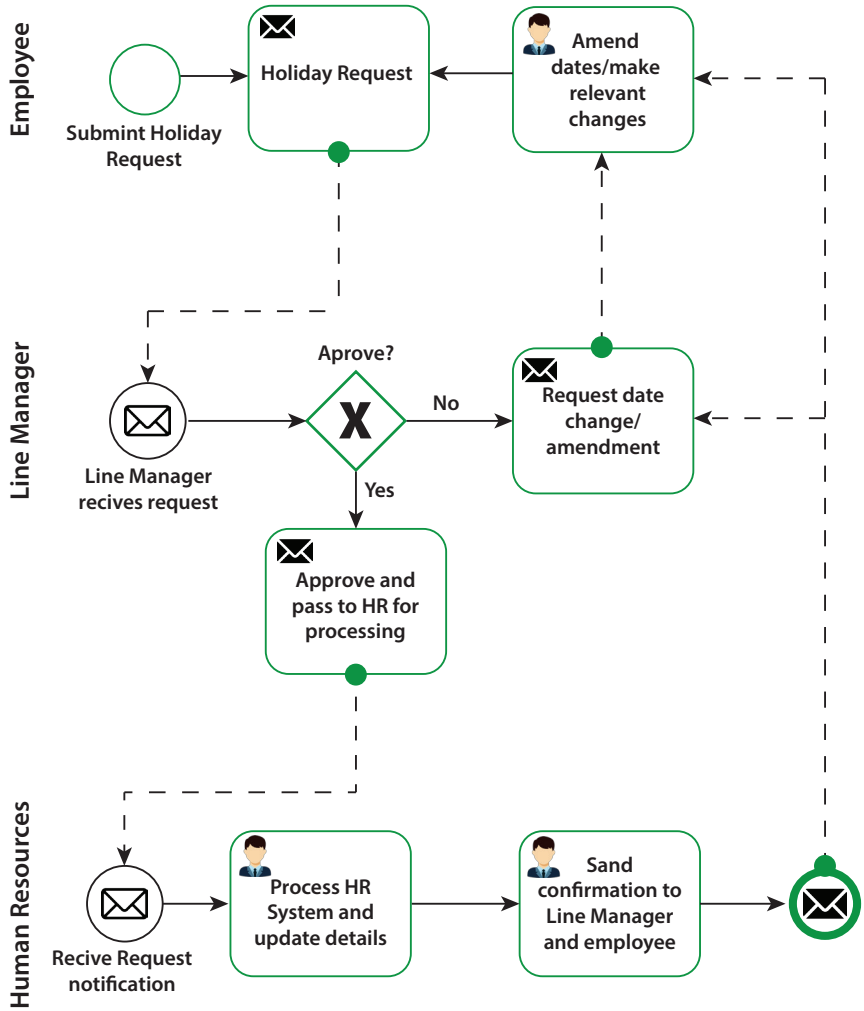
Potrebno je modelovati proces podnošenja i obrade zahteva za godišnji odmor u jednoj kompaniji. U procesu učestvuju tri strane: **Zaposleni, Linijski menadžer** i **Služba ljudskih resursa** (*Human Resource, HR*).

Tok procesa:

1. **Pokretanje procesa:** Zaposleni započinje proces slanjem zvaničnog zahteva za godišnji odmor.
2. **Validacija menadžera:** Kada linijski menadžer primi zahtev, on mora da donese odluku o odobravanju:
3. **Ukoliko zahtev nije odobren:** Menadžer šalje povratnu informaciju zaposlenom sa zahtevom za izmenu datuma ili dopunu zahteva. Zaposleni vrši potrebne izmene i ponovo šalje ažurirani zahtev menadžeru na pregled.
4. **Ukoliko je zahtev odobren:** Menadžer prosleđuje odobrenje službi ljudskih resursa (HR) na dalju obradu.
5. **Administrativna obrada:** HR služba prima obaveštenje o odobrenom odmoru, unosi podatke u HR sistem i ažurira evidenciju zaposlenog.
6. **Završetak procesa:** Nakon ažuriranja sistema, HR šalje finalnu potvrdu o realizaciji odmora i zaposlenom i njegovom linijskom menadžeru, čime se proces završava.

Zahtevi za izradu:

- Koristiti **bazene** (*Pools*) ili **staze** (*Lanes*) za svakog učesnika.
- Pravilno upotrebiti ***Message Events*** (simbole koverte) za komunikaciju između različitih učesnika.



Slika 4.36. BPMN za proces podnošenja i obrade zahteva za godišnji odmor

Zadatak 4.1. Kreirati BPMN procesa refundacije troškova zaposlenih

Kompanija želi da digitalizuje svoj interni proces obrade putnih troškova i drugih izdataka zaposlenih. Proces uključuje interakciju između zaposlenog koji podnosi zahtev, neposrednog rukovodioca i odeljenja finansija.

Opis procesa:

1. **Pokretanje**: Zaposleni popunjava formular za troškove i šalje ga svom linijskom menadžeru na odobrenje.
2. **Pregled menadžera:** Linijski menadžer prima i pregleda formular. Nakon pregleda, on donosi jednu od tri odluke:
   - **Odobravanje**: Ako je sve u redu, menadžer šalje dokumentaciju Odeljenju finansija na dalju obradu.
   - **Odbijanje**: Ako nedostaju informacije, menadžer traži od zaposlenog da unese izmene i ponovo podnese formular.
3. **Obrada u finansijama:** Odeljenje finansija prima odobreni zahtev i vrši finalnu reviziju. Ukoliko je provera uspešna, vrši se isplata troškova i zaposlenom se šalje potvrda o uplati. U slučaju da finansije odbiju zahtev, proces se terminira uz obaveštenje.

Koristiti bazene (*Pools*) ili staze (*Lanes*) za svakog učesnika.

POGLAVLJE 5



MODELOVANJE PODATAKA

Modelovanje podataka

5.1. Uvod u modelovanje podataka
5.2. Analiza modela podataka primenom normalizacije
5.3. Primer modelovanja podataka

Cilj poglavlja:

Pored procesa, ključni deo su i podaci. Nijedan sistem ne može funkcionisati bez podataka. Modelovanje podataka je preduslov za razvoj kvalitetne baze podataka. Modelima podataka se uglavnom bave administratori baza podataka ili bilo koja rola čija odgovornost su podaci. U ovom delu će se objasniti osnovni pojmovi modela podataka, prikazati osnovne notacije i analizirati primenom normalizacije baze podataka. Počeće se od objašnjenja modelovanja relacione baze podataka, kako bi se kasnije razumele i druge strukture baza podataka, kao što su skladište podataka (*Data Warehouse*), jezero podataka (*Data Lake*) kojeg čine ne relacione baze podataka (NoSQL), kombinacija oba (*Data Lakehouse*), Data Fabric i Data Mash.

5. MODELOVANJE PODATAKA

5.1. UVOD U MODELOVANJE PODATAKA

Dobili ste zadatak da razvijete bazu podataka za jedan manji hotel. Odakle ćete početi? Da li ćete odmah otvoriti alat za upravljanje bazama podataka (*Database Management System, DBMS*), kao što su na primer, Microsoft Access ili SQL Server, MySQL i drugi i odmah krenuti sa kreiranjem tabela i njihovih atributa sa primarnim i stranim ključevima ili ćete ipak pre razvoja, razmisliti prvo šta tačno sve treba da bude u bazi podataka, koji tačno podaci su potrebni korisnicima, kako su navikli da pretražuju određene podatke i sl. Kao što smo i spomenuli ranije, softversko inženjerstvo se ne razlikuje puno od drugih inženjerskih disciplina. Kao što, na primer, u automobilskoj industriji prvo prave modele kola, testiraju, poboljšavaju pre nego što se upuste u proces proizvodnje, tako i u softverskom inženjerstvu, prvo se prave modeli koji se testiraju (u slučaju baze podataka, normalizuju), a kasnije koristeći neki od alata i proizvode.

Pre nego što se upustite u modelovanje podataka, potrebno je prikupiti korisničke zahteve i razumeti sistem za koji se ravija baza podataka. Od naručioca posla, dobićete neke opšte informacije o tome šta baza podataka treba da sadrži, eventualno će vam biti data organizaciona struktura preduzeća. Zatim ćete organizovati sastanke sa zaposlenima ili ćete koristiti neku od spomenutih tehnika prikupljanja podataka. Na tim sastancima ideja je da se prikupi dokumentacija, npr. ako je u pitanju hotel, od recepcionera ćete prikupiti podatke koje uzima od gostiju, zatražiti mu blanko rezervaciju, od računovodstva zatražiti blanko fakture, prijemnice i druga dokumenta koje sistem ima potrebe da trajno čuva. Ono što se u organizaciji nalazi po fasciklama, dosijeima, Excel fajlovima, to su uglavnom buduće baze podataka.

Kada ste prikupili potrebna dokumenta i korisničke zahteve, sledeći korak je modelovanje baze podataka. **Modelovanje podataka je tehnika za organizovanje i dokumentovanje strukture podataka sistema**. Modelovanje podataka se ponekad zove modelovanje baze podataka jer se model podataka na kraju implementira kao baza podataka.

Model podataka je pojednostavljeno predstavljanje realnog sistema preko skupa objekata (entiteta), veza između objekata i njihovih atributa. Postoji nekoliko notacija modela objekti-veze (*Entity-Relationship diagram*, ERD), gde su mnogi imenovani prema njihovim tvorcima (npr. Chen, Martin, Bachman, Merise) ili prema objavljenim standardima (npr., IDEF1X). Svi „jezici" modelovanja podataka podržavaju iste fundamentalne koncepte i konstrukcije.

Dakle, nakon što ste prikupili podatke, skiciraćete logički model klasa, gde ćete odrediti objekte, atribute, primarne i strane ključeve, tj. povezati objekte. Model podataka se zatim testira postupkom normalizacije, odnosno primenom normalnih formi. U slučaju da imate dosta iskustva u modelovanju baza podataka, najverovatnije ćete u startu praviti normalizovane baze podataka. Međutim, ukoliko nemate iskustva, onda je potrebno testirati model, prolaskom kroz svih sedam normalnih formi, i to u slučaju da razvijate relacionu bazu podataka (*Relational database*).

Modeli se najčešće crtaju u nekom od specijalizovanih alata za modelovanje podataka, npr. SQL Developer Data Modeler[1], koji omogućavaju i generisanje DDL (*Data Definition Language*) jezika, na osnovu modela podataka. Neki alati vam omogućavaju da selektujete i DBMS sistem gde želite da vam se generiše baza podataka. Taj postupak se zove *forward engineering*. U slučaju da želite da pređete odnosno migrirate na drugi DBMS sistem, možete postupkom *reverse engineering*-a vratiti se na model i selektovati željeni sistem, bez gubitka podataka. Često se ovim postupkom vrše i neke izmene i dopune u bazi podataka. Ovo su samo od nekih prednosti postupka modelovanja podataka.

5.1.1. Osnovni pojmovi modela podataka

Ljudi rado klasifikuju stvari, ne bi li našli sličnosti među njima i prema tome ih grupisali. Stvari sa sličnim svojstvima se grupišu zajedno. Na primer, Petar Petrović i Marija Marjanović imaju svoj broj indeksa, ime, prezime, adresu, telefon, email i sl. i pošto su oboje studenti na istom fakultetu, možemo ih svrstati u objekat Student. Konkretan student se naziva primerkom ili instancom (*instance*) objekta. Na primer, Petar Petrović je instanca objekta Student.

Objekat

Objekat je klasa osoba, mesta, objekata, događaja ili koncepata o kojima treba da prikupljamo i skladištimo podatke. Tipovi objekata se mogu klasifikovati u:

- Objekte tipa „osobe“, kao što su klijenti, zaposleni, studenti i sl.
- Objekti tipa „mesto“: skladišta, zgrade, sobe i sl.
- Objekti tipa „stvari“: proizvod, vozilo, računar i sl.
- Objekti tipa „događaj“: rezervacija, porudžbina, plaćanje, registracija i sl.

Objekte možemo saznati iz:

- Zahteva korisnika – tako što će nam naručioci i budući korisnici baze podataka reći o čemu žele da prikupljaju podatke. Ponekad korisnici mešaju objekte i atribute, te je potrebno razgraničiti objekte.
- Dokumenata – U organizaciji se poslovi evidentiraju u dokumentima bilo u papirnoj ili elektronskoj formi. Potrebno je prikupiti sva blanko dokumenta (npr. Fakture, Rezervacije, Dosijee zaposlenih, Evidencije o klijentima i sl.), kao i pogledati Excel fajlove, izveštaje, e-mailove i druga elektronska dokumenta koja reflektuju poslovanje, a koja će pomoći u identifikovanju objekata.

Grafički prikaz objekta po *Chen*-ovoj notaciji je u obliku pravougaonika. U DBMS sistemima objekat je predstavljen kao tabela (slika 5.1). Termin objekat se inače koristi kod logičkog modelovanja baze podataka, a termin tabela kod fizičkog modela baze, tj. njene implementacije u nekom od konkretnih DBMS sistema. Atributi objekta predstavljaju zaglavlja tabele, tj. nazive kolona, dok se vrednosti atributa popunjavaju po redovima tabele. Redove tabele često nazivamo zapisima ili rekordima (*records*).

1 https://www.oracle.com/database/sqldeveloper/technologies/sql-data-modeler/

Objekat: STUDENT

Broj indeksa	Ime	Prezime	Adresa	PttBr	Broj telefona
122/03	Ana	Antić	Nušićeva 2	11000	011/123456
130/04	Pera	Perić	Nemanjina 3	21000	011/234567
140/02	Lea	Leić	Bul. umetnosti 9	18000	018/456901

Atributi

Instance

Slika 5.1. Objekat student kao tabela u bazi sa popunjenim vrednostima

Atribut

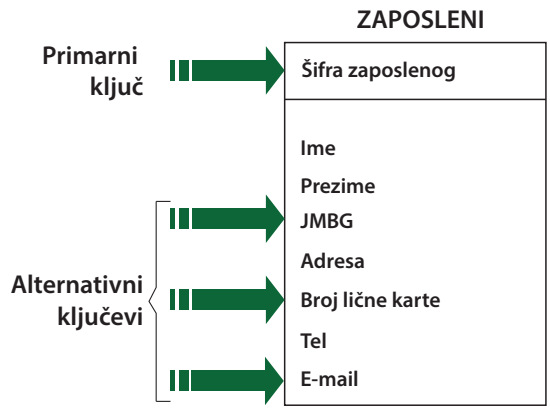
Atributi su podaci koji predstavljaju karakteristike objekta, odnosno to su podaci o kojima želimo da imamo informacije o nekom objektu u bazi podataka. Šta mislite, koji podaci se čuvaju o objektu Student u bazi podataka fakulteta? Koje ste podatke prilikom upisa na fakultet dostavili studentskoj službi fakulteta? Atributi objekta Student su uglavnom broj indeksa, ime studenta, prezime, datum rođenja, adresa stanovanja, kontakt telefon, email i drugi relevantni podaci koji se odnose samo na studenta. Ukoliko bi neko pomislio da su atributi objekta Student i predmeti koje je pohađao i polagao, odnosno naziv predmeta, ocena, ime profesora kod koga je polagao, datum ispita i dr. – to ne bi bilo tačno. Ovi podaci pripadaju objektu Ispit. Da zaključimo, kada određujete atribute, morate razmisliti da li se ti atributi odnose samo na taj konkretan objekat.

Atribute uglavnom saznajemo iz dokumenata, npr. ako pogledate dokument Račun, videćete da se tu nalaze atributi kao što su broj računa, datum računa, iznos računa i dr. Svi ti atributi koji se nalaze na tom dokumentu su potencijalni atributi objekta. Svakako atribute možete saznati i iz zahteva korisnika, na primer, menadžer može reći da o svojim zaposlenima želi da ima sledeće podatke u bazi, kao što su ime, prezime, datum zaposlenja, poziciju, platu i sl.

Primarni ključ

Zamislite sada da u bazi podataka imate oko 10.000 studenata i jedan konkretan student traži da mu se izda uverenje o redovnom studiranju. Kako ćete među 10.000 studenata pronaći ovog konkretnog studenta. Vama je verovatno logično, da svaki student ima svoj jedinstveni broj indeksa te se lako može pronaći pomoću tog atributa. Taj atribut, koji će nam poslužiti za pronalaženje konkretnih objekata, odnosno instanci objekata, nazivamo primarni ključ.

Primarni ključ (*Primary key, PK*) je jedan ili više atributa čija vrednost jedinstveno identifikuje jednu istancu objekta (jedan red u tabeli). U jednom objektu može postojati više različitih atributa koje zadovoljavaju definiciju ključa - svi se nazivaju kandidati za ključ. Na primer, konkretan Zaposleni se može jedinstveno identifikovati preko njegovog JMBG-a, broja lične karte, jedinstvene e-mail adrese ili preko šifre zaposlenog (Slika 5.2).



Slika 5.2. Alternativni ključevi objekta Zaposleni

Primarni ključ je kandidat za ključ koji će se najčešće koristiti da jedinstveno identifikuje dati objekat. Primarni ključ se definiše kao jedan ili više atributa čija vrednost jedinstveno identifikuje jednu instancu objekta. Difoltna vrednost primarnog ključa je *Not Null*, odnosno ključ ne sme da bude prazno polje, jer onda neće moći da jedinstveno identifikuje datu instancu objekta. Svi drugi kandidati za ključ koji nisu izabrani za primarni ključ se zovu **alternativni ključevi**.

Vrednost primarnog ključa obično reflektuje potrebe poslovanja organizacije. Na primer, da li smo za primarni ključ objekta Student, mogli da odredimo JMBG studenta? Svaki student ima svoj jedinstveni JMBG. Međutim šta možemo saznati iz njega, jedino datum rođenja studenta, eventualno u kom gradu je rođen i sl. Postavlja se pitanje, da li je ta informacija relevantna za poslovanje fakulteta? Fakultet će pre odrediti za primarni ključ, atribut broj indeksa, iz koga može da sazna koje školske godine je student upisao fakultet, te stoga može videti po kom akreditovanom studijskom programu studira, zatim pod kojim rednim brojem je upisan, a često, ako fakultet ima više centara u različitim gradovima, iz broja indeksa može se videti u kom centru student studira itd. Svi ovi podaci su relevantni za poslovanje fakulteta, te stoga je broj indeksa određen za primarni ključ objekta Student.

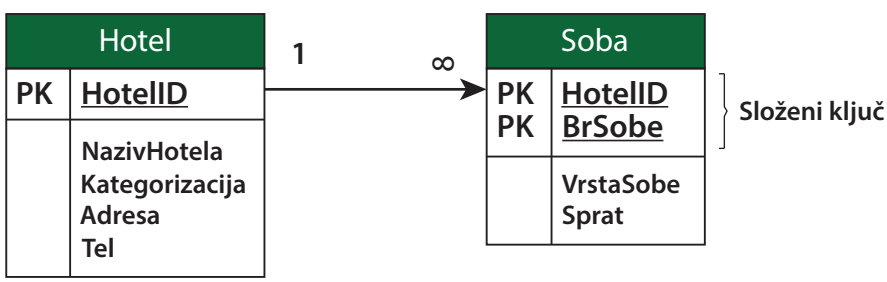
Osnovna namena primarnog ključa je pretraživanje i izvlačenje podataka iz baze podataka.

Da bi neki atribut mogao da bude primarni ključ, on mora da zadovolji minimum sledeće uslove:

- jednoznačnost (jedinstvenost) – ne smeju da postoje dve istance za istim vrednostima svih ključnih atributa; Da li može da se desi da dvoje studenata imaju identičan broj indeksa?
- određenost – postojanje vrednosti u trenutku stvaranja instance; Kako primarni ključ služi za pronalaženje konkretne instance, on ne može biti NULL vrednost (prazno polje), već se obavezno mora uneti konkretna vrednost.

- stabilnost – otpornost na promene tokom vremena; Nepromenljivost kroz vreme je ključna osobina. Zamislite da je je u bazi podataka hotela, za primarni ključ Gosta, stavljeno da je to broj pasoša. Prema programu lojalnosti, na svaku desetu posetu dobija se značajan popust. Međutim, u međuvremenu gost promeni pasoš i prilikom sledeće posete, s obzirom da ima novi broj pasoša, recepcioner, usled nemogućnosti da ga pronađe, kreiraće ga kao novog gosta u bazi podataka, pa samim tim gost će ostati bez popusta.
- raspoloživost – dostupnost svim korisnicima;
- neutralnost – prema značenju vrednosti ključa (samogovoreće šifre).

Nekada je neophodno da više od jednog atributa jedinstveno identifikuje instancu objeta. Ta grupa atributa koja jedinstveno identifikuju objekat se zove složeni ključ. **Prost primarni ključ** podrazumeva da samo jedan atribut čini primarni ključ konkretnog objekta. Na primer, broj računa je primarni ključ objekta Račun. **Složeni primarni ključ** podrazumeva da dva ili više atributa jedinstveno identifikuju konkretan objekat. Na primer, ako razvijate bazu podataka za turističku agenciju koja sarađuje sa hiljade hotela, tada ćete u bazi imati objekat HOTEL i objekat SOBA. Međutim stavili ste da je primarni ključ objekta HOTEL, šifra hotela, a za primarni ključ objekta SOBA, redni broj sobe. Zamislite da klijent traži od turističkog agenta opis sobe broj 5. Da li će agent znati o kojoj tačno sobi je reč? Najverovatnije će tražiti od klijenta i informacije o hotelu. Dakle, nije dovoljno da primarni ključ objekta SOBA bude samo redni broj sobe već i šifra hotela. Ta dva atributa čine složeni primarni ključ objekta SOBA.



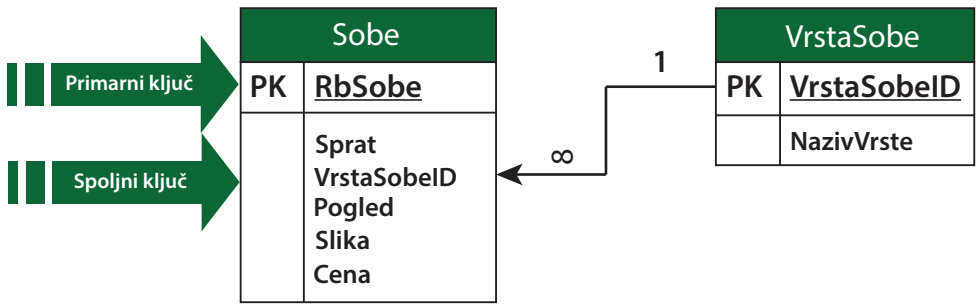
Slika 5.3. Složeni primarni ključ objekta SOBA

Potreba za složenim primarnim ključem dolazi iz činjenice da objekat SOBA potpuno zavisi od objekta HOTEL, odnosno da brisanjem konkretnog hotela iz baze, želimo da se po automatizmu obrišu i sve njegove sobe. Na primer, ako turistička agencija ne želi više da sarađuje sa hotelom Holiday Inn iz Beograda i želi da ga obriše iz baze, najverovatnije će želeti da se automatski obrišu i sve sobe tog hotela.

Odnos potpune zavisnosti između objekata je vrlo retka, uglavnom su objekti međusobno nezavisni. Nezavisnost podrazumeva da životni ciklus jednog objekta ne utiče na životni ciklus drugo objekta. Na primer, brisanjem konkretnog hotela iz baze podataka, ne želimo da brišemo i sve rezervacije koje su do sada bile kreirane sa tim hotelom.

Strani ključ

Objekti ne ekzistiraju samostalno već moraju biti u nekoj relaciji ili vezi sa drugim objektima. **Strani** (spoljni, preneseni) **ključ** (*Foreign Key*, FK) je atribut ili grupa atributa jednog objekta, čija se vrednost koristi za povezivanje sa vrednošću primarnog ključa u drugom objektu.



Slika 5.4. Strani ključ

Na primer, na slici 5.4, vidimo da je u objektu SOBE primarni ključ RbSobe, a u objektu VRSTA SOBE, primarni ključ je VrstaSobeID. Objekat VRSTA SOBE ima samo dva atributa i to VrstaSobeID (npr. STD) i NazivVrste (npr. standardna soba), to bi bio kao neki šifarnik različitih vrsta soba. Ova dva objekta su povezana tako što je primarni ključ objekta VRSTA SOBE dodat u objekat SOBE (na šta pokazuje strelica relacije na slici) – što znači da za konkretnu sobu (npr. soba broj 101) tačno znamo preko stranog ključa (VrstaSobeID) koja je ona vrste.

Verovatno se pitate, a kako da znam u kom objektu treba da stoji strani ključ? U nastavku teksta daje se detaljnije obrazloženje načina povezivanja objekata, kao i neka pravila kod povezivanja i određivanja položaja stranog ključa.

Povezivanje objekata

Povezivanje ili uspostavljanje relacije (veze) između objekata predstavlja najkritičniji deo posla. Kada se povezuju dva objekta, potrebno je sagledati sledeće:

1. Koje objekte treba povezati?
2. Da li su veze između objekata zavisne ili nezavisne?
3. Koji je tip veze između objekata?

1. Koje objekte treba povezati?

Postavite sebi pitanje: Da li je povezivanje nekih objekata bitno za bazu podataka koju pravite?

Na primer, ukoliko pravite bazu podataka turističke agencije, da li ćete povezati objekat ZAPOSLENI direktno sa objektom HOTEL? Međutim, prethodi pitanje, čije zaposlene vi želite da pamtite u vašoj bazi? Da li zaposlene u hotelima ili zaposlene u turističkoj agenciji? S obzirom da pravite bazu podataka za turističku agenciju, vas ne zanimaju svi zaposleni po hotelima, već zaposleni u turističkoj agenciji. Međutim, da li se mogu direktno povezati zaposleni u turističkoj agenciji sa hotelima sa kojima agencija sarađuje. Očigledno je da je zaposleni u relaciji sa objektom HOTEL tek kada, na primer, kreira rezervaciju za taj hotel. Dakle, zaposleni se ne povezuje direktno sa objektom HOTEL, već će između njih biti objekat tipa događaj, a to je u ovom slučaju REZERVACIJA.

Pokušajte sada Vi da odgovorite na sledeća pitanja:

1. Ukoliko razvijate bazu podatakata za jedan fakultet, da li možete povezati direktno objekat STUDENT sa objektom PROFESOR?
2. Ukoliko razvijate bazu podataka za jednu rentacar agenciju, da li možete povezati objekat AUTOMOBIL, koje agencija iznajmljuje, sa objektom KLIJENTI, koji rentiraju kola?
3. Ukoliko razvijate bazu podataka za jednu prodavnicu tehničke opreme, da li možete direktno povezati objekat KUPAC, sa objektom ZAPOSLENI (u toj prodavnici) ili sa objektom PROIZVOD (koji kupac kupuje)?

Odgovori su:

1. S obzirom da se između profesora i studenata uvek odvija neki događaj kao što su predavanja, konsultacije, ispiti, prijemni ispit i sl. stoga, se ova dva objekta ne mogu direktno povezati. Kada postoji izuzeće od ovog pravila, ukoliko, na primer, Fakultet daje popust studentima koji su deca profesora, u tom slučaju veza može da postoji, kako bi taj student zadobio popust.
2. S obzirom da su automobili vlasništvo rentacar agencije, a klijent samo iznajmljuje vozilo, što se vidi preko ugovora koji klijent sklapa sa agencijom, tako da se objekat KLIJENT i objekat AUTOMOBIL, ne mogu direktno povezati. Mogu se povezati preko objekta UGOVOR.

3. Kada kupac kupuje proizvod, a za koji je zaposleni izdao račun, tada se sva tri objekta KUPAC, ZAPOSLENI i PROIZVOD povezuju preko objekta RAČUN. Dakle, ne mogu se direktno povezati već preko objekta tipa transakcija ili događaj, što je u ovom slučaju RAČUN.

Može se zaključiti da postoji neko nepisano pravilo da se objekti tipa „osobe" nikada ne povezuju direktno, kao ni objekti tipa „osobe" i tipa „stvari" (npr. Proizvod), već se skoro uvek povezuju preko nekog objekta tipa „događaj".

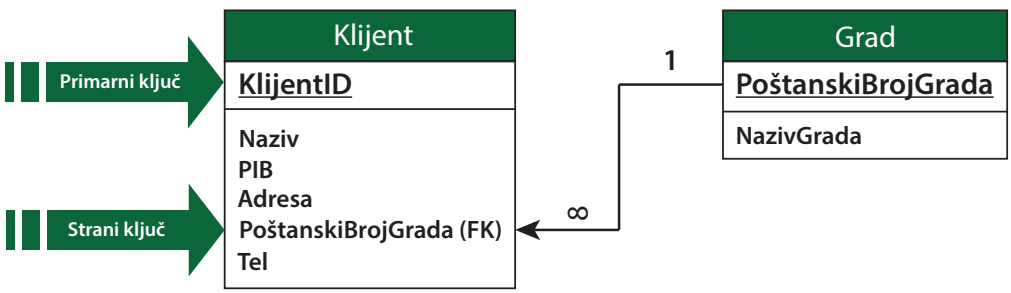
2. Da li su veze između objekata zavisne ili nezavisne?

Zavisnost podrazumeva, da životni vek jednog objekta utiče na životni vek drugog. Ovakva veza se još naziva i identifikaciona (*identifying relationship*) i tipična je kada objekat ne može da postoji bez svog „nadređenog" objekta. Primeri zavisnih veza su:

- Objekat SOBA zavisi od objekta HOTEL - Sobe ne mogu da postoje bez hotela, odnosno ako se obriše konkretan hotel iz baze podataka, automatski će se brisati i sve njegove sobe.
- Objekat PLATNE_KARTICE zavisi od objekta KLIJENT – Ukoliko omogućite registrovanom klijentu da u vašoj aplikaciji može da pamti više svojih platnih kartica, tada ste najverovatnije odvojili platne kartice kao poseban objekat. U poglavlju o Normalizaciji baze podataka, konkretnije u okviru Prve normalne forme, možete pročitati zašto je važno da se da se razdvoje ovi podaci.

Objekat koji zavisi od drugog objekta, naziva se još i slab objekat.

U većini slučajeva, objekti su uglavnom nezavisni jedni od drugih. Na primer, da li je objekat KLIJENT zavisan od objekta GRAD? Ukoliko obrišete klijenta sa sedištem u Beogradu, da li ćete želeti da obrišete i opciju Beograd iz objekta GRAD? Sigurno nećete, jer postoje i drugi klijenti iz Beograda.



Slika 5.5. Nezavisnost između objekata KLIJENT i GRAD

Pokušajte da odgovorite na pitanje: Zašto je važno da znamo da li su objekti zavisni ili nezavisni?

Usled toga što će to odrediti položaj stranog ključa! Naime, ukoliko su objekti zavisni, tada strani ključ postaje deo primarnog ključa. Na primer, ukoliko imate objekat HOTEL (gde je primarni ključ HotelID) i objekat SOBA (gde je RedniBrojSobe primarni ključ), tada će objekat SOBE naslediti primarni ključ objekta HOTEL i biće deo njegovog primarnog ključa. Dakle, objekat SOBE će imati složen primarni ključ, koji se sastoji od dva atributa i to HotelID i RedniBrojSobe (Slika 5.5).

Ukoliko su objekti nezavisni, tada se strani ključ samo doda među ostale (neključne) atribute. Na primer, objekat KLIJENT i objekat GRAD su nezavisni, te će primarni ključ objekta GRAD biti dodat među ostale atribute objekta KLIJENT.

Ukoliko ste se pitali, a što nije obrnuto, tj. da strani ključ stoji u objektu GRAD. U nastavku vam je odgovor.

3. Koji je tip veze između objekata?

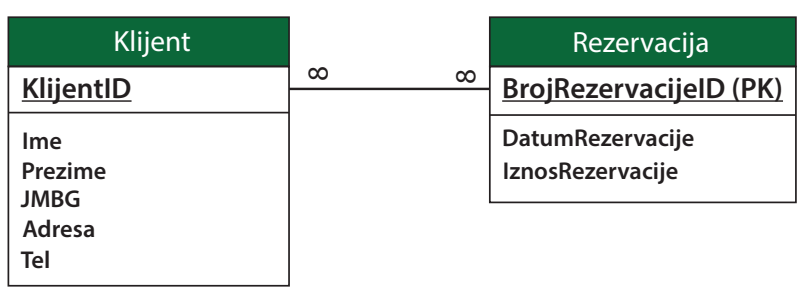
Tipovi veza između objekata mogu biti: jedan prema jedan (1:1), jedan prema više (1:M - inače u zavisnosti od notacija i alata „više" se može predstaviti različitim simbolima, npr. ∞ ili *), više prema jedan (M:1) ili više prema više (M:N).

Jedna od bitnih karakteristika veza između objekata je kardinalnost (tip veze). Kardinalnost definiše minimalni i maksimalni broj događaja jednog objekta koji se nalazi u konkretnoj relaciji sa drugim objektom. Pošto su sve relacije dvosmerne, kardinalnost se mora definisati za oba smera. Na primeru, sa slike 5.4, jedna konkretna soba (npr. soba broj 4) može biti samo jedne konkretne vrste (npr. jednokrevetna), dok jedne vrste (npr. dvokrevetnih) može biti više soba u jednom hotelu. Dakle jedan (1) je na strani objekta VRSTA SOBE, dok je više (M) na strani objekta SOBE. Ovo je tipična veza jedan prema više (1:M).

Zašto je tip veze bitan? Usled toga što određuje položaj stranog ključa. Gde god je više, kao u prethodnom primeru, pošto je „više" na strani SOBE, tu će biti dodat strani ključ. U ovom slučaju u okviru objekta SOBE dodat je primarni ključ objekta VRSTA SOBE, koji se zove vrstaSobeID i on predstavlja strani ključ.

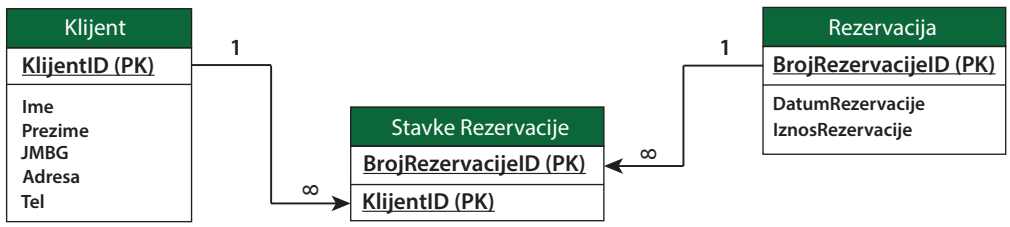
Međutim, šta se dešava u slučaju da je tip veze „više prema više", gde će tada stajati strani ključ?

Na slici 5.6 se prikazuje primer, gde Klijent u svom životnom veku može da napravi više rezervacija, dok kod konkretne rezervacije se ispisuju svi klijenti, na primer cela porodica.



Slika 5.6. Problem više prema više

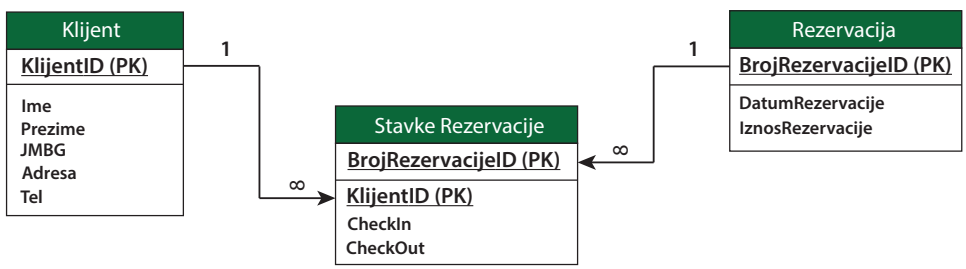
Veze tipa "više prema više" nisu podržane u relacionim bazama podataka, te se ovaj problem rešava uvođenjem posrednog objekta (međuobjekta) koji sadrži primarne ključeve povezanih objekata kao strane ključeve. U prethodnom primeru, kako jedan klijent može da kreira više rezervacija, a na jednoj konkretnoj rezervaciji može da se nalazi više klijenata, tada se kreira međuobjekat koji ćemo nazvati STAVKE_REZERVACIJE i on će naslediti primarne ključeve objekata između kojih je nastao i to BrojRezervacijeID i KlijentID, koji će postati njegov složeni primarni ključ.



Slika 5.7. Rešenje problema tipa veze „više prema više"

Ukoliko je porodica rezervisala dve sobe, a dolaziće u različitim vremenima, gde bi u tom slučaju postavili atribute kao što su prijavljivanje (*check-in*) **i odjavljivanje** (*check-out*)?

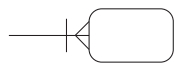
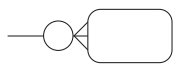
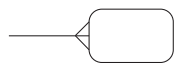
Međuobjekat se može iskoristiti za dodavanje novih atributa, u prethodnom primeru, *check-in* i *check-out* će stajati u objektu STAVKE_REZERVACIJE.



Slika 5.8. Dodavanje atributa u međuobjekat

Problem određivanja stranih ključeva će biti detaljnije obrazložen kroz postupak normalizacije.

Grafički prikaz kardinalnosti zavisi od notacije koja se koristi u modelu objekti-veza (*Entity-Relationship*, ER). Na slici 5.9 je prikazana kardinalnost prema Martinovoj notaciji ER modela.

Interpretacija kardinalnosti	Minimum	Maksimum	Grafička notacija
Tačno jedan	1	1	
Nula ili jedan	0	1	
Jedan ili više	1	>1	
Nula, jedan ili više	0	>1	
Više od jedan	>1	>1	

Slika 5.9. Martinova notacija kardinalnosti (prilagođeno Coronel & Morris, 2015)

Generalizacija i specijalizacija

Generalizacija predstavlja važan koncept u modelovanju baza podataka. To je tehnika kojom se objekti sa zajedničkim atributima grupišu (generalizuju) u jedan opštiji objekat, nazvan nadtip (*supertype*). Inverzni postupak se naziva specijalizacija gde se za nadtip kreriraju podtipi (*subtypes*) ili specijalizovani objekti koji nasleđuju zajedničke atribute, ali mogu imati i svoje specifične.

Na primer, ukoliko razvijate bazu podataka za jednu stomatološku ordinaciju, koja želi da zna ko su mu klijenti, da li su to studenti, penzioneri, deca ili zaposleni, kako bi im dali adekvatan popust. U ovom slučaju vi ćete kreirati posebne objekte Student, Zaposleni, Dete i Penzioner, ali ćete isto tako kreirati i generalizovani objekat Klijenti u okviru koga će stajati svi njihovi zajednički atributi, kao što su na primer ime, prezime, datum rođenja, adresa i telefon. Međutim, kod specijalizovanih objekata će stajati njima svojstveni atributi (koji nisu zajednički), kao na primer, objekat Student će imati naziv fakulteta koji studira, Penzioner će imati godinu penzionisanja i sl.

Kod specijalizacije, podtipovi nasleđuju zajedničke atribute iz nadtipa, uz dodatak svojih specifičnih atributa. Veza između nadtipa i podtipova često se predstavlja kao „Is-a" odnos. Kardinalnost ove veze može varirati: u slučaju potpune i disjunktne[2] specijalizacije, svaki objekat nadtipa pripada tačno jednom podtipu, pa veza može biti 1:1. Međutim, u delimičnoj ili preklapajućoj specijalizaciji, neki entiteti nadtipa mogu pripadati nijednom ili više podtipova, pa tada veza nije strogo 1:1.

2 Disjunktna specijalizacija je vrsta specijalizacije u modelovanju baza podataka gde se objekti nadtipa mogu pojaviti u najviše jednom od podtipova. Drugim rečima, objekat iz nadtipa ne može istovremeno biti član dva ili više podtipova. Ova vrsta specijalizacije je „isključiva" — svaki objekat pripada tačno jednoj (i*li nijednoj, ako nije potpuna specijalizacija*) podgrupi.

5.1.2. Postupak modelovanja podataka

Nakon što su prikupljeni relevantni podaci i definisani korisnički zahtevi, započinje proces modelovanja podataka. U ovoj fazi oblikuje se struktura podataka koja odražava poslovne potrebe, ali i omogućava efikasno upravljanje podacima.

U početnoj fazi modelovanja kreira se **konceptualni model podataka**, koji identifikuje osnovne objekte i njihove međusobne veze, ali ne ulazi u tehničke detalje poput atributa ili ključeva. Ovakav pristup omogućava da se prvo razume šta sistem treba da beleži, bez razmatranja kako će to biti implementiran.

Sledeća faza je izrada **logičkog modela podataka**, koji dodatno razrađuje prethodnu strukturu dodavanjem atributa, primarnih i stranih ključeva, kao i definisanjem relacija među objektima. Ovaj model je nezavisan od konkretne baze podataka, ali je bliži njenoj implementaciji.

Fizički model podataka podrazumeva prilagođavanje logičkog modela konkretnom DBMS (*Database Management System*) sistemu (npr. Oracle, MySQL, PostgreSQL...). Uključuje optimizaciju performansi, indeksiranje, načine skladištenja itd.

Modelovanje podataka je proces analize i dizajna strukture podataka koji se koristi u informacionom sistemu. Koraci u postupku modelovanja podataka su:

1. Prikupljanje podataka i korisničkih zahteva (*Requirements Gathering*) - Razumevanje poslovnih procesa i informacija koje sistem mora da podrži.
2. Identifikacija **objekata** - Koji su to osnovni poslovni objekti o kojima sistem treba da čuva podatke?
3. Identifikacija **atributa** - Koje informacije se vezuju za svaki objekat?
4. Određivanje **primarnih ključeva** - Kako se svaki objekat jednoznačno identifikuje?
5. Prepoznavanje **veza među objektima** (*Relationship Identification*) - Kako su objekti međusobno povezani (1:1, 1:N, M:N)?
6. Definisanje **ograničenja i poslovnih pravila** - npr, obaveznost veze, ograničenja domena atributa, itd.
7. **Normalizacija** – podrazumeva primenu normalnih formi (1NF, 2NF, 3NF...) radi uklanjanja redundanse i povećanja koherentnosti.

Kada kažemo da je model „koherentan", to podrazumeva:

- Doslednost – podaci se unose i čuvaju na način koji sprečava kontradiktorne informacije. Npr. ne može jedan korisnik imati dve različite prijavljene adrese stanovanja u isto vreme ako je predviđen samo jedan atribut za adresu.
- Jasna struktura odnosa – svi objekti i njihovi odnosi su logično povezani, bez nejasnoća ili nedovoljno definisanih veza.
- Organizacija prema stvarnim pravilima poslovanja – podaci „imaju smisla" jer model odražava pravila stvarnog sveta (tzv. semantička ispravnost).
- Minimizacija redundanse – isti atribut ne bi trebalo da se nalazi u više različitih objekata, jer ako neko promeni njegovu vrednost na jednom mesto, pitanje je da li nova vrednost upisana i na drugim mestima, tj. u okviru objekata gde se taj isti atribut nalazi. Dakle, jedan atribut treba da se nalazi samo na jednom mestu!

Upravo normalizacija doprinosi koherentnosti modela jer omogućava da:

- svaka informacija bude zabeležena na jedinstvenom mestu
- veze među objektima budu logične i funkcionalno zavisne,
- model bude lakši za održavanje, proširenje i razumevanje.

Kada je model podataka testiran primenom normalizacije, **finalni rezultat modelovanja je izrađen ER dijagram**.

Poslednji korak je prevođenje logičkog modela podataka u fizički model (šema baze podataka) u zavisnosti od izabranog sistema za upravljanje bazom podataka (*Database Management System, DBMS*).

5.2. ANALIZA MODELA PODATAKA PRIMENOM NORMALIZACIJE

Ukoliko model podataka efektivno izražava poslovne zahteve za podacima, to ujedno ne znači da je takav model i dobar model. Njegova struktura može biti takva da umanjuje fleksibilnost, proširljivost ili stvara nepotrebnu redundantnost. Tehnika koja se koristi za poboljšanje modela podataka u fazi pripreme za fizički dizajn baze podataka se naziva analiza podataka. **Analiza podataka** je proces koji priprema model podataka za implementaciju jednostavne, neredundantne, fleksibilne i adaptibilne baze podataka. Ova tehnika se naziva normalizacija.

Normalizacija je tehnika analize podataka koja organizuje, grupiše atribute podataka tako da formiraju neredundantne, stabilne, fleksibilne i prilagodljive objekte. Ovde nećemo prikazati matematički pristup koji obuhvata relacionu algebru i relacioni račun, već ćemo se osvrnuti samo na primenu normalnih formi. Pravilno projektovana baza podataka mora imati strukturu koja osigurava da u radu sa njom ne može doći do neželjenih anomalija, kao što su na primer, uništavanje određenih podataka ili neusklađenost između memorisanih podataka kao posledica ažuriranja baze podataka.

Tvorac normalizacije je Codd koji je definisao postupak koji dovodi projektanta relacione baze podataka do željenog rezultata. Codd je definisao tri nivoa normalizacije, da bi kasnije bilo dokazano da u nekim izuzetnim primerima, relacije koje su u trećoj normalnoj formi, još uvek pokazuju izvesne anomalije. Na treću normalnu formu nadovezuje se Boyce-Codd-ova normalna forma koja se bavi problemima preklapajućih kandidata za ključ. Probleme višeznačne zavisnosti rešava Fagin definišući četvrtu normalnu formu (4NF). Kasnije je formulisana i peta normalna forma koja definiše zavisnost spajanja relacija. Da bi se stalo u definisanju novih normalnih formi, Fagin je 1981. godine dao najopštiju definiciju normalne forme ključeva i domena.

Primena normalizacije predstavlja transformaciju početnog objekta u jednu ili više korektnih objekata (relacija) sa ciljem da eliminiše redundansu u atributima i sa time omogući lakše održavanje integriteta podataka i jednostavniju manipulaciju.

Greške početnika

Na osnovu prikupljenih zahteva i dokumentacije, početnici u razvoju baze podataka često prave neke osnovne greške, kao što su:

- da se svi atributi sa konkretnog dokumenta stave u istoimeni objekat. Na primer, ukoliko je potrebno kreirati rezervaciju u bazi, neko će sve elemente sa dokumenta Rezervacija, kao što su, broj rezervacije, datum, ime klijenta, prezime klijenta, broj sobe, vrsta aranžmana, iznos rezervacije i ostalo, staviti u objekat REZERVACIJA.
- pogrešno određenje stranog ključa, odnosno da se postavi u pogrešan objekat.
- usled ne razumevanja da li je objekat zavisan ili ne, pogrešno se odrede prostili ili složeni primarni kljjučevi.
- nepotrebno kreiranje novog atributa kao primarni ključ iako već postoji atribut, koji ima sve karakteristike primarnog ključa, a sa koji su korisnici već navikli da pretražuju podatke u bazi.
- kao i mnoge druge greške koje će se spomenuti u postupku normalizacije, a koje dovode do anomalija u održavanju baze podataka.

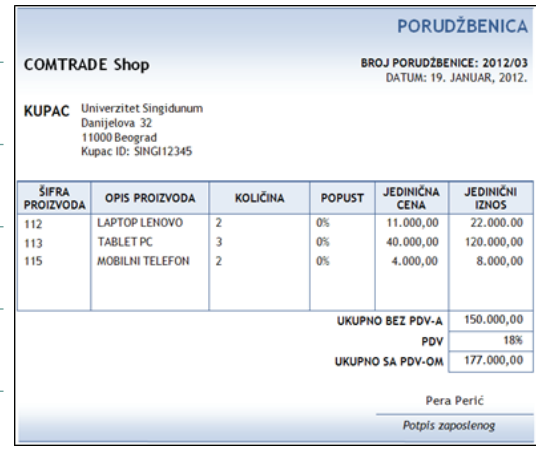
U ovakvim situacijama, neopohodno je da se primene makar prve tri normalne forme.

Ukoliko želite da testirate vaše predznanje o modelovanju baze podataka uradite sledeću vežbu.

Vežba 5.1. Na osnovu sledećeg dokumenta, odgovorite na pitanja i kreirajte model podataka.

Pitanja:

1. Koliko objekata uočavate na ovoj porudžbenici?
2. Da li možete da navedete sve atribute objekta PORUDŽBENICA?
3. U kom objektu se nalazi atribut Količina (poručena količina proizvoda)?
4. U kom objektu se nalazi atribut Jedinčni_iznos?
5. Da li je PDV objekat ili atribut, ukoliko znamo da je fiksan za sve račune?
6. U kom objektu se nalazi Ime_zaposlenog?



PORUDŽBENICA

COMTRADE Shop

BROJ PORUDŽBENICE: 2012/03
DATUM: 19. JANUAR, 2012.

KUPAC Univerzitet Singidunum
Danijelova 32
11000 Beograd
Kupac ID: SINGI12345

ŠIFRA PROIZVODA	OPIS PROIZVODA	KOLIČINA	POPUST	JEDINIČNA CENA	JEDINIČNI IZNOS
112	LAPTOP LENOVO	2	0%	11.000,00	22.000.00
113	TABLET PC	3	0%	40.000,00	120.000,00
115	MOBILNI TELEFON	2	0%	4.000,00	8.000,00

UKUPNO BEZ PDV-A	150.000,00
PDV	18%
UKUPNO SA PDV-OM	177.000,00

Pera Perić

Potpis zaposlenog

Ukoliko niste sigurni da li ste dobro uradili ovu vežbu, pokušajte da primenite sledeće normalne forme.

5.2.1. Prva normalna forma

Prva normalna forma (1NF) **proverava da li za jednu vrednost primarnog ključa, postoji više vrednosti drugih atributa**. Ukoliko postoje, te atribute treba izbaciti iz tog objekta i postaviti ih u novi. U tom novom objektu treba dopisati primarni ključ početnog objekta, odakle su ovi atributi preuzeti.

Na primer, u objektu STUDENT nalaze se sledeći atributi: broj indeksa, ime, prezime, adresa, šifra predmeta, naziv predmeta i ocena. Za primarni ključ je određen broj indeksa. Da li za konkretnu vrednost primarnog ključa, na primer 2020/111222, postoji više vrednosti drugih atributa?

Objekat i atribute ćemo predstaviti u sledećem obliku, dok ćemo primarni ključ podvući:

STUDENT (BrIndeksa, Ime, Prezime, Adresa, ŠifraPredmeta, NazivPredmet, Ocena)

Ukoliko ne možete da uočite da li postoje, popunićemo ovaj objekat sa nekim konkretnim vrednostima:

STUDENT

BrIndeksa	Ime	Prezime	Adresa	ŠifraPredmeta	NazivPredmeta	Ocena
2020/111222	Ana	Petrović	Makedonska 2	101	Matematika	8
				131	Programiranje	10
				…	…	…

Da li sada uočavate, da li za konkretnu vrednost primarnog ključa (npr. broj indeksa 2020/111222), postoji više vrednosti drugih atributa?

Očigledno da studentkinja Ana Petrović, sa brojem indeksa 2020/111222, može imati više položenih predmeta, stoga u bazi podataka se može pojaviti više vrednosti sledećih atributa: šifra predmeta, naziv predmeta i ocena. Prema pravilima 1NF te atribute treba izbaciti iz objekta STUDENT i postaviti ih u neki novi objekat, koji ćemo u ovom slučaju imenovati kao ISPIT.

Možda ćete pomisliti da je rešenje ukoliko se ponove podaci o studentkinji onoliko puta koliko ima položenih predmeta, kao na primer:

STUDENT

BrIndeksa	Ime	Prezime	Adresa	ŠifraPredmeta	NazivPredmeta	Ocena
2020/111222	Ana	Petrović	Makedonska 2	101	Matematika	8
2020/111222	Ana	Petrović	Makedonska 2	131	Programiranje	10
2020/111222	Ana	Petrović	Makedonska 2	…	…	…

Pokušajte da odgovorite na sledeća pitanja:

- Ukoliko studentska služba želi da doda novi predmet u bazu podataka, da li mora da čeka da ga neki student položi?
- Ukoliko Ana promeni adresu stanovanja, na koliko mesta je potrebno uneti tu izmenu?
- Ukoliko se Ana ispiše sa fakulteta i služba je obriše iz baze, da li će se izgubiti i podaci o predmetu (šifra predmeta i naziv) koji je samo ona položila?

Ukoliko ste tačno odgovorili na ova pitanja, sigurno ste i sami uočili anomalije prilikom održavanja ovakve baze podataka, a to su:

- Studentska služba neće moći da doda informacije o novom predmetu, sve dok ga neki student ne položi.
- Ukoliko student promeni adresu, to se mora učiniti na onoliko mesta koliko je predmeta student položio.
- Ukoliko se iz baze obriše student koji je jedini položio određeni predmet, gube se sve informacije o tom predmetu.

Osnovne operacije održavanja baze podataka obuhvataju:

- dodavanje nove instance (n-torke[3]) u relaciju,
- brisanje postojeće n-torke iz relacije,
- izmenu (ažuriranje) vrednosti jednog ili više atributa u okviru relacije.

Problemi pri izvođenju ovih operacija kao što su ponavljanje ovih operacija više puta, izbacivanje jednog logičkog skupa podataka koji dovodi do neželjenog izbacivanja drugih podataka ili logičko dodavanje se ne može izvršiti, nazivaju se **anomalije u održavanju baze podataka**.

Dakle, **relacija je u prvoj normalnoj formi** (1NF) ako su **sve vrednosti njenih atributa atomske**, odnosno **nedeljive**. To znači da za **svaku vrednost primarnog ključa** postoji **tačno jedna vrednost za svaki drugi atribut**.

U prethodnom primeru mogli smo uočiti da u objektu (relaciji) STUDENT, za konkretnu vrednost atributa BrIndeksa (koji predstavlja primarni ključ), postoje višestruke vrednosti atributa: ŠifraPredmeta, NazivPredmeta i Ocena. Zbog toga relacija ne zadovoljava uslove prve normalne forme. Početna relacija izgleda ovako:

STUDENT (<u>BrIndeksa</u>, Ime, Prezime, Adresa, ŠifraPredmeta, NazivPredmeta, Ocena)

Logično je da jedan student može položiti više predmeta, pa se podaci o ispitima prirodno ponavljaju za istog studenta, čime dolazi do višestrukih vrednosti u atributima, što narušava 1NF.

3 n-torka (engl. *n-tupl*) predstavlja jedan red (*instancu*) u tabeli, npr. 2020/111222, Ana, Petrović, Makedonska 2, predstavlja četvoročlanu torku (*4-torku*), jer sadrži četiri vrednosti.

Kako rešiti ovaj problem?

Da bi se relacija dovela u prvu normalnu formu, potrebno je izdvojiti atribute sa višestrukim vrednostima u posebnu relaciju. U ovom slučaju, formira se nova relacija koju ćemo nazvati ISPIT, a koja inicijalno sadrži atribute: ŠifraPredmeta, NazivPredmeta i Ocena. Međutim, kako bi se očuvala veza sa studentom i obezbedila identifikacija svake pojedinačne instance polaganja ispita, neophodno je u relaciju ISPIT uključiti i primarni ključ iz početne relacije STUDENT, tj. BrIndeksa. Na taj način, nova relacija ISPIT dobija složeni primarni ključ koji se sastoji od atributa BrIndeksa i ŠifraPredmeta, čime se osigurava jedinstvenost svakog ispita koji je student polagao.

Rezultat normalizacije u 1NF:

STUDENT (BrIndeksa, Ime, Prezime, Adresa)

ISPIT (BrIndeksa, ŠifraPredmeta, NazivPredmeta, Ocena)

Primena 1NF na vežbu 5.1.

Pretpostavimo da su svi atributi iz dokumenta smešteni u jedan objekat PORUDZBENICA, koji sadrži sledeće atribute:

PORUDZBENICA (BrojPorudzbenice, DatumPorudzbenice, NazivKupca, AdresaKupca, PostanskiBroj, NazivGrada, KupacID, SifraProizvoda, OpisProizvoda, PoručenaKolicina, PopustNaKol, JedinicnaCenaProizvoda, JedinicniIznos, UkupnoBezPDVa, PDV, UkupnoSaPDVom, ImeZaposlenog, PrezimeZaposlenog, ZaposleniID)

Da li uočavate višestruke vrednosti određenih atributa, za konkretnu vrednost broja porudžbenice, kao primarnog ključa?

Ukoliko pogledate i sam dokument porudžbenice, videćete da za jednu konkretnu porudžbenicu, broj 2012/03, postoji više poručenih proizvoda, odnosno postoji više redova sa različitim ŠifraProizvoda, OpisProizvoda, PoručenaKoličina itd. Takođe, atributi kao što su JediničniIznos, PopustNaKol, JediničnaCenaProizvoda se odnose na svaki pojedinačan proizvod, a ne na celu porudžbinu. Dakle, za jednu vrednost BrojPorudzbenice, postoje višestruke vrednosti za proizvode, što krši 1NF.

Rešenje je da se izbace ti atributi iz objekta Porudžbenica i postave u novi objekat, koji ćemo nazvati STAVKE_PORUDZBNICE. U taj novokreirani objekat, dopisaćemo i primarni ključ početne tabele, kako se ne bi izgubila početna informacija.

PORUDŽBENICA (BrojPorudzbenice, DatumPorudzbenice, NazivKupca, AdresaKupca, PostanskiBroj, NazivGrada, KupacID, ~~SifraProizvoda~~, ~~OpisProizvoda~~, ~~PorucenaKolicina~~, ~~PopustNaKol~~, ~~JedinicnaCenaProizvoda~~, ~~JedinicniIznos~~, UkupnoBezPDVa, PDV, UkupnoSaPDVom, ImeZaposlenog, PrezimeZaposlenog, ZaposleniID)

Rešenje zadatka je:

PORUDZBENICA (BrojPorudzbenice, DatumPorudzbenice, NazivKupca, AdresaKupca, PostanskiBroj, NazivGrada, KupacID, UkupnoBezPDVa, PDV, UkupnoSaPDVom, ImeZaposlenog, PrezimeZaposlenog, ZaposleniID)

STAVKE_PORUDZBENICE (BrojPorudzbenice, SifraProizvoda, OpisProizvoda, Porucena Kolicina, PopustNaKol, JedinicnaCenaProizvoda, JedinicniIznos)

Zadatak 5.1. Primeniti 1NF na sledeću relaciju:

FUDBALSKI_SUDIJA (SudijaID, ImeSudije, AdresaSud, PoštanskiBroj, NazivGrada, email, tel, UtakmicaID, DatumUtakmice, TimDomaćin, TimGost, Rezultat)

5.2.2. Druga normalna forma

Relacija je u **Drugoj normalnoj formi** (2NF) **ako i samo ako je prethodno u Prvoj normalnoj formi** (1NF) **i svi njeni neključni atributi su potpuno funkcionalno zavisni od celokupnog primarnog ključa**. Drugim rečima, nijedan neključni atribut ne sme zavisiti samo od dela složenog ključa, već mora zavisiti od celog ključa (Elmasri & Navathe, 2016).

Potpuna funkcionalna zavisnost (*full functional dependency*) znači da se vrednost neključnog atributa može jedinstveno odrediti samo kada su poznate sve komponente složenog ključa. Ako atribut zavisi samo od jednog dela složenog ključa, tada postoji nepotpuna funkcionalna zavisnost, što krši 2NF (Coronel & Morris, 2015).

Postupak normalizacije do 2NF se obično vrši dekompozicijom relacije, pri čemu se:

1. u jednoj relaciji zadržava primarni ključ i svi atributi koji su potpuno funkcionalno zavisni od njega,
2. dok se u drugim relacijama izdvajaju oni atributi koji su nepotpuno funkcionalno zavisni, zajedno sa atributima od kojih zavise (Date, 2012).

Na primer, nakon primene 1NF, dobijene su sledeće relacije:

STUDENT (BrIndeksa, Ime, Prezime, Adresa)

ISPIT (BrIndeksa, ŠifraPredmeta, NazivPredmeta, Ocena)

Relacija ISPIT ima složeni primarni ključ: (BrIndeksa, ŠifraPredmeta). Da bismo utvrdili da li je relacija u 2NF, moramo proveriti da li postoje neključni atributi koji zavise samo od dela ključa.

Posmatrajmo najpre atribut NazivPredmeta. Ako znamo samo BrIndeksa, ne možemo odrediti naziv predmeta — što znači da NazivPredmeta ne zavisi od BrIndeksa. Međutim, ako znamo ŠifruPredmeta, možemo odrediti i naziv, što implicira da NazivPredmeta funkcionalno zavisi samo od ŠifrePredmeta, tj. od dela složenog ključa. Ova zavisnost krši 2NF. Usled toga, atribut NazivPredmeta se izdvaja u novu relaciju:

PREDMET (ŠifraPredmeta, NazivPredmeta)

A relacija ISPIT se sada svodi na:

ISPIT (BrIndeksa, ŠifraPredmeta, Ocena)

Preostaje da analiziramo da li je atribut Ocena potpuno funkcionalno zavisan od primarnog ključa. Ako znamo samo BrIndeksa ili samo ŠifruPredmeta, ne možemo jednoznačno odrediti ocenu. Ali ako su poznati oba atributa, BrIndeksa i ŠifraPredmeta, tada je moguće tačno odrediti ocenu. Dakle, Ocena je potpuno funkcionalno zavisna od celokupnog primarnog ključa i ostaje u relaciji ISPIT.

Konačne relacije, nakon primene 2NF:

ISPIT (BrIndeksa, ŠifraPredmeta, Ocena)

PREDMET (ŠifraPredmeta, NazivPredmeta)

STUDENT (BrIndeksa, Ime, Prezime, Adresa)

Relacija STUDENT već zadovoljava 2NF, jer ima prost ključ (BrIndeksa), pa svi neključni atributi zavise upravo od celog ključa.

Primena 2NF na vežbu 5.1.

Početni objekat PORUDZBENICA je, primenom Prve normalne forme (1NF), dekomponovan u sledeće relacije:

PORUDZBENICA (BrojPorudzbenice, DatumPorudzbenice, NazivKupca, AdresaKupca, PostanskiBroj, NazivGrada, KupacID, UkupnoBezPDVa, PDV, UkupnoSaPDVom, ImeZaposlenog, PrezimeZaposlenog, ZaposleniID)

STAVKE_PORUDZBENICE (BrojPorudzbenice, SifraProizvoda, OpisProizvoda, Porucena Kolicina, PopustNaKol, JedinicnaCenaProizvoda, JedinicniIznos)

U relaciji STAVKE_PORUDZBENICE, primarni ključ je složen i sastoji se od dva atributa BrojPorudzbenice i SifraProizvoda. S obzirom na to, možemo primeniti Drugu normalnu formu (2NF) analizom funkcionalnih zavisnosti neključnih atributa:

- OpisProizvoda zavisi isključivo od atributa SifraProizvoda, jer se opis svakog proizvoda može odrediti na osnovu njegove šifre, nezavisno od broja porudžbenice. Dakle, ovaj atribut zavisi samo od dela primarnog ključa, što predstavlja nepotpunu funkcionalnu zavisnost i krši 2NF.
- JedinicnaCenaProizvoda takođe zavisi samo od atributa SifraProizvoda, jer svaki proizvod ima fiksnu jediničnu cenu, nezavisno od porudžbine. Takođe i ovde je u pitanju nepotpuna funkcionalna zavisnost.
- PorucenaKolicina zavisi od celog primarnog ključa, tj. jedinstveno je određena kombinacijom broja porudžbenice i šifre proizvoda, što znači da je ovaj atribut potpuno funkcionalno zavisan od složenog ključa.

- PopustNaKol je zavisan od poručene količine, odnosno, posredno zavisi i od celog primarnog ključa, jer je količina uslovljena konkretnom stavkom porudžbenice.
- JedinicniIznos predstavlja izračunatu vrednost kao rezultat izraza: PoručenaKoličina × JediničnaCenaProizvoda. Kao takav, ovaj atribut potpuno funkcionalno zavisi od celog složenog ključa, iako je izveden.

Atributi OpisProizvoda i JedinicnaCenaProizvoda su nepotpuno funkcionalno zavisni od atributa SifraProizvoda, tj. samo od dela složenog ključa. Da bi relacija STAVKE_PORUDZBENICE zadovoljila 2NF, ovi atributi se izdvajaju u zasebnu relaciju PROIZVOD.

Nove relacije su:

PORUDZBENICA (BrojPorudzbenice, DatumPorudzbenice, NazivKupca, AdresaKupca, PostanskiBroj, NazivGrada, KupacID, UkupnoBezPDVa, PDV, UkupnoSaPDVom, ImeZaposlenog, PrezimeZaposlenog, ZaposleniID)

STAVKE_PORUDZBENICE (BrojPorudzbenice, SifraProizvoda, PoručenaKolicina, PopustNaKol, JedinicniIznos)

PROIZVOD (SifraProizvoda, OpisProizvoda, JedinicnaCenaProizvoda)

Na ovaj način, svaka relacija ispunjava uslove druge normalne forme, budući da svi neključni atributi zavise isključivo i potpuno od celog primarnog ključa u svojim relacijama.

Zadatak 5.2. Primeniti 2NF na sledeću relaciju:

SOBA (HotelID, RbSobe, NazivHotela, AdresaHotela, KategorizacijaHotela, VrstaSobe, Pogled)

5.2.3. Treća normalna forma

Relacija R je u **Trećoj normalnoj formi** (3NF) ako i samo ako je prethodno u **Drugoj normalnoj formi** (2NF) i ako svi njeni **neključni atributi zavise netranzitivno** od **primarnog ključa** (Elmasri & Navathe, 2016). Drugim rečima, **ne sme postojati tranzitivna funkcionalna zavisnost** u kojoj jedan **neključni atribut zavisi od drugog neključnog atributa**, koji je pak zavisan od primarnog ključa.

Tranzitivna funkcionalna zavisnost se može formalno izraziti na sledeći način: Ako imamo relaciju sa primarnim ključem *K* i atributima *A* i *B*, pri čemu važi:

- K → A (K funkcionalno određuje A)
- A → B (A funkcionalno određuje B), onda je B tranzitivno zavisan od K, što krši uslove 3NF, jer *B* zavisi od *K* preko atributa *A*, a ne direktno.

Na primer, razmotrimo sledeću relaciju:

STUDENT (BrojIndeksa, Ime, Prezime, Adresa, PostanskiBr, NazivGrada)

Pretpostavimo da je BrojIndeksa primarni ključ, dok su ostali atributi neključni. Da bismo proverili da li relacija zadovoljava uslove 3NF, analizirajmo funkcionalne zavisnosti:

BrojIndeksa → Ime, Prezime, Adresa, PostanskiBr

PostanskiBr → NazivGrada

U ovoj relaciji postoji tranzitivna zavisnost, jer **NazivGrada zavisi od neključnog atributa PostanskiBr**, a PostanskiBr zavisi od BrojaIndeksa. Dakle:

BrojIndeksa → PostanskiBr → NazivGrada

Ova zavisnost krši uslove treće normalne forme jer NazivGrada nije direktno zavisan od primarnog ključa, već indirektno — preko drugog neključnog atributa (PostanskiBr).

Da bi se relacija dovela u treću normalnu formu (3NF), neophodno je izvršiti dekompoziciju tako što ćemo atribut NazivGrada izdvojiti u novi objekat GRAD. U ovu novu relaciju dodaćemo i atribut PoštanskiBroj, jer je NazivGrada zavisio upravo od njega. Međutim, atribut PoštanskiBroj ne sme biti uklonjen iz početne relacije, jer on ostaje kao strani ključ koji povezuje relaciju STUDENT sa relacijom GRAD.

Rezultat ove dekompozicije su sledeće dve relacije:

STUDENT (BrojIndeksa, Ime, Prezime, Adresa, PoštanskiBr)

GRAD (PoštanskiBr, NazivGrada)

U ovakvoj strukturi, relacija STUDENT zadovoljava uslove treće normalne forme, jer svi njeni neključni atributi zavise isključivo i direktno od primarnog ključa (BrojIndeksa). Relacija GRAD predstavlja nezavisnu tabelu u kojoj se čuva veza između poštanskog broja i naziva grada. Objekti koji sadrže dva međusobno zavisna atributa, gde jedan jednoznačno određuje drugi (kao što su PoštanskiBroj i NazivGrada), često se u praksi nazivaju **šifarnici** i obično se korisnicima prikazuju u vidu padajućih menija.

Ovakvom dekompozicijom eliminiše se tranzitivna zavisnost, čime se omogućava dosledno ažuriranje podataka i smanjuje rizik od anomalija pri unosu, izmeni i brisanju podataka (Date, 2012; Coronel & Morris, 2015).

Primena 3NF na vežbu 5.1.

Početni objekat PORUDZBENICA je, primenom Druge normalne forme (2NF), dekomponovan u sledeće relacije:

PORUDZBENICA (BrojPorudzbenice, DatumPorudzbenice, NazivKupca, AdresaKupca, PostanskiBroj, NazivGrada, KupacID, UkupnoBezPDVa, PDV, UkupnoSaPDVom, ImeZaposlenog, PrezimeZaposlenog, ZaposleniID)

STAVKE_PORUDZBENICE (BrojPorudzbenice, SifraProizvoda, PoručenaKolicina, PopustNaKol, JedinicniIznos)

PROIZVOD (SifraProizvoda, OpisProizvoda, JedinicnaCenaProizvoda)

Sada ćemo proveriti da li su relacije i u trećoj normalnoj formi (3NF), analizom tranzitivnih zavisnosti između atributa.

U relaciji PORUDZBENICA identifikujemo sledeće zavisnosti:

BrojPorudzbenice → KupacID

KupacID → NazivKupca, AdresaKupca, PostanskiBroj

PostanskiBroj → NazivGrada

BrojPorudzbenice → ZaposleniID

ZaposleniID → ImeZaposlenog, PrezimeZaposlenog

Na osnovu ovih zavisnosti možemo zaključiti da atributi NazivKupca, AdresaKupca, PostanskiBroj, NazivGrada, ImeZaposlenog i PrezimeZaposlenog nisu direktno zavisni od primarnog ključa (BrojPorudzbenice), već zavise od drugih neključnih atributa. Ove tranzitivne zavisnosti narušavaju pravila 3NF i moraju biti eliminisane dekompozicijom.

Na osnovu detektovanih zavisnosti, relacija PORUDZBENICA se dekomponuje na sledeći način:

PORUDZBENICA (BrojPorudzbenice, DatumPorudzbenice, KupacID, ZaposleniID, UkupnoBezPDVa, PDV, UkupnoSaPDVom)

KUPAC (KupacID, NazivKupca, AdresaKupca, PostanskiBroj)

MESTO (PostanskiBroj, NazivGrada)

ZAPOSLENI (ZaposleniID, ImeZaposlenog, PrezimeZaposlenog)

Relacije STAVKE_PORUDŽBENICE i PROIZVOD ostaju nepromenjene jer ne sadrže tranzitivne zavisnosti među neključnim atributima.

Konačan rezultat vežbe je:

PORUDŽBENICA (BrojPorudzbenice, DatumPorudzbenice, KupacID, ZaposleniID, UkupnoBezPDVa, PDV, UkupnoSaPDVom)

KUPAC (KupacID, NazivKupca, AdresaKupca, PostanskiBroj)

MESTO (PostanskiBroj, NazivGrada)

ZAPOSLENI (ZaposleniID, ImeZaposlenog, PrezimeZaposlenog)

STAVKE_PORUDŽBENICE (BrojPorudzbenice, ŠifraProizvoda, PoručenaKoličina, PopustNaKol, JediničnIlznos)

PROIZVOD (ŠifraProizvoda, OpisProizvoda, JediničnaCenaProizvoda)

Na ovaj način baza podataka je dovedena u oblik koji omogućava veću logičku konzistenciju, manju redundantnost i izbegavanje anomalija pri ažuriranju.

Zadatak 5.3. Primeniti 3NF na sledeću relaciju:

ZAPOSLENI (ZaposleniID, ImeZaposl, DatumZaposlenja, Plata, OdelenjeID, NazivODel)

5.2.4. Boyce-Coddova normalna forma

Edgar F. Codd je originalno formulisao definicije 2NF i 3NF, ali se pokazalo da te definicije nisu dovoljno precizne u slučajevima kada relacija ima **više kandidata za ključ koji se međusobno preklapaju**, tj. imaju bar jedan zajednički atribut (overlapping candidate keys).

Kandidat za ključ je bilo koji skup atributa koji može jednoznačno da identifikuje svaki red u tabeli, bez potrebe za dodatnim atributima. Na primer, zamislimo da imamo objekat u kojoj su dva kandidata za ključ:

- Kandidat 1: {A, B}
- Kandidat 2: {A, C}

Ovi kandidati za složen primarni ključ se preklapaju jer **A** (atribut) pripada oba skupa. To znači da je atribut **A** zajednički u oba kandidata za ključ, a to stvara situaciju u kojoj oba kandidata "sarađuju" u određivanju jednog reda, ali svaki od njih može da bude validan primarni ključ za objekat.

Kada se kandidati za ključ preklapaju, može doći do problema u modelovanju podataka jer se može desiti da normalne forme kao što su 2NF i 3NF ne mogu da reše sve redundanse i zavisnosti. Ovaj preklapajući skup atributa može izazvati nepotpunu zavisnost.

Ova nedorečenost dovela je do razvoja strožije normalne forme, **Boyce-Coddove normalne forme** (BCNF). BCNF normalna forma je dobila naziv po svojim tvorcima, Raymondu F. Boyceu i Edgaru F. Coddu, koji su je formalno definisali 1974. godine. Cilj je bio da se reše određene anomalije i redundantnosti koje 3NF nije mogla da obuhvati,

posebno u slučajevima kada postoje složeni i preklapajući kandidati za ključ. Zanimljivo je da je britanski informatičar Ian Heath, još 1971. godine, objavio definiciju koja je praktično opisivala BCNF, ali se naziv Boyce-Codd ustalio u literaturi (Date, 2004).

Relacija je u BCNF **ako i samo ako** su **sve determinante** kandidati za ključ. **Determinanta je atribut ili skup atributa od kojih neki drugi atribut potpuno funkcionalno zavisi**.

Razmotrimo sledeću relaciju koja ilustruje problem preklapajućih kandidata za ključ i BCNF:

PRIJAVA (BrIndeksa, PredmetID, ProfesorID)

Pretpostavimo da važe sledeća pravila:

1. Student se jedinstveno identifikuje brojem indeksa (BrIndeksa).
2. Predmet se jedinstveno identifikuje šifrom predmeta (PredmetID).
3. Student može prijaviti više predmeta.
4. Za svaki prijavljeni predmet, student ga polaže kod jednog profesora.
5. Jedan profesor može predavati samo jedan predmet.

Na osnovu ovih pravila, možemo izvući sledeće funkcionalne zavisnosti:

- {BrIndeksa, PredmetID} -> ProfesorID (Primarni ključ određuje profesora)
- ProfesorID -> PredmetID (Svaki profesor predaje samo jedan predmet)

Da bi relacija bila u BCNF, svaka determinanta mora biti kandidat za ključ.

U našem primeru, ProfesorID je determinanta jer određuje PredmetID. Međutim, ProfesorID nije kandidat za ključ. Iz toga sledi da relacija PRIJAVA nije u BCNF.

Ovo odstupanje dovodi do redundantnosti i anomalija. Na primer, podatak da profesor P1 predaje predmet PR1 ponavlja se za svakog studenta koji je taj predmet prijavio. Takođe, nije moguće evidentirati da profesor P1 predaje PR1 ako nijedan student nije prijavio taj predmet.

Da bismo relaciju doveli u BCNF, moramo je dekomponovati (razbiti) na dve manje relacije koje će eliminisati problematičnu zavisnost ProfesorID -> PredmetID. Koraci su:

1. Izdvajamo zavisnost u novu relaciju. Kreiramo novu relaciju koja će sadržati samo atribute iz sporne zavisnosti:

   PREDAJE (ProfesorID, PredmetID)

U ovoj relaciji, ProfesorID je primarni ključ i ujedno jedina determinanta, pa je ova relacija u BCNF.

2. Modifikujemo originalnu relaciju. Iz originalne relacije uklanjamo atribute koji su izdvojeni, osim onih koji su primarni ključ. U ovom slučaju, uklanjamo PredmetID jer je on određen sa ProfesorID.

   PRIJAVA (BrIndeksa, ProfesorID)

   Primarni ključ ove relacije je {BrIndeksa, ProfesorID}. Ova relacija je takođe u BCNF.

Konačno, relacija PRIJAVA (BrIndeksa, PredmetID, ProfesorID) je dekomponovana u dve BCNF relacije:

PRIJAVA (BrIndeksa, ProfesorID)

PREDAJE (ProfesorID, PredmetID)

Ove dve relacije zajedno čuvaju sve originalne informacije, ali bez redundantnosti i anomalija koje su postojale u početnoj relaciji.

Vežba 5.2. Primeniti BCNF na sledeću relaciju POSAO (ZaposleniID, ProjekatID, Lokacija), gde važe sledeća poslovna pravila:

- Svaki zaposleni radi na više projekata i na svakom projektu je angažovano više zaposlenih.
- Svakom zaposlenom je dodeljena tačno jedna lokacija gde obavlja posao (više zaposlenih može imati istu lokaciju), odnosno lokacija zaposlenog zavisi samo od zaposlenog, a ne od projekta na kome radi.

Rešenje vežbe

Postupak normalizacije relacije **POSAO** (ZaposleniID, ProjekatID, Lokacija) primenom Boyce-Coddove normalne forme (BCNF) uključuje sledeće korake:

Korak 1: Određivanje ključeva i funkcionalnih zavisnosti

Prvo, moramo identifikovati sve funkcionalne zavisnosti na osnovu datih poslovnih pravila:

1. Svaki zaposleni radi na više projekata i na svakom projektu je angažovano više zaposlenih.
   - Ovo ukazuje da ne postoji jednostavna zavisnost između ZaposleniID i ProjekatID.
   - Primarni ključ mora biti složen: {ZaposleniID, ProjekatID}. Ova kombinacija jedinstveno identifikuje svaki red.
2. Svakom zaposlenom je dodeljena tačno jedna lokacija gde obavlja posao Ovo je ključna funkcionalna zavisnost: **ZaposleniID** -> **Lokacija**.

Korak 2: Provera BCNF uslova

Da bi relacija bila u BCNF, svaka determinanta mora biti kandidat za ključ. U relaciji POSAO (ZaposleniID, ProjekatID, Lokacija), imamo dve determinante:

- {**ZaposleniID**, **ProjekatID**} je determinanta jer je to primarni ključ. Po definiciji, primarni ključ je kandidat za ključ. Ovaj deo je u skladu sa BCNF.
- **ZaposleniID** je takođe determinanta jer funkcionalno određuje atribut Lokacija. Međutim, **ZaposleniID nije kandidat za ključ** (sâm ne može jedinstveno identifikovati red u relaciji POSAO jer zaposleni radi na više projekata).

Zbog ove druge determinante, relacija **POSAO nije u BCNF**.

Korak 3. Dekompozicija relacije

Problematičnu zavisnost **ZaposleniID -> Lokacija** moramo razdvojiti u novu relaciju.

1. **Kreiranje nove relacije za zavisnost**. Napravimo novu relaciju koja sadrži samo atribute iz sporne zavisnosti. Nazovimo je **ZAPOSLENI_LOKACIJA**.

   ZAPOSLENI_LOKACIJA (ZaposleniID, Lokacija)

   - Primarni ključ u ovoj relaciji je ZaposleniID, a Lokacija zavisi od njega. Pošto je ZaposleniID primarni ključ, on je i kandidat za ključ.
   - Ova relacija je sada u BCNF.

2. **Modifikovanje originalne relacije**. Iz originalne relacije POSAO uklanjamo atribut Lokacija, jer je on premešten u novu relaciju: POSAO (ZaposleniID, ProjekatID). Ova relacija je sada takođe u BCNF.

Konačno rešenje

Originalna relacija POSAO (ZaposleniID, ProjekatID, Lokacija) je uspešno dekomponovana u dve relacije koje su u BCNF:

POSAO (ZaposleniID, ProjekatID)

ZAPOSLENI_LOKACIJA (ZaposleniID, Lokacija)

Ove dve relacije ne sadrže redundantnost (lokacija zaposlenog se ne ponavlja za svaki projekat) i omogućavaju ažuriranje podataka bez anomalija. Na primer, sada je moguće promeniti lokaciju zaposlenog u samo jednom redu relacije ZAPOSLENI_LOKACIJA, umesto u više redova u originalnoj relaciji POSAO.

Zadatak 5.4. Primeniti BCNF na sledeću relaciju:

Relacija SEMESTAR (ProfesorID, KursID, BrSale) beleži raspored kurseva po semestrima.
Poslovna pravila su sledeća:
- Svaki ProfesorID može držati više KursID.
- Svaki KursID može držati više ProfesorID.
- Svaki KursID se održava u tačno jednoj BrSale.

5.2.5. Četvrta normalna forma

Četvrta normalna forma (4NF) rešava specifičan problem redundantnosti koji ne može biti otklonjen primenom prethodnih normalnih formi, uključujući BCNF. To se dešava kada relacija sadrži dve ili više **nezavisnih višeznačnih zavisnosti**.

Višeznačna zavisnost (A↠B) postoji u relaciji R(A,B,C) ako za svaku vrednost atributa A postoji skup vrednosti atributa B, a taj skup vrednosti je potpuno nezavisan od vrednosti ostalih atributa (C) u relaciji. Drugim rečima, jedna vrednost A određuje više vrednosti B, a ta veza nema nikakve veze sa vrednostima iz skupa C.

Ključna razlika u odnosu na funkcionalnu zavisnost (A→B) je u tome što funkcionalna zavisnost zahteva da jedna vrednost A određuje tačno jednu vrednost B, dok višeznačna zavisnost dopušta da jedna vrednost A određuje više vrednosti B.

Dakle, relacija R je u **četvrtoj normalnoj formi** (4NF) **ukoliko u njoj nisu date dve ili više nezavisne višeznačne činjenice**.

Zamislite da imamo objekat koji čuva informacije o knjigama:

BIBLIOTEKA (KnjigaID, AutorID, Žanr)

U tom objektu, jedna knjiga ima više autora i pripada više žanrova. Veza između knjige i njenih autora nema nikakve veze sa žanrovima te knjige. Isto tako, veza između knjige i njenih žanrova je potpuno nezavisna od njenih autora. U ovakvim slučajevima, čak i ako je relacija u BCNF, ona i dalje sadrži mnogo ponavljanja (redundantnosti). Cilj 4NF je da se ova ponavljanja otklone.

Dakle, relacija BIBLIOTEKA (KnjigaID, AutorID, Žanr) ima sledeća pravila:

- Jedna knjiga (identifikovana sa KnjigaID) može imati više autora: KnjigaID →→ AutorID
- Jedna knjiga može pripadati više žanrova: KnjigaID →→ Žanr

Ovo nam daje dve nezavisne višeznačne zavisnosti.

Zamislimo da knjiga K01 ima autore A01 i A02 i pripada žanrovima Ž01 i Ž02. U tabeli BIBLIOTEKA, podaci bi izgledali ovako:

BIBLIOTEKA		
KnjigaID	**AutorID**	**Žanr**
K01	A01	Ž01
K01	A01	Ž02
K01	A02	Ž01
K01	A02	Ž02

Kao što vidite, podatak da K01 ima autora A01 ponavlja se dva puta, kao i podatak da K01 pripada žanru Ž01. Ova redundantnost otežava ažuriranje i unos podataka. Na primer, ako dodamo novog autora A03 za knjigu K01, morali bismo uneti dva nova reda, ponavljajući sve postojeće žanrove.

Da bismo rešili ovaj problem, moramo da razbijemo originalnu relaciju u dve manje, jednostavnije relacije. Svaka nova relacija će sadržati samo jednu od višeznačnih zavisnosti.

1. **Izdvojite prvu zavisnost u novu tabelu**. Kreiramo novu relaciju koja će beležiti samo vezu između knjiga i autora. Nazovimo je **KNJIGE_AUTORI**.
   KNJIGE_AUTORI (KnjigaID, AutorID)
2. **Izdvojite drugu zavisnost u drugu tabelu**. Napravimo još jednu relaciju koja će beležiti samo vezu između knjiga i žanrova. Nazovimo je **KNJIGE_ŽANROVI**.
   KNJIGE_ŽANROVI (KnjigaID, Žanr)

Nakon dekompozicije, podaci iz našeg primera bili bi smešteni u dve nove tabele:

KNJIGE_AUTORI	
KnjigaID	**AutorID**
K01	A01
K01	A02

KNJIGE_ŽANROVI	
KnjigaID	**Žanr**
K01	Ž01
K01	Ž02

Sada je svaka od ovih tabela u 4NF. Nema više ponavljanja. Dodavanje novog autora (A03) za knjigu K01 zahteva samo jedan unos u tabelu KNJIGE_AUTORI, bez ikakvog uticaja na tabelu KNJIGE_ŽANROVI. To olakšava održavanje baze i osigurava njenu konzistentnost.

Vežba 5.3. Primeniti 4NF na relaciju ZAPOSLENI_PROJEKTI (ZaposleniID, ProjekatID, Veština). Poslovna pravila glase:

- Jedan zaposleni može raditi na više projekata.
- Jedan zaposleni ima više veština.
- Veza između zaposlenog i projekta na kome radi je nezavisna od veština koje poseduje. Veza između zaposlenog i veština je nezavisna od projekata.

Rešenje vežbe

Na osnovu zadatih pravila, uočavaju se dve nezavisne višeznačne zavisnosti:

- ZaposleniID →→ ProjekatID (Zaposleni radi na više projekata)
- ZaposleniID →→ Veština (Zaposleni ima više veština)

Ovakva relacija dovodi do redundanse. Ako zaposleni Z1 radi na projektima P1 i P2 i poseduje veštine V1 i V2, tabela će izgledati ovako:

ZaposleniID	ProjekatID	Veština
Z1	P1	V1
Z1	P1	V2
Z1	P2	V1
Z1	P2	V2

Dodavanje novog projekta za zaposlenog Z1 zahtevalo bi ponavljanje svih njegovih veština.

Postupak dekompozicije u 4NF

Dekomponujemo relaciju na dve nove:

- Relacija 1: ZAPOSLENI_PROJEKTI (ZaposleniID, ProjekatID)
- Relacija 2: ZAPOSLENI_VEŠTINE (ZaposleniID, Veština)

Zadatak 5.5. Primeniti 4NF na sledeću relaciju:

Relacija RASPORED (Ku. rsID, UčesnikID, Instruktor) ima sledeća poslovna pravila:

- Jedan kurs (KursID) ima više učesnika (UčesnikID).
- Jedan kurs (KursID) ima više instruktora (Instruktor).
- Lista učesnika za jedan kurs je potpuno nezavisna od liste instruktora za taj isti kurs.

5.2.6. Peta normalna forma

Peta normalna forma (5NF), poznata i kao zavisnost spajanja (join dependency), predstavlja najviši nivo normalizacije u klasičnim relacionim bazama podataka. Njen cilj je eliminacija redundantnosti koja nije otklonjena u prethodnim normalnim formama (1NF, 2NF, 3NF, BCNF, 4NF). 5NF osigurava da se dekompozicija relacije bez gubitka informacija može izvesti samo ako je rezultat te dekompozicije direktno povezan s ključevima te relacije (Date, 2004).

Relacija je u 5NF ako i samo ako se svaka zavisnost spajanja u njoj može pripisati nekom kandidatu za ključ (Fagin, 1979).

Zavisnost spajanja je opštija od višeznačnih zavisnosti (4NF), koja je pak opštija od funkcionalnih zavisnosti (BCNF). 5NF je relevantna za situacije kada se podaci mogu dekomponovati na više od dve projekcije bez gubitka informacija.

Razmotrimo relaciju PROJEKTI_PROGRAMERA (ProgramerID, ProjekatID, Veština) sa sledećim pravilima:

- Programeri na projektu koriste samo veštine koje su im dodeljene.
- Projekti su dodeljeni samo onim programerima koji poseduju potrebne veštine.
- Veštine su dodeljene samo onim programerima koji rade na projektu.

Ovakva relacija se može razbiti na tri manje relacije bez gubitka informacija (Fagin, 1979):

1. PROJEKTI_PROGRAMERA (ProgramerID, ProjekatID)
2. PROGRAMERI_VEŠTINE (ProgramerID, Veština)
3. PROJEKTI_VEŠTINE (ProjekatID, Veština)

Spajanjem ove tri relacije, možemo rekonstruisati originalnu relaciju bez gubitka informacija, ali bez suvišnog ponavljanja podataka. Svaka od ove tri relacije sada se nalazi u 5NF, što je idealno stanje za bazu podataka. Primarni ključevi u sve tri nove relacije su složeni, formirani od parova atributa.

Vežba 5.4. Svesti sledeću relaciju na 5NF: RASPORED (StudentID, PredmetID, ProfesorID)**. Ova relacija beleži ko od studenata pohađa koji predmet i kod kog profesora. Poslovna pravila su:**

- Student može da pohađa više predmeta.
- Predmet pohađa više studenata
- Na predmetu može da predaje više profesora
- Student može pohađati predmet samo ako taj predmet predaje profesor koji je dodeljen tom studentu.

Rešenje vežbe

Korak 1. Identifikacija zavisnosti i ključeva

Prvo je neophodno shvatiti prirodu odnosa među atributima na osnovu poslovnih pravila:

- Student može da pohađa više predmeta. Predmet pohađa više studenata. Na predmetu može da predaje više profesora.
  - Ovo nam govori da ne postoji jednostavna funkcionalna zavisnost. Na primer, StudentID ne određuje jedinstveno PredmetID, niti obrnuto.
  - Složen primarni ključ je kombinacija svih atributa: (StudentID, PredmetID, ProfesorID). Ovo je neophodno jer je samo kombinacija sve tri vrednosti jedinstvena za svaki zapis.
- **Student može pohađati predmet samo ako taj predmet predaje profesor koji je dodeljen tom studentu.**
  - Ovo je ključno pravilo koje ukazuje na postojanje zavisnosti spajanja. Veza između StudentID, PredmetID i ProfesorID nije jednostavna, već je uslovljena svim trima entitetima istovremeno. Ovo je "skrivena" zavisnost koja se ne može rešiti prethodnim normalnim formama.
  - Ova zavisnost spajanja implicira da relaciju možemo bez gubitka informacija razložiti na tri manje relacije.

Korak 2. Provera uslova 5NF

Relacija **nije u 5NF** jer se njeni podaci mogu rekonstruisati spajanjem njenih projekcija na tri manja skupa atributa, a ta dekompozicija ne proizlazi iz jednostavnog kandidata za ključ. Prisustvo zavisnosti spajanja, bez da se može pripisati ključu, direktno krši uslove 5NF. Ovakva situacija dovodi do **redundantnosti** (ponavljanja podataka) i **anomalija ažuriranja**. Na primer, ako se studentu dodeli novi profesor, morali bismo da dodajemo redove za sve predmete koje taj student pohađa.

Korak 3. Dekompozicija na 5NF

Da bismo eliminisali redundantnost, relaciju RASPORED moramo dekomponovati na tri manje relacije, od kojih svaka modeluje po jednu od tri nezavisne veze:

1. Veze između studenta i predmeta:
   - Kreiramo novu relaciju STUDENTI_PREDMETI (StudentID, PredmetID).
   - Ova relacija čuva podatak da student StudentID pohađa predmet PredmetID.
   - Primarni ključ je složen: (StudentID, PredmetID).
2. Veze između studenta i profesora:
   - Kreiramo novu relaciju STUDENTI_PROFESORI (StudentID, ProfesorID).
   - Ova relacija čuva podatak da je profesoru ProfesorID dodeljen student StudentID.
   - Primarni ključ je složen: (StudentID, ProfesorID).

3. Veze između predmeta i profesora:
   - Kreiramo novu relaciju PREDMETI_PROFESORI (PredmetID, ProfesorID).
   - Ova relacija čuva podatak da profesor ProfesorID predaje predmet PredmetID.
   - Primarni ključ je složen: (PredmetID, ProfesorID).

Konačno rešenje

Originalna relacija **RASPORED** je uspešno dekomponovana u tri relacije koje su u 5NF:

STUDENTI_PREDMETI (StudentID, PredmetID)
STUDENTI_PROFESORI (StudentID, ProfesorID)
PREDMETI_PROFESORI (PredmetID, ProfesorID)

Svaka od ovih novih relacija ne sadrži redundantnost i može se ažurirati nezavisno. Spajanjem ove tri relacije, možemo rekonstruisati sve originalne informacije iz relacije RASPORED.

5.2.7. Normalna forma ključeva i domena

Normalna forma ključeva i domena (*Domain/Key Normal Form*, DK/NF), koju je formalno definisao Ronald Fagin 1981. godine, predstavlja najviši nivo normalizacije i nudi najširu definiciju relacione baze podataka bez anomalija. Faginova teza je da je relacija u DK/NF ako je svako ograničenje u njoj logična posledica definicije ključeva i domena (Fagin, 1981).

Relacija je u DK/NF ako i samo ako:

- **su sva ograničenja u relaciji direktna posledica definicije ključeva i domena** (ograničenja na domenima podrazumevaju skup dozvoljenih vrednosti za svaki atribut, npr. numeričke vrednosti, setovi dozvoljenih stringova).
- **Ako svaki atribut daje informaciju isključivo o ključu i o celom ključu**.

Drugim rečima, DK/NF obezbeđuje da relacija ne pati od anomalija pri održavanju (ubacivanju, brisanju ili ažuriranju podataka) jer se sve zavisnosti mogu utvrditi isključivo na osnovu jasno definisanih ključeva i domena. Fagin je tvrdio da nije neophodno primenjivati funkcionalne, višeznačne ili zavisnosti spajanja, već je dovoljno definisati ključeve i domene da bi se postigao najviši nivo normalizacije (Fagin, 1981). Relacija u DK/NF automatski zadovoljava sve prethodne normalne forme (1NF, 2NF, 3NF, BCNF, 4NF, 5NF).

Razmotrimo relaciju STAVKE_PORUDŽBINE (PorudžbinaID, ProizvodID, OpisProizvoda, PoručenaKoličina).

Ova relacija nije u DK/NF zbog sledećeg ograničenja:

- Jedan ProizvodID ima jedan OpisProizvoda.

Ovo ograničenje je zapravo funkcionalna zavisnost (ProizvodID → OpisProizvoda), ali ona nije direktna posledica definicije primarnog ključa relacije {PorudžbinaID, ProizvodID}. ProizvodID je samo deo ključa, pa ova relacija ne zadovoljava uslove 2NF, a time ni DK/NF. Da bi se postigla DK/NF, potrebno je dekomponovati relaciju i izolovati zavisnost u novu relaciju:

STAVKE_PORUDŽBINE (PorudžbinaID, ProizvodID, PoručenaKoličina)

PROIZVODI (ProizvodID, OpisProizvoda)

U relaciji PROIZVODI, primarni ključ je ProizvodID i ograničenje ProizvodID → OpisProizvoda je sada posledica definicije ključa te relacije.

Iako je DK/NF teorijski idealan model, njegova direktna primena u praksi je retka. Razlog tome je što je identifikacija i definisanje svih mogućih ograničenja (uključujući složene poslovne logike i izuzimanja) izuzetno složen zadatak. U većini slučajeva, praktičari modelovanja podataka se zaustavljaju na BCNF ili 3NF, jer one rešavaju najveći deo anomalija sa kojima se susreću. Otkrivanje zavisnosti i dekompozicija do 5NF je i dalje najpraktičniji način za postizanje visokog nivoa normalizacije, iako je teorijski Faginov pristup elegantniji (Date, 2004).

Vežba 5.5. Na osnovu prikupljenog dokumenta »Ugovor o rentiranju vozila« kreiran je početni objekat koji je potrebno analizirati primenom prve tri normalne forme:

UGOVOR (BrUgovoraID, DatumUgovora, KorisnikID, NazivKorisnika, AdresaKorisnika, TelKorisnika, VoziloID, TipVozila, MarkaVozila, DatumRent, VremeRent, DatumVraćanja, VremeVraćanja, IznosUgovora, AgentID, ImeAgenta, PrezimeAgenta)

Poslovna pravila retacar agencije su:

- Jedan klijent može iznajmiti više vozila
- Za svako pojedinačno vozilo je nephodno znati vreme i datum rentiranja i vraćanja

Rešenje vežbe

Korak 1: Primeniti 1NF

Kada se primenjuje 1NF, postavlja se pitanje da li postoje više vrednosti ostalih atributa za konkretnu vrednost primarnog ključa?

S obzirom da se na jednom ugovoru, koji se sklapa sa jednim klijentom, mogu nalaziti informacije o više vozila, to znači da sve atribute koji su vezani za vozilo treba izbaciti u novu relaciju.

UGOVOR (BrUgovoraID, DatumUgovora, KorisnikID, NazivKorisnika, AdresaKorisnika, TelKorisnika, ~~VoziloID~~, ~~TipVozila~~, ~~MarkaVozila~~, ~~DatumRent~~, ~~VremeRent~~, ~~DatumVraćanja~~, ~~VremeVraćanja~~, IznosUgovora, AgentID, ImeAgenta, PrezimeAgenta)

Kreiraćemo novu relaciju STAVKE_UGOVORA u koji ćemo staviti sve izbačene atribute:

STAVKE_UGOVORA (VoziloID, TipVozila, MarkaVozila, DatumRent, VremeRent, DatumVraćanja, VremeVraćanja)

Analizirajući postojeće atribute u relaciji STAVKE_UGOVORA, uočava se da VoziloID može biti primarni ključ koji će odrediti ostale atribute, pa ćemo VoziloID podvući:

STAVKE_UGOVORA (VoziloID, TipVozila, MarkaVozila, DatumRent, VremeRent, DatumVraćanja, VremeVraćanja)

Kada se primenjuje prva normalna forma, neophodno je dodati primarni ključ početne relacije, kako ne bi došlo do gubljenja početnih informacija i tada se u novoj relaciji dobija složen primarni ključ:

STAVKE_UGOVORA (BrUgovoraID, VoziloID, TipVozila, MarkaVozila, DatumRent, VremeRent, DatumVraćanja, VremeVraćanja)

Dakle, primenom 1NF, dobijaju se sledeće relacije:

- UGOVOR (BrUgovoraID, DatumUgovora, KorisnikID, NazivKorisnika, AdresaKorisnika, TelKorisnika, IznosUgovora, AgentID, ImeAgenta, PrezimeAgenta)
- STAVKE_UGOVORA (BrUgovoraID, VoziloID, TipVozila, MarkaVozila, DatumRent, VremeRent, DatumVraćanja, VremeVraćanja)

Korak 2: Primeniti 2NF

Kada se primenjuje 2NF ispitujemo da li neključni atributi zavise od celog složenog primarnog ključa?

S obzirom da UGOVOR nema složen primarni ključ, one je već u 2NF.

S obzirom da STAVKE_UGOVORA imaju složen primarni ključ, ispitaćemo da li neki od neključnih atributa zavisi od jednog dela primarnog ključa. Uočava se da kada je poznat atribut VoziloID, tada su poznate sve informacije o vozilu i to TipVozila i MarkaVozila, što znači da ovi neključni atributi zavise od jednog dela primarnog ključa te ih je potrebno izbaciti u novu relaciju koju možemo nazvati VOZILO:

STAVKE_UGOVORA (BrUgovoraID, VoziloID, ~~TipVozila~~, ~~MarkaVozila~~, DatumRent, VremeRent, DatumVraćanja, VremeVraćanja)

Prilikom izbacivanja ovih atributa, u novu relaciju je potrebno dodati deo primarnog ključa od koga su zavisili – u ovom slučaju VoziloID:

VOZILO (VoziloID, TipVozila, MarkaVozila)

Dakle, konačni rezultat nakon primene 2NF je:

- UGOVOR (BrUgovoraID, DatumUgovora, KorisnikID, NazivKorisnika, AdresaKorisnika, TelKorisnika, IznosUgovora, AgentID, ImeAgenta, PrezimeAgenta)
- STAVKE_UGOVORA (BrUgovoraID, VoziloID, DatumRent, VremeRent, DatumVraćanja, VremeVraćanja)
- VOZILO (VoziloID, TipVozila, MarkaVozila)

Korak 3: Primeniti 3NF

Primenom 3NF ispituju se tranzitivne funkcionalne zavisnosti, odnosno da li postoje zavisnosti između neključnih atributa?

Razmotrimo relaciju:

UGOVOR (**BrUgovoraID**, DatumUgovora, KorisnikID, NazivKorisnika, AdresaKorisnika, TelKorisnika, IznosUgovora, AgentID, ImeAgenta, PrezimeAgenta)

Uočavamo da neključni atributi NazivKorisnika, AdresaKorisnika i TelKorisnika mogu se odrediti na osnovu atributa KorisnikID. U ovom slučaju izdvajamo ove zavisne neključne atribute u novu relaciju KORISNIK i dopisujemo atribut od koga su zavisili - KorisnikID:

KORISNIK (KorisnikID, NazivKorisnika, AdresaKorisnika, TelKorisnika)

Takođe, u relaciji UGOVOR, atributi ImeAgenta i PrezimeAgenta, mogu se odrediti na osnovu AgentID, te i njih izbacujemo u novu relaciju koj ćemo imenovati kao AGENT:

AGENT (AgentID, ImeAgenta, PrezimeAgenta)

U relaciji UGOVOR ostaju sledeći nezavisni neključni atributi:

UGOVOR (BrUgovoraID, DatumUgovora, KorisnikID, IznosUgovora, AgentID)

Razmotrimo drugu relaciju STAVKE_UGOVORA. Datumi i vremena nisu zavisni jedni od drugih, te je ova relacija u 3NF. Slično je za relaciju VOZILO, neključni atributi TipVozila i MarkaVozila, ne određuju jedno drugo.

Konačan rezultat:

UGOVOR (BrUgovoraID, DatumUgovora, KorisnikID, IznosUgovora, AgentID)

STAVKE_UGOVORA (BrUgovoraID, VoziloID, DatumRent, VremeRent, DatumVraćanja, VremeVraćanja)

VOZILO (VoziloID, TipVozila, MarkaVozila)

KORISNIK (KorisnikID, NazivKorisnika, AdresaKorisnika, TelKorisnika)

AGENT (AgentID, ImeAgenta, PrezimeAgenta)

5.3. PRIMER MODELOVANJA PODATAKA

Primer ERD dijagrama, prikazan u nastavku, modeluje bazu podataka za zdravstvenu ustanovu sa ciljem praćenja informacija o pacijentima, lekarima, bolnicama, propisanim lekovima, istoriji poseta ordinacijama, izmenama u statusu izabranih lekara i porodičnim vezama.

Na osnovu prikupljenih zahteva, definisana su sledeća ključna poslovna pravila koja će usmeravati proces modelovanja:

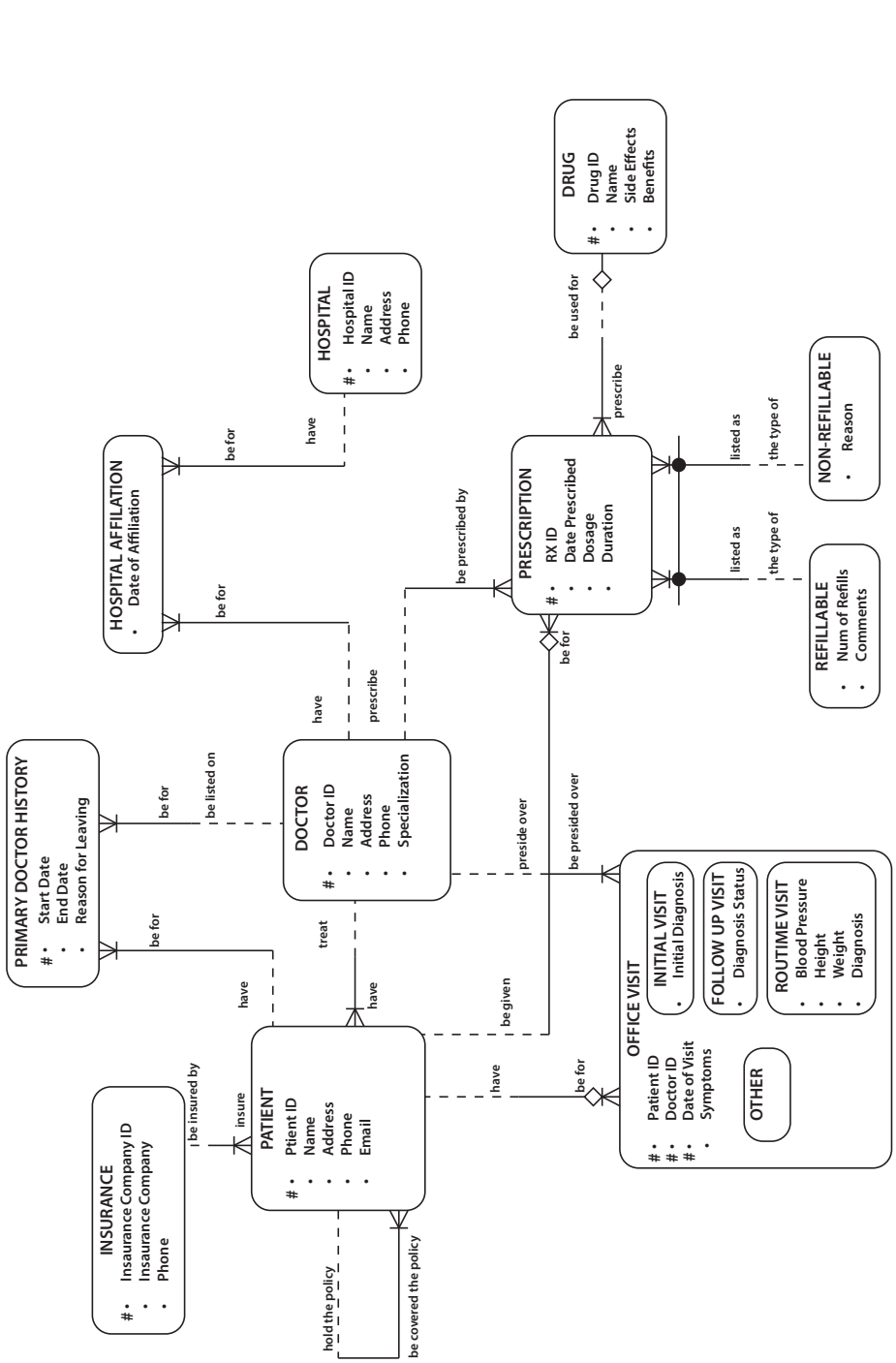
- Svaki pacijent ima tačno jednog izabranog lekara.
- Svaki lekar može imati više pacijenata.
- Svaki lekar može biti povezan sa više bolnica, i obrnuto – ova M:N veza se razrešava objektom SARADNJA_LEKAR_BOLNICA.
- Svaki pacijent može imati više poseta i više recepata.
- Svaki recept mora biti izdat tačno jednom pacijentu i od jednog lekara, za tačno jedan lek.
- Poseta može biti nove prirode, kontrolna ili redovna, što se modeluje pomoću podtipova ili dodatnih atributa.
- Ako je recept obnovljiv, beleže se broj i učestalost obnova.

Na osnovu definisanih zahteva i poslovnih pravila, identifikovani su sledeći entiteti sa pripadajućim atributima:

- **PATIENT**: Osnovni entitet za praćenje pacijenata.
  - PatientID: Jedinstveni identifikator (primarni ključ).
  - Name, Address, Phone, Email: Osnovni podaci o pacijentu.
  - InsuranceID, InsuranceCompanyID: Podaci o osiguranju.
  - PrimaryDoctorID: Strani ključ koji povezuje pacijenta sa izabranim lekarom.
  - PolicyHolderID: Koristi se za povezivanje članova porodice koji dele isto osiguranje.
- **DOCTOR**: Entitet za praćenje podataka o lekarima.
  - DoctorID: Jedinstveni identifikator (primarni ključ).
  - Name, Address, Phone: Kontakt podaci lekara.
  - Specialization: Specijalnost lekara.
- **HOSPITAL**: Entitet koji čuva podatke o bolnicama.
  - HospitalID: Jedinstveni identifikator (primarni ključ).
  - Name, Address, Phone: Podaci o bolnici.
- **DRUG**: Entitet za evidenciju propisanih lekova.
  - DrugID: Jedinstveni identifikator (primarni ključ).
  - Name, SideEffects, Benefits: Detalji o leku.

- **INSURANCE**: Skladišti podatke o osiguravajućim društvima.
  - InsuranceCompanyID: Jedinstveni identifikator (primarni ključ).
  - InsuranceCompanyName, Phone: Podaci o osiguravajućoj kući.
- **OFFICE_VISIT**: Evidencija poseta ordinacijama.
  - PatientID, DoctorID, VisitDate: Složen primarni ključ koji jedinstveno identifikuje posetu.
  - Symptoms, InitialDiagnosis, DiagnosisStatus, BloodPressure, Height, Weight, Diagnosis: Detalji posete.
- **PRESCRIPTION**: Evidencija izdatih recepata.
  - PrescriptionID: Jedinstveni identifikator (primarni ključ).
  - DatePrescribed, Dosage, Duration, Refillable, NumberOfRefills, Comments, Reason: Detalji recepta.
  - PatientID, DoctorID, DrugID: Strani ključevi koji povezuju recept sa pacijentom, lekarom i lekom.
- **DOCTOR_HISTORY**: Praćenje promena izabranih lekara.
  - PatientID, DoctorID, StartDate: Složen primarni ključ.
  - EndDate, ReasonForChange: Detalji o promeni.
- **DOCTOR_HOSPITAL_AFFILIATION**: Razrešenje M:N veze između lekara i bolnica.
  - DoctorID, HospitalID: Složen primarni ključ.
  - AffiliationDate: Datum uspostavljanja saradnje.

Na osnovu navedenih objekata, atributa i veza, moguće je kreirati Entity-Relationship (ER) dijagram koji vizuelno prikazuje strukturu baze podataka i odnose među entitetima. Ovaj dijagram služi kao osnova za kreiranje tabela i relacija u bazi podataka, osiguravajući integritet podataka i eliminaciju redundantnosti.



Slika 5.10. ERD za zdravstvenu ustanovu

POGLAVLJE 6



SPECIFIKACIJA I MODELOVANJE KORISNIČKIH ZAHTEVA

Specifikacija i modelovanje korisničkih zahteva

6.1. Specifikacija zahteva upotrebom UML-a
6.2. Dijagram slučajeva upotrebe
6.3. Dijagram aktivnosti
6.4. Dokument specifikacije softverskih zahteva

Cilj poglavlja:

Specifikacija zahteva je ključni korak u razvoju softverskog sistema, jer definiše šta sistem treba da radi. Ovo poglavlje prikazuje kako UML (*Unified Modeling Language*) omogućava efikasno modelovanje i dokumentovanje korisničkih zahteva.

6. SPECIFIKACIJA I MODELOVANJE KORISNIČKIH ZAHTEVA

6.1. SPECIFIKACIJA ZAHTEVA UPOTREBOM UML-A

U poglavlju „3.2.2. Rational Unified Process metodologija“ ukratko je dat uvid u nastanak i pojam UML jezika, tako da ćemo se u ovom delu baviti samo UML dijagramima koji nam pomažu da razumemo korisnika, njegove probleme, potrebe, zahteve i ciljeve. Treba naglasiti da UML ne zamenjuje analizu zahteva, već samo pomaže da se ti zahtevi formalizuju i vizuelizuju kako bi se umanjili eventualni nesporazumi između korisnika i razvojnog tima i uvideli potencijalni izazovi ili problemi u ranoj fazi projekta.

Faza specifikacije korisničkih zahteva smatra se jednom od najkritičnijih u celokupnom procesu razvoja softvera. Razlog tome je što korisnici i razvojni tim često imaju različite perspektive na problem i različito tumače potrebe organizacije ili krajnjih korisnika. Prema Sommervilleu (2016), većina grešaka koje nastanu u ovoj fazi prenosi se kroz čitav životni ciklus projekta i postaje znatno skuplja za ispravku u kasnijim fazama. Upravo zato UML dijagrami koji se koriste u ovoj fazi, posebno dijagrami slučajeva upotrebe (u nastavku teksta: use case dijagrami) i dijagrami aktivnosti, imaju ključnu ulogu u komunikaciji i validaciji zahteva. Ovi dijagrami omogućavaju razvojnom timu da na jasan i konzistentan način prikaže:

- ko koristi sistem (akteri)
- koje funkcionalnosti korisnici očekuju (*use case*)
- kako te funkcionalnosti teku u okviru poslovnih procesa
- na koji način sistem i spoljne komponente međusobno komuniciraju.

Korišćenjem UML-a u fazi specifikacije lakše se identifikuju nedorečeni ili kontradiktorni zahtevi, otkrivaju izuzeci i alternativni scenariji, kao i nefunkcionalni zahtevi koji su ključni za kvalitet sistema (npr. performanse, bezbednost, pouzdanost). Dakle, prednosti korišćenja UML-a u specifikaciji zahteva su:

- povećana preciznost i potpunost specifikacije zahteva
- otkrivanje nedoslednosti i nejasnoća
- otkrivanje grešaka u ranoj fazi
- standardizovan pristup koji je razumljiv svim članovima tima.

U ovoj fazi analize posebno je važno jasno razlikovati funkcionalne i nefunkcionalne zahteve, jer zajedno čine osnovu za precizno definisanje budućeg softverskog sistema.

6.1.1. Funkcionalni zahtevi

Funkcionalni zahtevi (*functional requirements*) opisuju šta sistem treba da radi, odnosno konkretno ponašanje sistema, usluge (servise) koje pruža i interakcije između aktera i sistema. Drugim rečima, funkcionalni zahtevi su uvek direktno povezani sa poslovnom logikom, scenarijima korišćenja, interakcijama sa korisnicima ili komunikacijom sa drugim sistemima.

Najčešće vrste funkcionalnih zahteva su:

1. **Poslovne funkcije i procesi** - opis konkretnih funkcionalnosti koje sistem mora da podrži. Na primer, sistem mora omogućiti korisniku da kreira novi nalog, da kreira porudžbinu, da izvrši plaćanje i slično.
2. **Tokovi aktivnosti i pravila ponašanja** - definišu kako sistem treba da reaguje na određene događaje. Na primer, nakon što korisnik unese korisničko ime i lozinku, sistem mora verifikovati podatke i odobriti pristup ako su validni.
3. **Validacija podataka** - pravila o ispravnosti, formatu i ograničenjima ulaznih podataka. Na primer, polje „Cena“ mora da sadrži pozitivnu numeričku vrednost.
4. **Interakcija korisnika i sistema** - sve funkcije dostupne korisnicima putem korisničkog interfejsa. Na primer, korisniku treba da se omogući pretraživanje proizvoda po nazivu ili kategoriji.
5. **Interfejs prema drugim sistemima** (integracije) - opis načina razmene podataka sa eksternim servisima. Na primer, sistem mora da prosledi podatke o transakcijama eksternom platnom servisu.
6. **Uslovi i ograničenja ponašanja u posebnim situacijama** - alternativni tokovi, izuzeci i granični slučajevi. Na primer, ako je broj pokušaja logovanja na sistem veći od tri, sistem mora da zaključa nalog na tri sata.

U UML-u se funkcionalni zahtevi najčešće prikazuju use case dijagramima, jer ti dijagrami jasno povezuju aktere sa funkcijama i omogućavaju lakšu validaciju sa korisnicima.

6.1.2. Nefunkcionalni zahtevi

Nefunkcionalni zahtevi (*non-functional requirements, NFR*) opisuju kvalitet, ograničenja, performanse i uslove pod kojima sistem treba da radi, ali ne opisuju konkretno ponašanje ili funkcije sistema. Drugim rečima, dok funkcionalni zahtevi govore šta sistem radi, nefunkcionalni zahtevi govore kako dobro, pod kojim uslovima i sa kakvim ograničenjima to radi.

Najčešće vrste nefunkcionalnih zahteva su:

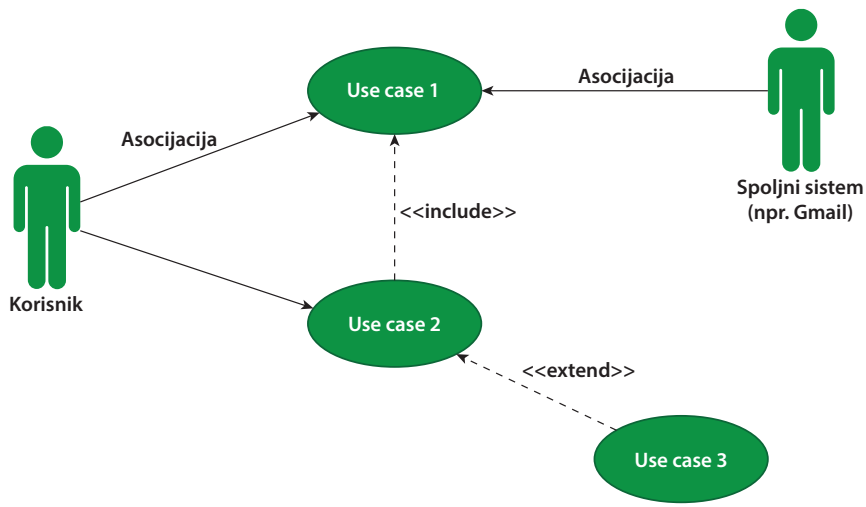
- **Performanse** – odnose se na očekivano ponašanje sistema pod različitim opterećenjima, uključujući vreme odziva, propusni opseg ili broj korisnika koji mogu istovremeno da koriste sistem. Na primer, sistem mora da odgovori u roku od 2 sekunde za 95% upita.
- **Bezbednost** – obuhvata mehanizme zaštite sistema i podataka, kao što su autentikacija i autorizacija korisnika, enkripcija podataka ili vođenje evidencije o aktivnostima (*audit log*). Na primer, sistem mora da beleži sve pokušaje prijave, bez obzira na to da li su bili uspešni ili ne.
- **Pouzdanost i dostupnost** – odnose se na stabilnost sistema i njegovu spremnost za rad u svakom trenutku, uključujući maksimalno dozvoljeno vreme nedostupnosti, toleranciju na greške i brzinu oporavka nakon incidenta. Na primer, sistem mora biti dostupan 99,9% vremena tokom godine.
- **Upotrebljivost** (*usability*) – tiče se jednostavnosti korišćenja, intuitivnosti interfejsa i pristupačnosti za različite kategorije korisnika. Na primer, korisnik treba da završi proces registracije u najviše tri koraka.
- **Održivost i održavanje** (*maintainability*) – obuhvata lakoću uvođenja izmena, modularnost sistema, jasnoću koda i mogućnost brze identifikacije problema. Na primer, promena poslovnog pravila ne sme zahtevati izmene u više od jedne komponente.
- **Skalabilnost** (mogućnost povećanja kapaciteta) – podrazumeva sposobnost sistema da efikasno funkcioniše i pri znatno većem obimu podataka ili broju korisnika. Na primer, sistem mora podržati povećanje broja korisnika sa 10.000 na 100.000 bez značajne degradacije performansi.
- **Portabilnost** – odnosi se na mogućnost da sistem radi na različitim platformama ili operativnim sistemima bez potrebe za velikim izmenama. Na primer, aplikacija mora da omogući instalaciju na Windows, Linux i macOS operativnim sistemima.

Iako UML nije primarno namenjen za modelovanje nefunkcionalnih zahteva, njegova fleksibilnost omogućava da se ovi zahtevi prikažu kroz napomene, stereotipe, ograničenja ili dodatne oznake, čime se precizno definišu kriterijumi kvaliteta i ograničenja funkcionalnosti sistema.

6.2. DIJAGRAM SLUČAJEVA UPOTREBE

Prvi dijagram koji se crta u fazi analize zahteva je dijagram slučajeva upotrebe (u daljem tekstu: use case dijagram). Osnovni elementi ovog dijagrama su (Slika 5.10):

- **Akteri** – predstavljeni stilizovanim figurama („čiča gliše"), koji označavaju korisnike ili postojeće (već razvijene) aplikacije koje komuniciraju sa sistemom.
- **Slučajevi upotrebe** (*use cases*) – predstavljeni elipsama, koje opisuju funkcionalnosti koje sistem pruža akterima.
- **Relacije** – veze između aktera i *use case*-ova i između samih use *case*-ova. Neke od osnovnih relacija su:
  - **Asocijacija** – osnovna veza između aktera i *use case*-a.
  - ***Include*** (uključuje) – označava da jedan *use case* uvek uključuje drugi. Na primer, da bi korisnik koristio *use case* „Plaćanje", neophodno je da prethodno uspešno izvrši „Prijavljivanje na sistem" (*login*), što se modeluje *include* relacijom.
  - ***Extend*** (proširuje) – označava opcione ili dodatne funkcionalnosti *use case*-a. Na primer, *use case* „Podizanje novca sa bankomata" može biti proširen (*extend*) sa funkcionalnošću „Štampanje slipa". Ova relacija je opciona, jer korisnik može odabrati i da ne želi da mu se štampa slip (račun transakcije).



Slika 6.1. Osnovni elementi *use case* dijagrama

Iako je *use case* dijagram naizgled jednostavan, on je ujedno i najkritičniji, jer ukoliko se u njemu nešto izostavi ili pogreši, to direktno utiče i na tačnost i potpunost svih ostalih dijagrama koji se izrađuju tokom razvoja softvera. Drugi razlog, njegove jednostavnosti je ta, da bi se i sami korisnici uključili u njihovo kreiranje. Videvši kružiće i čiča gliše, korisnik je slobodniji da i sam koriguje postojeće ili nacrta nove elemente *use case* dijagrama.

Dakle, *use case* dijagram predstavlja:

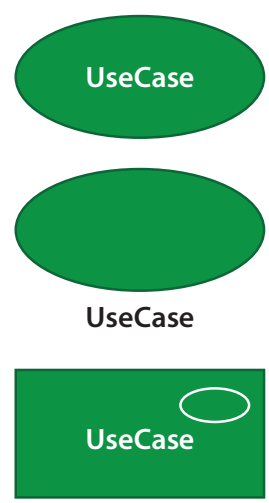
- osnovu za razumevanje funkcionalnosti sistema iz perspektive korisnika.
- glavne funkcionalnosti (*use cases*) sistema
- hronološku sekvencu *use case*-ova, a koje su od posebne važnosti za aktera
- osnovni dogovor između naručioca sistema i razvojnog tima šta treba da se razvije.

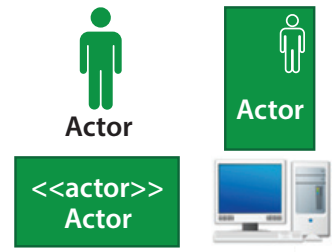
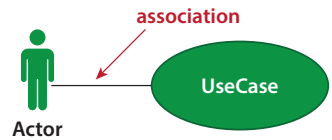
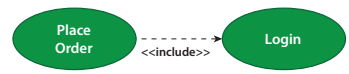
Use case dijagram opisuje šta sistem radi, ali ne i kako, tj. ne pokazuje kako će ponašanje (funkcionalnosti) sistema biti implementirane.

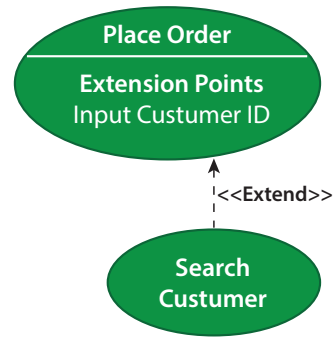
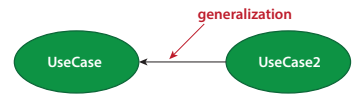
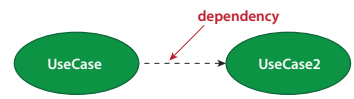
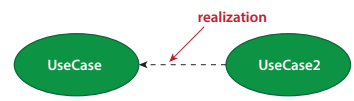
6.2.1. Osnovni elementi use case dijagrama

Najčešći problemi pri kreiranju *use case* dijagrama odnose se na neadekvatno definisanje *use case*-ova, jer se često mešaju sa aktivnostima (npr. unos šifre) i pogrešno imenuju, pri čemu se umesto glagolskih naziva (npr. pretraga proizvoda) koriste imenice (npr. proizvod), te nije jasno koju funkcionalnost *use case* zapravo treba da realizuje. Takođe, *use case* dijagram se ponekad pogrešno posmatra kao algoritam ili detaljan opis toka izvršavanja, umesto kao prikaz funkcionalnih zahteva iz perspektive korisnika, tj. funkcionalnosti sa kojima korisnik stupa u interakciju. Dodatne česte greške uključuju predstavljanje spoljnih aktera (npr. Gmail) kao *use case*-ove, što bi značilo da ih treba razviti, kao i nabrajanje *use case*-ova bez logičnog ili korisnički motivisanog redosleda, čime se gubi njihova preglednost i svrha. Kako bi se izbegle ove i druge uobičajene greške, u tabeli 6.1 detaljnije su objašnjeni osnovni elementi *use case* dijagrama i dati njihovi mogući grafički prikazi u zavisnosti od korišćenih alata.

Tabela 6.1. Opis i grafički prikaz osnovnih elemeanta *use case* dijagrama

Naziv	Opis	Grafički prikaz
Use case	• Jedna funkcionalnost koja je od važnosti za aktera. • Jedan servis koji se sastoji od više aktivnosti. Na primer, servis logovanje, sastoji se od aktivnosti: unos korisničkog imena, unos lozinke, provera parametara i dr. • Željeno ponašanje sistema koji se razvija. • Naziv *use case*-a izražava se glagolom u infinitivu, čime se jasno označava akcija koju sistem izvršava (npr. „Praćenje proizvoda“, „Plaćanje“). • Naziv treba da bude kratak, jednoznačan i da odražava očekivanu funkcionalnost, bez tehničkih detalja implementacije.	

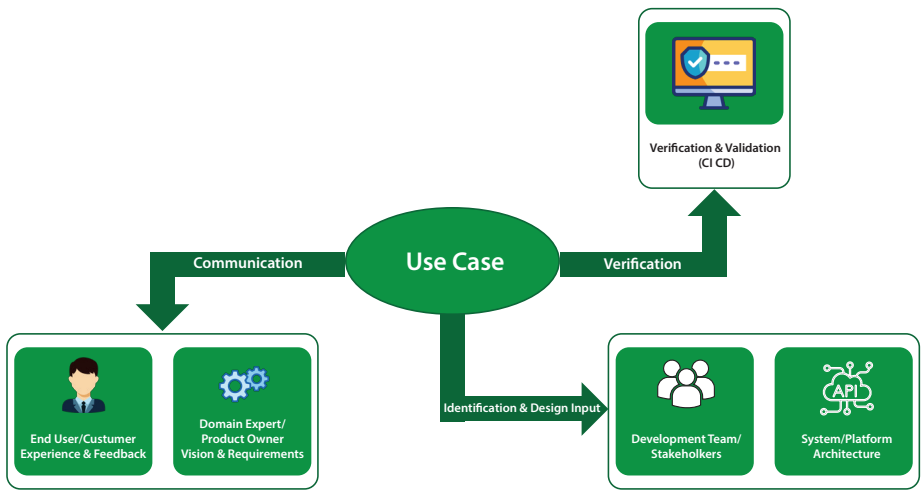
Naziv	Opis	Grafički prikaz
Akter	• Korisnik aplikacije • Postojeći, već razvijeni sistem, sa kojim će buduća aplikacija da komunicira putem API-ja (npr. Facebook) • Interni korisnik koji ima određenu ulogu (rolu) u sistemu • Hardverski uređaj • Akteri se imenuju imenicama, jer predstavljaju uloge ili entitete koji ostvaruju interakciju sa sistemom (npr. „Korisnik", „Administrator", „Spoljni sistem plaćanja").	
Asocijacija	• Relacija komunikacije između aktera i *use case*-a. • Može biti jednosmerna ili dvosmerna. • Može postojati jedino između aktera i *use case*-a, nikako između *use case*-ova!	
Uključivanje (*include*)	• Ova relacija može biti samo između *use case*-ova. • Pokazuje na use case koji prvo treba prvo proveriti pre nego što se korisniku dozvoli pristup željenom *use case*-u. Na primer, ako include relacija ide od *use case*-a „Plaćanje" do *use case*-a „Autentikacija korisnika", to znači da će se prvo izvršiti autentikacija korisnika, pa će tek onda korisnik moći da pristupi plaćanju. • Ukoliko se u okviru više *use-case*-ova se nalazi ista funkcionalnost (npr. login), tada se ona može izdvojiti u poseban *use case* (*Login*), a *include* relacija će biti usmerena od osnovnog do izdvojenog *use case*-a.	

Naziv	Opis	Grafički prikaz
Proširivanje (*extend*)	• Ova relazcija može da postoji samo između *use case*-ova. • Ukazuje se na alternativni, opcioni ili onaj use case koji se događa pod određenim uslovima. Na primer, ako *use case* „Podizanje novca sa bankomata“ ima u okviru sebe dodatnu, opcionalnu funkcionalnost, kao što je na primer, „štampanje slipa“ (opcionalnost ukazuje na to da *use case* „štampanje slipa“ i ne mora da se realizuje ukoliko korisnik to ne želi), tada *use case* „Podizanje novca“ biti proširen sa ovom opcionalnom funkcionalnošću, odnosno relacija *extend* će pokazivati na *use case* koji tu opciju nudi.	
Generalizacija	• Ova relacija se može prikazati bilo između aktera ili između *use case*-ova. • Ukazuje na opšti *use case* i njemu određenije / konkretnije *use case*-ove • Specifični *use case* nasleđuje opis opšteg *use case*-a, odnosno pokazuje konkretnu realizaciju opšteg *use case*-a.	
Zavisnost (*Dependency*)	• Između *use case*-ova • Ređe se koristi u fazi analize zahteva • Ukazuje na to da *use case* zahteva druge *use case*-ove za svoju određeniju specifikaciju ili implementaciju	
Realizacija (*Realization*)	• Između *use case*-ova i ređe se koristi u fazi analize zahteva. • Povezuje specifikaciju i implementaciju te specifikacije. • Pokazuje detaljniju realizaciju ili implementaciju apstraktnog *use case*-a.	

6.2.2. Kreiranje *use case* dijagrama

Namena use case dijagrama je da (Slika 6.2):

- **identifikuje** aktere i funkcionalnosti koje očekuju od sistema, čime se precizira šta sistem treba da obezbedi u pogledu ponašanja;
- **verifikuje** dogovorene zahteve, odnosno služi kao potvrda da su korisnici i razvojni tim usaglašeni oko obima i svrhe funkcionalnosti koje se razvijaju;
- služi kao sredstvo **komunikacije** između korisnika i tima, a i između samih članova tima.



Slika 6.2. Namena use case dijagrama

Broj i nivo detalja *use case* dijagrama zavisi od složenosti sistema koji se modeluje. Kod jednostavnijih aplikacija dovoljan je jedan dijagram koji obuhvata sve ključne aktere i njihove interakcije. Kod složenih sistema, preporučuje se izrada više komplementarnih *use case* dijagrama kako bi se izbegla pretrpanost i omogućio pregledan prikaz različitih perspektiva sistema. U praksi se najčešće izrađuju sledeće vrste dopunskih ili fokusiranih dijagrama:

- **Dijagram po akteru**: koristi se kada jedan akter ostvaruje interakciju sa većim brojem *use case*-ova, pa se na posebnom dijagramu prikazuju sve njegove relevantne funkcionalnosti.
- **Dijagram po *use case*-u**: izrađuje se kada jedan *use case* ima veliki broj relacija (npr. *include*, *extend*) ili kada je potrebno detaljnije predstaviti njegovu lokalnu mrežu povezanih funkcionalnosti.
- **Dijagram po iteraciji**: koristi se u iterativnom i agilnom razvoju kako bi se prikazali samo oni *use case*-ovi koji se realizuju u okviru određene iteracije ili sprinta.

Ovako strukturisana dokumentacija omogućava jasniji pregled sistema, povećava konzistentnost modela i olakšava komunikaciju u timu tokom celog procesa razvoja.

Raspored elemenata na use case dijagramu prati sledeće smernice:

- **Sa leve strane dijagrama prikazuju se akteri koji iniciraju izvršavanje *use case*-ova**, odnosno oni koji pokreću interakciju prema sistemu (npr. korisnici koji zahtevaju određenu funkcionalnost).
- **U centralnom delu dijagrama prikazuju se ključni *use case*-ovi**, najčešće postavljeni hronološkim ili logičkim redosledom, jedan ispod drugog, dok se pomoćni ili sekundarni *use case*-ovi raspoređuju bočno, vodeći računa o njihovim relacijama (*include, extend*).
- **Na desnoj strani dijagrama prikazuju se akteri koji reaguju na realizaciju *use case*-ova**, odnosno sistemi ili uloge koje pružaju podršku, povratne informacije ili dodatne usluge tokom izvršavanja funkcionalnosti.

Identifikovanje aktera

Akteri (*Actors*) nisu deo sistema već predstavljaju bilo koga ili bilo šta što mora da uspostavlja interakciju sa sistemom. Oni mogu biti korisnici, organizacione uloge, spoljni sistemi ili uređaji koji iniciraju ili primaju informacije od sistema. Dakle, **akter je uloga koju spoljašnji entitet ima u interakciji sa sistemom i taj entitet može unositi podatke, zahtevati određenu funkcionalnost ili primati rezultate iz sistema**.

Proces identifikacije aktera zasniva se na razumevanju okruženja sistema i njegovih interfejsa. Sledeća pitanja pomažu u sistematskom prepoznavanju aktera:

- Ko je zainteresovan za određeni zahtev ili funkcionalnost?
- Ko u organizaciji koristi *use case*-ove i u kojim kontekstima?
- Ko će koristiti aplikaciju ili imati koristi od njenog izvršavanja?
- Ko unosi, ažurira ili briše podatke u sistemu?
- Ko je odgovoran za pružanje podrške, administraciju ili održavanje sistema?
- Da li sistem sarađuje sa spoljnim servisima, uređajima ili drugim softverskim komponentama?
- Da li jedna osoba može obavljati više uloga (više akterskih perspektiva)?
- Da li više različitih korisnika može deliti istu ulogu/rolu u odnosu na sistem?
- Da li sistem uspostavlja komunikaciju sa postojećim (nasleđenim) sistemima?

Korišćenje ovih pitanja omogućava doslednu identifikaciju svih relevantnih aktera i doprinosi kvalitetnom definisanju granica sistema i njegovih funkcionalnih zahteva.

Identifikovanje *use case*-ova

Use case se definiše kao sekvenca interakcija koja vodi ka ostvarenju vrednog i merljivog ishoda za aktera. *Use case* obično:

- modeluje tok komunikacije između aktera i sistema, formalizujući očekivanu razmenu informacija;
- predstavlja celovitu funkcionalnost „od početka do kraja", odnosno logički zaokruženu sekvencu aktivnosti koja vodi ka jasno definisanom rezultatu;
- opisuje skup aktivnosti koje sistem izvršava kako bi akter ostvario cilj, pri čemu rezultat mora imati prepoznatljivu vrednost za tog aktera.

Proces identifikacije *use case*-ova obično se zasniva na analizi ciljeva aktera, poslovnih pravila i očekivanog ponašanja sistema. U tome mogu pomoći sledeća pitanja:

- Koji su zadaci svakog aktera i koje ciljeve želi da ostvari korišćenjem sistema?
- Da li akter kreira, čuva, menja, briše ili čita informacije u sistemu?
- Koji *use case*-ovi omogućavaju kreiranje, čuvanje, menjanje, brisanje ili čitanje ovih informacija?
- Da li neki akter treba da obavesti sistem o iznenadnim spoljnim događajima ili promenama?
- Da li je potrebno da sistem automatski obavesti aktera o određenim događajima?
- Koji *use case*-ovi su povezani sa administracijom, podrškom ili održavanjem sistema?
- Da li se svi funkcionalni zahtevi mogu realizovati kroz jasno definisane *use case*-ove?

Vežba 6.1. Kreiranje *use case* dijagrama za e-*commerce* aplikaciju (primer: Amazon)

Identifikovati aktere i *use case*-ove, a potom kreirati *use case* dijagram za pojednostavljenu verziju e-*commerce* aplikacije tipa Amazon. Dijagram treba da prikazuje funkcionalne zahteve sistema iz perspektive korisnika i eventualnih spoljnih servisa. Aplikacija treba da omogući korisniku da:

- pretražuje ili filtrira proizvode,
- kreira šoping korpu i uređuje njen sadržaj (dodavanje proizvoda, uklanjanje proizvoda i izmena količina),
- izvrši proces plaćanja putem eksternog provajdera plaćanja,
- kreira i uređuje svoj korisnički profil (lične informacije, adrese, preferencije),
- prati isporuku naručenih proizvoda.

Korisnik može pretraživati proizvode i kreirati šoping korpu bez potrebe da bude prijavljen na sistem. Međutim, za realizaciju plaćanja, uređivanje korisničkog profila i praćenje isporuka, neophodno je da se prethodno uspešno prijavi. Ukoliko korisnik nema postojeće parametre za prijavu, sistem treba da omogući registraciju putem Gmail naloga, Facebook naloga ili popunjavanjem standardnog registracionog formulara.

Rešenje vežbe

Akteri:

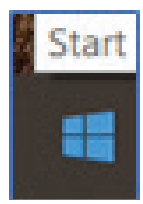
- Primarni akter (leva strana dijagrama):
  - Korisnik – inicira sve upotrebe sistema (pretraživanje, kupovina, uređivanje profila itd.).
- Sekundarni / spoljni akteri (desna strana dijagrama):
  - Eksterni provajder plaćanja (npr. *Stripe, Braintree*)
  - Gmail autentikacioni servis (*Google OAuth* servis)
  - Facebook autentikacioni servis

Use case-ovi:

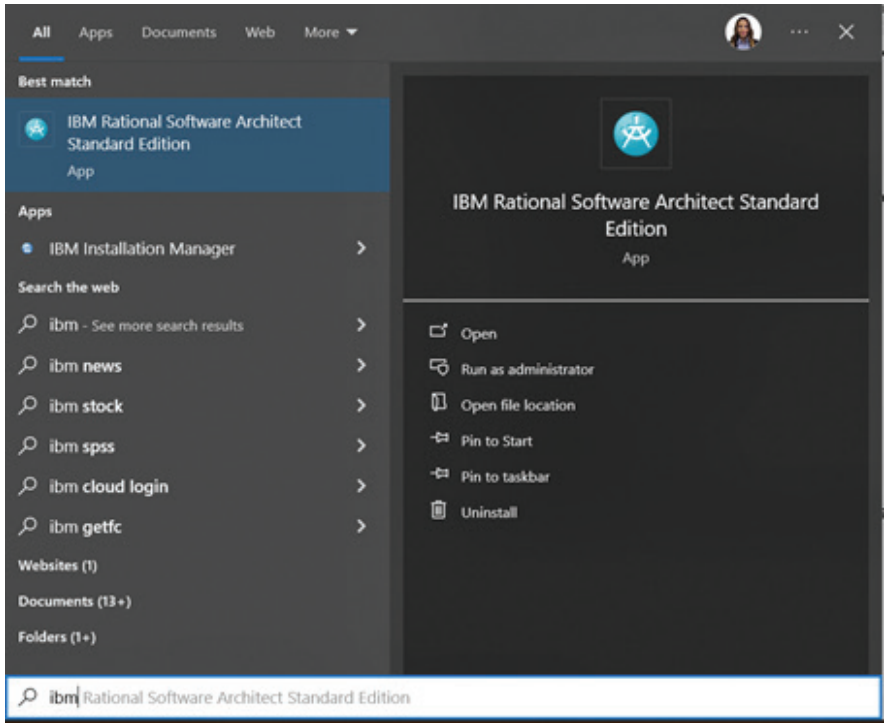
- Pretraživanje i filtriranje proizvoda
- Kreiranje i uređivanje šoping korpe
- Plaćanje
- Praćenje isporuke
- Uređivanje korisničkog profila
- Prijavljivanje (Logovanje)
- Registracija

Koraci kreiranja projekta i *use case* **dijagrama u IBM *Ratonal Software Architect*** (RSA) **alatu.**

Korak 1. Kliknuti na Start ikonu na donjem levom uglu ekrana, ukucati IBM i kliknuti na alat IBM *Rational Software Architect Standard Edition.*

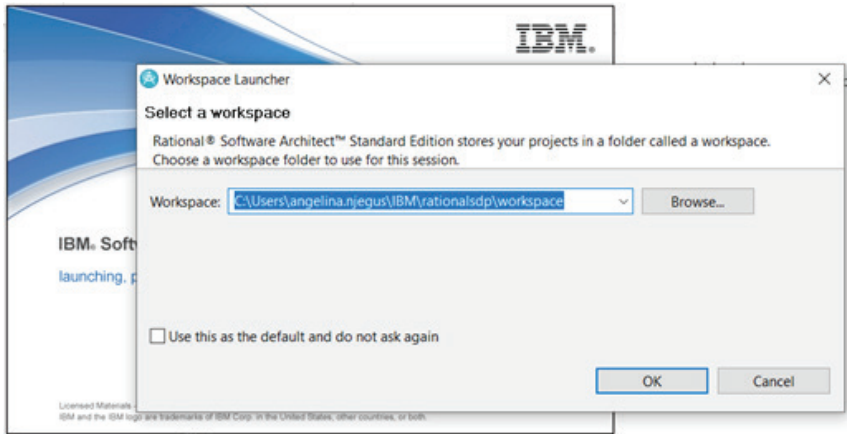


Slika 6.3. Start ikona kod Windows sistema



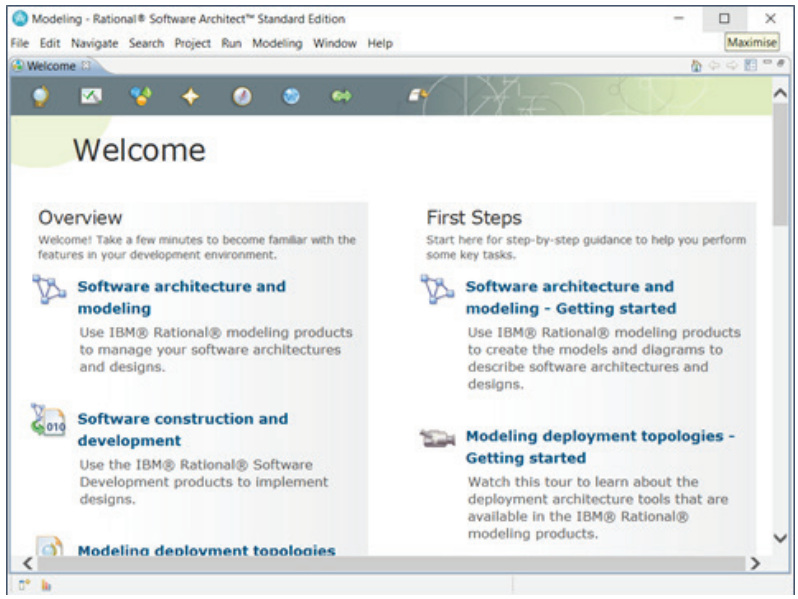
Slika 6.4. Pretraga i pokretanje IBM RSA alata

Korak 2. Ostavite po difoltu ponuđeni *workspace*, samo pogledajte putanju gde se čuvaju vaši projekti. U slučaju da ste iskoristili jedan *workspace* za neki konkretan projekat i želite novi da kreirate, dovoljno je da preimenujete *workspace*.

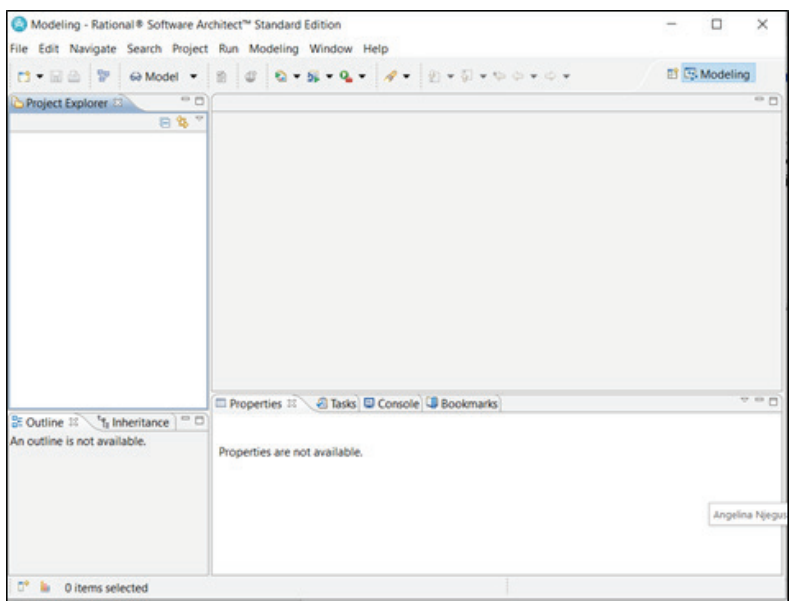


Slika 6.5. Odabir radnog prostora (*workspace*) za kreiranje i rad na projektu

Korak 3. Prilikom prvog pokretanja alata, pojaviće vam se *Welcome* prozor. Potrebno je samo zatvoriti prozor, kako bi se prikazalo okruženje za rad.



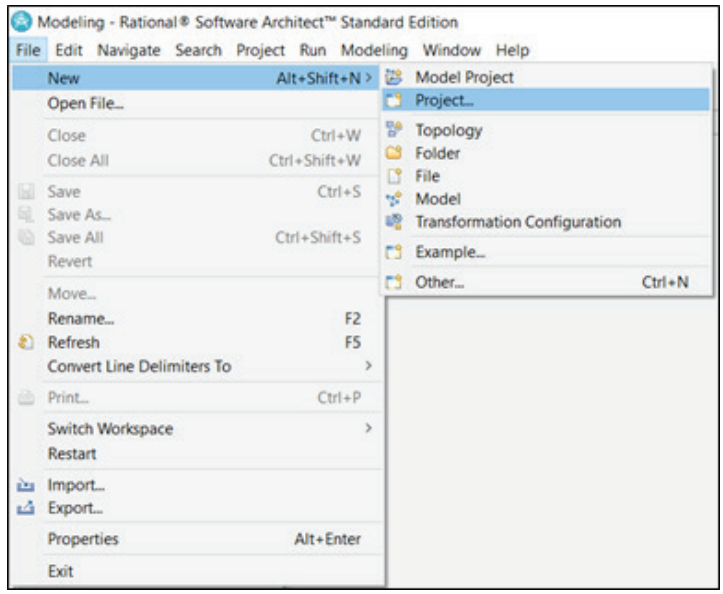
Slika 6.6. Prozor dobrodošlice (*welcome*) prilikom prvog pokretanja alata



Slika 6.7. Okruženje IBM RSA alata

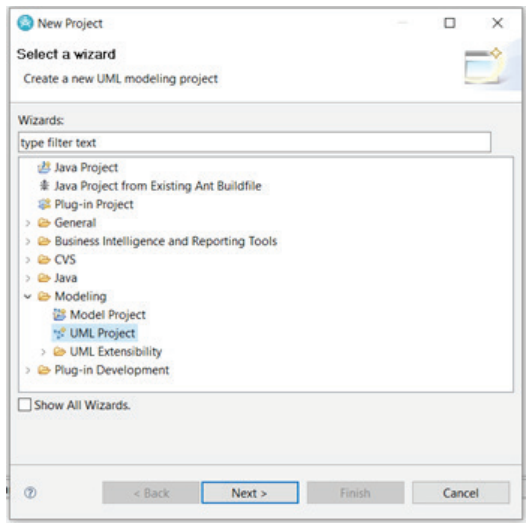
Korak 4: Kreirati projekat i model.

Projekat se kreira klikom pod *File* na *New*, a zatim *Project*.



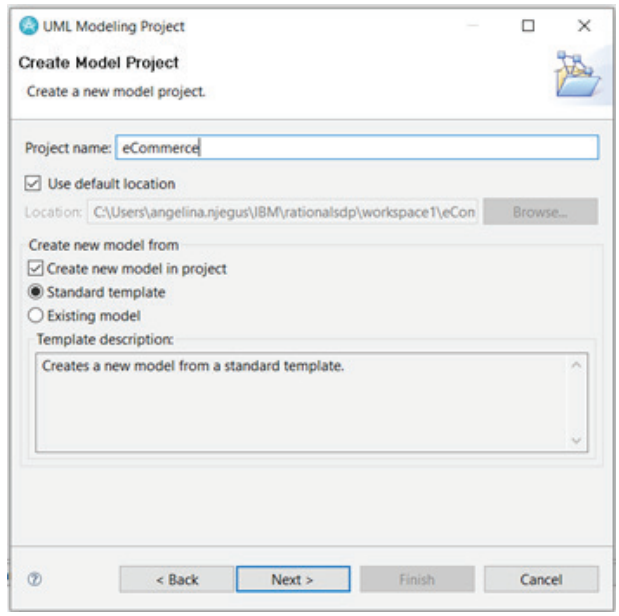
Slika 6.8. Kreiranje projekta u IBM RSA alatu

Proširiti *Modeling* i selektovati UML *Project*:



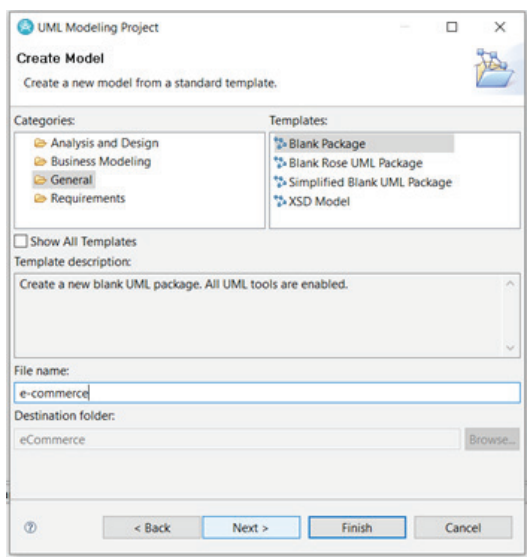
Slika 6.9. Kreiranje UML projekta u IBM RSA alau

Imenovati projekat, npr. *eCommerce:*



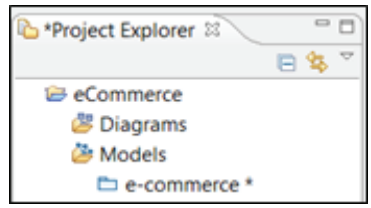
Slika 6.10. Imenovanje projekta

Odabrati pod kategorijom *General*, šablon (*template*) *Blank Package* i imenovati ga.



Slika 6.11. Kreiranje UML modela i paketa

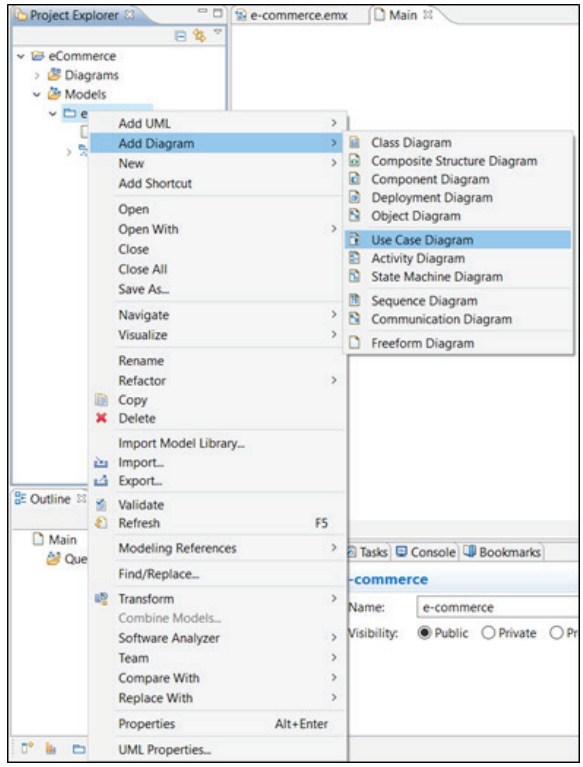
Nakon kreiranja projekta i modela pogledajte njegovu osnovnu strukturu u krajnjem levom prozoru tzv. *Project Explorer*-u. Ukoliko nije po difoltu proširena struktura, potrebno je proširiti sve elemente:



Slika 6.12. Početna struktura projekta

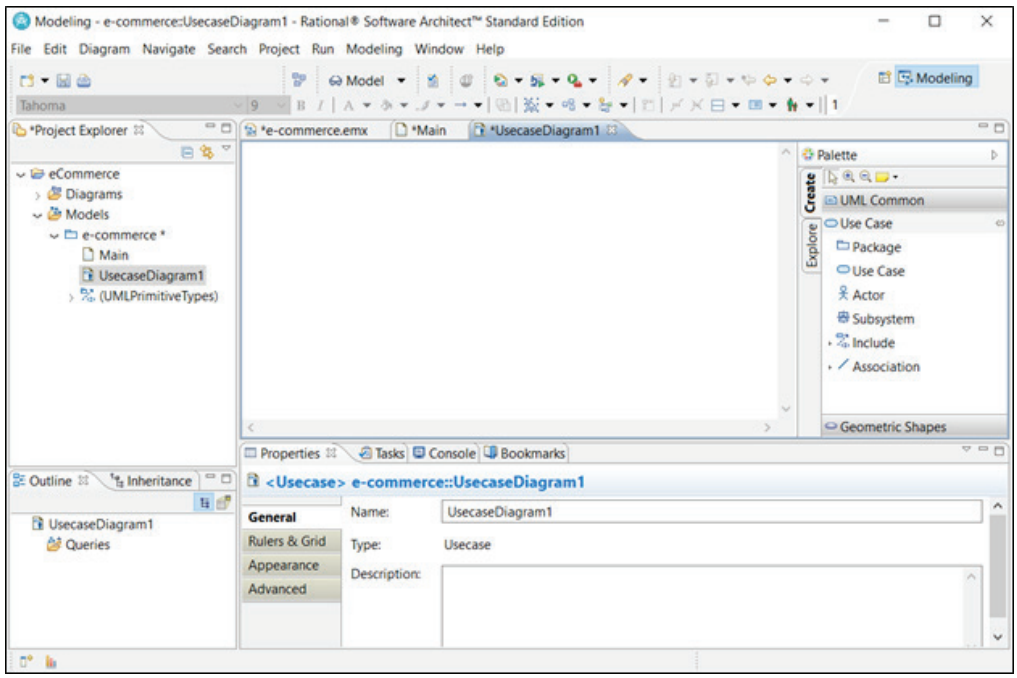
Korak 5. Kreirati *use case* dijagram

U okviru modela (*Models*), nalazi se prazan paket kojeg ste imenovali isto kao projekat *ecommerce*. Potrebno je kliknuti, desnim klikom na taj paket i iz padajućeg menija odabrati *Add Diagram*, a zatim *Use case Diagram*.



Slika 6.13. Kreiranje *use case* dijagrama

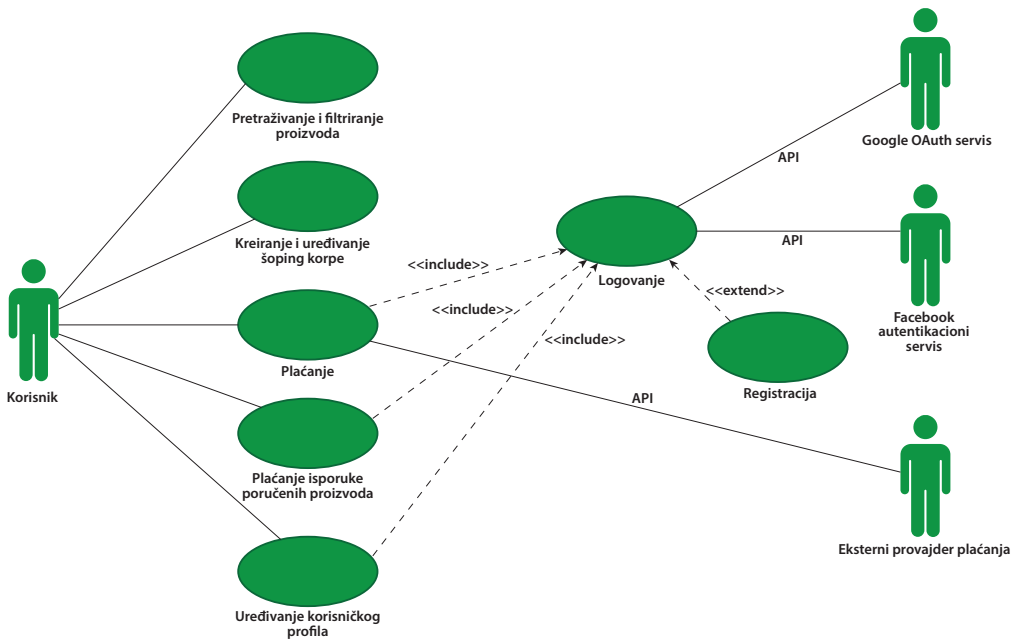
Možete i imenovati vaš *use case* dijagram, u slučaju da ih imate više u okviru vašeg projekta. Na slici 6.14 ostavljen je difoltni naziv. Na levom prozoru „Palette" nalaze se osnovni elementi za kreiranje *use case* dijagrama.



Slika 6.14. Paleta alatki za kreiranje *use case* dijagrama

Kreirati *use case* dijagram, tako što će te iz palete izabrati želeni element i prebaciti ga na centralni prostor za crtanje UML dijagrama (*Diagram Editor*).

Prema zahtevima ove vežbe, nacrtan je sledeći use case dijagram:



Slika 6.15. *Use case* dijagram za tipičnu e-commerce aplikaciju

Vežba 6.2. Kreirati *use case* dijagram za aplikaciju „Airbnb za rezervaciju smeštaja"

Kreirati *use case* dijagram za aplikaciju koja omogućava korisnicima da pretražuju, rezervišu, plaćaju i ocenjuju smeštaj, dok domaćini mogu da upravljaju svojim smeštajem i komuniciraju sa gostima.

Aplikacija treba da omogući korisnicima (gostima) da:

- Pretražuju smeštaj prema lokaciji, datumu i broju osoba.
- Filtriraju smeštaj po ceni, tipu smeštaja, broju soba, pogodnostima (Wi-Fi, parking, doručak).
- Pregledaju detalje smeštaja (opis, fotografije, ocene, recenzije prethodnih gostiju).
- Rezervišu smeštaj nakon što su prijavljeni na sistem.
- Plaćaju rezervaciju putem različitih metoda (PayPal, kreditna/debitna kartica, bankovni transfer).
- Komuniciraju sa domaćinom preko ugrađenog sistema poruka.
- Ocenjuju i komentarišu svoj boravak nakon odjave.

Domaćini (Hostovi) mogu da:

- Kreiraju i uređuju profil smeštaja (cena, opis, fotografije, dostupni datumi).
- Pregledaju dolazne rezervacije i potvrđuju ili odbijaju rezervacije.
- Komuniciraju sa gostima.
- Pregledaju ocene i komentare svojih smeštaja.

Sistem/platforma treba da:

- Upravlja korisničkim nalozima i autentifikacijom (login/registracija preko e-maila, Google ili Facebook naloga).
- Obrađuje plaćanja i evidentira transakcije.
- Čuva i prikazuje recenzije i ocene.
- Šalje notifikacije (npr. potvrda rezervacije, podsetnik za check-in).

Rešenje vežbe

Akteri su:

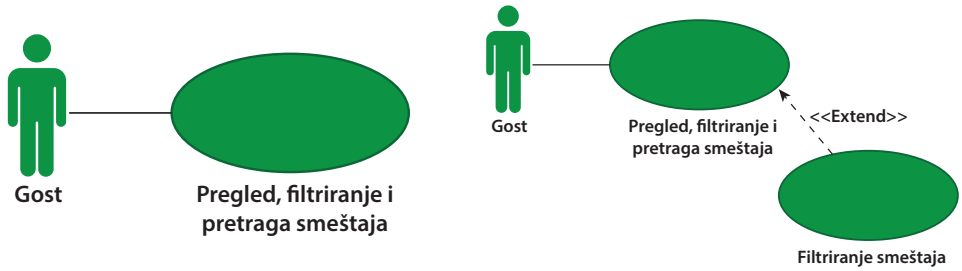
- Gost (Korisnik)
- Domaćin (Host)
- Sistem plaćanja

Use case-ovi su:

- Pregled, filtriranje i pretraga smeštaja
- Rezervacija
- Plaćanje
- Komunikacija sa domaćinom
- Ocenjivanje i recenzija boravka
- Kreiranje ili uređivanje smeštaja (Host)
- Pregled rezervacija (Host)
- Login
- Registracija

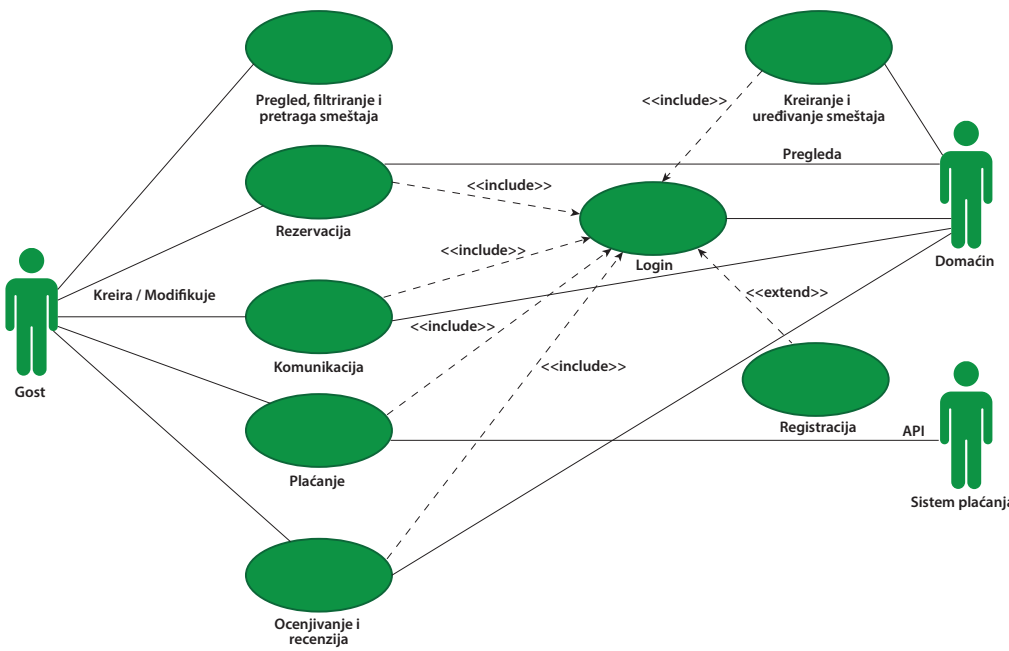
Napomena:

Ako se pitate zašto pregled, filtriranje i pretraga smeštaja nisu prikazani kao tri odvojena *use case*-a, razlog je taj što se *use case*-ovi koji dele istu osnovnu imenicu, u ovom slučaju smeštaj, mogu grupisati u jednu funkcionalnu celinu. To znači da se u okviru jednog *use case*-a mogu razviti sve mogućnosti pristupa informacijama o smeštaju. Međutim, ako je idejno rešenje takvo da filtriranje treba da bude poseban ekran sa dodatnim opcijama koje korisnik može selektivno da uključi i koristi, tada se filtriranje može definisati kao poseban *use case* i prikazati kao opciona funkcionalnost. Dakle, izbor zavisi od koncepta same aplikacije, odnosno od potreba i zahteva korisnika. Da bi napomena bila jasnija, u nastavku će biti prikazane obe alternative i njihov mogući grafički prikaz.



Slika 6.16. Dve alternative prikaza *use case*-ova vezanih za pregled, filtriranje i pretragu smeštaja

Pojednostavljeni *use case* sa prvom alternativom je prikazan na donjoj slici 6.17.



Slika 6.17. *Use case* dijagram za aplikaciju za rezervaciju smeštaja

Zadatak 6.1. Kreirati *use case* dijagram za HR aplikaciju koja podržava procese selekcije, regrutovanja i zapošljavanja kandidata. Prikupljeni su sledeći zahtevi od korisnika:

Kandidat može da:

- Prijavi se na otvorenu poziciju.
- Pregleda status svoje prijave.
- Prima obaveštenja (notifikacije).
- Učestvuje u intervjuima (uživo, pozivom ili online linkom).

HR osoblje može da:

- Kreira i objavljuje nove pozicije.
- Pregleda pristigle prijave kandidata.
- Zakazuje intervjue sa kandidatima.
- Ažurira status kandidata (prihvaćen, odbijen, u procesu).
- Sprovodi intervju.
- Ocenjuje kandidata nakon intervjua.
- Vodi proces zapošljavanja.

Menadžer HR odeljenja može da:

- Pregleda listu kandidata koje je HR tim selektovao.
- Daje ocenu i učestvuje u donošenju konačne odluke o zapošljavanju.

Dodatni sistemski uslovi:

- Korisnici moraju biti prethodno ulogovani da bi koristili funkcionalnosti sistema.
- Ako nemaju nalog, mogu se registrovati putem izdvojene funkcije Registracija.
- Registracija i Logovanje su potpuno nezavisni *use case*-ovi (korisnik može odabrati bilo koji iz menija).

6.3. DIJAGRAM AKTIVNOSTI

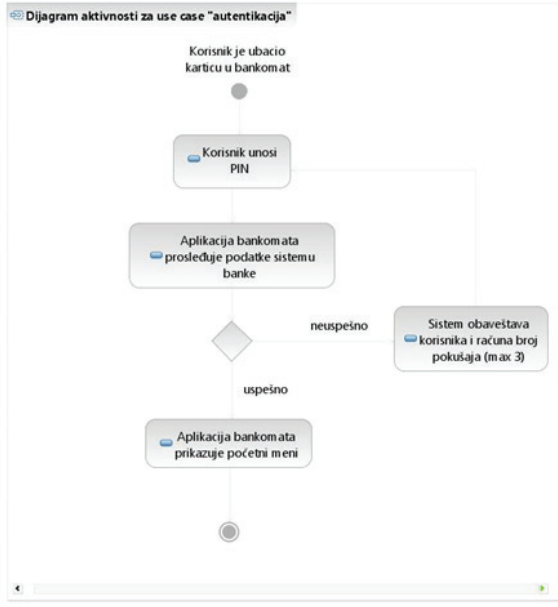
Tokom analize korisničkih zahteva, pored dogovora oko glavnih funkcionalnosti buduće aplikacije, neophodno je sa korisnikom definisati i opšte korake ili aktivnosti svakog pojedinačnog *use case*-a.

Dijagram aktivnosti predstavlja grafički prikaz toka rada i može se razumeti kao vizuelni algoritam koji prikazuje redosled izvršavanja aktivnosti u okviru jednog *use case*-a. Na primer, ukoliko se sa korisnikom definiše *use case* „plaćanje", neophodno je razložiti ovaj slučaj na konkretne korake, tj. na *workflow* koji opisuje kako se inicira, verifikuje i realizuje proces plaćanja. U praksi se dijagram aktivnosti koristi za razradu normalnih, alternativnih i izuzetnih scenarija izvršavanja *use case*-a, što omogućava detaljnije razumevanje funkcionalnih zahteva i potencijalnih odstupanja u toku rada.

Dijagram aktivnosti je izuzetno fleksibilan i primenljiv u različitim fazama razvoja softvera. Može se koristiti za opis:

- poslovnih procesa,
- tokova poslova (*workflow*),
- složenih slučajeva korišćenja (*use cases*),
- proceduralne i kontrolne logike,
- algoritama i drugih sekvencijalnih ili paralelenih aktivnosti.

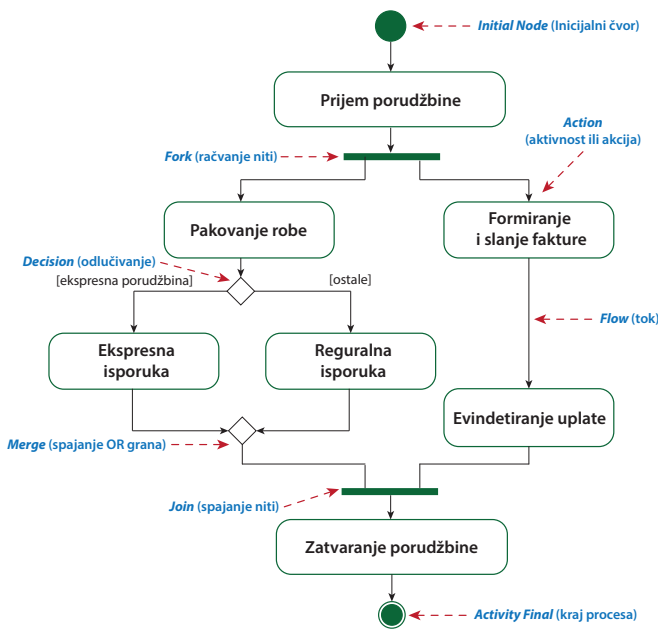
Zahvaljujući svojoj jasnoći i intuitivnosti, ovaj dijagram predstavlja jedan od najčešće korišćenih UML alata u komunikaciji između analitičara, korisnika i razvojnih timova.



Slika 6.18. Dijagram aktivnosti za *use case* „Autentikacija"

Osnovni elementi dijagrama aktivnosti su (Slika 6.19):

- Početni čvor (*initial node*) — označava početak toka.
- Akcije/aktivnosti (*action/activity nodes*) — zaokruženi pravougaonici koji predstavljaju izvršne korake.
- Kontrolni tokovi (*control flows*) — strelice koje povezuju aktivnosti i označavaju redosled.
- Odluke i spojevi (*decision/merge nodes*) — dijamanti za grananje toka i ponovno spajanje.
- Fork i join — za razdvajanje na paralelne tokove i njihovo sinhronizovanje.
- Objektni tokovi (*object nodes/object flows*) — prikazuju protok podataka između aktivnosti.
- Finalni čvor (*final node*) — označava završetak toka.
- Particije (*swimlanes*) — vertikalne ili horizontalne oblasti koje dodeljuju odgovornost (aktor, sistemski komponenti, uloga).

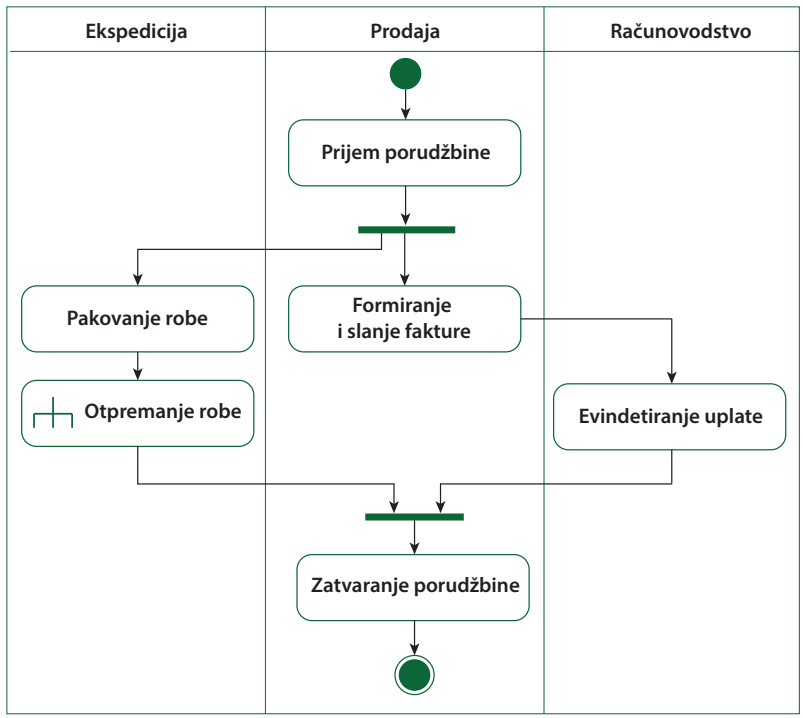


Slika 6.19. Osnovni elementi dijagrama aktivnosti

Aktivnosti se u dijagramu obično odvijaju sekvencijalno (jedna aktivnost započinje tek nakon što je prethodna završena). Međutim, ukoliko aktivnosti nisu zavisne jedna od druge i mogu se izvršavati paralelno, koristi se Fork Node (Čvor račvanja). Fork Node služi za račvanje toka kontrole i prikaz svih nezavisnih, paralelnih aktivnosti. Nakon završetka svih tih paralelnih aktivnosti, neophodno je koristiti Join Node (Čvor spajanja). Join Node je poseban jer čeka da se sve prethodne paralelne aktivnosti završe pre nego što omogući nastavak toka kontrole na narednu, zajedničku aktivnost.

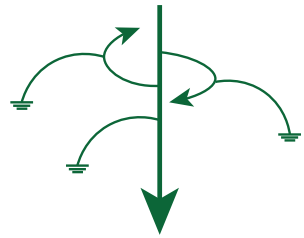
Treba obratiti pažnju da *Join node* (čvor spajanja) nije element bez funkcije. Za razliku od *Merge node*-a (čvor spajanja), čija je svrha samo da usmeri jedan tok kontrole nakon tačke odluke, *Join* ima dodatnu, kritičnu funkciju: on služi za sinhronizaciju, čekajući da se sve prethodne paralelne aktivnosti (koje su započete *Fork node*-om) završe pre nego što omogući nastavak toka na narednu zajedničku aktivnost.

Ukoliko se na prethodno prikazani dijagram primene particije (*Swimlanes*), rezultujući dijagram aktivnosti je prikazan na narednoj slici 6.20.



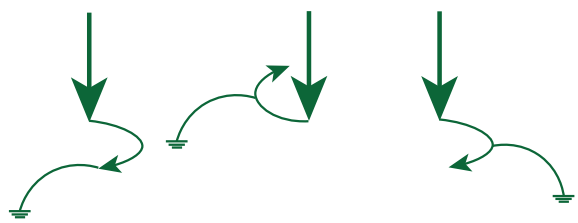
Slika 6.20. Particije (*swimlines*)

Jedan *use case* može imati više scenarija izvršavanja (*flows of events*), a oni se na dijagramu aktivnosti prikazuju kroz različite grane tokova, uslovne prelaze, *fork/join* čvorove i druge kontrolne konstrukte. Dijagram aktivnosti omogućava vizualizaciju kako osnovnog toka, tako i alternativnih i izuzetnih tokova izvršavanja.



Slika 6.21. Scenariji (tokovi događaja) jednog *use case*-a predstavljeni slikovito dijagramom aktivnosti

Scenario predstavlja instancu *use case*-a, odnosno jedan konkretan tok događaja. Scenario može obuhvatiti osnovni tok i pripadajuće alternativne ili izuzetne tokove. U praksi se detaljno razrađuju scenariji za one *use case*-ove koji nose visok rizik, imaju poslovni značaj ili utiču na ključne funkcionalnosti sistema. Scenariji su korisni i za razumevanje ponašanja sistema i kao osnova za validaciju specifikacije i sistemsko testiranje.



Slika 6.22. Različiti tokovi scenarija koji se kasnije mogu posebno prikazati dijagramom sekvenci

Primer: Scenariji ATM sistema

Osnovni scenario sistema bankomata (ATM) za podizanje gotovine obuhvata sledeće aktivnosti:

- **Provera kartice** – Klijent ubacuje karticu u bankomat. Sistem očitava i proverava da li je kartica važeća. Ako je kartica prihvatljiva, klijent se upućuje na unos PIN-a.
- **Provera PIN-a** – Klijent unosi PIN. ATM prosleđuje PIN bankarskom sistemu radi autentikacije. Ako je PIN ispravan, prikazuje se glavni meni sa vrstama dostupnih transakcija.
- **Izbor transakcije** – Klijent bira opciju „podizanje novca". Sistem prikazuje ponuđene iznose ili mogućnost unosa željenog iznosa.
- **Obrada podizanja novca** – Klijent bira iznos. ATM šalje zahtev za povlačenje sredstava bankarskom sistemu. Nakon potvrde da je iznos raspoloživ, prikazuje se opcija za štampanje potvrde o transakciji.
- **Završetak transakcije** – Sistem vraća karticu, isporučuje gotovinu i (po izboru klijenta) štampa potvrdu.

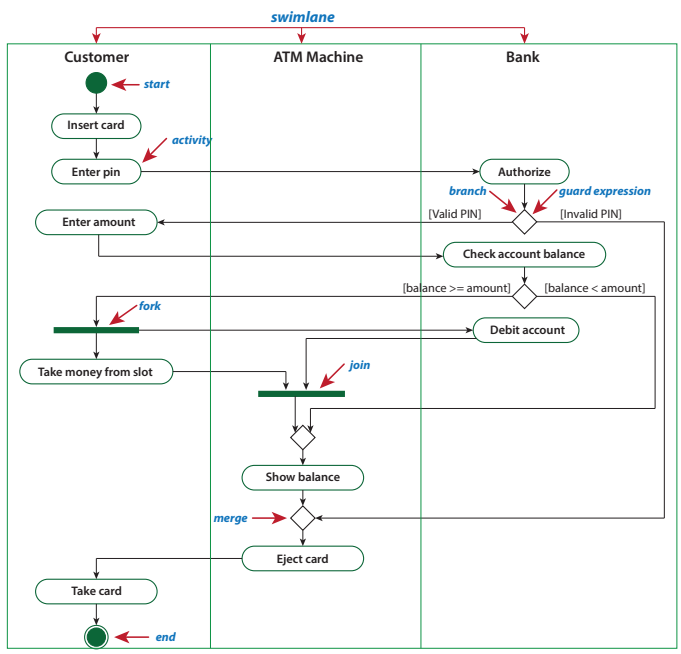
Alternativni scenariji za *use case* „Podizanje novca“ uključuju:

- **Kartica nije prihvatljiva** – Sistem odbacuje karticu i vraća je klijentu uz odgovarajuće obaveštenje.
- **Pogrešan PIN** – Prikazuje se poruka o grešci i omogućava ponovni unos. Nakon tri neuspešna pokušaja, kartica se vraća klijentu.
- **Prekid od strane klijenta** – Klijent može prekinuti transakciju u bilo kojoj fazi. U tom slučaju sistem poništava sve započete aktivnosti i vraća karticu.

Dijagram aktivnosti koji modeluje navedene tokove prikazan je na sledećoj slici 6.23.

Dokumentacija toka aktivnosti razvija se iterativno i inkrementalno, u skladu sa principima modernog softverskog inženjerstva i UML praksi. Tipičan proces obuhvata sledeće korake:

- Inicijalna razrada osnovnog toka – Na početku se formuliše kratak, visokonivojski opis ključnih koraka normalnog toka *use case*-a. Ovaj opis služi kao polazna osnova za modelovanje ponašanja.
- Proširivanje i detaljisanje koraka – Kako analiza napreduje i kako se prikupljaju dodatne informacije, osnovni tok se postepeno proširuje dodavanjem preciznijih aktivnosti, uslova, interakcija i definisanjem ulazno-izlaznih podataka.
- Identifikovanje i dodavanje alternativnih i izuzetnih tokova – Nakon stabilizacije osnovnog toka razmatraju se odstupanja, granični slučajevi i potencijalne greške, čime se definišu alternativni i izuzetni tokovi koji povećavaju potpunost i robustnost specifikacije.



Slika 6.23. Dijagram aktivnosti za *use case* “Podizanje novca sa bankomata”

Primer detaljno razrađene specifikacije jednog *use case*-a prikazan je na sledećim slikama.

1.0. Naziv use case-a (*Use case name*)

Odabir kurseva koji će predavati

1.1. Kraći opis use case-a (*Brief Description*)

Ovaj use case pokreće Profesor. Omogućava da profesor izabere do četiri kursa koja će predavatitokom izabranogsemestra.

2.0. Tok događaja (*Flow of Events*)

2.1. Osnovni tok (*basicflow*)

2.1.1. Prijava na sistem

Ovaj use case počinje kada se profesor prijavi na sistem za registraciju i unese svoju lozinku.
Sistem verifikuje da je lozinka valjana (ukoliko lozinka nije valjana izvršava se alternativni tok 2.2.1)

2.1.2. Unos tipa transakcije: Izbor semestra

Sistem signalizira da profesor izabere tekući ili budući semestar (ukoliko se unese pogrešan semestar, izvršava se alternativni tok 2.2.2).
Profesor unosi željeni semestar. Sistem signalizira da profesor izabere željenu aktivnost: add, delete, review, print ili quit.

2.1.3. Izbor aktivnosti: Add

Ukoliko je izabrana aktivnost Add, sistem prikazuje ekran sa poljem za unos naziva i broja kursa. Ukoliko profesor unese nevažeću kombinaciju ime/broj, izvršava se alternativni tok 2.2.3). Sistem prikazuje nuđene opcije za taj kurs (ako ime kursa ne može da se prikaže izvršava se alternativni tok 2.2.4).
Profesor bira neku od ponuda. Sistem povezuje tog profesora sa izabranom ponudom (ukoliko veza ne može da se napravi, izvršava se alternativni tok 2.2.5).

2.1.4. Izbor aktivnosti: Delete

Ukoliko je izabrana opcija Delete, sistem prikazuje ekran sa nuđenim opcijama za taj kurs sa poljem za ime opcije i za broj. Profesor unosi ime i broj opcije (ako se unese nevažeća kombinacija ime/broj, izvršava se alternativni tok 2.2.3). Sistem uklanja vezu sa tim profesorom (ako veza ne može da se ukloni, izvršava se alternativni tok 2.2.6).

2.1.5. Izbor aktivnosti: Review

Ukoliko je izabrana aktivnost Review, sistem poziva (ako informacija o kursu ne može biti pozvana, izvršava se alternativni tok 2.2.7) i prikazuje sledeće informacije o svim opcijama kurseva koje su njemu dodeljene: ime kursa, broj kursa, broj opcije kursa, dane u nedelji, vreme i lokaciju.

2.1.6. Izbor aktivnosti: Print

Ukoliko je izabrana aktivnost Print, sistem štampa profesorov raspored (ukoliko raspored ne može da bude odštampan, izvršava se alternativni tok 2.2.8).

2.1.7. Izbor aktivnosti: Quit

Ukoliko je izabrana aktivnost Quit, use case se završava.

Slika 6.24. Primer detaljne specifikacije jednog *use case*-a

2.2. Alternativni tokovi

2.2.1. **Nevažeća lozinka** - Unešena je nevažeća lozinka. Korisnik može ponovo da je unese ili da okonča use case.

2.2.2. **Nevažeći semestar** - Sistem obaveštava korisnika da je semestar nevažeći. Korisnik može ili ponovo da unese semestar ili da okonča use case.

2.2.3. **Nevažeće ime/broj kursa** - Sistem obaveštava korisnika da je nevažeće ime/broj kursa. Korisnik može ponovo da unese valjanu kombinaciju ili da okonča use case.

2.2.4. **Opcija kursa ne može da se prikaže** - Korisnik biva informisan da ta opcija nije trenutno dostupna. Use case ponovo počinje.

2.2.5. **Ne može da se uspostavi veza između profesora i opcije kursa** - Ova informacija je zapamćena i sistem će kasnije uspostaviti tu vezu.

2.2.6. **Veza između profesora i opcije kursa ne može da se ukloni** - Ova informacija ostaje zapamćena i sistem će kasnije ukloniti tu vezu.

2.2.7. **Ne može da se dobije informacija o rasporedu** - Korisnik se obaveštava da ova opcija trenutno nije dostupna.

2.2.8. **Raspored ne može da se odštampa** - Korisnik se obaveštava da ova opcija trenutno nije dostupna.

3.0. Specijalni zahtevi (*Special Requirements*)

Za ovaj use case ne postoje posebni zahtevi.

4.0. Preduslovi (*Preconditions*)

4.1. Pre nego što se pokrene ovaj use case mora biti završen podtok Kreiranje ponude kursa koji se nalazi u okviru use case-a Održavanje informacija o kursevima.

5.0. Post stanja (*Post Conditions*)

Ne postoje krajnji uslovi.

6.0. *Extension Points*

Ne postoje tačke proširivanja.

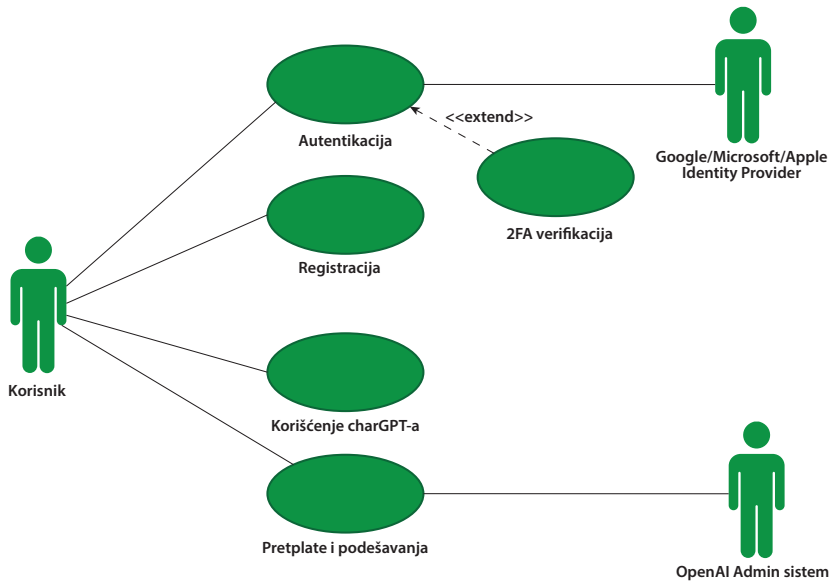
Slika 6.25. Alternativni tokovi

Vežba 6.3. Kreirati dijagram aktivnosti za *use case* „Autentikacija“ kod chatGPT sistema

Ukoliko ste prethodno kreirali *use case* dijagram za chatGPT sistem, koji sadrži glavne *use case*-ove, kao što je prikazano na slici 6.26.

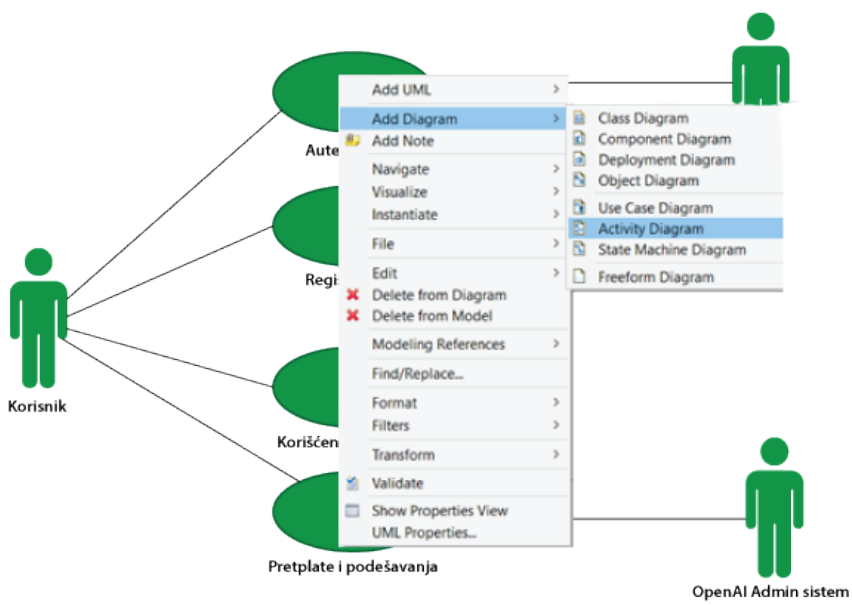
Učesnici (*Actors*) su:

- Korisnik – primarni akter koji koristi ChatGPT.
- Google/Microsoft/Apple Identity Provider – spoljni sistem za autentikaciju (SSO).
- Administrativni sistem OpenAI-a (opciono) – za upravljanje nalozima (npr. *subscription*).



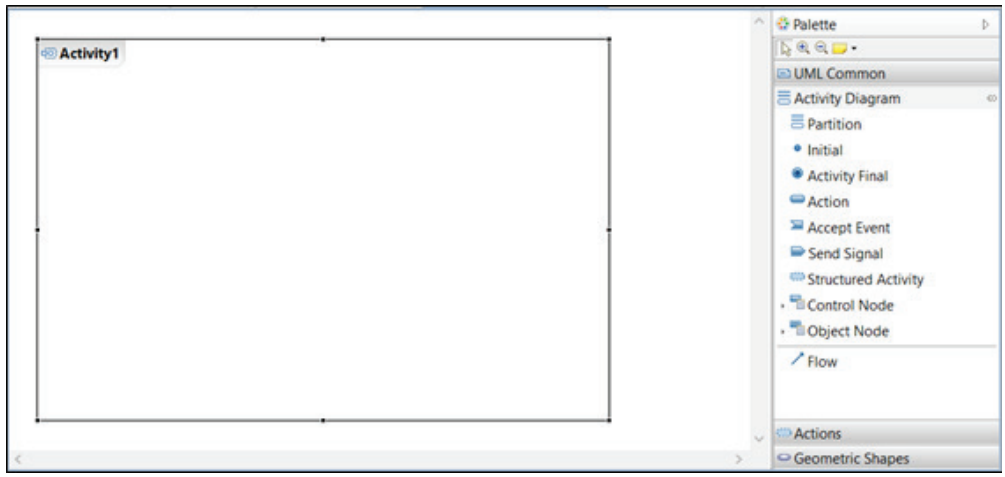
Slika 6.26. *Use case* dijagram chatGPT sistema

Potrebno je kliknuti na *use case* „Autentikacija" i desnim klikom odabrati *Add Diagram*, a zatim *Activity Diagram* (Slika 6.27).



Slika 6.27. Kreiranje dijagrama aktivnosti za konkretan *use case* prikazanog na use case dijagramu

Pojaviće se radno okruženje (Slika 6.28), sa paletom elemenata za kreiranje dijagrama aktivnosti.



Slika 6.28. Paleta UML elemenata za kreiranje dijagrama aktivnosti

Dijagram aktivnosti za proces autentikacije na ChatGPT sistem prikazan je na slici 6.29. Ukoliko korisnik izabere SSO (*Single Sign-On*) logovanje, sistem ga preusmerava na eksternog SSO provajdera. Na dijagramu se zatim vidi apstrakcija procesa: interni koraci autentikacije na strani provajdera (npr. provera lozinke i 2FA) se ne prikazuju, jer ChatGPT (*Relying Party*) nema uvid u njih. ChatGPT se oslanja na spoljni Identity Provider (IdP) i samo prima potvrdu o tome da li je korisnik uspešno autentifikovan ili ne. Time je obezbeđen glavni princip OAuth protokola: delegiranje provere identiteta bez razotkrivanja osetljivih akreditiva (lozinki) klijentskoj aplikaciji. Na primer, ukoliko je korisnik odabrao da se loguje preko svog *Google* naloga, tada proces logovanja funkcioniše preko *OAuth 2.0* protokola u kojem se Google i ChatGPT nalaze u sledećim ulogama:

- Google = *Identity Provider* (IdP) — Proverava lozinku, 2FA kodove itd.
- ChatGPT = *Relying Party* (RP) — Nikada ne vidi korisnikovu lozinku, već samo dobija potvrdu od Google-a: "korisnik je uspešno autentifikovan".

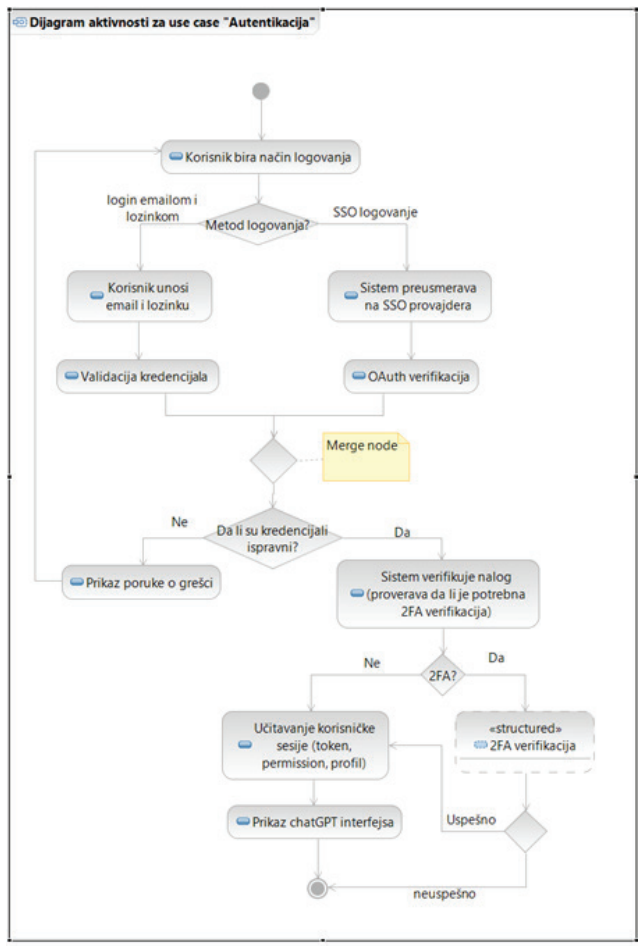
Konkretno, ChatGPT dobija od Google-a:

- OAuth token (kratkotrajni)
- ID token (sa osnovnim podacima: email, ime, Google user ID)

Životni ciklus lozinke, 2FA, sigurnosna pitanja i drugo, ostaje u nadležnosti Google-a i ChatGPT nema pristup tim podacima.

S obzirom da se nakon „Validacije kredencijala (email i lozinka)“ i „OAuth verifikacije“ nastavlja na isti tok aktivnosti, dodat je *Merge Node* (čvor spajanja). U UML-u, ovaj čvor nema nikakvu logiku i njegova jedina svrha je da prikupi dva ili više prethodnih tokova kontrole i usmeri ih dalje na istu aktivnost.

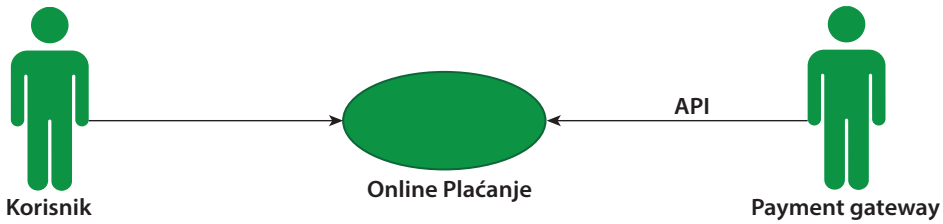
Ukoliko je potrebna dvofaktorska verifikacija (2FA), na dijagramu je prikazano da se poziva složeniji potproces 2FA verifikacija, čije su aktivnosti detaljno opisane posebnim dijagramom aktivnosti. Za predstavljanje ovakvih složenih potprocesa koristi se UML element *Structured Activity* (Strukturisana aktivnost). Rezultat iz tog potprocesa može biti uspešna ili neuspešna verifikacija. Ukoliko je 2FA verifikacija neuspešna, tu se tok procesa autentikacije mora prekinuti. S obzirom da neuspešna autentikacija prekida ceo proces (celu aktivnost), a ne samo jedan tok, zbog toga je prikazan element *Activity Final* (Kraj aktivnosti), a ne *Flow Final* (prekid jednog toka). Ukoliko je uspešna 2FA verifikacija, nastavlja se dalje sa učitavanjem sesije.



Slika 6.29. Dijagram aktivnosti za *use case* „Autentikacija“ korisnika na chatGPT sistem

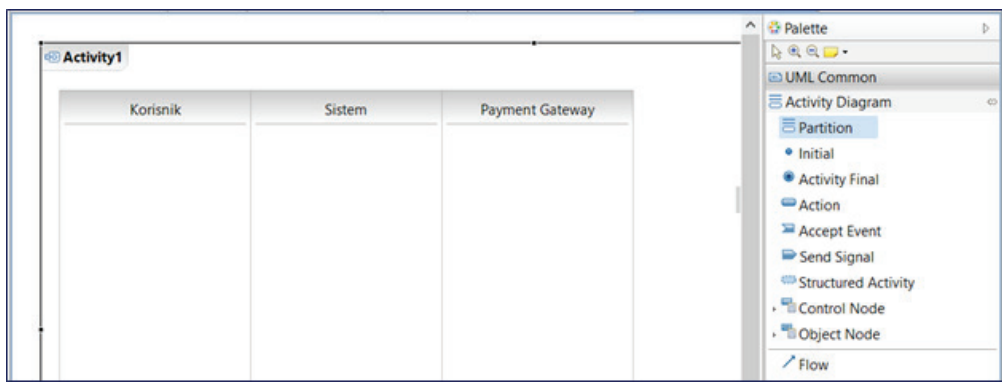
Vežba 6.4. Kreirati dijagram aktivnosti za *use case* „Plaćanje kreditnom karticom" koristeći particije (*swimlines*).

U procesu online plaćanja, komunikacija sa bankama se delegira eksternom finansijskom servisu poznatom kao *Payment Gateway* (ili procesor plaćanja). Na *use case* dijagramu to se prikazuje kao:



Slika 6.30. Jednostavan *use case* dijagram vezano za onlajn plaćanje

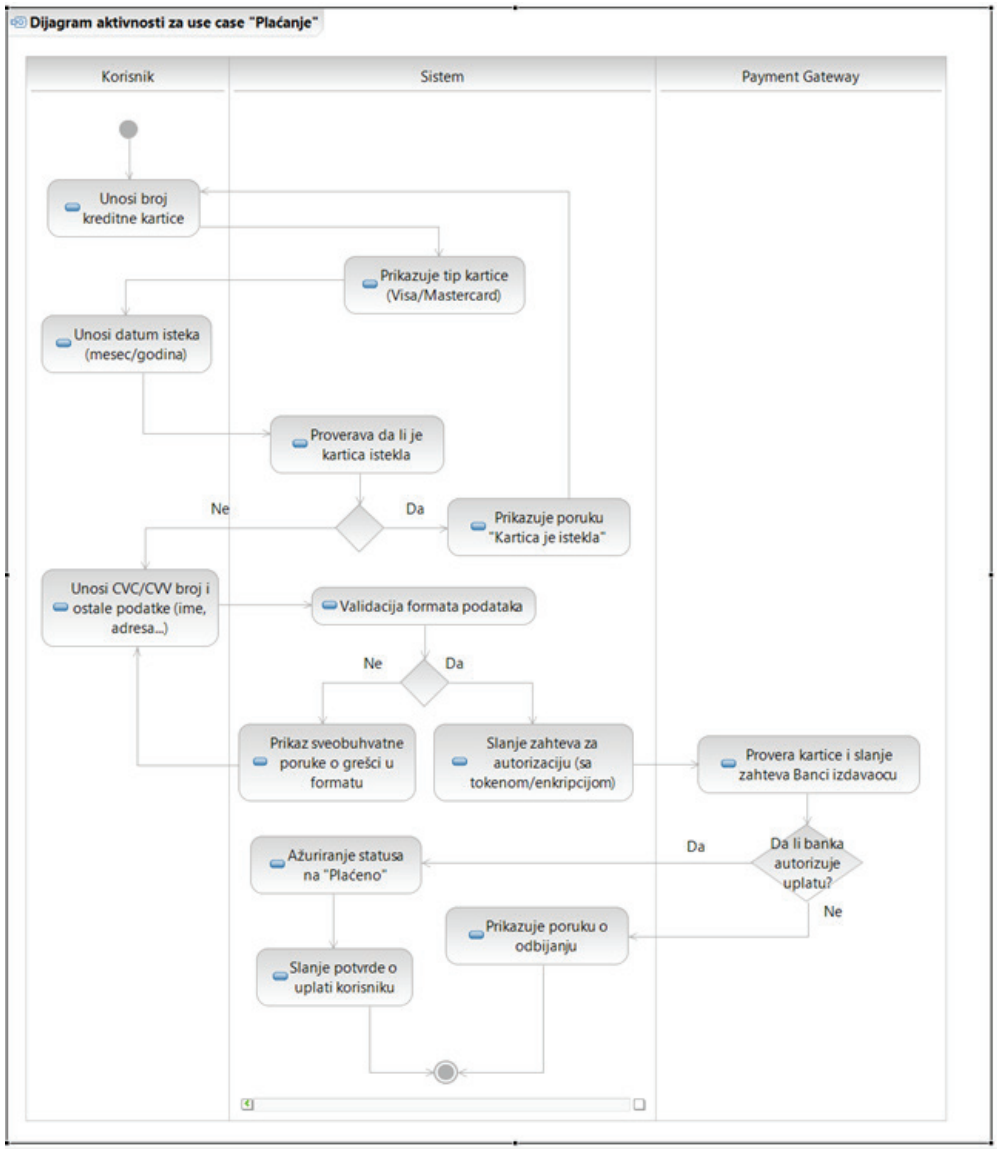
Desnim klikom miša na *use case* „Onlajn plaćanje", odaberite opciju *Add Diagram*, a zatim kliknite na *Activity Diagram*. Zatim, sa palete odabrati Partition i dodati ga na dijagram aktivnosti. Potrebno je kreirati tri particije: Korisnik, Sistem i Payment Gateway.



Slika 6.31. Dodavanje particija na dijagram aktivnosti

S obzirom da ste kreirali particije, nije neophodno da u nazivima aktivnosti pišete ko tu aktivnost obavlja, npr. umesto „Korisnik unosi podatke", dovoljno je da stoji „Unosi podatke", jer je ta aktivnost već grafički smeštena u particiju „Korisnik".

Postojeći dijagram aktivnosti za plaćanje može se proširiti kako bi uključio 3D Secure verifikaciju (npr. *Visa Secure*). Ukoliko je ta verifikacija neophodna, Banka Izdavalac (banka korisnika) generiše Jednokratnu lozinku (OTP) i šalje je korisniku putem SMS poruke. Korisnik zatim unosi taj kod u zaštićeni prozor (koji tehnički pripada domenu banke) prikazan unutar aplikacije. Ako je verifikacija uspešna, Banka Izdavalac obaveštava *Payment Gateway*, koji zatim signalizira sistemu (aplikaciji) da nastavi sa obradom transakcije.



Slika 6.32. Dijagram aktivnosti za *use case* „Plaćanje kreditnom karticom“

Zadatak 6.2. Kreirati dijagram aktivnosti za *use case* "Online zahtev za odobravanje kredita", koji prati sledeća scenarija:

- Klijent popunjava online zahtev za odobravanje kredita.
- Sistem verifikuje klijenta i zahtev.
  - Ukoliko su podaci i zahtev nevalidni, sistem obaveštava klijenta o nevalidnosti podataka i zahteva i tu se tok prekida.
  - Ukoliko su podaci i zahtev validni, sistem izvršava sledeće sinhronizovane aktivnosti:
    - Analiza platežne sposobnosti klijenta
    - Analiza imovine klijenta
    - Analiza namene kredita
- Sistem prosleđuje rezultate svih analiza Kreditnom odboru
- Kreditni odbor donosi odluku.
  - Ukoliko je rizik visok, sistem obaveštava klijenta o rezultatima odluke i tu se tok aktivnosti završava.
  - Ukoliko je rizik nizak, sistem generiše ugovor i obaveštava klijenta o rezultatima odluke.

6.4. DOKUMENT SPECIFIKACIJE SOFTVERSKIH ZAHTEVA

Svi prethodno spomenuti funkcionalni i nefunkcionalni zahtevi ne ostaju u „vazduhu", već se strukturirano objedinjuju u formalni dokument. **Specifikacija softverskih zahteva** (*Software Requirements Specification*, SRS) predstavlja „ustav" projekta koji sadrži sve zahteve, dijagrame i opise interfejsa. On služi kao ključni komunikacioni kanal i ugovor između klijenta i razvojnog tima (Pressman & Maxim, 2020).

Da bi SRS bio upotrebljiv u inženjerskoj praksi, svaki zahtev unutar njega mora ispunjavati ključne kriterijume kvaliteta (Sommerville, 2016):

- **Jednoznačnost** (*Unambiguous*): Svaki zahtev mora imati samo jedno moguće tumačenje za sve zainteresovane strane.
- **Kompletnost** (*Complete*): Dokument mora obuhvatiti sve funkcije, performanse, hardverska ograničenja i atribute kvaliteta sistema.
- **Proverljivost** (*Verifiable*): Mora postojati objektivan proces (testiranje ili demonstracija) kojim se može utvrditi da li je finalni softver ispunio zahtev.
- **Doslednost** (*Consistent*): Zahtevi ne smeju biti u kontradikciji; podaci i termini moraju biti ujednačeni kroz ceo dokument.
- **Modifikabilnost** (*Modifiable*): Struktura dokumenta treba da dozvoli izmene bez narušavanja logičke celovitosti sistema.

Iako se forma dokumenta prilagođava izabranoj metodologiji, tipičan SRS sadrži sledeće standardne celine (IEEE, 1998):

Sekcija	Sadržaj i namena
1. Uvod	Svrha softvera, obuhvat projekta, definicije i skraćenice.
2. Opšti opis	Perspektiva proizvoda, korisničke karakteristike, opšta ograničenja i pretpostavke.
3. Specifični zahtevi	Detaljan opis **funkcionalnih zahteva** (često kroz *Use case* opise) i **nefunkcionalnih zahteva** (performanse, sigurnost).
4. Interfejsi	Opis korisničkog interfejsa (UI), hardverskih i softverskih interfejsa, kao i komunikacionih protokola.
5. Modeli i dijagrami	Integrisani UML modeli poput dijagrama aktivnosti koji vizuelizuju kompleksne tokove rada.

Jedna od najvažnijih uloga SRS-a je uspostavljanje **matrice sledljivosti** (*Requirements Traceability Matrix*). Svaki zahtev dobija svoj jedinstveni identifikator (npr. REQ-001), što omogućava arhitektama da prate koji deo koda realizuje konkretan zahtev, a testerima da potvrde pokrivenost zahteva test-slučajevima (Wiegers & Beatty, 2013).

Savremena specifikacija softverskih zahteva danas se oslanja na standard **ISO/IEC/IEEE 29148:2018**, koji je evoluirao iz ranijeg IEEE 830 standarda pružajući bolji okvir za verifikaciju i validaciju u složenim sistemima (ISO/IEC/IEEE, 2018). Za praktičan rad, preporučuje se korišćenje standardizovanog šablona koji osigurava kompletnost dokumentacije. Primer takvog šablona dostupan je na adresi: https://www.cse.msu.edu/~cse870/IEEEXplore-SRS-template.pdf.

POGLAVLJE 7



OBJEKTNO-ORIJENTISANA ANALIZA SOFTVERA

Objektno-orijentisana analiza softvera

7.1. Arhitektura softvera
7.2. Dijagrami interakcija
7.3. Dijagram klasa

Cilj poglavlja:

Nakon razgovora sa korisnikom i kreiranja početnih modela koji reflektuju potrebe i zahteve klijenata, prelazi se u fazu analize softvera. Prvi korak u analizi je da se tim dogovori oko arhitekture softvera, a potom da kreiraju i testiraju logičke modele aplikacije, koje će kasnije, u fazi projektovanja softvera, biti prevedene u fizičke modele. Dakle, cilj ovog poglavlja je da se razume i prepozna odgovarajuća arhitektura softvera, prema kojoj će se kreirati dijagrami sekvenci i na kraju sagledati klase aplikacije i njihove veze na dijagramu klasa.

7. OBJEKTNO-ORIJENTISANA ANALIZA SOFTVERA

7.1. ARHITEKTURA SOFTVERA

Nakon razgovora sa korisnikom i kreiranja početnih modela (*use case* dijagrama i dijagrama aktivnosti), softver arhitekta zajedno sa razvojnim timom dogovara okvir po kome će se razvijati aplikacija, odnosno definiše i postavlja arhitekturu softverskog rešenja. Arhitektura predstavlja dobar temelj buduće aplikacije koja se razvija, kao i dogovor tima kako će podeliti posao i razvijati aplikaciju.

Postoji puno šablona (*patterns*) arhitekture softvera, koji se mogu preuzeti u nekoj početnoj fazi, da bi se kasnije taj šablon prilagodio specifičnostima buduće aplikacije. Svaki šablon je namenjem razvoju određene vrste aplikacije. Na primer, ako se razvija interaktivna aplikacija, gde će aplikacija uglavnom reagovati na zahteve korisnika (korisnik će svojim akcijama uticati na tok izvršavanja aplikacije, a aplikacija će odmah reagovati na te akcije), tada je preporuka da se koristi šablon *Model-View-Controller* (MVC) arhitektura.

Arhitektura nije statička struktura aplikacije, već se posmatra kao dinamičan okvir koji usmerava razvoj, održavanje i dalju nadogradnju sistema tokom celog životnog ciklusa.

Arhitektura softvera sadrži nekoliko ključnih komponenti:

- **Strukturni elementi sistema** - Ovi elementi predstavljaju osnovne građevinske blokove softvera. Najčešće se radi o modulima, komponentama, servisima, slojevima ili podsistemima koji zajedno obrazuju celinu. Svaki od njih poseduje sopstvenu odgovornost i interfejs preko kojeg komunicira sa drugim delovima sistema.
- **Relacije i interakcije između komponenti** - Arhitektura eksplicitno definiše kako se elementi međusobno povezuju. Način na koji su interakcije organizovane utiče na performanse, integraciju i buduću evolutivnost sistema.
- **Principi i smernice za projektovanje** - Ovo su apstraktnija pravila koja oblikuju arhitekturu, kao što su razdvajanje odgovornosti, hijerarhijska organizacija, ograničavanje zavisnosti ili definisanje jasnih interfejsa i dr. Ovi principi služe kao vodič za celokupni razvojni tim.
- **Skup kritičnih odluka** - arhitektura je pre svega set najvažnijih odluka donesenih u ranom stadijumu razvoja, a koje će imati dugoročne posledice. Odluke koje se odnose na izbor stilova arhitekture (npr. mikroservisi vs. monolit), protok podataka, integracione mehanizme ili tehnologije često su vrlo skupe za promenu kasnije u projektu.
- **Evolutivni karakter arhitekture** - U praksi, arhitektura se retko definiše jednom za sva vremena. Ona se razvija kako sistem raste i kako se menjaju zahtevani funkcionalni i nefunkcionalni kriterijumi. Savremene agilne metodologije podstiču iterativni pristup arhitekturi, gde se odluke donose postepeno, u skladu sa potrebama i rizicima.

Arhitektura softvera zauzima **centralno mesto u procesu razvoja** jer utiče na kvalitet sistema, međusobnu komunikaciju timova, tehnološke odluke i organizacione aspekte projekta. Ona služi kao veza između poslovnih ciljeva i tehničke realizacije. Jasno definisana arhitektura pomaže u prevazilaženju nedoumica, posebno kada učesnici dolaze iz različitih domena znanja. Vizuelni dijagrami, opis komponenti i standardizovani modeli olakšavaju planiranje i razumevanje sistema.

Arhitektura direktno **utiče i na nefunkcionalne zahteve**, kao što su: performanse, skalabilnost, pouzdanost, sigurnost, održivost i interoperabilnost. Odluka o arhitekturi softvera određuju kako će se sistem ponašati pod opterećenjem, kako će reagovati na greške, koliko će biti jednostavno prilagoditi ga novim zahtevima. Na primer, ukoliko se odabere mikroservisna arhitektura, podela aplikacije na mikroservise može olakšati održavanje, ali može povećati složenost integracije. Veći softverski sistemi obično imaju veliki broj međuzavisnih delova, te arhitektura pomaže u strukturiranju ove složenosti definisanjem granica, hijerarhija i veza među delovima sistema. Na primer, slojevita arhitektura omogućava jasan prelaz između prezentacionog, poslovnog i dela podataka sistema. Dakle, izbor adekvatne arhitekture omogućava identifikovanje i minimiziranje rizika pre nego što sistem uđe u fazu implementacije.

Arhitektura obezbeđuje **da tehničko rešenje bude usklađeno sa strateškim ciljevima organizacije**. Ukoliko su poslovni ciljevi usmereni na brzu isporuku funkcionalnosti, arhitektura mora podržati iterativni razvoj. Ukoliko je prioritet skalabilnost, arhitektura će biti dizajnirana sa fokusom na raspodelu opterećenja.

Arhitektura softvera predstavlja fundamentalnu organizacionu strukturu sistema. Ona predstavlja skup osnovnih odluka o tome šta će sistem raditi, kako će biti podeljen na delove i kako će ti delovi komunicirati. Arhitektura je most između zahteva sistema i njegove konačne implementacije. Ključni značaj arhitekture:

- **Komunikacija sa stejkholderima**: Služi kao apstraktni model koji omogućava svim zainteresovanim stranama (developerima, menadžerima, klijentima) da razumeju sistem.
- **Odluke o organizaciji sistema**: Arhitektonske odluke utiču na odabir tehnologije, programskih jezika i okvira (*frameworks*).
- **Atributi kvaliteta** (*Quality Attributes*): Direktno utiče na performanse sistema, sigurnost, skalabilnost i održivost. Kroz planiranje arhitekture, tim mora balansirati između konkurentnih zahteva kvaliteta (Tabela 7.1).

Tabela 7.1. Opis atributa kvaliteta softvera i primer izabranih taktika

Atributi kvaliteta	Definicija	Primer rešenja
Skalabilnost	Sposobnost sistema da se nosi sa povećanjem opterećenja.	Korišćenje distribuiranog keširanja ili *load balancing*-a (distribucija opterećenja).
Sigurnost	Sposobnost sistema da štiti informacije od neovlašćenog pristupa.	Korišćenje *Gatekeeper* šeme (obavezna autorizacija kroz jednu tačku).
Održivost	Lakoća izmene, popravke ili prilagođavanja sistema.	Korišćenje modularnog dizajna i labavog povezivanja.
Performanse	Sposobnost sistema da odgovori na zahteve u definisanom vremenskom roku.	Upotreba asinhronih operacija (*Queuing*) ili optimizacija algoritama.

Kvalitetna arhitektura omogućava da se sistem lako proširi, refaktoriše ili integriše sa novim tehnologijama. **Slaba arhitektura**, s druge strane, dovodi do tzv. arhitektonske erozije — postepenog gubitka konzistentnosti, što može imati ozbiljne posledice na održavanje i kvalitet sistema.

7.1.1. Stilovi i obrasci arhitekture softvera

Stilovi i obrasci arhitekture su opšta, ponovljiva rešenja koja se primenjuju na česte probleme u razvoju softvera. Stilovi arhitekture (*Architecture Styles*) više ukazuju na konceptualni model organizacije softverskog sistema, dok su obrasci arhitekture (*Architectural patterns*) konkretna rešenja problema. Na primer, ukoliko se odabere stil arhitekture „slojeviti sistem", onda se u okviru tog stila može koristiti MVC obrazac arhitekture.

Stil arhitekture predstavlja visokonivojski, konceptualni tip organizacije softverskog sistema, koji propisuje osnovne vrste komponenti, njihove interakcije i pravila povezivanja. Drugim rečima, stil opisuje kako je ceo sistem organizovan i koje vrste elemenata u njemu postoje.

Osnovne karakteristike stilova su:

- Rade na najvišem nivou apstrakcije.
- Definišu tipičnu strukturu celog sistema.
- Ograničavaju način na koji komponente mogu biti povezane.
- Fokus im je na globalnoj organizaciji.

Odabir stila, a samim tim i obrasca zavisi od primarnih zahteva kvaliteta (npr., da li je kritična skalabilnost, sigurnost ili brzina razvoja). Ključne prednosti pojednih stilova ili obrazaca su navedene u tabeli 7.2.

Tabela 7.2. Ključne prednosti određenih stilova ili obrazaca arhitekture softvera

Stil/Obrazac	Opis	Ključna prednost
Slojevita arhitektura (*Layered Architecture*)	Podela sistema na horizontalne slojeve (prezentacija, poslovna logika, pristup podacima). Tok kontrole je jednosmeran.	**Održivost i testiranje**. Svaki sloj je nezavisan i zadužen za specifičnu funkciju (npr., Troslojna arhitektura).
Klijent-Server arhitektura	Komponente su podeljene na pružaoce usluga (Server) i potrošače usluga (Klijent).	**Centralizovana kontrola i skladištenje podataka.**
Monolitna arhitektura (*Monolithic*)	Sva funkcionalnost je objedinjena u jednoj aplikacionoj jedinici.	**Jednostavnost razvoja i implementacije** (za male projekte).
Mikroservisna arhitektura (*Microservices*)	Sistem je sastavljen od skupa malih, nezavisnih servisa, koji komuniciraju preko laganih protokola (npr. REST API).	**Skalabilnost, fleksibilnost i otpornost na greške.**
Arhitektura vođena događajima (*Event-Driven*)	Komponente asinhrono komuniciraju preko događaja (*Events*) putem kanala (*Message Queues*).	**Visoka reaktivnost i paralelnost.**

Za razliku od opštih stilova, **obrasci arhitekture softvera** (*Architectural patterns*) predstavljaju dokazano, ponovljivo konkretno rešenje određenog problema u kontekstu arhitekture sistema.

Karakteristike obrazaca arhitekture ili arhitektonskih obrazaca su:

- Manje su apstraktni od stilova
- Rešavaju konkretne probleme projektovanja softvera
- Obično se koriste unutar arhitektonskih stilova, a ne umesto njih
- Uključuju opis problema, konteksta, rešenja i posledica primene.

Primeri arhitektonskih obrazaca (paterna) su:

- MVC (*Model–View–Controller*)
- PAC (*Presentation–Abstraction–Control*)
- Microkernel pattern
- Broker pattern
- Pipes-and-Filters kao obrazac (kada se koristi za deo sistema)
- CQRS (*Command Query Responsibility Segregation*)
- Event Sourcing

Većina obrazaca arhitekture se mogu videti na narednoj slici 7.1, gde su grupisani prema stilovima arhitekture.

VRSTA APLIKACIJE / SISTEMA		PREPORUČENI OBRASCI ARHITEKTURE SOFTVERA	OPIS I PRIMENA
	Interaktivni sistemi (Web i mobilne aplikacije)	• MVC (*Model-View-Controller*) • MVVM • PAC (*Presentation-Abstraction-Control*)	Fokus je na odvajanju logike od prikaza kako bi se omogućilo lako ažuriranje korisničkog interfejsa bez menjanja poslovne logike.
	Distribuirani sistemi (*Cloud* i *Enterprise* rešenja)	• Microservices • Broker • Client-Server • SOA	Omogućavaju sistemu da bude sastavljen od nezavisnih komponenti koje komuniciraju preko mreže, čime se postiže visoka dostupnost.
	Sistemi za obradu podataka (Big Data, BI i drugi analitički sistemi)	• Pipes and Filters • Batch Sequential	Idealni za sisteme gde podaci prolaze kroz niz transformacija. Svaki "filter" obavlja jedan zadatak pre slanja podataka dalje.
	Sistemi sa deljenom memorijom (AI, Ekspertni sistemi)	• Blackboard • Shared Repository	Koriste se kada više nezavisnih komponenti (agenata) treba da radi na rešavanju složenog problema koristeći zajedničku bazu znanja.
	Sistemi sa implicitnim pozivom (Event-driven)	• Publish-Subscribe	Komponente ne pozivaju jedna drugu direktno, već reaguju na događaje (events). Ključno za moderne "real-time" aplikacije.
	Sistemi visokih performansi (Streaming, Gaming)	• Reactor • Proactor • Non-blocking IO	Fokusirani na efikasno upravljanje resursima i istovremenu obradu velikog broja zahteva bez zagušenja procesora.
	Adaptibilni i sistemski softver	• Microkernel • Reflection • Proxy	Omogućavaju proširivost sistema putem dodataka (plug-ins) i skrivanje kompleksnosti pristupa resursima.

Slika 7.1. Paterni arhitekture prema stilovima arhitekture

7.1.2. Popularni obrasci arhitekture softvera

U praksi se najčešće koriste sledeći arhitektonski obrasci:

- **Model–View–Controller** (MVC) - Jedan od najpoznatijih obrazaca. Razdvaja prikaz podataka (*View*), poslovnu logiku (*Controller*) i model podataka (*Model*) sa ciljem da se olakša razvoj, održavanje i testiranje aplikacije. Koristi se u veb aplikacijama (npr. Django, Ruby on Rails) i desktop aplikacijama.
- **Model–View–ViewModel** (MVVM) - Slično MVC-u, ali uvodi ViewModel kao most između View-a i Modela. Koristi se u mobilnim aplikacijama (npr. Android), modernim UI framework-ovima (npr. React, Angular, WPF).
- **Microkernel Pattern** - Deljenje sistema na osnovni (kernel) i dodatne plug-in module. Koristi se u IDE-ovima (npr. Eclipse), OS arhitekturama, sistemima koji se lako proširuju.
- **Broker Pattern** - Upravlja komunikacijom između distribuiranih komponenti preko posrednika (brokera). Koristi se u distribuiranim sistemima, middleware platformama (npr. CORBA).

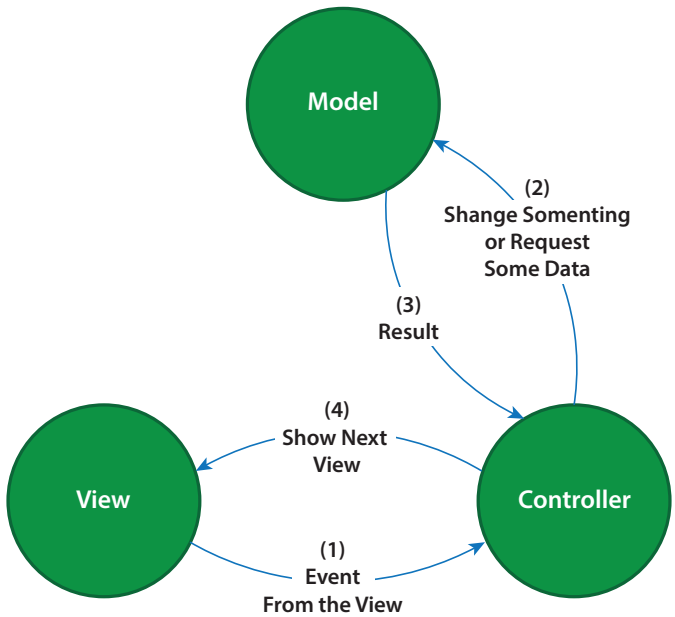
- **Pipes and Filters Pattern** - Sekvenca komponenti (filtera) kroz koje podaci prolaze putem cevi (pipes). Koristi se u obradi podataka, ETL sistemima, streaming aplikacijama.
- **Event-Bus Pattern** - Komponente reaguju na objavljene događaje preko centralnog event bus-a. Koristi se u aplikacijama u realnom vremenu, UI framework-ovima, mobilnim aplikacijama.
- **Event Sourcing** - Sve promene stanja ne čuvaju se kao gotov rezultat, već kao niz događaja koji se mogu ponovo proigrati. Koristi se u: finansijskim sistemima, sistemima koji zahtevaju audit trail.
- **CQRS** (*Command Query Responsibility Segregation*) - Razdvajanje komandi (write operacija) i upita (read operacija) u dva dela sistema. Često se kombinuje sa Event Sourcing-om. Koristi se u sistemima visokih performansi i skalabilnosti.
- **Blackboard Pattern** - Multiple specijalizovane komponente sarađuju rešavajući problem preko zajedničke "tabla" strukture. Koristi se u ekspertnim sistemima, AI (Artificial Intelligence) rešenjima i sl.
- *Client–Proxy Pattern* (*Remote Proxy*) - Omogućava lokalnoj komponenti da koristi udaljeni servis preko proxy-ja koji "maskira" udaljenost. Koristi se u distribuiranim sistemima, mikroservisima, gRPC i RMI okruženjima.
- **Mikroservisi** - Najpoznatiji obrasci mikroservisne arhitekture su:
  - **API *Gateway Pattern*** - Upravlja svim ulaznim pozivima ka mikroservisima kroz jedinstvenu tačku.
  - ***Service Registry & Discovery*** - Servisi se automatski pronalaze i međusobno povezuju.
  - **Circuit Breaker Pattern** - Sprečava propadanje sistema pri greškama u mreži — prekida pozive ka problematičnom servisu.

S obzirom da obrazaca arhitekture softvera ima puno, u nastavku teksta će se detaljnije opisati MVC obrazac.

Model View Controller obrazac arhitekture

MVC se sastoji od tri ključne komponente (Slika 7.2):

- ***View*** – obezbeđuje korisnički interfejs preko koga korisnik komunicira sa sistemom.
- ***Controller*** – upravlja izvršavanjem sistemskih operacija. Prihvata zahtev od klijenta i izvršava zadatke:
  - ažurira *Model* i *View*, na osnovu korisničkog inputa
  - poziva izabrani Model i izvršava logiku, npr., dok ažurira podatke zaposlenih, *Controller* odlučuje koju veb stranu treba prikazati.
- ***Model*** – predstavlja podatke i poslovnu logiku sistema. Upravlja pristupom podacima, pravilima poslovanja i stanjima aplikacije. Ova komponenta sadrži strukturu sistema tj. podatke i njene operacije (metode).



Slika 7.2. MVC arhitektura softvera

Zahvaljujući ovom razdvajanju, moguće je menjati prikaz bez menjanja logike, menjati logiku bez uticaja na interfejs i lakše testirati svaki deo posebno. MVC se široko koristi u web i desktop aplikacijama.

Primer: Spring MVC

Spring MVC je modul, unutar Spring framework-a, koji služi za razvoj web aplikacija koristeći MVC obrazac. Primer tok rada Spring MVC aplikacije:

1. Korisnik šalje HTTP zahtev (npr. GET/POST).
2. *DispatcherServlet* (glavni front controller Spring MVC-a) prima zahtev.
3. *DispatcherServlet* prosleđuje zahtev odgovarajućem *Controller*-u.
4. Controller obrađuje zahtev, pristupa Model-u ili servisima i priprema podatke.
5. Podaci se prosleđuju u *View*, koji generiše HTML ili JSON odgovor.
6. Odgovor se šalje korisniku.

Detaljnije karakteristike Spring MVC u okviru Spring okvira su opisane u tabeli 7.3.

Tabela 7.3. Pregled karakteristika *Spring Framework*-a i *Spring MVC*-a

Karakteristika	Spring framework (Core/običan)	Spring MVC
Definicija	Osnovni framework za razvoj Java aplikacija sa fokusom na modularnost, IoC/DI i AOP.	Modul unutar Spring framework-a za razvoj web aplikacija koristeći MVC obrazac.
Cilj	Olakšava upravljanje objektima, zavisnostima, transakcijama i integraciju sa resursima.	Razdvaja aplikaciju na Model, View i Controller i olakšava razvoj interaktivnih web aplikacija.
Glavne komponente	Core/Beans, Context, AOP, Data Access, Transaction.	DispatcherServlet, Controller, Model, View, View Resolver.
Tip aplikacija	Desktop, enterprise i backend sistemi.	Web aplikacije, REST API-ji, veb servisi.
Arhitektonski obrazac	Ne primenjuje MVC sam po sebi (ali podržava modularni dizajn).	MVC (*Model–View–Controller*).
Primer tehnologija	JDBC, JPA, JMS, AOP, *Dependency* Injection.	JSP, Thymeleaf, REST, JSON/ HTML prikazi.
Veza sa drugim modulima	Osnova za sve druge Spring module (MVC, Security, Boot).	Zavisi od Spring Core za DI i konfigurisanje bean-ova.

Napomene:

- *Controller* je samo posrednik između korisničkog interfejsa *(UI/View*) i ostatka aplikacije. Njegova uloga je da primi zahtev, eventualno proveri osnovne ulazne podatke (validacija), i prosledi ih *Service* sloju. *Controller* **ne treba da sadrži poslovna pravila, izračunavanja, popuste, validaciju stanja u bazi itd**. Tipične funkcije *controller*-a:
  - Prima HTTP/REST zahtev (npr. POST /cart/add)
  - Parsira podatke iz zahteva (request body, query param)
  - Poziva odgovarajući *service* metod
  - Prima rezultat iz servisa
  - Vraća odgovor (HTTP 200, JSON sa statusom, grešku)
- *Model* je sloj aplikacije koji čuva podatke i poslovna pravila. On sadrži stanje aplikacije, obrađuje logiku poslovanja i obaveštava *View* o promenama kada dođe do izmene podataka. Istovremeno, Model ne zavisi ni od View-a ni od Controller-a i samostalno izvršava sve zadatke vezane za podatke i pravila aplikacije. Model bi se u praksi podelio na tri stvari:
  - *Service* – predstavlja poslovnu logiku (pravila)
  - *Repository* – pristup bazi (CRUD – *create, read, update, delete*)
  - *Entity/Domain Model* – predstavljanje podataka

Dakle, Model u praksi = Entity + Repository + Service. Zato *Service* jeste deo Model-a, ali ne Model-a kao „tabela u bazi", već Model-a kao celokupnog domena aplikacije.

Kada se primenjuje MVC arhitektura na neku aplikaciju, tada se ona sastoji iz sledećih klasa:

- *View* – prikaz korisničkog interfejsa i prezentacija podataka korisniku.
- *Controller* – prima korisničke zahteve, prosleđuje ih *Service* sloju i vraća rezultat *View*-u; ne sadrži poslovnu logiku.
- *Service* – deo modernog *Model* sloja; sadrži poslovna pravila, izračunavanja, validacije i „kako aplikacija misli".
- *Repository* – deo modernog Model sloja; radi sa bazom podataka i čuva podatke („kako podaci fizički žive").

Dakle, *Service* i *Repository* su deo Model sloja, ali su izdvojeni radi čiste arhitekture i lakšeg testiranja. U originalnom MVC-u Model je sveobuhvatan i obuhvata i poslovnu logiku i pristup podacima.

Kratka analogija:

- *Controller* = konobar → prima porudžbinu i prenosi je kuhinji
- *Service* = kuvar → pravi jelo prema pravilima kuhinje
- *Repository* / DB = magacin + kuhinjski inventar → gde se čuvaju sastojci

Dakle, konobar ne kuva, kuvar ne priča sa gostima, a magacin ne odlučuje o receptu.

Imajući sve ovo u vidu, navode se konkretnije karakteristike elemenata MVC arhitekture:

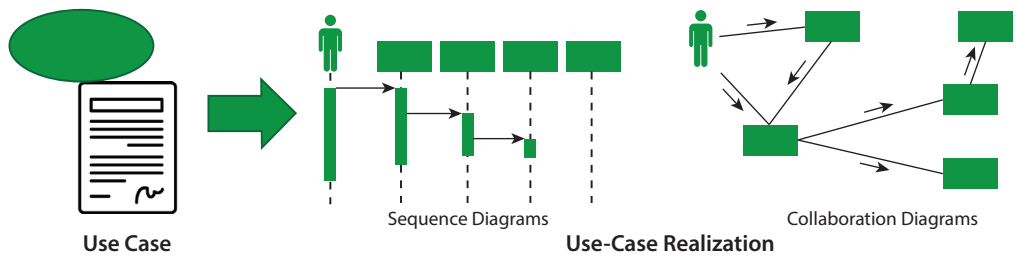
- *View*
  - Odgovoran je za prikaz korisničkog interfejsa.
  - Ne sadrži poslovna pravila.
  - Primeri: web stranica, mobilna aplikacija, desktop GUI.
- *Controller*
  - Posrednik između korisnika i aplikacije.
  - Prima korisničke zahteve (klik, forma, REST API poziv).
  - Validira osnovne podatke (npr. da li je polje prazno, da li je ID broj).
  - Prosleđuje zahtev Service sloju (ili direktno Model-u u teorijskom MVC-u).
  - Ne sadrži poslovnu logiku.
  - Primeri: Spring MVC @Controller, Django view funkcija, ASP.NET Controller.
- *Model*
  - Sadrži poslovnu logiku i podatke.
  - U modernom web razvoju Model je često podeljen:
    - *Entity / Domain Model* - struktura podataka (npr. ShoppingCart, Product).
    - *Service / Business Logic* - pravila aplikacije, izračunavanja, validacije.
    - *Repository* / DAO → rad sa bazom podataka (CRUD operacije).
  - Dakle, *Service* sloj i *Repository* su deo *Model*-a u modernom MVC-u.

7.2. DIJAGRAMI INTERAKCIJA

Nakon definisanja arhitekture sistema, pristupa se identifikovanju klasa buduće aplikacije i njihovih odgovornosti i operacija. U ranim fazama analize, jedan od prvih UML dijagrama koji se izrađuje jeste dijagram sekvenci (*Sequence Diagram*). Ovaj dijagram modeluje hronološki redosled poruka koje objekti razmenjuju tokom izvršavanja **jednog *use case*-a**, čime se precizno opisuje dinamika ponašanja sistema u interakciji sa korisnikom.

Dijagram sekvenci vizuelizuje jedan specifičan scenario – odnosno, konkretan tok događaja koji ilustruje kako sistem reaguje na određeni zahtev korisnika. Na taj način omogućava razumevanje međusobnih zavisnosti, redosleda akcija i odgovornosti pojedinačnih objekata u okviru poslovnog procesa.

Za svaki identifikovani *use case* može se kreirati jedan ili više dijagrama sekvenci i/ili dijagrama kolaboracije (*Collaboration Diagram*). Ovi dijagrami, kao deo iste grupe UML dijagrama, nazivaju se **dijagrami interakcije**, jer **modeluju tok komunikacije između instanci klasa**. Iako prikazuju iste informacije, razlikuju se po fokusu: dijagram sekvenci naglašava vremenski redosled poruka, dok dijagram kolaboracije više ističe strukturne veze i organizaciju objekata koji učestvuju u interakciji.



Slika 7.3. Dijagrami interakcije koji opisuju realizaciju konkretnog *use case*-a

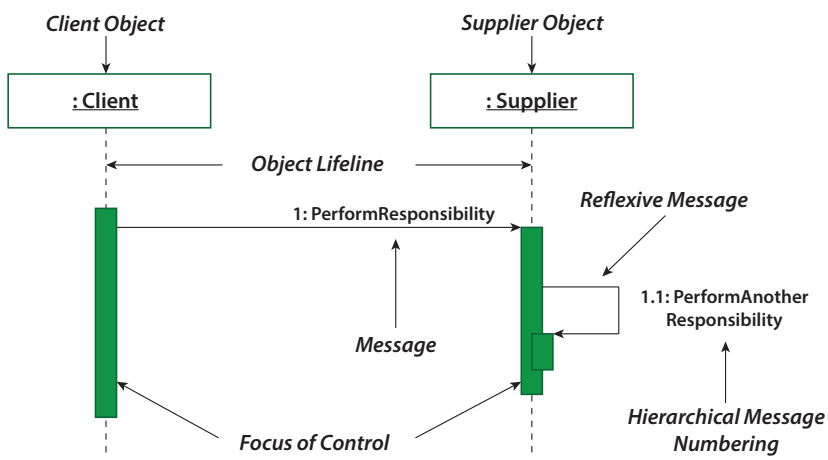
7.2.1. Dijagram sekvenci

Dijagram sekvenci (*Sequence Diagram*) pripada grupi UML bihevioralnih dijagrama, čija je osnovna svrha modelovanje načina na koji objekti međusobno komuniciraju tokom realizacije određenog funkcionalnog zahteva (*use case*-a). Za razliku od statičkih dijagrama, poput dijagrama klasa, dijagram sekvenci opisuje dinamičko ponašanje sistema, pri čemu se fokusira na hronološki redosled poruka koje objekti razmenjuju u toku izvršavanja scenarija.

Ovaj dijagram je posebno koristan u fazi analize i projektovanja, jer pomaže da se razume kako se konkretan *use case* realizuje i koje odgovornosti imaju pojedinačni objekti.

Osnovni elementi dijagrama sekvenci su (Slika 7.4):

- Objekti (*Participants*) — učesnici interakcije, prikazani kao pravougaonik sa nazivom.
- Životna linija (*Lifeline*) — vertikalna isprekidana linija koja prikazuje postojanje objekta tokom vremena.
- Poruke (*Messages*) — horizontalne ili dijagonalne strelice koje predstavljaju komunikaciju između objekata. Opisuju komandu, poziv metode, odgovor ili povratnu vrednost.
- Aktivacioni blok (*Activation bar*) — uzani vertikalni pravougaonik na *lifeline*-u, koja prikazuje period kada je objekat aktivan (izvodi određenu operaciju).
- Vremenska osa — implicitna, uvek ide odozgo nadole.
- Interakcione fragmente (opciono) — konstrukti poput *alt, opt* ili *loop* koji omogućavaju opis uslova, alternativa i ponavljanja.



Slika 7.4. Osnovni elementi dijagrama sekvenci

Prilikom izrade dijagrama sekvenci preporučuje se sledeći pristup (Larman, 2012):

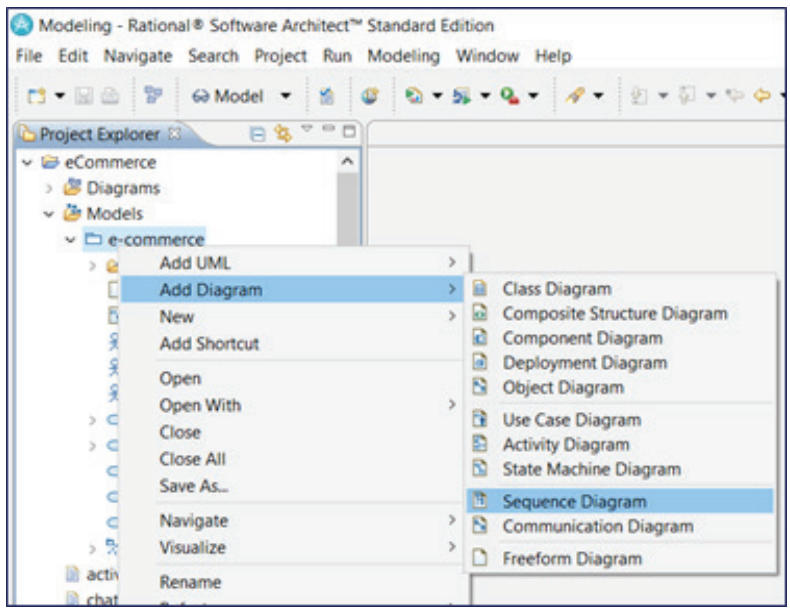
1. Identifikovati *use case* ili scenario koji se modeluje.
2. Prepoznati ključne klase koji učestvuju u realizaciji tog scenarija.
3. Definisati tok poruka: ko šalje poruku, kome, kojim redosledom i sa kojim parametrima.
4. Prevlačenjem poruke, kreira se aktivacioni blok (uzak, vertikalni prevougaonik na *lifeline*-u klase) koji pokazuje koji objekti obavljaju operacije u kom trenutku.
5. Opcionalno uključiti uslove, petlje i alternative putem interakcionih fragmenata (npr. *alt, opt, loop*).

Ovakvim pristupom dobija se jasna i hronološki organizovana specifikacija ponašanja sistema, koja je izuzetno korisna za dalji dizajn klasa i definisanje njihovih odgovornosti.

Koraci kreiranja dijagrama sekvenci u IBM RSA alatu

Korak 1. Kreiranje dijagrama sekvenci

Iskoristićemo postojeći projekat za *eCommerce* i desnim klikom na istoimeni paket odabraćete *Add Diagram*, a zatim *Sequence Diagram*:



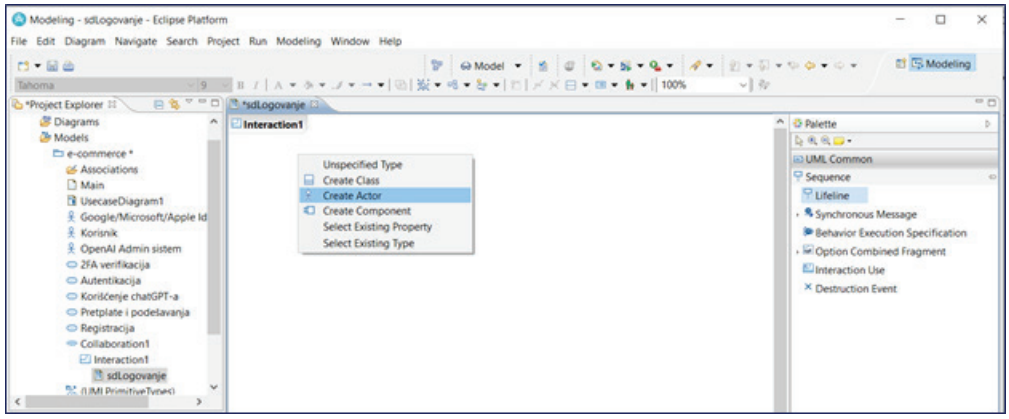
Slika 7.5. Kreiranje dijagrama sekvenci u IBM RSA alatu

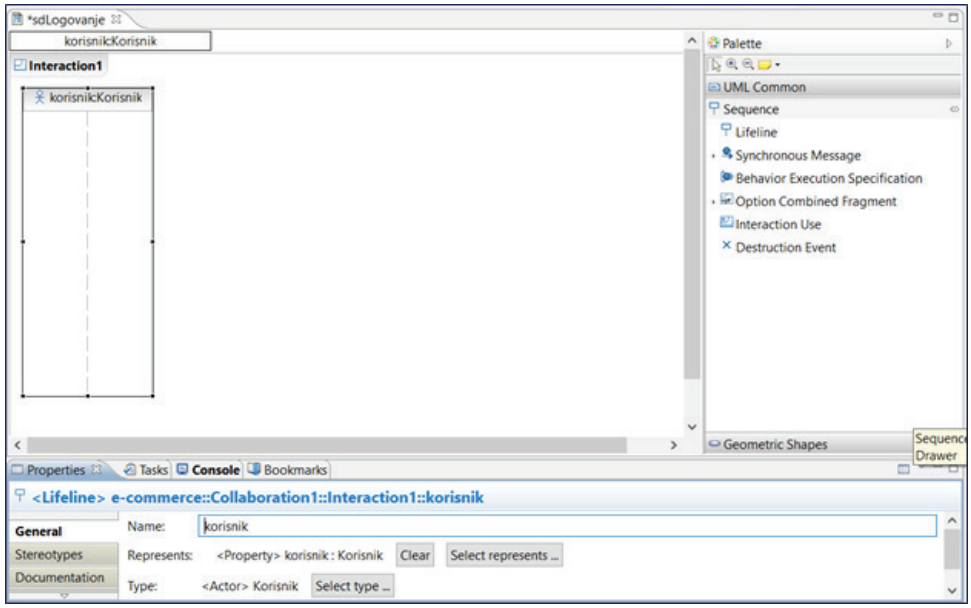
Dijagram sekvenci možete imenovati kao sdLogin (od reči: *sequence diagram* za login).

Korak 2. Kreiranje klasa prevlačenjem odgovarajućih *lifelines* na dijagram

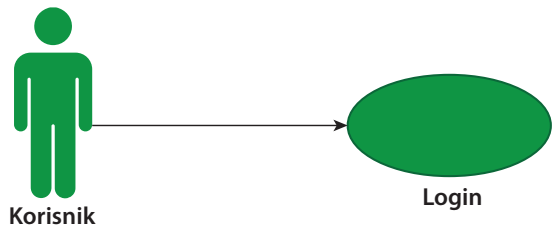
Lifelines su konkretno buduće klase aplikacije, s tim što prevlačenjem *lifeline* na deo za kreiranje dijagrama, imate mogućnost da odaberete tip *lifeline*-a, npr. da li želite da bude akter ili komponenta i sl. Ukoliko prvi put učite kako da kreirate ovaj dijagram, preporuka je da prvi *lifeline* bude tipa akter, kako bi zamislili npr. nekog korisnika koji klikće po korisničkoj formi. U ovom slučaju akter je čovek i nije deo aplikacije, jedino u slučaju AI sistema gde se na primer prenosi glasom poruka do korisnika, a ne preko interfejsa.

Dakle, prevlačenjem lifeline na centralni deo selektovaćemo *Create Actor* i imenovati ga kao Korisnik.



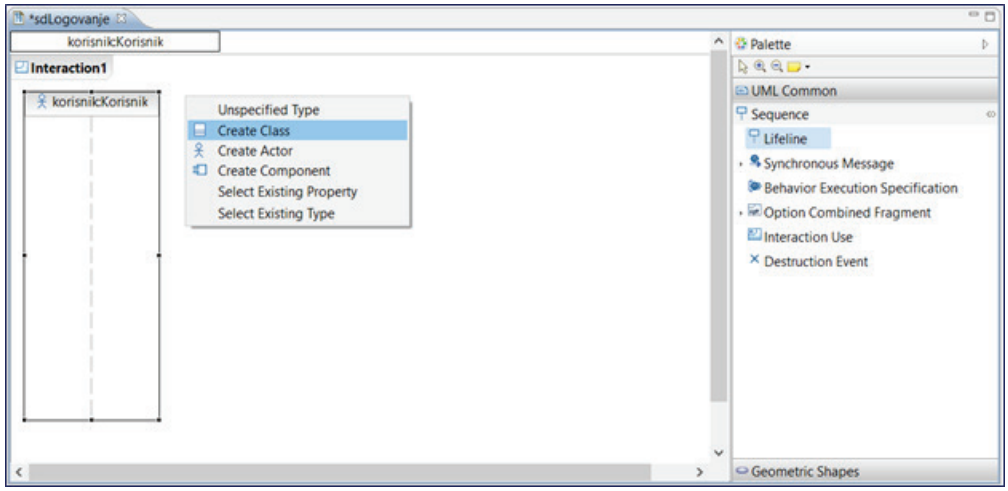


Ukoliko niste sigurni kako da imenujete prvog aktera, pogledajte vaš *use case* dijagram. Ukoliko deo dijagrama izgleda nešto slično kao na slici 7.7, tada ćete imati na početku tri *lifeline*-a. Prvi je tipa akter. Relacija asocijacije se posmatra kao korisnički interfejs, te je drugi lifline tipa klasa koju možemo imenovati kao LoginUserForm (korisnička forma za logovanje). Treći lifline je tipa klasa i njega možemo imenovati kao use case „Login" s tim što ćemo mu dodeliti stereotip „controller".

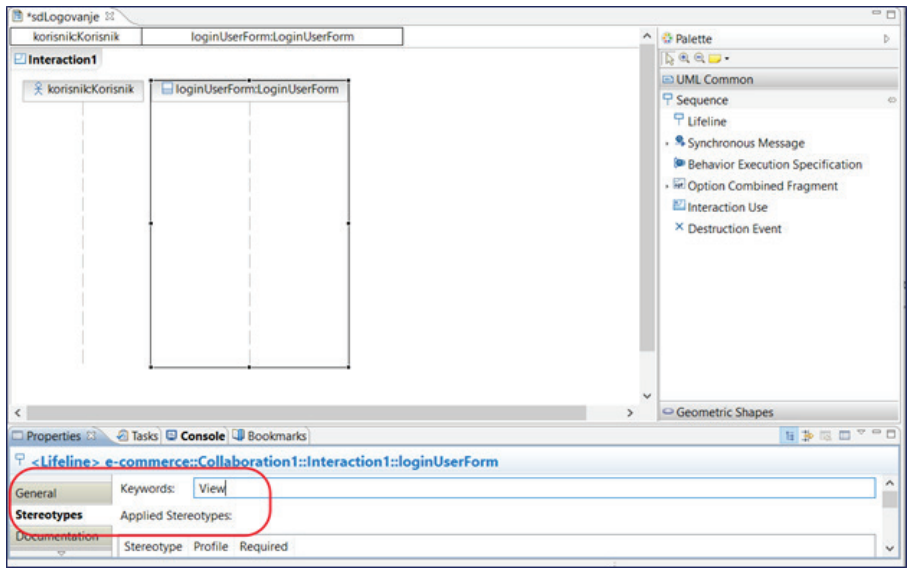


Slika 7.6. Pojednostavljeni prikaz use case dijagrama, sa akcentom na login *use case*

Nakon dodavanja lifeline tipa akter, kojeg smo imenovali korisnik (kao sa *use case* dijagrama), drugi lifeline koji prevlačimo je tipa klasa, kojeg ćemo imenovati kao LoginUserForm:

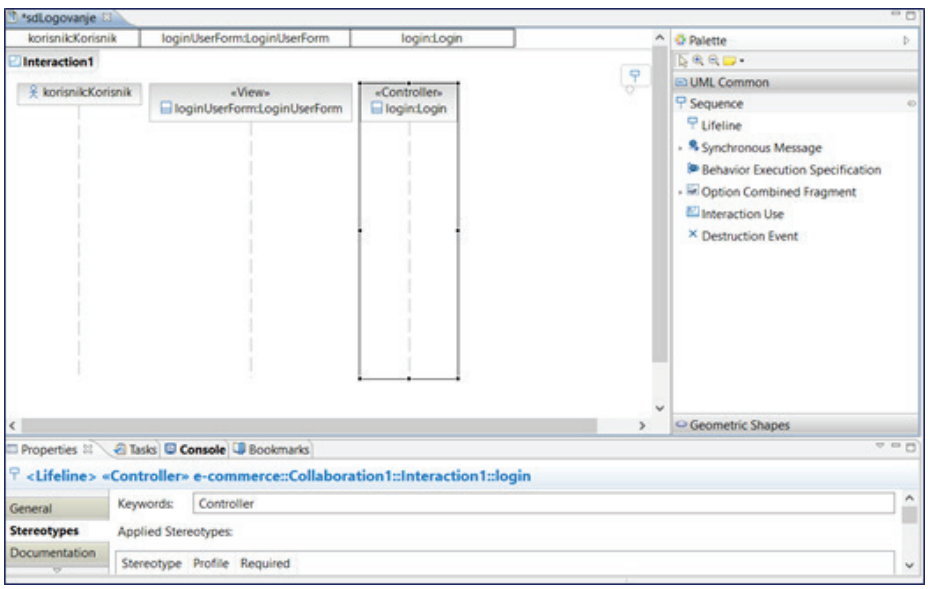


Dodavanje stereotipa koji dodatno opisuje klasu, se imenuje u *Properties* prozoru pod *Stereotypes*:



Obratite pažnju da lifeline treba da bude selektovan da bi se video Stereotypes za taj lifeline.

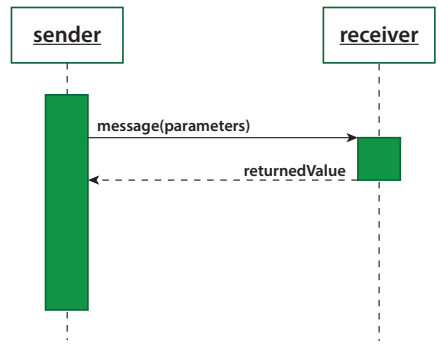
Finalni izgled dodavanja *lifelines* na dijagram bi izgledao ovako:



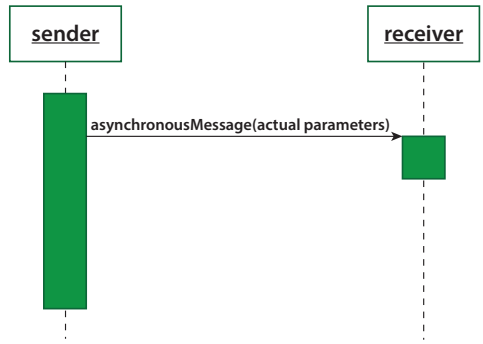
Korak 3. Dodavanje poruka između klasa, hronološkim redom.

U UML dijagramima sekvenci prikazuju se različite vrste poruka koje objekti međusobno razmenjuju tokom izvršavanja jednog scenarija. Najčešće korišćene vrste su:

- **Sinhrone poruke** (*Synchronous Message*) - Sinhrona poruka označava situaciju kada pošiljalac mora da sačeka odgovor pre nego što može da nastavi dalje. Na primer: Korisnik unosi email i lozinku → kontroler šalje Modelu poruku validateCredentials(*email, password*) i čeka rezultat (tačno / netačno) pre nego što prikaže naredni ekran.

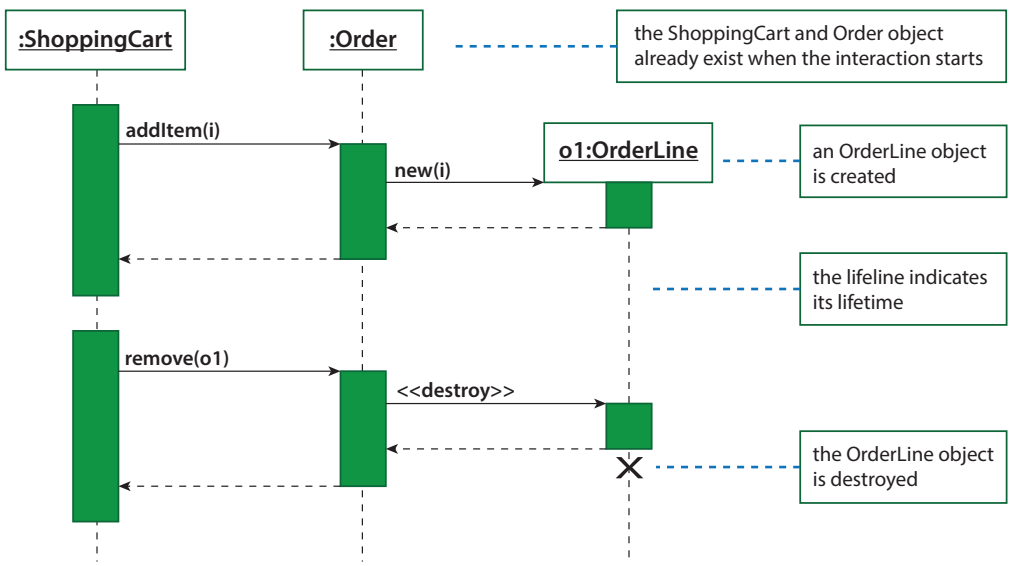


- **Asinhrone poruke** (*Asynchronous Message*) - Asinhrona poruka se koristi kada pošiljalac ne čeka da primi odgovor. Na primer: View šalje kontroleru poruku showErrorMessage() da prikaže grešku, ali ne stoji „na čekanju" – View odmah nastavlja sa radom (npr. omogućava korisniku novi unos podataka).



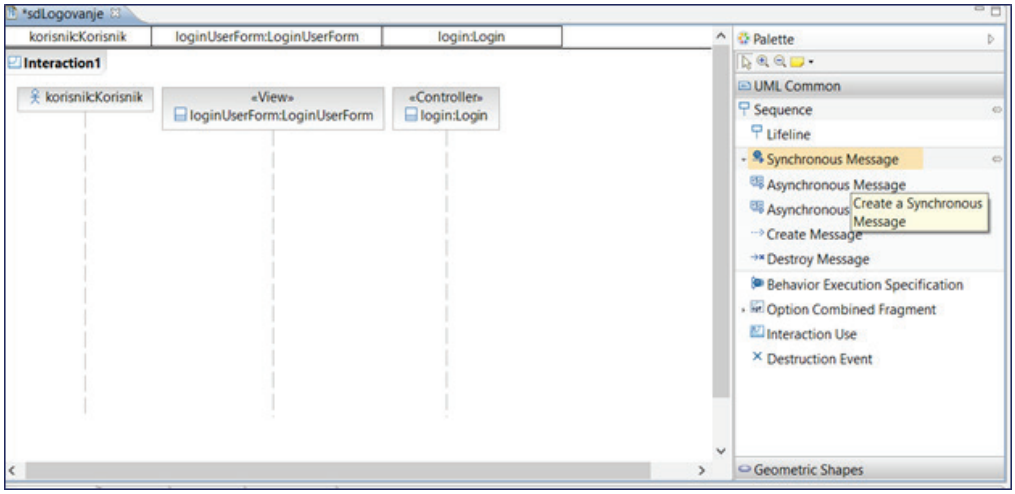
- **Asinhrona signalna poruka** (*Asynchronous Signal Message*) - Ova poruka predstavlja slanje signala bez očekivanja odgovora, najčešće u distribuiranim i event-driven sistemima. Može predstavljati događaj koji aktivira neki proces. Na primer: Sistem šalje signal loginFailed komponenti koja evidentira statistiku neuspelih pokušaja prijave, bez čekanja povratne poruke.

- **Poruka kreiranja** (*Create Message*) - Ova poruka se koristi kada jedan objekat kreira novi objekat. Na primer: Kontroler tokom procesa registracije šalje poruku new User() da bi kreirao novi User objekat u Modelu.
- **Poruka uništenja ili brisanja** (*Destroy Message*) - Ova poruka označava trenutak kada objekat prestaje da postoji. Obično se obeležava simbolom X na kraju lifeline-a. Na primer: Nakon logout procesa kontroler šalje poruku destroy() objektu sesije, čime se sesija korisnika uništava.

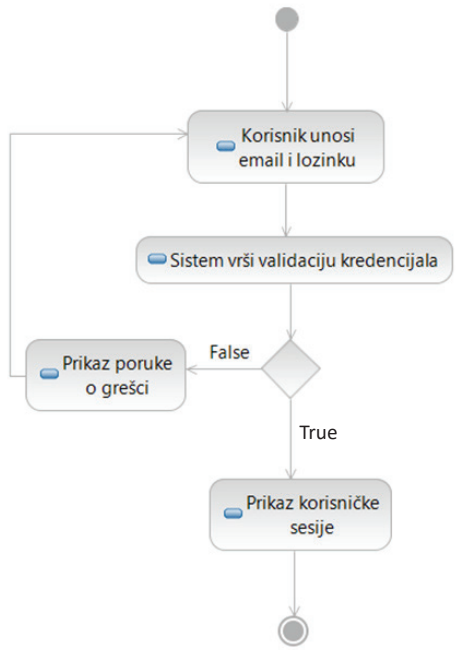


Slika 7.7. *Create* i *Destroy* poruke na primeru porudžbine

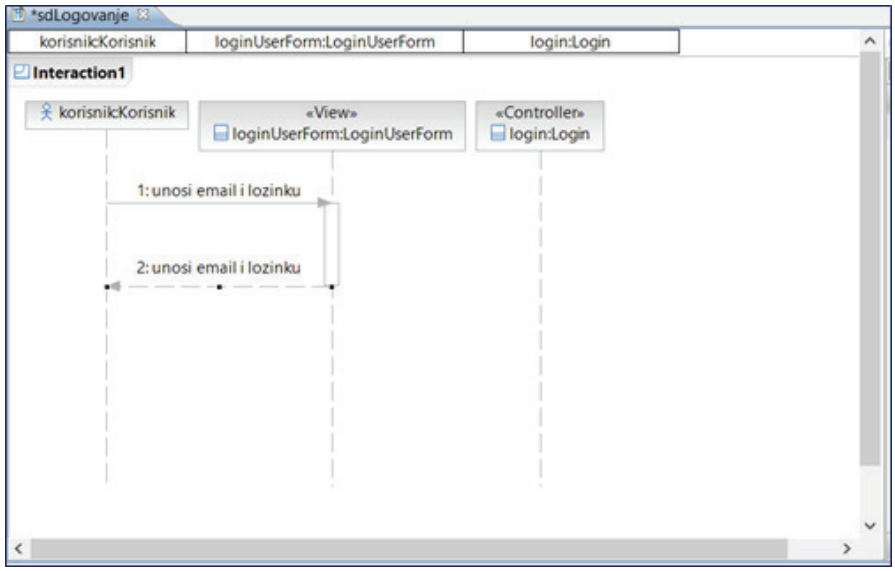
S obzirom da korisnik prilikom unosa korisničkog imena i šifre, mora da čeka da aplikacija proveri parametre, sa palete selektujemo *Synchronous Message* i vučemo od *lifeline*-a Korisnika do *lifeline*-a tipa *View*.



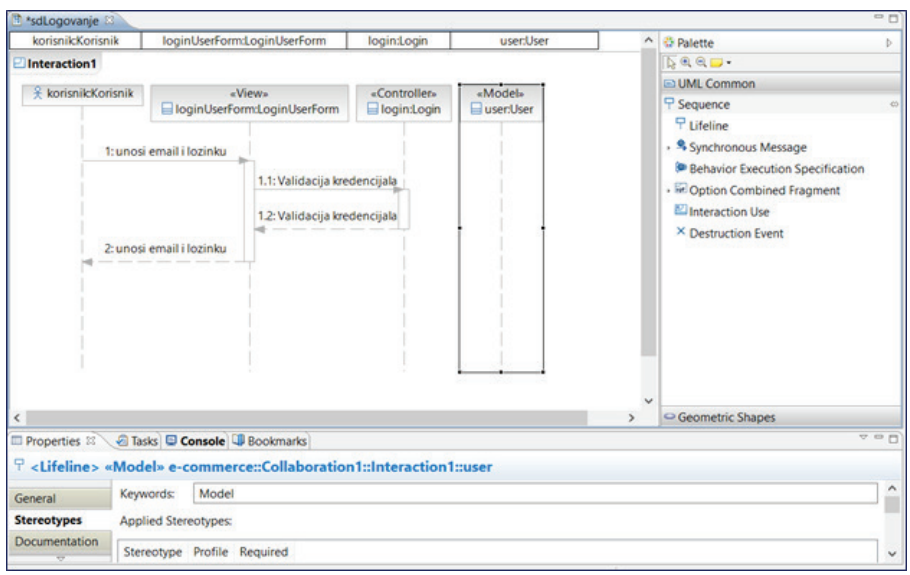
Ovu prvu poruku imenujemo, makar u početnim fazama crtanja dijagrama sekvenci, kao što je imenovana prva aktivnost u okviru dijagrama aktivnosti. Inače, da bi znali kako da imenujemo poruke i koju vrstu da selektujemo, te informacije se mogu videti na dijagramu aktivnosti.



Prva aktivnost je prva poruka na dijagramu sekvenci:



Ono što je novo na dijagramu sekvenci su klase tipa Model ili buduće tabele u bazi podataka. Vi treba da date logične nazive tim klasama. Ukoliko je u pitanju prosleđivanje polja username i password, dovoljno je da se pitate, u okviru koje klase će ta polja stajati. Logično je da u klasi User ili korinsik se nalaze ova polja. Na dijagramu ispod dodajemo novi lifeline User koji je tipa Model.

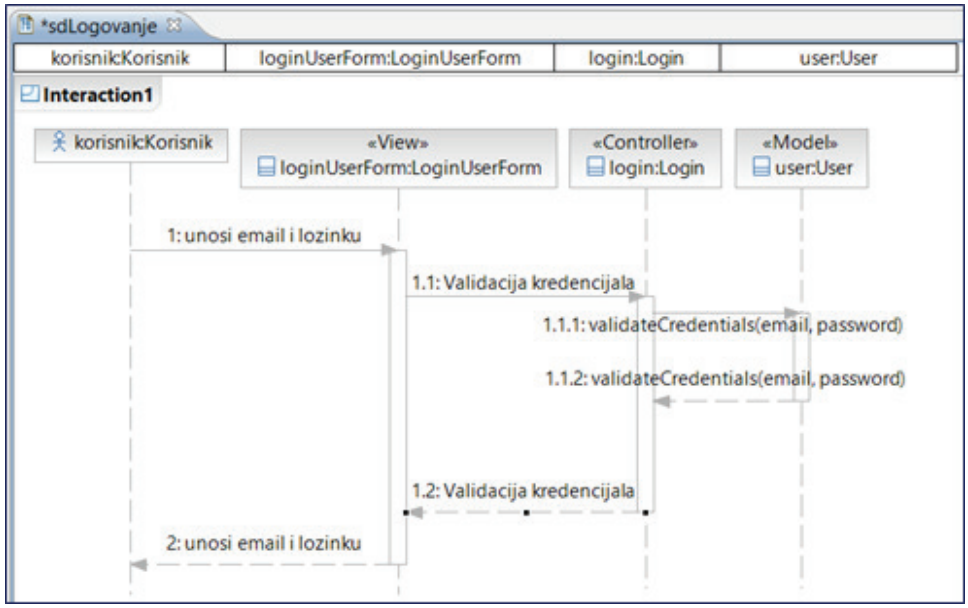


Kontroler će modelu proslediti poruku da autentifikuje korisnika slanjem konkretnih podataka. Na dijagramu smo imenovali tu poruku, kao *validateCredentials*(*email, password*). Tokom analize softvera, ne morate razmišljati o preciznim nazivima poruka. Poruke su buduće metode klasa. Precizne nazive metoda ćete dati na fizičkom dijagramu sekvenci, koji se crta u fazi projektovanja softvera (*software design*).

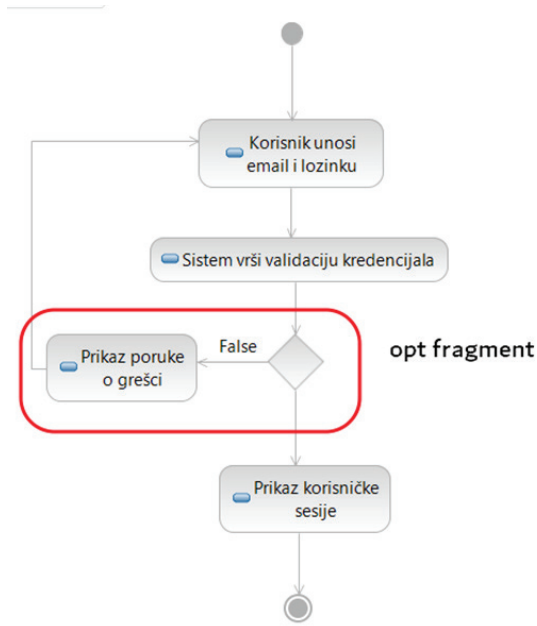
Poruka koju *controller* šalje klasi *User* (tipa Model), *validateCredentials*(*email, pass*), podrazumeva nekoliko koraka, npr:

- Model proverava u bazi: SELECT * FROM User WHERE email=? AND password=?
- vraća rezultat (objekat korisnika ili null).

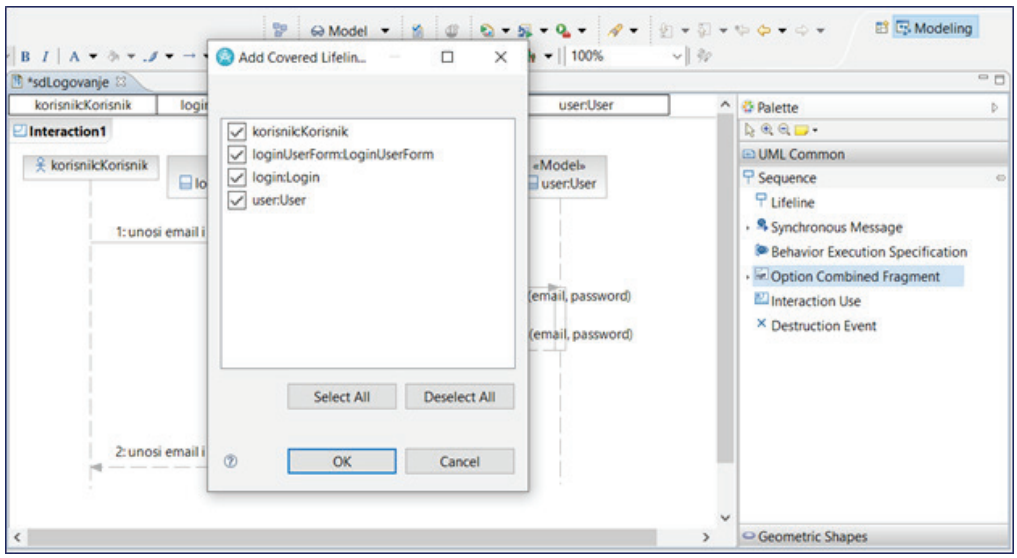
Tek nakon dobijenih rezultata iz modela (povratna poruka), kontroler dalje može da nastavi sa radom.



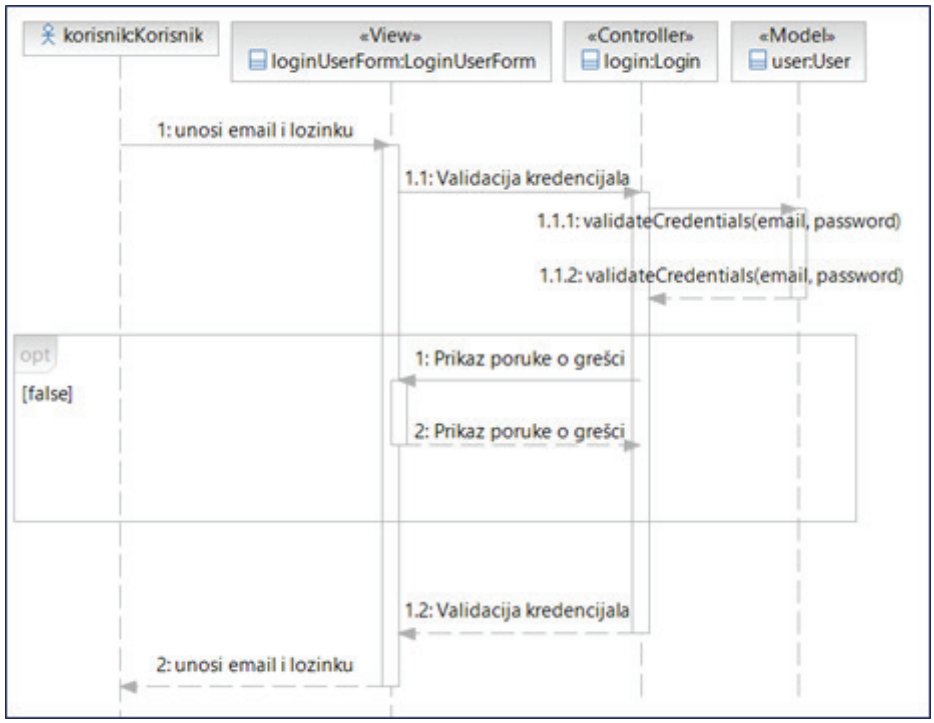
S obzirom da na dijagramu aktivnosti imamo proveru da li su kredencijali ispravni (*decision node*), taj node se na dijagramu sekvenci prikazuje kroz kombinovane fragemente (*Combined Fragment*).



Ukoliko imamo jednu proveru, onda ćemo odabrati *Option Combined Fragment*, a ukoliko imamo opciju sa više grananja (*if then else*), tada će se koristi alternativni fragment (*Alternative Combined Fragment*). Prilikom odabira fragmenta, prevućićemo preko svih *lifline*-ova, nakon ćega će se otvoriti prozor, gde možete da isključite neke *liflines*, na koje se ove provere ne odnose.



U okviru *opt* fragmenta dodaćemo sinhronu poruku od kontrolera do interfejsa (*View*) o prikazu grešci:



Iako je intuitivno pomisliti da *Controller* "ne čeka odgovor" od *View*-a, **u UML-u se logika strelice zasniva na mehanizmu izvršavanja, a ne na realnoj percepciji komunikacije**. U tipičnoj MVC aplikaciji (JSP, Spring MVC, .NET MVC), *Controller* poziva *View* sinhrono:

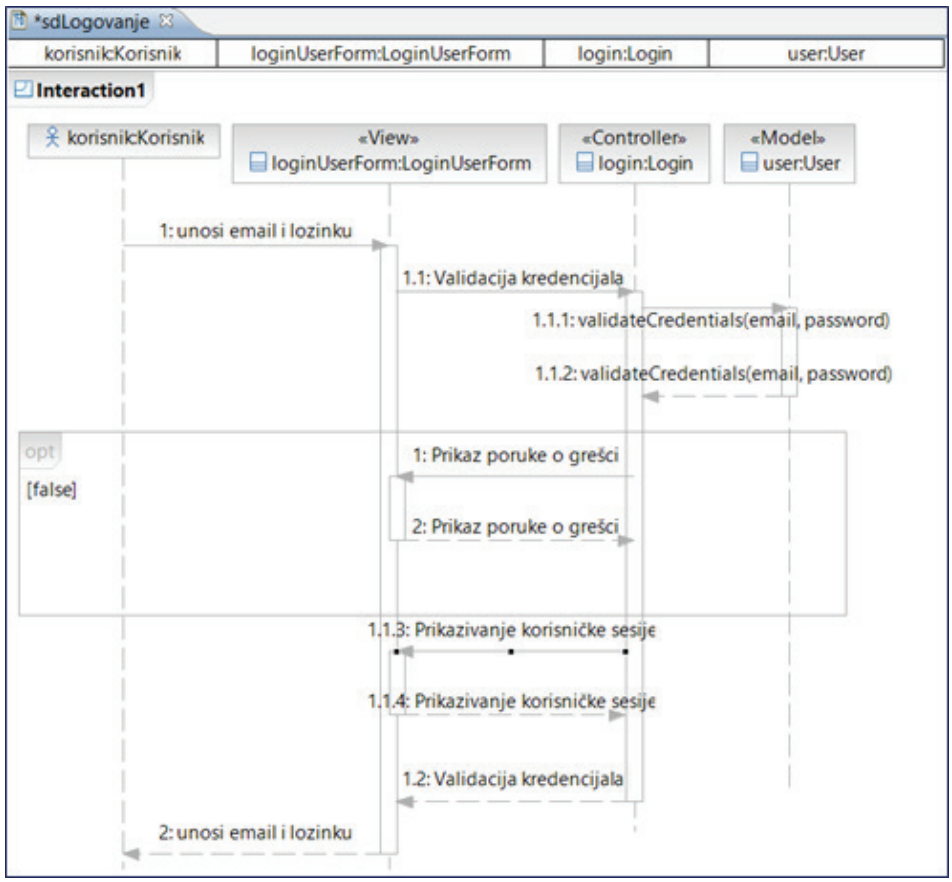
- renderuj stranicu,
- View je generiše,
- pa se vraća rezultat (HTML ili UI komponenta).

To znači da je sekvenca blokirajuća, pa se koristi sinhrona poruka.

Asinhrone poruke se koriste samo kada ima asinhroni mehanizam, npr:

- event loop,
- message queue,
- concurrency (*thread*-ovi),
- publish/subscribe,
- mikroservisi sa eventovima.

Nakon tog fragmenta ide **glavni tok**, tj. scenario koji se izvršava kada je uslov istinit (odnosno kada su kredencijali ispravni).



U standardnim MVC aplikacijama:

- *Controller* kreira ili osvežava korisničku sesiju nakon uspešne autentikacije.
- Zatim preusmerava/vrši renderovanje odgovarajućeg *View*-a (npr. DashboardView, HomePage i sl.).
- View prikazuje interfejs novom, ulogovanom korisniku.

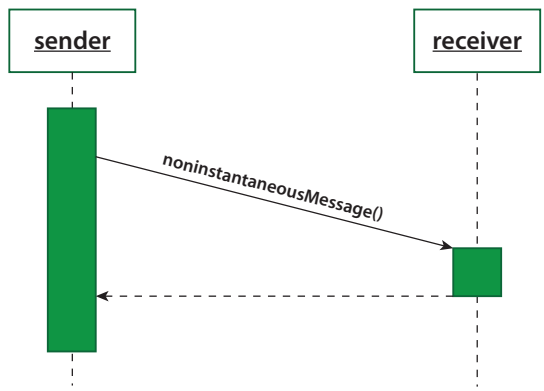
Precizniji prikaz poruka na dijagramu sekvenci bi bio:

- Od modela ka kontroleru dodati poruku: *loginSuccess*()
- Nakon čega se uspostavlja sesija *setSession*(*user*) između *Controller–Model*, gde se učitavaju korisnički podaci (npr. setovanje User Session-a)
- Na kraju od kontrolera do *View*-a se prikazuje poruka *showDashboard*(*user*) ili *renderDashboard*(), u okviru koga se preusmerava na novi *View* (npr. HomePage, Dashboard, UserProfile…).

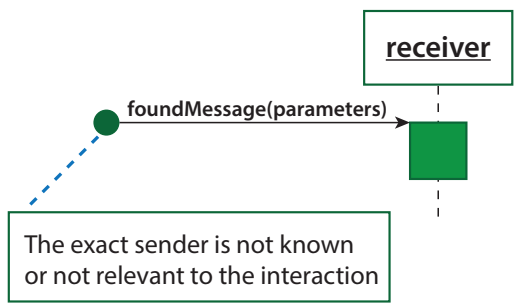
Dodatne opcije na dijagramu sekvenci

Pored osnovnih poruka, dostupnih u okviru IBM RSA alata, postoje i druge vrste poruka i to:

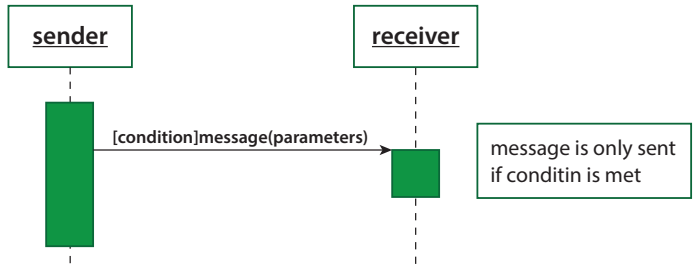
- **Trenutne poruke** (*Instantaneous message*) - Poruke se smatraju trenutnim ukoliko je vreme koje je potrebno da stignu do primaoca zanemarljivo.
- **Netrenutne poruke** (*NonInstantaneous message*) - Nekim porukama je potrebno određeno vreme da stignu do primaoca – takve poruke se crtaju kosom linijom. Ovakve poruke bi trebalo crtati jedino ukoliko se želi naglasiti da poruka putuje sporim komunikacionim kanalom (i eventualno se želi dati iskaz o mogućem kašnjenju poruke).



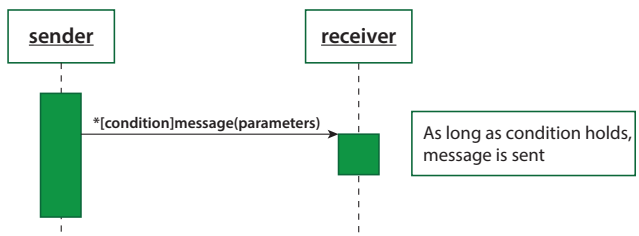
- **Pronađena poruka** (*Found message*) je poruka kod koje pozivaoc nije prikazan (ili pošiljaoc nije poznat ili nije relevantno za interakciju ko je pošiljaoc). Prikazuje se kao pun krug na početku poruke:



- **Uslovne poruke ili uslovne interakcije** (*conditional interaction*) gde se poruka šalje samo ukoliko je ispunjen određeni uslov. Uslov se ispisuje u srednjoj zagradi ispred naziva poruke:

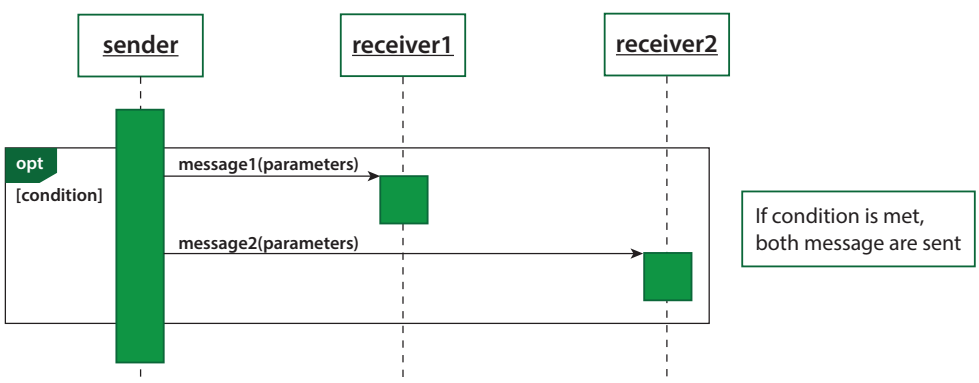


- **Ponavljajuće poruke ili ponavljajuće interakcije** Ukoliko ispred naziva poruke stoji simbol zvezdice, znači da se poruka šalje uzastopnim ponavljanjem (*loop* petlja). Dok god je uslov ispunjem poruka se šalje: while (*true*) { }

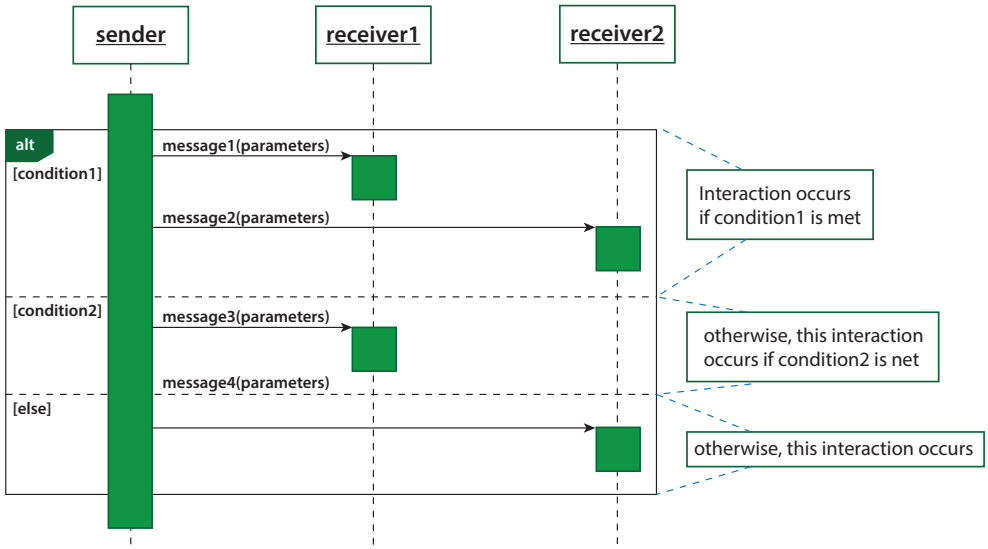


Najčešće korišćeni kombinovani fragmenti su:

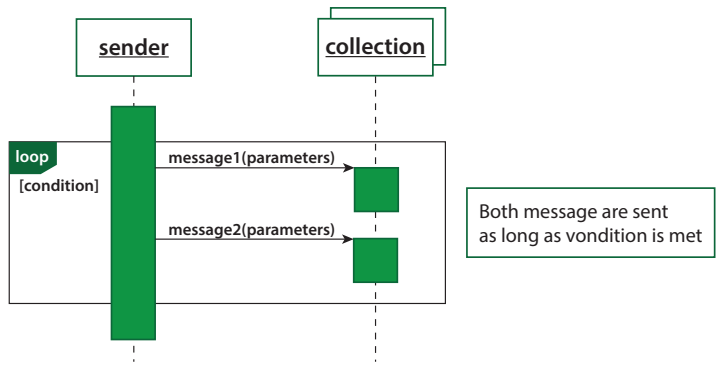
- **Option Combined Fragment** - Ukoliko se nekoliko poruka šalje pod istim uslovom onda se koristi "opt" fragment. Uslovne poruke ili "opt" fragment je sličan if konstrukciji u programskim jezicima *if-then*.



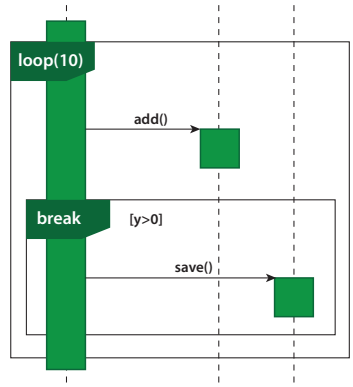
- **Alternative Combined Fragment** - Ukoliko se žele prikazati nekoliko alternativnih interakcija, koristi se "alt" fragment. Svaka alternativa ima uslov i sadrži interakcije koje se događaju kada se željeni uslov ispuni. Ovaj fragment je sličan ugnježdenim if-then-else ili switch/case konstrukcijama kod programskih jezika.



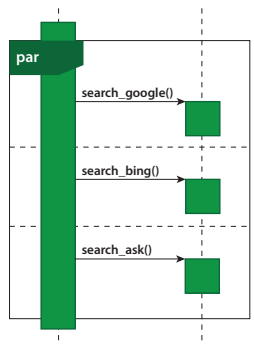
- **Loop Combined Fragment** - Ukoliko se više poruka šalje u istoj iteraciji, koristi se "loop" kombinovani fragment. Primaoc ponavljajuće poruke je kolekcija, a uslov uglavnom predstavlja određeni filter za elemente kolekcije:



- **Break Combined Fragment** - Kada je uslov u okviru *break* fragmenta ispunjen, zaustavlja se prethodno izvršavanje toka.

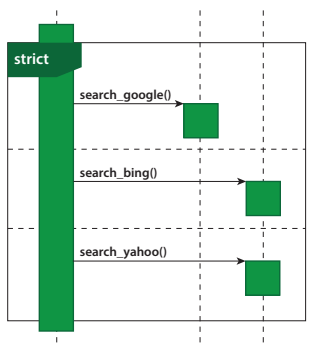


- **Parallel Combined Fragment** - označava potencijalno paralelno izvršavanje poruka.



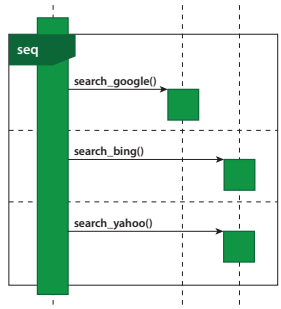
Search Google, Bing, and Ask in any order, possibly in parallel.

- **Strict sequencing** – označava obavezan (striktan) redosled izvršavanja poruka



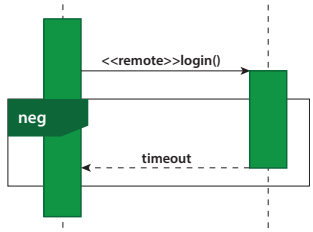
Search Google, Bing, and Yahoo in the strict sequential order.

- **Weak sequencing** - Ne mora da se izvršava striktnim redosledom.



Search Google, possibly parallel with Bing and Yahoo, but search Bing before Yahoo.

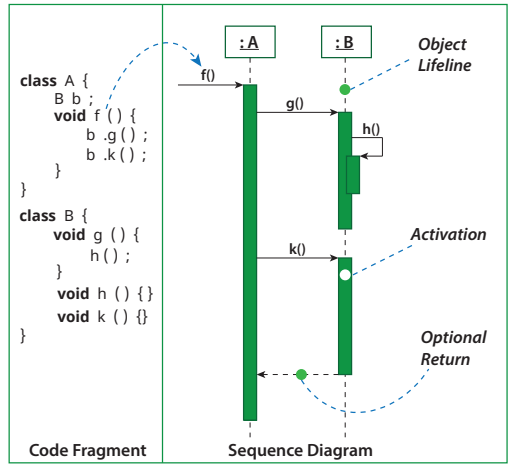
- **Negative** – predstavlja scenario koji je nedozvoljen, nepoželjan ili predstavlja grešku. Koristi se da bi se eksplicitno prikazalo šta ne sme da se desi u sistemu.



Should we receive a timeout message, it means the system has failed.

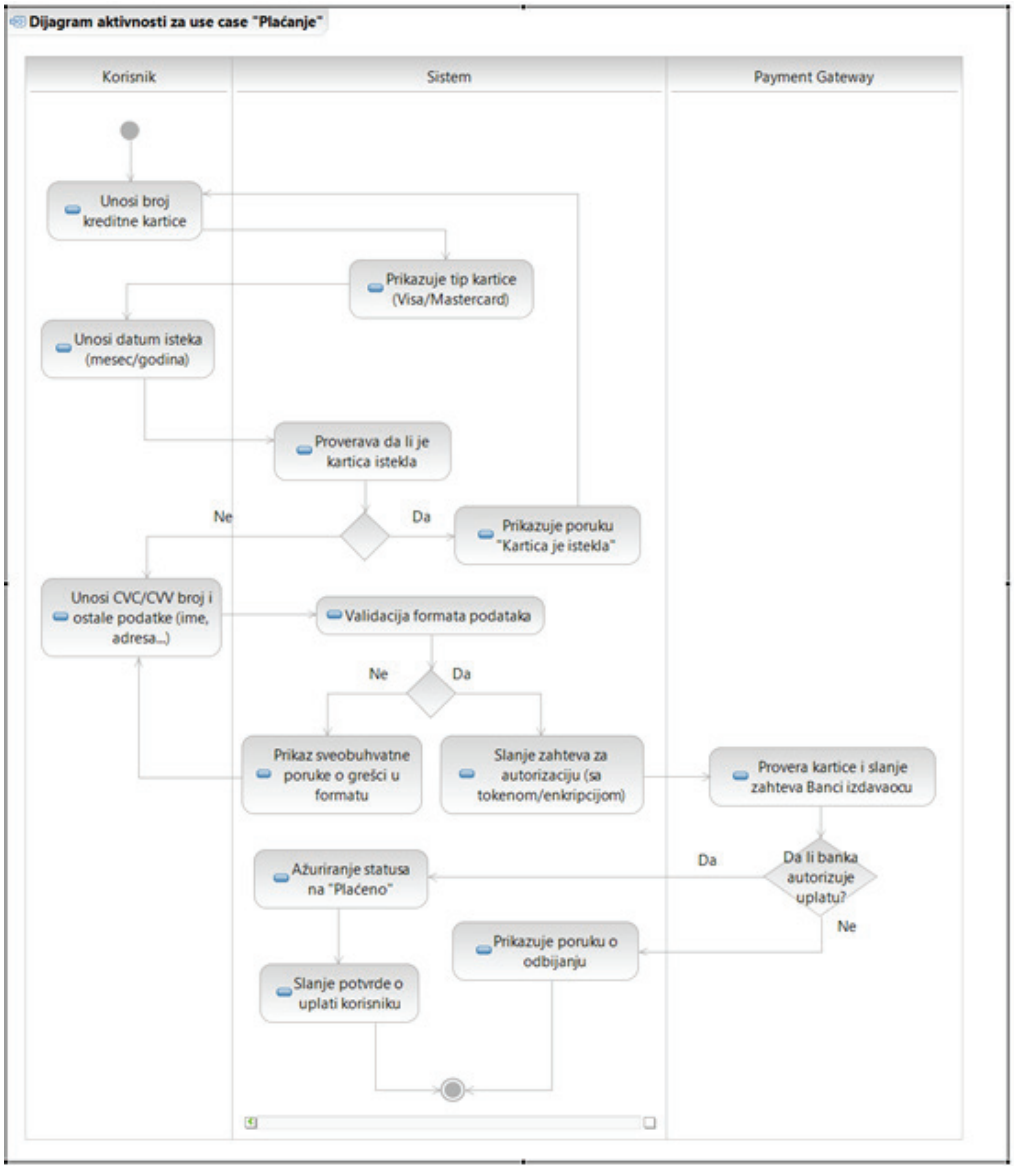
Značenje poruka za klase

Poruke na dijagramu sekvenci zapravo predstavljaju pozive metoda — svaka poruka odgovara metodi koja postoji u klasi objekta koji poruku prima.



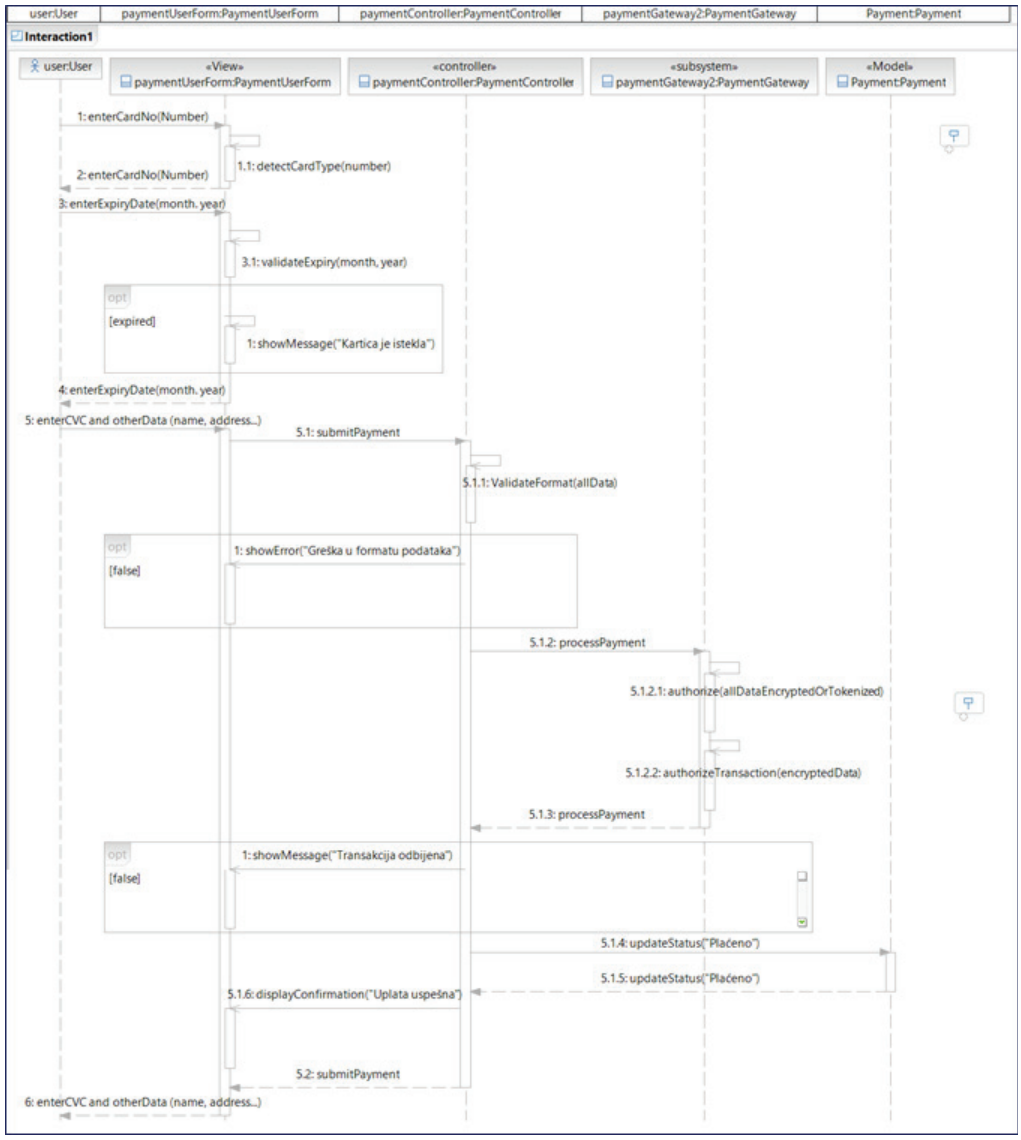
Vežba 7.1. Kreirati dijagram sekvenci za *use case* „Plaćanje“

Ukoliko vam je poznat scenario aktivnosti ka) za *use case* „Plaćanje“, potrebno je kreirati logički dijagram sekvenci.



Rešenje vežbe

Na osnovu priloženog dijagrama aktivnosti mogu se uočiti sledeće klase: Korisnik kao primarni akter, sistem koji se u okviru tradicionalne MVC arhitekture deli na klase tipa *View* i *Controller*, kao i spoljni akter *PaymentGateway* koji predstavlja eksterni sistem sa kojim aplikacija komunicira. Aktivnosti koje pojedine klase obavljaju prikazane su kao poruke na dijagramu sekvenci.



Napomene:

- Rekurzivne poruke na *View* klasi su sinhrone, jer se obrada izvodi odmah na *frontend*-u (npr. JavaScript). IBM RSA kod sinhronog *self-call*-a automatski generiše ulaznu i povratnu strelicu kao dve odvojene, iako se u UML-u radi o jednoj sinhronoj operaciji. Međutim, radi preciznijeg i preglednijeg prikaza razmene informacija na dijagramu, u ovoj vežbi je umesto toga korišćena asinhrona poruka.
- U okviru *opt* fragmenta, na primer kada se proverava validnost formata podataka, prikazna je samo ova poruka showError("Greška u formatu podataka"). Ukoliko pogledate dijagram aktivnosti, videćete da **sistem treba da vrati korisnika na prethodnu aktivnost** kako bi uneo ispravan podatak. Međutim, na dijagramu sekvenci **ove vrste povratnih poruka** ka korisniku ili interne kontrole toka obično **se ne prikazuju,** jer se **dijagram sekvenci fokusira na međusobnu interakciju objekata ili komponenti, a ne na detalje korisničkog interfejsa ili tok kontrole unutar pojedinačnih aktivnosti**. Dakle, prikazivanje samo poruke *showError* u *opt* fragmentu služi da se naznači da postoji kontrola greške, dok povratak korisnika na prethodnu aktivnost ostaje implicitno razumljiv iz konteksta dijagrama aktivnosti.
- Poruke koje se razmenjuju unutar *PaymentGateway* podsistema spadaju u njegovu internu logiku i nisu deo naše aplikacije, zbog toga se one uobičajeno ne prikazuju na dijagramu sekvenci. U okviru ove vežbe su prikazane samo radi ilustracije – da bi se videlo kakav tip odgovora naš sistem može očekivati od *PaymentGateway*-a.

Prikazani dijagram je pojednostavljena verzija interakcije u skladu sa tradicionalnim MVC pristupom, u kome *View* direktno komunicira sa *Controller*-om, a *Controller* dalje upravlja osnovnom poslovnom logikom i modelom.

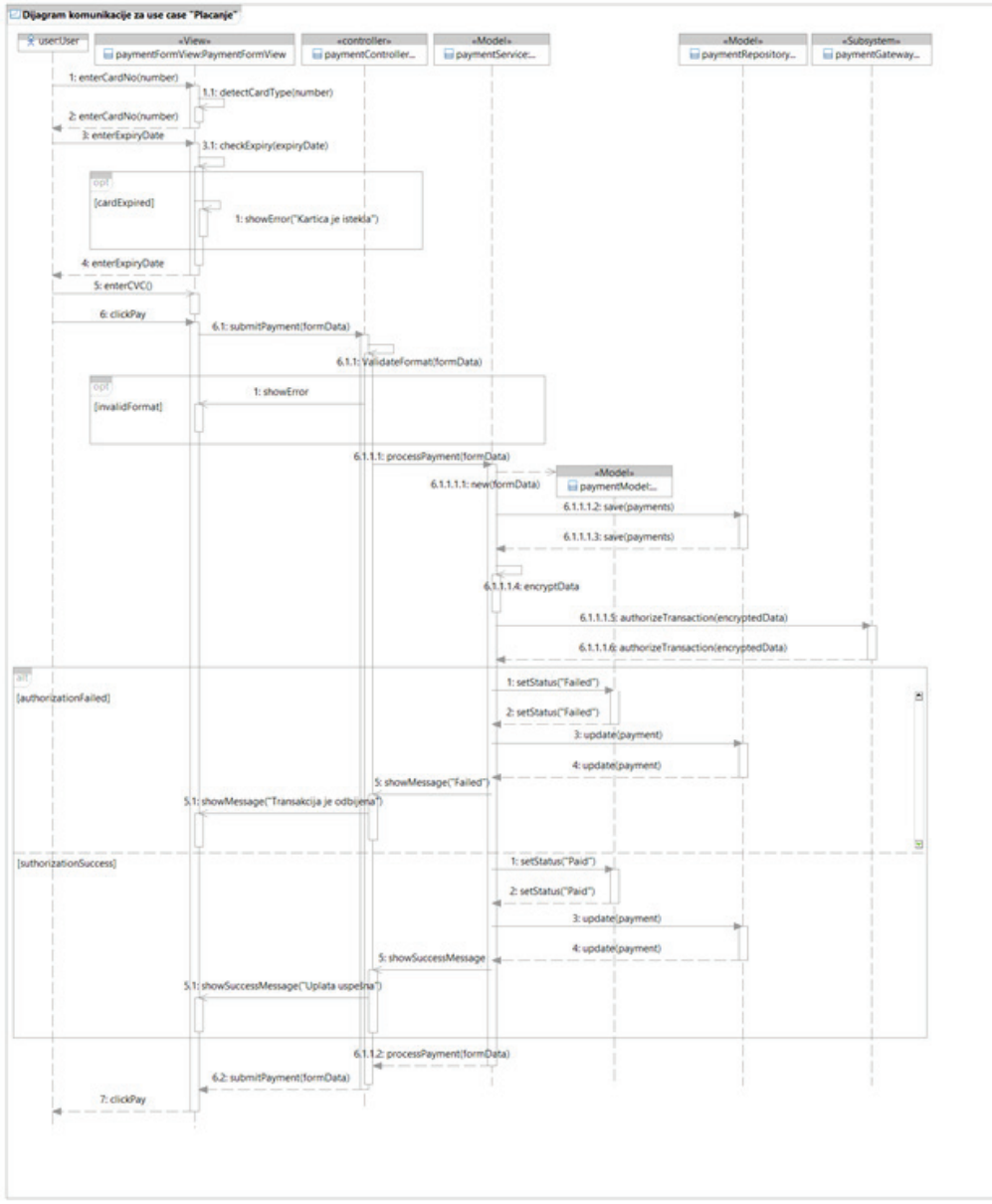
Prošireni dijagram sekvenci primenom savremene MVC arhitekture

U savremenim arhitekturama (npr. Clean Architecture, Spring layered, Domain-Driven Design), nakon Controller-a bi se u komunikacioni tok uveo dodatni sloj — Service, koji sadrži poslovnu logiku, dok bi pristup podacima bio izdvojen u poseban Repository sloj. Controller bi u tom slučaju bio samo koordinacioni sloj, bez poslovnih pravila. Zbog toga bi u potpunoj implementaciji sekvencijalni dijagram sadržao dodatne lifeline-e:

- PaymentService (poslovna logika)
- PaymentRepository (čuvanje i čitanje podataka)
- eventualno EncryptionService, ValidationService, itd.

Prethodni dijagram prikazuje samo osnovnu putanju radi preglednosti, dok bi potpuna enterprise verzija imala više slojeva i razdvojenih odgovornosti. U proširenoj verziji dijagrama, Controller više ne sadrži poslovnu logiku, već sve poslovne odluke delegira Service sloju. Service sloj zatim koristi Repository sloj za rad sa podacima i Gateway za komunikaciju sa eksternim servisima.

Prošireni dijagram sekvenci je prikazan na slici 7.9.



Slika 7.8. Prošireni dijagram sekvenci za *use case* "Plaćanje"

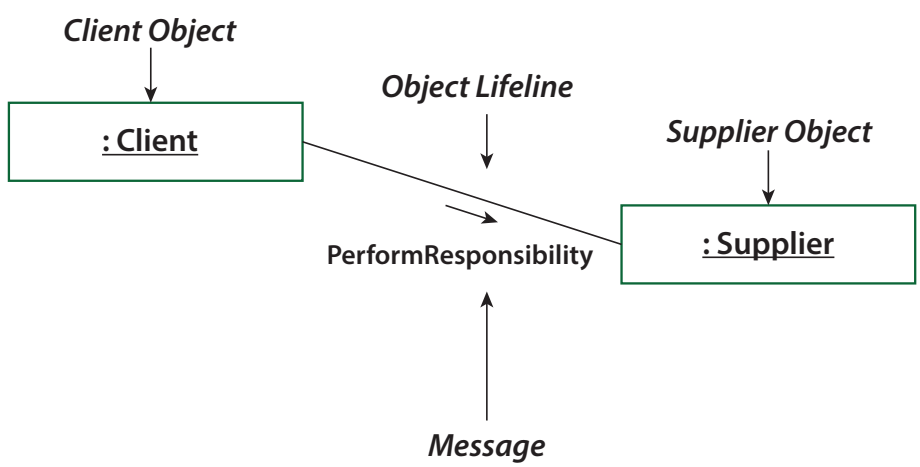
7.2.2. Dijagram komunikacije

Nakon što smo detaljno analizirali dijagram sekvenci, koji se fokusira na vremenski redosled poruka, važno je upoznati se sa drugim jednako značajnim načinom modelovanja interakcija - dijagramom komunikacije (*Communication Diagram*), koji se u starijim UML verzijama nazivao dijagram saradnje (*Collaboration Diagram*).

Dijagrami sekvenci i komunikacije prikazuju iste informacije, ali na različite načine. Dok dijagram sekvenci naglašava vremensku osu i vertikalni redosled izvršavanja poruka, dijagram komunikacije stavlja akcenat na strukturu i veze između objekata u interakciji, sa posebnim fokusom na učestalost komunikacije između objekata (broj poruka koje objekti razmenjuju). Zbog ovakve perspektive, dijagram komunikacije omogućava da se lako identifikuju **zavisne klase**, odnosno one koje međusobno razmenjuju veliki broj poruka. Takve klase je preporučljivo grupisati u isti paket, modul ili servis, umesto da se distribuiraju na različite lokacije, čime se smanjuje spoljna zavisnost i poboljšava kohezija.

Za prikaz redosleda izvršavanja na dijagramu komunikacije se koristi sledeći sistem numeracije poruka:

- Sekvencijalna numeracija: svaka poruka dobija broj (npr. 1:, 2:).
- Hijerarhija poziva: u slučaju ugnježdenih (rekurzivnih) poziva koristi se decimalna struktura (npr. 1.1:, 1.1.1:). Na primer, ako KorpaController šalje poruku proveriZalihe (1:), a unutar te metode se poziva dohvatiZalihe, ta poruka će biti označena kao 1.1:.

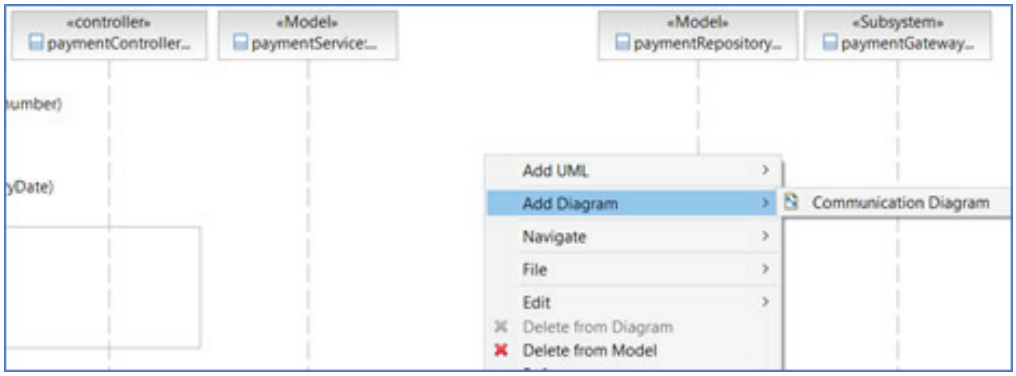


Slika 7.9. Pojednostavljeni prikaz dijagrama komunikacije

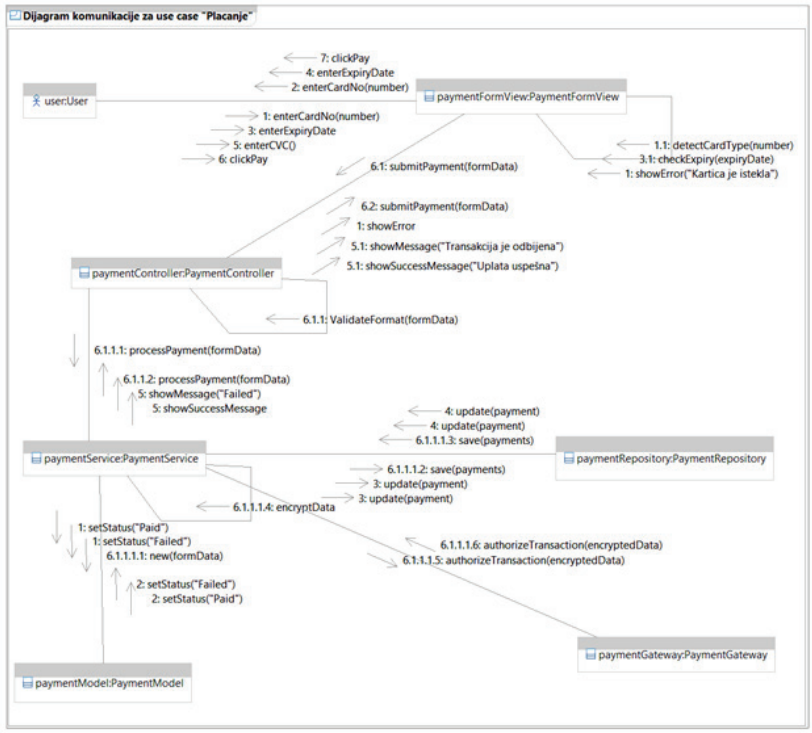
Dijagram komunikacije je koristan kada želimo da se fokusiramo na:

- Organizacione aspekte: ko sa kim direktno komunicira u sistemu.
- Identifikovanje zavisnosti: pregled toga koje klase zavise od drugih klasa pri realizaciji pojedinih funkcionalnosti.

U alatu IBM *Rational Software Architect* (RSA) dijagram komunikacije se može automatski generisati iz postojećeg dijagrama sekvenci. Potrebno je desnim klikom odabrati dijagram sekvenci, zatim *Add Diagram*, nakon čega će biti ponuđena opcija ***Communication Diagram***, a alat će automatski generisati odgovarajući prikaz.



Slika 7.10. Generisanje dijagrama komunikacije sa dijagrama sekvenci



Slika 7.11. Dijagram komunikacije za *use case* „Plaćanje“

Zadatak 7.1. Kreirati dijagram sekvenci za *use case* "Kreiranje i editovanje šoping korpe"

Dijagram sekvenci treba da prikaže sledeće klase i scenarije šoping korpe:

1. Definisani učesnici (*Lifelines*) – Na dijagramu prikazati sledeće objekte:
   - Korisnik (Akter)
   - KorpaView
   - KorpaController
   - KorpaService (poslovna logika)
   - KorpaRepository (pristup podacima)
   - KorpaModel (za kreiranje instance korpe)
   - ArtikalModel (za proveru zaliha i dohvat cene)
2. Zahtevani tokovi interakcije - Modelujte sledeće tokove koristeći odgovarajuće fragmente (opt, alt):

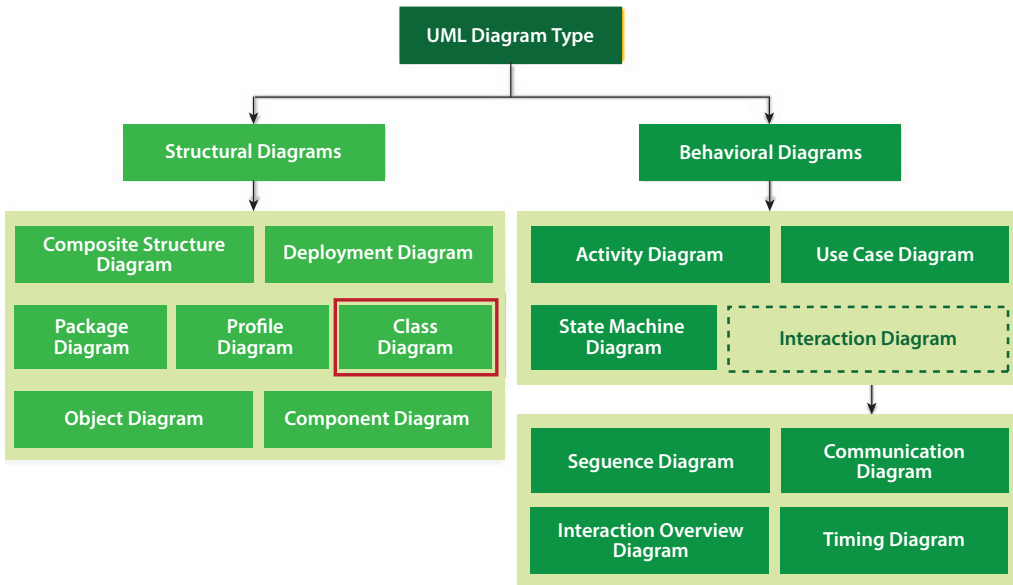
   A. Tok: Dodavanje Artikla (Kreiranje/Ažuriranje korpe)
   - Korisnik pokreće akciju dodajArtikal(ID, količina).
   - KorpaController poziva KorpaService radi provere zaliha.
   - Provera zaliha: KorpaService poziva ArtikalModel da proveri zalihe. Upotrebite odgovarajući fragment za sledeća dva uslova: [Nema zaliha] (vraća grešku) ili [Ima zaliha] (nastavlja se).
   - Ako ima zaliha, KorpaService dohvata korpu iz KorpaRepository, dodaje artikal i vraća je na čuvanje.
   - Ključna Logika Kreiranja: Ako korpa ne postoji ([Korpa ne postoji (null)]), KorpaRepository mora eksplicitno kreirati instancu KorpaModel Instance (new(userID)).
   - Kalkulacija Iznosa: Nakon dodavanja artikla, u KorpaService se vrši kalkulacija (dohvat cene iz ArtikalModel, izračunavanje ukupnog iznosa).
   - Ažurirana korpa se čuva (sacuvajKorpu) i KorpaView prikazuje poruku o uspehu.

   B. Tok: Ažuriranje Količine
   - Korisnik menja količinu artikla (azurirajKolicinu(ID, novaKoličina)).
   - Pre ažuriranja, ponoviti proveru zaliha.
   - Logika Ažuriranja: Koristiti odgovarajući fragment za uslove [Nova količina = 0] (ukloni artikal) ili [Ažuriraj] (ažuriraj količinu).
   - Ponovna Kalkulacija ukupnog iznosa: Nakon ažuriranja, obavezno je modelovati grupisanu ponovnu kalkulaciju totala u KorpaService (dohvat ažurirane korpe, dohvat cena, izračunavanje totala).
   - KorpaService vraća ažurirani DTO[4] objekat (korpaDTO sa novim ukupnim iznosom) Kontroleru, koji ga šalje Prikazu (KorpaView) za osvežavanje.

4 *Data Transfer Object* (*Objekat za prenos podataka*) sadrži samo neophodne podatke za prikaz, kao što su: lista artikala (*sa ID-om, nazivom i količinom*), ukupan iznos korpe (*total*), iznos popusta (*ako postoji*) i sl.

7.3. DIJAGRAM KLASA

Dijagram klasa je statčki (strukturni) UML dijagram koji modeluje ključne klase (entitete), njihove atribute, operacije (metode) i relacije (asocijacije, generalizacije, zavisnosti itd.). Koristi se tokom više faza razvoja softvera i to: za modelovanje domena (konceptualni model), analizu zahteva (logički model) i detaljni dizajn pre implementacije (fizički model).



Slika 7.12. Položaj dijagrama klasa u klasifikaciji UML dijagrama

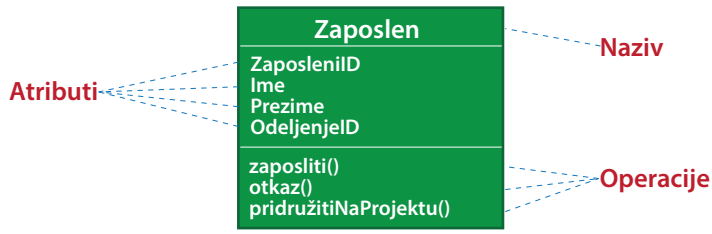
7.3.1. Osnovni elementi dijagrama klasa

Modelovanje statičke strukture softverskog sistema u UML-u zasniva se na jasno definisanim osnovnim elementima dijagrama klasa, među kojima centralno mesto zauzimaju klasa i objekat, kao i njihovi atributi i operacije.

Klasa je osnovni element dijagrama klasa i opisuje skup objekata koji dele zajedničke karakteristike i ponašanja. Konkretno, klasa opisuje grupu objekata sa:

- sličnim osobinama (atributima)
- zajedničkim ponašanjima (operacijama ili metodama)
- zajedničkim relacijama ka drugim objektima
- jedinstvenim značenjem ("semantika").

Na primer, zaposleni imaju ID, ime i prezime, odelenjeID. Jedan zaposleni se može zaposliti, dobiti otkaz ili biti pridružen na projektima.

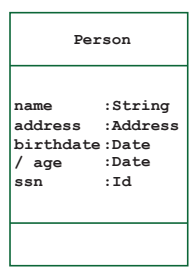


Objekat je osnovna jedinica objektno-orijentisanog sistema. Predstavlja konkretnu instancu klase, odnosno stvarnu pojavu nečega što klasa opisuje. Ako je klasa šablon ili tip, onda je objekat njegov konkretni primer. U akademskoj literaturi objekat se opisuje kao entitet koji poseduje:

- **Identitet** - Svaki objekat ima jedinstveni identifikator (ID) koji ga razlikuje od svih drugih objekata, čak i ako imaju iste atribute (Larman, 2004).
- **Stanje** - Stanje objekta čine vrednosti njegovih atributa u datom trenutku. Na primer, objekat *zaposleni1* može imati:
  - zaposleniID = 15
  - ime = „Ana"
  - prezime = „Jovanović"
  - odeljenjeID = 3
- **Ponašanje** - Ponašanje se opisuje operacijama (metodama) koje objekat može da izvrši. Npr. zaposleni1.otkaz() menja stanje objekta ili sistema. Na primer, ukoliko imamo klasu Zaposleni, onda su objekti konkretni zaposleni u sistemu:
  - zaposleni1: Zaposleni (ID=25, ime="Milan", odeljenjeID=2)
  - zaposleni2: Zaposleni (ID=26, ime="Teodora", odeljenjeID=5)

Dakle, klasa predstavlja šablon ili tip, a objekat je konkretan entitet (istanca klase) sa svojim stanjem i ponašanjem.

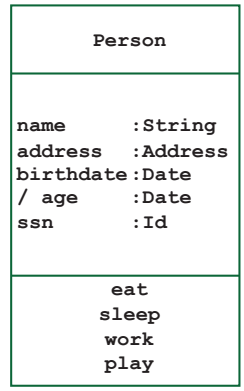
Atributi predstavljaju podatke koje objekti klase čuvaju. Standardna UML notacija koristi format *attributeName : Type*. Izvedeni atributi (*derived attributes*) su oni koji se ne skladište eksplicitno, već se mogu izračunati iz drugih podataka. Na primer, starost se može izračunati na osnovu datuma rođenja. Izvedeni atribut se označava sa kosom crtom, npr. */ age : Date*.



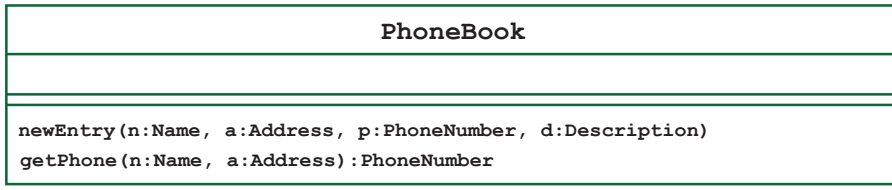
Operacije (metode) opisuju ponašanje objekata, tj. funkcionalnosti koje klasa može da obavlja (Booch et al., 2005). U UML-u se obično navode sa kompletnim potpisom:

```
nazivOperacije(parametar : Tip = podrazumevano) : TipPovratka
```

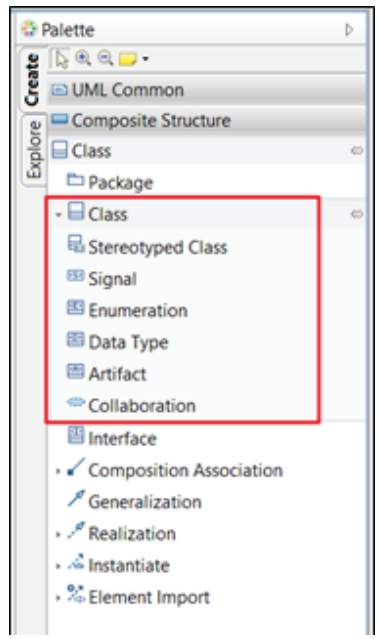
U ranim fazama analize metode mogu biti opisne, ali u fazi projektovanja softvera je preporučeno potpuno navođenje parametara i povratnih tipova (Larman, 2004), kako bi kasnije generisanje koda bilo čisto i nedvosmisleno (Fowler, 2004). Metode se prikazuju u trećem odeljku klase:



Primer klase *PhoneBook* sa jasno definisanim potpisima metoda:

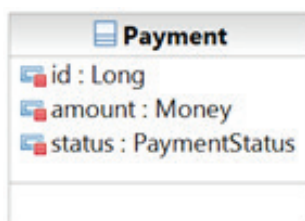


Kako bi se obezbedilo precizno i semantički bogato modelovanje softverskog sistema, UML dijagram klasa razlikuje više tipova klasa i povezanih elemenata, od kojih svaki ima jasno definisanu namenu:

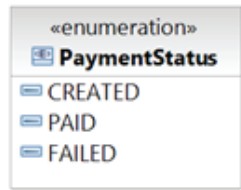


Slika 7.13. Vrste klasa u okviru IBM RSA alata

- ***Stereotyped Class*** - Klasa proširena stereotipom (npr. «entity», «controller», «service»), koji precizira njenu ulogu u arhitekturi ili domenu. Stereotipi omogućavaju semantičko obogaćivanje modela bez menjanja osnovne UML notacije.
- ***Enumeration*** - Enumeracije se koriste kada je skup mogućih vrednosti unapred poznat i konačan. One povećavaju čitljivost modela i sprečavaju nevalidne vrednosti atributa u dizajnu sistema. U klasama se najčešće koristi kao **tip atributa**. Na primer: klasa *Payment* ima atribute *id:long, amount:Money* i status:*PaymentStatus*.

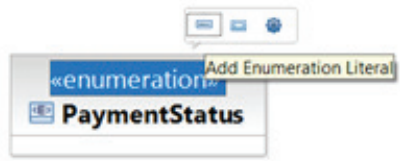


S obzirom da *PaymentStatus* ima konačnu skup dozvoljenih vrednosti npr. CREATED, PAID, FAILED, tada se može kreirati *enumeration* tip klase:

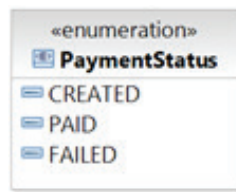


Koraci kreiranja enumeration klase u IBM RSA alatu su:

- Dodati Enumeration na dijagram klasa – iz palete izabrati Enumeration, postaviti na dijagram i promeniti ime u *PaymentStatus*.
- Dodati vrednosti (*literals*) – U prečicama odabrati opciju Add Enumeration Literal:



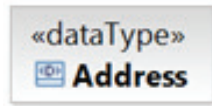
Uneti redom: CREATED, PAID i FAILED. Konačan izgled bi bio:



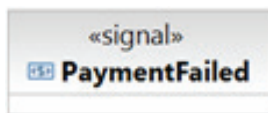
Dakle, *PaymentStatus* je tip aributa *status* i nema potrebe da se crta asocijacija, agregacija ili kompoziicja. Sama činjenica da je naveden kao tip je dovoljna.

- ***Data Type*** – UML element koji predstavlja vrednosni tip (*value object*) i služi za opis strukture podataka bez sopstvenog identiteta i nezavisnog životnog ciklusa. Data Type se koristi kao tip atributa u drugim klasama i uvek postoji u okviru objekta koji ga sadrži. Tipični primeri Data Type elemenata su:
- Address (ulica, grad, poštanski broj, država)
- Money (iznos, valuta)
- Period (datumOd, datumDo)

U UML dijagramu klasa, Data Type se najčešće označava stereotipom «dataType».



- **Signal** - predstavlja događaj ili poruku koja se šalje asinhrono, što znači da pošiljalac ne čeka da primalac završi obradu poruke. Signal samo obaveštava da se nešto desilo. Za razliku od poziva metode, signal ne poziva konkretnu operaciju, već pokreće reakciju kod objekta koji ga primi. Na primer, kada je plaćanje završeno, sistem može poslati signal „PaymentCompleted". Taj signal mogu da „slušaju" različiti delovi sistema, npr. sistem za slanje email potvrde, sistem za ažuriranje statusa porudžbine, sistem za analitiku i dr. Pošiljalac ne zna ko će sve reagovati na signal — samo ga emituje. Na primer:



PaymentFailed kao signal označava da je došlo do neuspešnog plaćanja. Objekti koji primaju signal mogu reagovati, npr. promeniti stanje (Order → FAILED), obavestiti korisnika, zapisati događaj u log i sl. Najčešće se koristi u *state machine* dijagramima (kao okidač promene stanja), u event-driven sistemima, kod sistema sa notifikacijama i asinhronim procesima i sl.

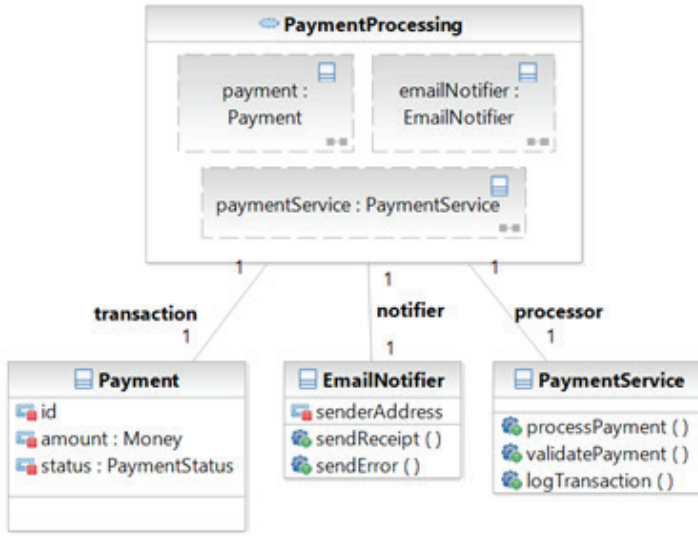
Dakle, signal predstavlja asinhroni događaj koji obaveštava sistem da se nešto desilo, bez direktnog pozivanja konkretne metode.

- **Artifact** - Predstavlja fizički ili logički rezultat razvoja softvera, kao što su fajl, biblioteka, izvršni program ili konfiguracioni resurs. Češće se koristi u *deployment* i *component* dijagramima nego u dijagramima klasa. Drugim rečima, Artifact je ono što se dobije kao izlaz iz razvoja softvera. Na primer, u aplikaciji za plaćanje, artefakti mogu biti:
  - *payment-service.jar* — izvršna Java aplikacija
  - *application.properties* — konfiguracioni fajl
  - *email-notifier.dll* — biblioteka za slanje email-a.



Ovi elementi nisu klase, već fizičke ili logičke jedinice koje sadrže implementaciju klasa.

- **Collaboration** – opisuje grupu elemenata (najčešće objekata ili klasa) koji zajedno učestvuju u realizaciji jedne funkcionalnosti sistema. Fokus nije na unutrašnjoj strukturi pojedinačnih klasa, već na načinu na koji one međusobno sarađuju. Na primer, za funkcionalnost „Obrada plaćanja", u saradnji mogu učestvovati:
  - *PaymentService* — koordinira proces
  - *Payment* — sadrži podatke o plaćanju
  - *EmailNotifier* — šalje potvrdu korisniku

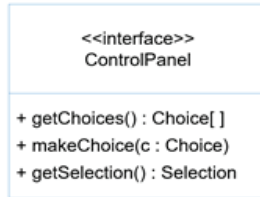


Slika 7.14. Primer *collaboration* klase

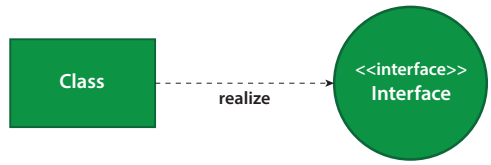
Dakle, *collaboration* ne opisuje šta klasa jeste, već kako klase zajedno rade da bi se ostvarilo određeno ponašanje sistema.

Interfejs predstavlja skup javno dostupnih operacija bez njihove implementacije, čineći time ugovor koji svaka klasa mora da ispuni ukoliko ga realizuje (OMG, 2017). Interfejsi ne sadrže atribute i ne mogu se instancirati, već služe isključivo za definisanje ponašanja koje konkretne klase treba da obezbede. U UML-u, interfejs se može prikazati na dva načina:

- kao pravougaonik sa stereotipom «interface» iznad njegovog naziva



- u obliku tzv. "lollipop" notacije, gde se interfejs prikazuje kao mali krug povezan linijom sa klasom.

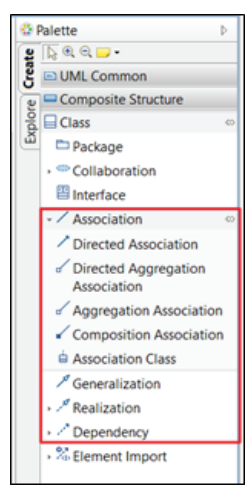


Veza između klase i interfejsa prikazuje se relacijom realizacije («realize»), koja je predstavljena isprekidanom linijom sa praznim trouglom okrenutim ka interfejsu (Booch et al., 2005).

Relacije između klasa određuju kako objekti međusobno komuniciraju i zavise jedni od drugih. UML definiše različite vrste veza, ali multiplicitet i navigabilnost (strelica) su ključni za razumevanje modela (Larman, 2004; OMG, 2017). Multiplicitet (npr. 0..* ili 1..*) određuje koliko instanci jedne klase može biti povezano sa instancom druge. Ispravno navođenje ovih elemenata utiče na semantiku modela, implementaciju, integritet podataka i arhitekturu sistema (Fowler, 2004).

7.3.2. Vrste relacija i zavisnosti između klasa u UML dijagramima

Da bi se precizno opisali odnosi između elemenata sistema, dijagram klasa razlikuju više tipova relacija koje se koriste u zavisnosti od namene modela i nivoa apstrakcije. U objektno orijentisanom modelovanju, UML omogućava prikazivanje kako trajnih strukturalnih odnosa između klasa, tako i privremenih veza koje nastaju kroz njihovu međusobnu upotrebu.



Slika 7.15. Vrste relacija na dijagramu klasa u okviru IBM RSA alata

Asocijacija

Asocijacija predstavlja osnovnu semantičku vezu između dve ili više klasa i koristi se za opisivanje činjenice da su objekti tih klasa međusobno povezani i svesni jedni drugih u kontekstu sistema. Drugim rečima, asocijacija odgovara na pitanje: „Koje klase su u trajnoj ili konceptualnoj vezi?“ Na primer, porudžbina je vezana za kupca. U ovoj relaciji Kupac — Porudžbina, jedan kupac može imati više porudžbina i svaka porudžbina se odnosi na tačno jednog kupca. U UML-u se ova asocijacija može prikazati uz navođenje:

- naziva asocijacije (npr. naručuje ili ima porudžbinu)
- uloga (rola) na krajevima veze (kupac, porudžbina)
- multipliciteta (npr. 1 — 0..*)

Na ovaj način se precizira semantičko značenje odnosa između klasa, a ne samo njegovo postojanje.



Dakle, asocijacija može imati:

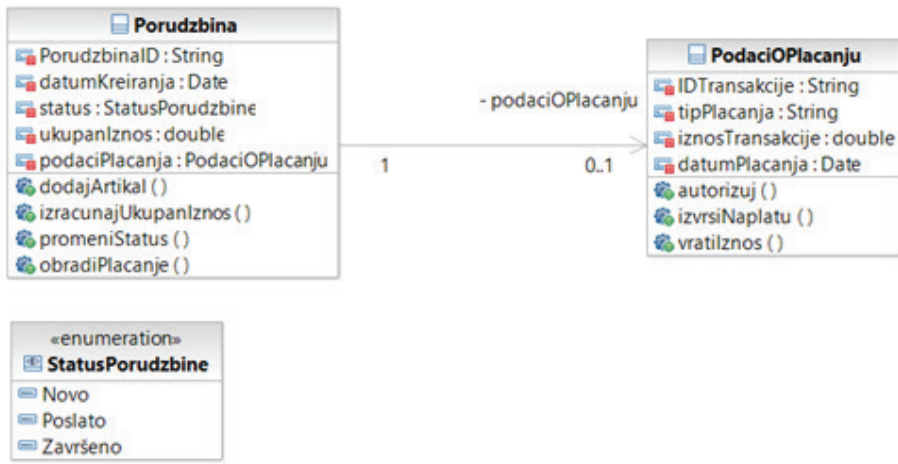
- ime — opisuje značenje veze (glagolska fraza)
- uloge — označavaju ulogu koju klasa ima u relaciji
- multiplicitete — određuju koliko instanci jedne klase može biti povezano sa instancom druge klase
- navigabilnost (opciono) — pokazuje smer pristupa objektima

Asocijacija je, dakle, najopštija i najčešća relacija na dijagramima klasa i predstavlja UML relaciju kojom se modeluje semantička veza između klasa, pri čemu se može precizirati njen naziv, uloge i multiplicitete radi jasnijeg tumačenja odnosa.

Usmerena asocijacija

Usmerena asocijacija (*Directed Association*) je specijalizovani oblik opšte asocijacije koja dodaje koncept **navigabilnosti** (smera pristupa) relaciji. Dok opšta asocijacija implicira da su dve klase međusobno povezane i da je veza u načelu dvosmerna, usmerena asocijacija jasno definiše da **samo jedna klasa poznaje drugu klasu i ima sposobnost da joj pristupi**. Ako klasa A poznaje klasu B, promena u B **ne utiče direktno** na A.

Na primer, u sistemu za onlajn kupovinu, klasa Porudžbina (*Order*) mora poznavati klasu PodaciOPlaćanju (*PaymentDetails*) kako bi obradila transakciju. Međutim, klasa *PodaciOPlaćanju* ne mora da poznaje sve detalje porudžbine (poput liste artikala, adrese dostave, itd.) — ona samo treba da obradi ukupan iznos porudžbine.



Slika 7.16. Primer usmerene asocijacije

Klasa Porudžbina poznaje i drži referencu na tačno jedan objekat PodaciOPlaćanju. Navigacija je ovde neophodna: Porudžbina treba da pozove metodu *obradiPlacanje*() na objektu PodaciOPlaćanju. Klasa PodaciOPlaćanju ne poznaje objekat Porudžbina (uopšte ili kao trajni atribut), čime ostaje nezavisna i može se ponovo koristiti za plaćanje i drugih stvari (pretplate, donacije, itd.) bez potrebe da se menja svaki put kada se promeni struktura klase Porudžbina.

Ovaj smer asocijacije obezbeđuje modularnost: klasa za plaćanje ostaje "čista" i fokusirana isključivo na transakciju, dok je odgovornost za obradu i povezivanje sa kupcem, adresama i artiklima ostavljena klasi Porudžbina.

Usmerena asocijacija je često model koji se koristi u praksi kada modelar želi da izbegne složenost dvosmerne navigacije, posebno u velikim sistemima.

Agregacija

Agregacija je slaba relacija između klasa koja prikazuje da jedan objekat (celina) sadrži ili se sastoji od drugih objekata (delova), ali ti delovi mogu postojati i samostalno. Drugim rečima, agregacija odgovara na pitanje: „Koji objekti čine deo nekog drugog objekta, a da pri tome zadržavaju sopstveni životni ciklus? Na primer, jedan tim se sastoji od više igrača. Veza između njih je agregacija, jer igrači mogu postojati i samostalno, čak i ako tim prestane da postoji. Treba imati na umu, da tim ne može da postoji bez igrača, igrači čine njegov sastavni deo.

Relacija agregacije se u UML-u prikazuje kao prazni romb na kraju veze kod klase tipa “celina” (npr. Tim). Strelica pokazuje na deo (npr. Igrač). Multipliciteti se mogu dodati kao i kod asocijacije. Jedan tim se sastoji od više igrača, a jedan igrač igra tačno u jednom timu.



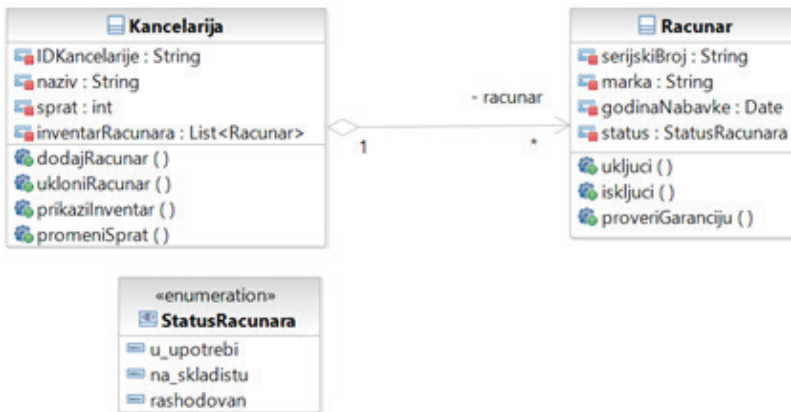
Dakle, agregacija je UML relacija tipa „celina–deo" kojom se prikazuje da jedan objekat sadrži druge objekte, ali da ti delovi mogu postojati i samostalno.

Usmerena agregacija

Usmerena agregacija (*Directed Aggregation Association*) je specijalizacija agregacije koja, pored odnosa celina-deo, dodatno specificira **smer navigacije**. To znači da samo celina (*agregator*) poznaje i ima pristup delovima, ali delovi ne moraju nužno poznavati celinu. Usmerena Agregacija kombinuje:

- Odnos celina-deo (agregacija): Prikazana praznim rombom na strani celine.
- Smer navigacije (usmerenost): Prikazana strelicom koja pokazuje od celine ka delu.

Zamislite sistem za praćenje imovine (*Asset Management*) u jednoj kompaniji. Imamo klasu koja predstavlja fizičku lokaciju - Kancelarija i klasu koja predstavlja opremu – konkretno računar.



Slika 7.17. Primer usmerene agregacije

Na dijagramu se uočava:

- Agregacija (Prazan romb):
  - Kancelarija sadrži Računare.
  - Ako se Kancelarija zatvori ili ugasi (prestane da postoji kao objekat u sistemu), Računar (deo) i dalje postoji. On će biti premešten u drugu kancelariju, prodat, ili uskladišten. Njegov životni ciklus je nezavisan od kancelarije.

- Usmerenost (Strelica):
  - Navigacija je jednosmerna: Kancelarija poznaje svoje Računare (ima listu Računara). To omogućava menadžeru da lako vidi sav inventar u određenoj kancelariji.
  - Klasa Računar ne poznaje (ne sadrži referencu na) Kancelariju. Kada razmišljamo o Računaru, ne zanima nas koja ga kancelarija trenutno koristi; zanimaju nas njegovi atributi (serijski broj, procesor, godina nabavke). Time Računar ostaje fleksibilan i "čist" entitet koji se lako može dodeliti drugim celinama.

Ovaj primer je praktičan jer jasno pokazuje da je celina odgovorna za upravljanje i pregled delova, dok delovi zadržavaju potpunu autonomiju i nezavisnost.

Usmerena agregacija služi za precizno definisanje relacije "deo-celina" sa nezavisnim životnim ciklusom delova, dok istovremeno ograničava navigaciju na smer od celine ka delovima.

Kompozicija

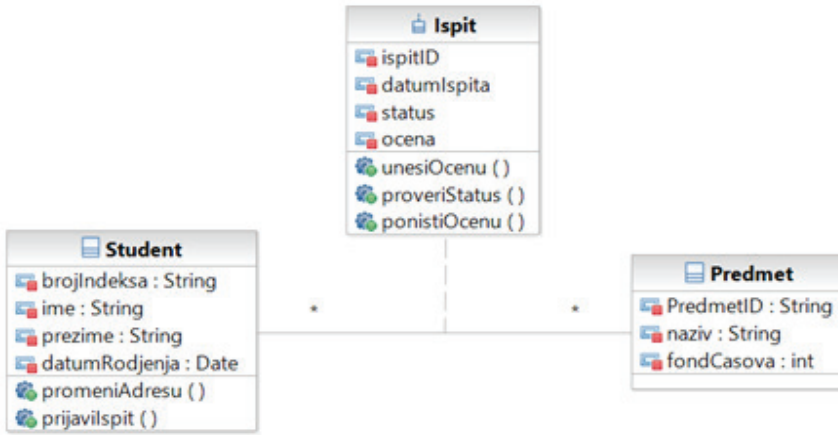
Kompozicija je jaka „celina–deo“ relacija između klasa, kod koje deo ne može postojati samostalno, već je njegov životni ciklus potpuno vezan za ceo objekat. Drugim rečima, kompozicija odgovara na pitanje: „Koji objekti čine deo celog sistema i prestaju da postoje kada sistem (celina) prestane da postoji?“ Na primer, kuća se sastoji iz više soba. Sobe ne mogu postojati samostalno bez kuće (ne postoji „neovisna soba“ van kuće). Ako se kuća uništi, sobe takođe prestaju da postoje. U UML-u se kompozicija prikazuje ispunjenim rombom na kraju veze kod klase tipa “celina”:



Kompozicija se koristi kada deo živi i umire zajedno sa celinom. Dakle, kompozicija je UML relacija tipa „celina–deo“ kojom se prikazuje da deo ima životni ciklus vezan za celinu, odnosno da ne može postojati nezavisno od nje.

Klasa asocijacije

Klasa Asocijacije ili asocijativna klasa (*Association Class*) je sledeći element sa palete i predstavlja napredan koncept koji se koristi kada je sama veza između dve klase složena i zahteva da ima sopstvene atribute i ponašanje. Uzmimo klasičnu asocijaciju između klase **Student** i klase **Predmet**. Ukoliko student polaže ispit iz određenog predmeta, **gde se čuva datum polaganja i ocena?** Ti podaci ne pripadaju samo Studentu (jer ocena zavisi od predmeta), niti samo Predmetu (jer ocena zavisi od studenta). Oni pripadaju **konkretnom Ispitu**. U ovom slučaju, kreiramo klasu asocijacije **Ispit** (*Exam*) koja povezuje Studenta i Kurs. Klasa Ispit će imati svoje atribute i metode:



Slika 7.18. Primer klase asocijacije

Pravila:

- Životni ciklus: Instanca klase asocijacije (npr. jedan konkretan Ispit) može postojati samo ako postoje instance klasa koje povezuje (u ovom slučaju, ako postoji taj Student i taj Predmet).
- Jedinstvenost: Svaki par objekata (Student i Predmet) može biti povezan samo jednom instancom klase asocijacije (Ispit).

Dakle, klasa asocijacije se koristi kada veza između dve klase ima sopstvenu semantičku vrednost i zahteva da bude modelovana kao poseban entitet sa jedinstvenim atributima i ponašanjem. Ona efikasno rešava problem *many-to-many* asocijaciju (M:N), gde je informacija o vezi specifična za svaki par povezanih objekata.

Generalizacija

Generalizacija (nasleđivanje) predstavlja „is-a“ („je-tip“) vezu između nadklase i potklase. Na primer, pas je tip životinja. Životinja je nadklasa (superclass), a pas je potklasa (subclass). Pas nasleđuje sve atribute i metode životinje (npr. disanje(), kretanje()). Pas takođe može imati svoje dodatne atribute ili metode (npr. lajanje()). U UML-u ova relacija se prikazuje kao puna linija sa praznim trouglom okrenutim ka nadklasi. Strelica pokazuje da potklasa nasleđuje nadklasu.



Primer u programskom jeziku Java:

```
// Nadklasa
class Zivotinja {
    String ime;
    int godine;
    float tezina;
    void disanje() {}
    void kretanje() {}
    // Konstruktor i druge metode
}

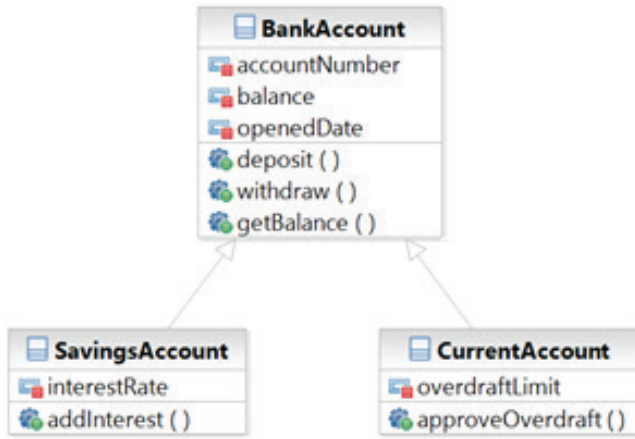
// Potklasa
class Pas extends Zivotinja {
    String rasa;
    String bojaKrzna;
    void lajanje() {}
    // Konstruktor i druge metode
}
```

Generalizacija omogućava ponovnu upotrebu koda, jer potklasa ne mora ponovo definisati atribute i metode nadklase. Takođe olakšava logičko grupisanje sličnih klasa i podržava polimorfizam, što znači da objekti potklase mogu biti korišćeni kao objekti nadklase u metodama i kolekcijama.

Radi dodatnog pojašnjenja, navodimo još jedan primer iz bankarskog sistema, gde različite vrste računa nasleđuju zajedničke karakteristike nadklase. Generalizacija se često koristi u modelovanju finansijskih sistema, gde različite vrste računa imaju zajedničke karakteristike, ali i svoje specifične atribute i metode. Na slici 7.19 je prikazan primer:

- BankAccount je nadklasa koja sadrži zajedničke atribute i metode svih računa:
  - Atributi: accountNumber, balance, openedDate
  - Metode: deposit(), withdraw(), getBalance()
- SavingsAccount (štedni račun) i CurrentAccount (tekući račun) su potklase:
  - SavingsAccount dodaje atribut interestRate i metodu addInterest()
  - CurrentAccount dodaje atribut overdraftLimit i metodu approveOverdraft()

U UML dijagramu generalizacija se prikazuje punom linijom sa praznim trouglom okrenutim ka nadklasi BankAccount. Strelice pokazuju da potklase nasleđuju nadklasu i njene atribute i metode.

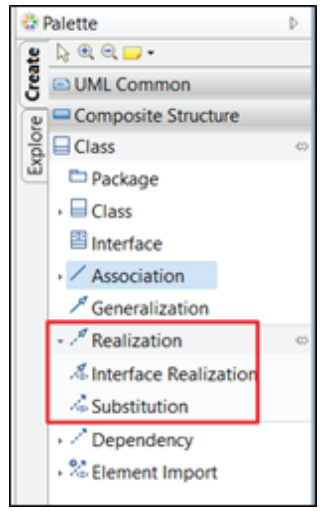


Slika 7.19. Primer generalizacije

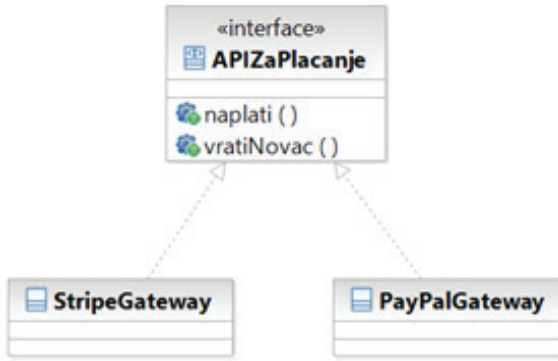
Ovaj primer jasno pokazuje princip „is-a / je-tip" veze: svaki štedni ili tekući račun je tip BankAccount, nasleđuje zajedničke osobine i dodaje svoje specifične karakteristike. Dakle, generalizacija je UML relacija „is-a" kojom potklasa nasleđuje atribute i metode nadklase, omogućavajući specijalizaciju i ponovnu upotrebu koda.

Realizacija

Relacija realizacije modeluje odnos u kojem jedan element ispunjava ugovor definisan drugim elementom. Najčešće se koristi za prikaz veze gde klasa obezbeđuje implementaciju za operacije definisane u interfejsu. Grupa dostupnih relacija koje se koriste za modelovanje ispunjavanja ugovora su: Realizacija, Interfejs Realizacija i Supstitucija (*Substitution*), kao što je prikazano na paleti alata.



Relacija realizacije je fundamentalni koncept koji modeluje da jedan element (najčešće klasa) ispunjava ugovor definisan drugim elementom (najčešće interfejsom). Na primer, interfejs APIZaPlacanje sadrži opšte metode naplati() i vratiNovac(). Komponente realizacije, StripeGateway i PayPalGateway, realizuju celokupan ugovor API-ja, tj. prikazuju konkretnu realizaciju metoda.



Slika 7.20. Primer generalne realizacije

Interfejs realizacija

Relacija interfejs realizacija (*Interface Realization*) pokazuje da klasa implementira (realizuje) sve operacije koje su deklarisane u nekom interfejsu. Ona je obavezna da ispoštuje ugovor. Na dijagramu klasa izgleda kao ispresecana linija sa praznim trouglom na strani interfejsa. Na primer, Interfejs pod nazivom Vozilo definiše metode: startujMotor() i ubrzaj(). Klasa Automobil realizuje interfejs Vozilo. To znači da klasa Automobil mora da pruži konkretnu implementaciju za startujMotor() i ubrzaj(). Interfejsi služe za definisanje opšteg ponašanja (ugovora), a klase koje ih realizuju pružaju specifičnu implementaciju tog ponašanja.

Substitucija

Relacija substitucije (*Substitution*) je formalno definisana kao vrsta realizacije, ali se najčešće odnosi na princip supstitucije (zamene). Relacija koja potvrđuje da objekat zamenjuje drugi objekat. U UML-u, supstitucija formalizuje Liskov princip supstitucije (LSP). Koristi se ako klasa A ima supstitucionu vezu sa klasom B, to znači da svaka instanca A može biti zamenjena instancom B (u smislu da instance B mogu korektno da izvršavaju sve operacije definisane za A). Na primer, ako klasa Krug ima supstitucionu vezu sa klasom Oblik, to znači da gde god u kodu očekujemo objekat tipa Oblik, možemo bez problema koristiti objekat tipa Krug. Iako se ova veza može formalno prikazati na dijagramima (koristeći isti simbol kao realizacija ili strelicu sa ključnom reči «substitute»), u praksi se češće modeluje kroz Generalizaciju (nasleđivanje), jer nasleđivanje automatski podrazumeva princip supstitucije.

Zavisnost

Zavisnost (*Dependency*) je veza tipa „koristi" i predstavlja situaciju u kojoj jedna klasa kratkotrajno koristi drugu kroz parametar, lokalnu promenljivu ili povratnu vrednost. Označava se isprekidanom strelicom ka elementu koji se koristi. Ova relacija ne označava vlasništvo, ne znači da se čuva objekat, niti da postoji trajna veza, već ukazuje na to da klasa zavisi od druge jer koristi neki njen element (metod, tip, parametar). Na primer, klasa PaymentService predstavlja servis koji obrađuje plaćanja i koja zavisi ili koristi klasu EmailNotifier.



Na primer, ukoliko u kodu imamo ovu situaciju:

```
class PaymentService {
    void processPayment(Payment payment, EmailNotifier notifier) {
        notifier.sendReceipt(payment); }
}
```

Vidimo da klasa *PaymentService* ima metodu *processPayment*() koja ima dva parametra: payment (objekat klase Payment koji predstavlja informacije o plaćanju) i *notifier* (objekat klase EmailNotifier koji šalje email potvrdu). Ova metoda koristi objekat klase *EmailNotifier* tako što poziva metodu sendReceipt() da pošalje email potvrdu koristeći informacije iz objekta *payment*:

```
notifier.sendReceipt(payment);
```

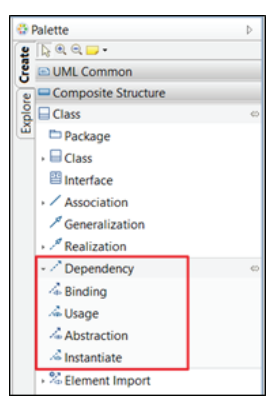
Dakle, metod *processPayment*() ne šalje email direktno, već poziva *sendReceipt*() tj. delegira taj zadatak *EmailNotifier* objektu. To znači da *PaymentService* zavisi od *EmailNotifier*-a samo tokom izvršavanja metode. Zavisnost je privremena i dešava se jer jedna klasa koristi drugu.

U UML-u se ovakva vrsta zavisnosti prikazuje isprekidanom strelicom od *PaymentService* ka *EmailNotifier*. Ova veza ne može biti asocijacija jer:

- *PaymentService* nema atribut tipa *EmailNotifier*
- ne poseduje ga
- ne čuva ga u stanju objekta
- već ga samo koristi kao parametar metode.

Kratkotrajna upotreba druge klase je upravo ono što UML definiše kao zavisnost (*dependency*). Dakle, relaciju zavisnosti u ovom kontekstu tumačimo kao: *PaymentService* obrađuje plaćanje, a u procesu koristi *EmailNotifier* da pošalje potvrdu o plaćanju.

UML definiše više specijalizovanih podtipova zavisnosti, koje su izvedene iz osnovne (generičke) zavisnosti:



Slika 7.21. Podtipovi relacije zavisnosti

- **Generička zavisnost** (*Dependency*) označava da jedan element zavisi od drugog jer ga koristi, referencira ili mu treba za ispravno funkcionisanje. Koristi se kada **nije potrebno precizirati specifičniji tip zavisnosti**.
- **Usage Dependency** («use») je specijalizovana verzija zavisnosti koja znači da jedan element koristi drugi prilikom izvršavanja (ali ga ne poseduje, ne kreira i ne čuva kao stanje). Na primer, *PaymentService* → «use» → *EmailNotifier*, jer ga dobija kao parametar u metodi i poziva njegovu operaciju. Ovo je najčešći tip zavisnosti u praksi.



Slika 7.22. Primer usage relacije

- ***Binding*** je posebna vrsta zavisnosti koja se koristi kod generičkih (parametarskih, template) klasa. Njegova svrha je da prikaže trenutak kada se apstraktni tip (parametar) generičke klase zameni konkretnim tipom. Drugim rečima, binding odgovara na pitanje: „Koji se stvarni tip koristi umesto generičkog parametra?" Generička klasa je „šablon", a *binding* predstavlja popunjavanje tog šablona konkretnim tipom. Na primer, ako imamo generičku klasu *Repository*<T> :

```
class Repository<T> {
  void save(T entity) { }
}
```

U aplikaciji koristimo:

```
Repository<User> userRepository;
```

U tom trenutku T više nije apstraktan, već je T konkretizovan kao User i tada generička klasa Repository<T> postaje konkretna instanca tipa Repository<User>. Primer zapisa:

```
Repository<User>  «bind»  →  Repository<T>
[T → User]
```

Time se eksplicitno pokazuje koja generička klasa se koristi i kojim konkretnim tipom je parametar popunjen. Dakle, Repository<T> ne zna unapred kojim tipom će upravljati, tek kada napišemo Repository<User>, mi vežemo generički parametar T za konkretan tip User. To „vezivanje" (T → User) je upravo binding.

Iako je *binding* tehnički zavisnost, on ne znači da jedna klasa poziva metode druge klase. Za razliku od zavisnosti «use» koja se koristi tokom izvršavanja, «bind» konkretizuje tip na nivou modela. Binding se dešava pri definisanju tipova, a ne tokom izvršavanja programa.

Binding se najčešće koristi prilikom modelovanja generičkih (parametarskih) elemenata sistema, kao što su *framework*-ovi, repozitorijumi, kolekcije i drugi infrastrukturni ili aplikativni obrasci koji koriste generičke tipove. Takođe se primenjuje za povezivanje šablona projektovanja (*design patterns*) zasnovanih na generičkim klasama sa konkretnim klasama sistema. U jednostavnim UML dijagramima ova relacija se često izostavlja radi preglednosti, ali u didaktičkim, analitičkim i dizajnerskim modelima ima značajnu ulogu u razjašnjavanju načina konkretizacije generičkih tipova.

- ***Abstraction*** je posebna vrsta zavisnosti koja se koristi da prikaže vezu između: apstraktnog modela ili koncepta i njegove konkretne realizacije ili, šire posmatrano, između dva modela na različitim nivoima apstrakcije, gde jedan model predstavlja pojednostavljeni ili apstraktniji pogled na drugi. Drugim rečima, *abstraction* odgovara na pitanje: „Kako je jedan apstraktni opis povezan sa konkretnim načinom njegove realizacije?" Apstrakcija opisuje šta sistem treba da radi, dok realizacija opisuje kako se to konkretno ostvaruje. Na primer, apstraktni model su pravila, ugovori, koncepti, a konkretna realizacija su klase, metode, strukture koje ta pravila sprovode.

Najčešće korišćena podvrsta *abstraction* relacije je realization, kojom se prikazuje da klasa implementira interfejs ili konkretna komponenta realizuje apstraktni opis. Abstraction se ne koristi samo za interfejse, već može da prikaže vezu između analitičkog modela i dizajna, između konceptualnog pojma i implementacione klase ili mapiranje poslovnog pravila na konkretni algoritam.

U UML-u ova relacija se prikazuje kao isprekidana linija sa strelicom koja ide ka apstraktnijem elementu. Označava se stereotipom «realize» (najčešće) ili opštim «abstraction». Ova relacija je važna jer jasno razdvaja šta sistem radi od toga kako je implementiran, zatim olakšava promene implementacije bez menjanja apstraktnog modela i poboljšava razumljivost i održivost dizajna.

- ***Instantiate*** je specijalizovana vrsta zavisnosti koja se koristi kada jedna klasa direktno kreira instancu druge klase. Ova relacija eksplicitno prikazuje da jedna klasa poziva konstruktor druge klase i time je odgovorna za njeno instanciranje. Drugim rečima, instantiate odgovara na pitanje: „Ko kreira objekat ove klase?" Kada klasa sama pozove *new* nad drugom klasom ona kontroliše stvaranje objekta, odlučuje kada i kako će objekat biti napravljen i preuzima deo odgovornosti za njegov životni ciklus. U UML-u se ta odluka o kreiranju objekta modeluje relacijom instantiate. Na primer,

```
class OrderService {
    void confirmOrder() {
        EmailNotifier notifier = new EmailNotifier();
        notifier.sendConfirmation();
    }
}
```

OrderService kreira objekat klase EmailNotifier, gde instanca ne dolazi kao parametar niti se dobija spolja već se kreiranje dešava unutar metode. Na dijagramu klasa ovaj primer bi izgledao:



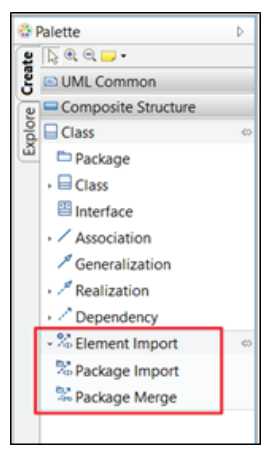
Ova relacija ne znači da OrderService trajno poseduje EmailNotifier, niti podrazumeva da čuva atribut objekta EmailNotifier, već ukazuje na to da je samo lokalno kreira i koristi. Ukoliko uporedimo sa drugim relacijama:

- «use» → klasa koristi drugu klasu (npr. kao parametar)
- «instantiate» → klasa kreira drugu klasu (*new*)
- asocijacija → klasa poseduje ili dugoročno referencira drugu klasu

Instantiate je, dakle, konkretnija informacija od obične zavisnosti kojom se označava da jedna klasa direktno kreira instancu druge klase, odnosno da je odgovorna za njeno instanciranje.

Elementi za rad sa paketima

Paket (*Package*) u UML-u služi za grupisanje logički povezanih elemenata sistema (klasa, interfejsa, drugih paketa) u jednu celinu, slično folderu u operativnom sistemu ili namespace-u u programskom jeziku. *Element Import, Package Import* i *Package Merge* se koriste u Dijagramima paketa (*Package Diagrams*) za modelovanje načina na koji veće logičke celine (paketi) međusobno sarađuju.



Slika 7.23. Vrste elemenata za rad sa paketima

Uvoz elementa (*Element Import*)

Ova relacija se koristi kada želite da **samo jedan specifičan element** iz jednog paketa učinite direktno dostupnim (vidljivim) u drugom paketu. Dakle, element import je mehanizam kojim jedan paket (uvoznik) dobija pristup pojedinačnom elementu (npr. klasi ili interfejsu) iz drugog paketa (izvora). Označava se kao isprekidana linija sa otvorenom strelicom koja ukazuje na uvezeni element, sa ključnom reči «elementimport». Na primer, zamislite da razvijate finansijsku aplikaciju koja mora da radi sa datumima i vremenom. Dostupni paketi i njihovi elementi su:

- BibliotekaDatumi - veliki paket koji sadrži sve funkcije za datume i vreme: Datum, Vreme, PoređenjeDatuma, FormatirajDatum i dr.
- formatirajDatum() - jedna specifična klasa/funkcija koja je potrebna aplikaciji koju razvijate.
- FinansijskiIzveštaji - paket koji treba da koristi samo jednu funkciju, konkretno formatirajDatum().



Slika 7.24. Primer elementi import relacije

Ukoliko ne bi koristili «elementimport», unutar paketa FinansijskiIzveštaji morali biste pisati BibliotekaDatumi::FormatirajDatum.izvrši(). Zahvaljujući «elementimport», možete direktno pisati FormatirajDatum.izvrši().

Relacija Uvoz Elementa služi za čišćenje koda. Umesto da uvozite čitav paket («package-import») i rizikujete da se nazivi klasa preklapaju (npr. imate i FinansijskiIzveštaji::Izveštaj i BibliotekaDatumi::Izveštaj), vi selektivno uvozite samo ono što vam je neophodno za jednostavnu referencu, a da pritom zadržavate punu kontrolu nad *namespace*-om.

Uvoz paketa (*Package Import*)

Ovo je uobičajenija relacija kojom se omogućava korišćenje **svih javnih elemenata** jednog paketa unutar drugog. Relacija između paketa uvoznika i paketa izvora, omogućava da se neosnovana imena (bez prefiksa paketa) koriste za referenciranje **javnih članova** paketa izvora. Označava se isprekidanom linijom sa otvorenom strelicom koja ukazuje na uvezeni paket, sa ključnom reči «import». Na primer, razvijate aplikaciju za upravljanje logistikom koja treba da pristupi svim klasama *eCommerce* paketa koji sadrži klase Kupac, Porudžbina i Artikal.



U paketu UpravljanjeLogistikom sada možete koristiti klase Kupac i Porudžbina bez prefiksa. Ovo je ekvivalentno using namespace u C++ ili *import package*.* u Javi.

Spajanje paketa (*Package Merge*)

Ovo je najsloženija relacija. Koristi se za spajanje sadržaja dva paketa u jedan, pri čemu se elementi istog imena spajaju (implicitna generalizacija). Package Merge je odnos koji definiše da se sadržaj izvornog paketa **širi** tako da uključuje elemente ciljnog paketa. Ako i izvorni i ciljni paket imaju elemente istog imena, definicije elemenata izvora se proširuju kako bi uključile definicije iz cilja. Prikazuje se kao isprekidana linija sa otvorenom strelicom koja ukazuje na paket cilj, sa ključnom reči «**merge**». Na primer, imamo dva paketa koja želimo da spojimo:

- paket A (StandardniKlijent) sadrži klasu Klijent (sa osnovnim atributima: ime, adresa)
- paket B (PremiumKlijent) sadrži klasu Klijent (sa dodatnim atributima: VIPstatus, datumRegistracije).



Kao rezultat spajanja, sve klase iz StandardniKlijent postaju dostupne u PremiumKlijent. Elementi istog imena se spajaju, tj. krajnji paket PremiumKlijent sada efektivno ima jednu klasu Klijent koja uključuje atribute i iz StandardniKlijent i iz PremiumKlijent. Koristi se za modelovanje proširenja osnovnih modela.

7.3.3. Koraci modelovanja dijagrama klasa

Modelovanje klasa se posmatra kao most između faze **analize zahteva** i faze **implementacije**. Pravilno definisan dijagram klasa služi kao 'nacrt' baze podataka i objekata u kodu, obezbeđujući arhitektonsku konzistentnost i olakšavajući razumevanje kompleksnih sistema. Kreiranje dijagrama klasa nije jednofazni proces, već se sprovodi kroz iterativni set koraka. Iako se specifična terminologija može razlikovati, akademski pristup modelovanju obično sledi pet ključnih, međusobno zavisnih faza. Cilj je da se apstraktni koncepti poslovnog domena sistematski prenesu u precizne, tehničke strukture koje će činiti osnovu objektnog koda.

Koraci modelovanja dijagrama klasa su:

1. Identifikacija ključnih entiteta (Klasa)
2. Definisanje atributa
3. Identifikacija ponašanja (Metoda)
4. Definisanje relacija (Asocijacije, Multiplicitet)
5. Uvođenje specijalnih relacija (Nasleđivanje, Asocijativne Klase)

Na primeru, jedne Rent-A-Car agencije, prikazaćemo korake modelovanja dijagrama klasa.

Korak 1: Identifikacija ključnih entiteta (Klasa)

Identifikujemo glavne aktere i objekte u sistemu:

Klasa	Opis entiteta
Vozilo	Vozilo koje je dostupno za iznajmljivanje.
Klijent	Osoba koja iznajmljuje vozilo.
Iznajmljivanje	Zapis o jednom ugovoru o iznajmljivanju (od-do, cena).
Filijala	Lokacija (poslovnica) gde se vozilo preuzima i vraća.

Korak 2: Definisanje atributa

Dodajemo informacije koje su bitne za praćenje svakog entiteta:

Klasa	Atributi
Vozilo	`registarskaOznaka: String, marka: String, model: String, godinaProizvodnje: int, trenutnaKilometraza: int, dostupnost: boolean`
Klijent	`JMBG: String, ime: String, prezime: String, datumRođenja: Date, brojTelefona: String`
Iznajmljivanje	`ugovorID: String, datumPreuzimanja: Date, datumVraćanja: Date, očekivanaCena: double, stvarnaCena: double`
Filijala	`filijalaID: String, naziv: String, adresa: String`

Korak 3: Identifikacija ponašanja (Metoda)

Definišemo operacije koje svaka klasa može da izvrši:

Klasa	Metode
Vozilo	`+ beležiServis(datum: Date): void, + proveriStatus(): String`
Klijent	`+ kreirajRezervaciju(datumi, model): Iznajmljivanje`
Iznajmljivanje	`+ izračunajCenu(kilometraza): double, + obradiPreuzimanje(): void, + obradiVraćanje(): void`
Filijala	`+ dodajAutomobil(auto): void`

Korak 4: Definisanje relacija (Asocijacije, Multiplicitet)

Klase u relaciji	Tip relacije	Multiplicitet	Objašnjenje
Klijent i Iznajmljivanje	Asocijacija	1..*	Jedno iznajmljivanje pripada jednom klijentu; klijent može imati više iznajmljivanja.
Vozilo i Iznajmljivanje	Asocijacija	1..*	Jedno iznajmljivanje se odnosi na tačno jedan automobil; Vozilo može da se iznajmi (rentira) više puta.
Filijala i Vozilo	Agregacija	1..*	Jedna filijala drži mnogo vozila. Vozilo može postojati i preći u drugu filijalu (životni ciklus je nezavisan).
Filijala i Iznajmljivanje (Preuzimanje)	Asocijacija	1..*	Jedno iznajmljivanje (preuzimanje vozila) se može obaviti u jednoj filijali; dok u jednoj filijali se može obaviti više iznajmljivanja (preuzimanja).

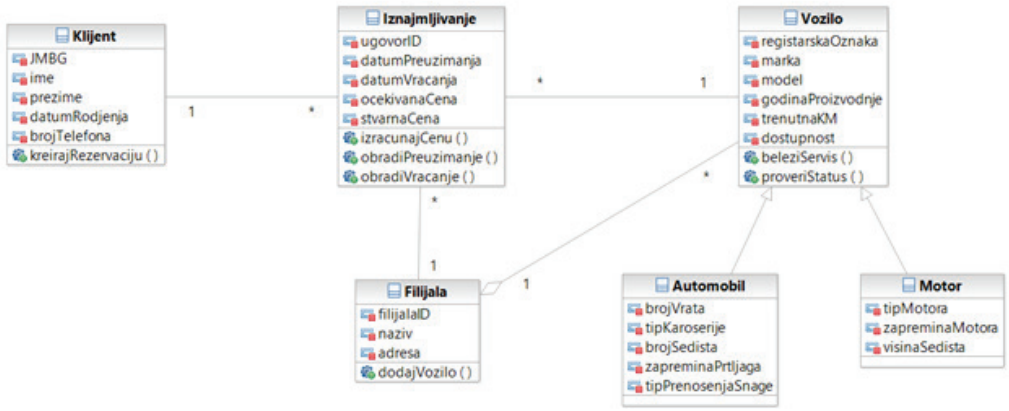
Korak 5: Uvođenje specijalnih relacija (Generalizacija)

Generalizacija (Nasleđivanje):

- Klasa Vozilo se generalizuje.
- Podklase: Automobil i Motor. One nasleđuju opšte atribute (npr. registarskaOznaka, marka) i dodaju specifične atribute (npr. brojVrata za automobile).

Ovaj proces omogućava kreiranje kompletnog i robusnog dizajna sistema pre samog kodiranja.

Finalni dijagram klasa bi izgledao:



Slika 7.25. Dijagram klasa rentacar agencije

Vežba 7.2. Kreirati prošireni dijagram klasa za rentacar agenciju koja je uvela nova pravila poslovanja.

Iskoristićemo prethodni primer da proširimo sa dodatnim pravilima rentacar agencije:

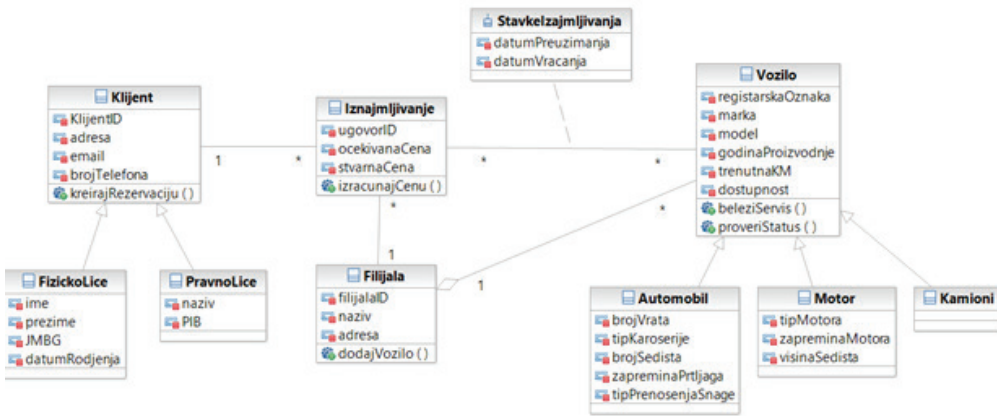
1. Rentacar agencija je proširila svoje usluge i sada želi da posluje i sa pravnim licima (preduzećima), prema kojima ima posebne pogodnosti.
2. Ne žele da limitiraju klijente po broju rentiranih vozila u jednoj rezervaciji, tako da bilo da je klijent pravno ili fizičko lice, mogu u jednoj rezervaciji rentirati više vozila (npr. dva automobila i jedan motor).
3. U slučaju rentiranja više vozila, nije neophodno da se sva vozila preuzmu u isto vreme. Datumi preuzimanja i vraćanja će se beležiti posebno po vozilima.
4. Žele da prošire delatnost i sada imaju mogućnost iznajmlijivanja i kamiona.

Potrebno je kreirati prošireni dijagram klasa koji će reflektovati nove zahteve i potrebe rentacar agencije.

Rešenje vežbe

Na osnovu novih pravila poslovanja, izmene na dijagramu klasa su sledeće.

Pravilo	Rešenje
Uvođenje poslovanja sa pravnim licima	S obzirom da rentacar posluje sada i sa fizičkim i sa pravnim licima, klasa Klijenti će biti generalizovana klasa (nadklasa) nad potklasama FizickaLica i PravnaLica.
U jednoj transakciji iznajmljivanja moguće je rentirati više vozila.	S obzirom da se jedno vozilo može rentirti više puta, a pri jednom izajmljivanju se mogu rentirati više vozila, relacija se menja u više prema vše.
Mogućnost rentiranja više vozila u jednoj transakciji iznajmljivanja.	S obzirom da su datumi preuzimanja i vraćanja vozila vezani za vozilo, u tom slučaju će se kreirati asocijativna klasa između Izajmljivanja i Vozila, gde će stajati dodatni atributi: datumPreuzimanja i datumVraćanja.
Žele da prošire delatnost i sada imaju mogućnost iznajmlijivanja i kamiona.	Dodaće se još jedna potklasa Kamioni koja će nasleđivati klasu Vozila.



Slika 7.26. Prošireni model dijagrama klasa rentacar agencije

Dostupni alati za modelovanje dijagrama klasa kao što su Enterprise Architect, IBM RSA, Visual Paradigm, Sparx, i IDE-integracije (npr. IntelliJ, Eclipse) omogućavaju *model-to-code* i *reverse-engineering*.

Vežba 7.3. Kreirati konceptualni dijagram klasa bankarskog sistema i transakcija

Na osnovu dole navedenih poslovnih pravila, kreirati kompletan i konzistentan konceptualni dijagram klasa za bankarski informacioni sistem.

Osnovne poslovne funkcije i entiteti:

1. **Banka i Filijale**: Banka posluje kroz mrežu Filijala. Svaka Filijala je povezana sa tačno jednom Bankom. Banka upravlja i odgovara za Filijale.
2. **Bankomati**: Bankomati su smešteni ili u okviru Filijale ili na udaljenoj lokaciji, ali se uvek administrativno vode pod jednom Filijalom. Fizička lokacija Bankomata se može promeniti (može preći pod nadležnost druge Filijale).

Vlasništvo i računi:

1. **Klijent**: Banka vodi evidenciju o Klijentima. Klijent mora imati jedinstvenu identifikaciju (JMBG/PIB) i kontakt podatke.
2. **Račun**: Svaki Račun mora pripadati barem jednom Klijentu. Jedan Klijent može imati više Računa. Račun može pripadati i više Klijenata (u slučaju ovlašćenja nad računom).
3. **Vlasnički odnosi:** Sistem mora biti u stanju da zabeleži dodatne informacije o odnosu Klijenta i Računa, kao što su: datum kada je Klijent postao vlasnik računa i uloga Klijenta (npr. vlasnik, ovlašćeno lice, punomoćnik).

Ugovori i obaveze:

1. **Ugovor**: Klijent stupa u pravni odnos sa Bankom putem Ugovora (npr. Ugovor o kreditu, Ugovor o štednji).
2. **Povezivanje ugovora**: Svaki Ugovor mora biti vezan za tačno jednog Klijenta (pravna odgovornost). Ipak, Ugovor se mora vezati i za tačno jedan Račun, jer se finansijsko sprovođenje ugovora (isplata, naplata rate) vrši preko tog računa.

Transakcije i načini izvršenja:

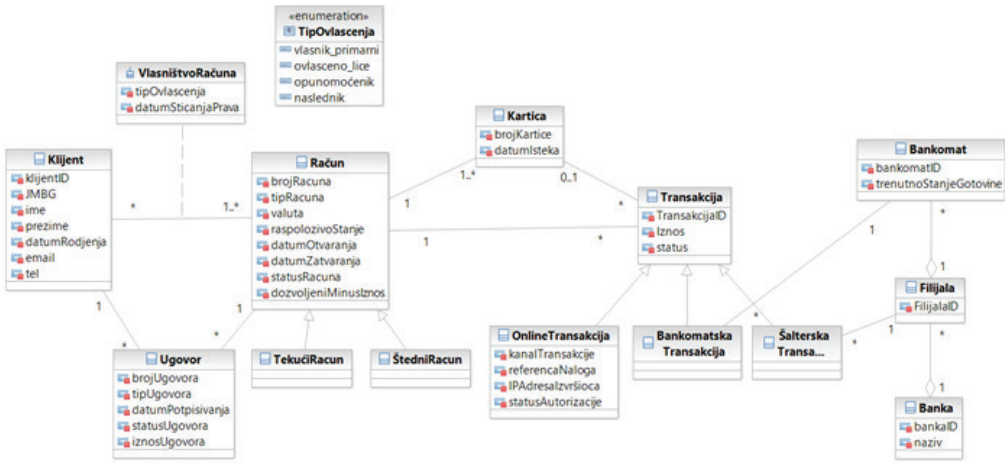
1. **Primarna veza Transakcije**: Svaka Transakcija mora biti obavezno i direktno povezana sa jednim Računom, jer menja njegovo stanje. Transakcije postoje i kada ne uključuju kartice (npr. obračun kamate).
2. **Vrste transakcija**: Transakcije se dele na tri glavne kategorije, zavisno od načina izvršenja:
   - **Online transakcije**: Izvršavaju se digitalno (putem aplikacija ili online bankarstva).
   - **Šalterske transakcije**: Izvršavaju se u Filijali, uz direktno učešće službenika.
   - **Bankomatske transakcije**: Izvršavaju se putem Bankomata.
3. **Lokacija i Instrumenti:**
   - Online transakcija mora imati zabeležene detalje o digitalnom kanalu i može biti povezana sa Karticom (ako je to bio instrument plaćanja).
   - Šalterska transakcija mora biti vezana za Filijalu u kojoj je izvršena.
   - Bankomatska transakcija mora biti vezana za Bankomat na kome je izvršena, kao i za Karticu koja je korišćena.

Rešenje vežbe

Konceptualni dijagram klasa je model na visokom nivou koji se koristi u ranim fazama analize softvera. Njegova glavna svrha je da prikaže poslovne entitete (ključne koncepte) u sistemu i odnose među njima. Osnovne karakteristike su:

- **Fokus na poslovnoj logici**: Modeluje šta sistem radi i koje koncepte koristi, a ne kako će biti implementiran (npr., ne obazire se na detalje programiranja ili baze podataka).
- **Klase**: Predstavljaju glavne koncepte iz domena (kao što su Klijent, Račun, Transakcija).
- **Relacije**: Definišu veze između entiteta, uz obavezno navođenje Multipliciteta (npr. 1..*).
- **Konceptualni atributi**: Uključuje samo najvažnije atribute koji služe za jedinstvenu identifikaciju i opis entiteta.

U kontekstu banke, konceptualni dijagram nam pomaže da se dogovorimo o tome šta tačno znači "Klijent", kako se definiše "Vlasništvo Računa" i koji su svi mogući oblici "Transakcije" pre nego što počnemo da pišemo ijednu liniju koda. Služi kao nacrt za komunikaciju između poslovnih analitičara, korisnika i razvojnog tima.



Slika 7.27. Dijagram klasa bankarskog sistema

Na narednoj slici 7.28 je ovaj konceptalni model dopunjen sa dodatnim atributima i metodama, koji bi se dalje mogao proširiti na logički dijagram klasa.

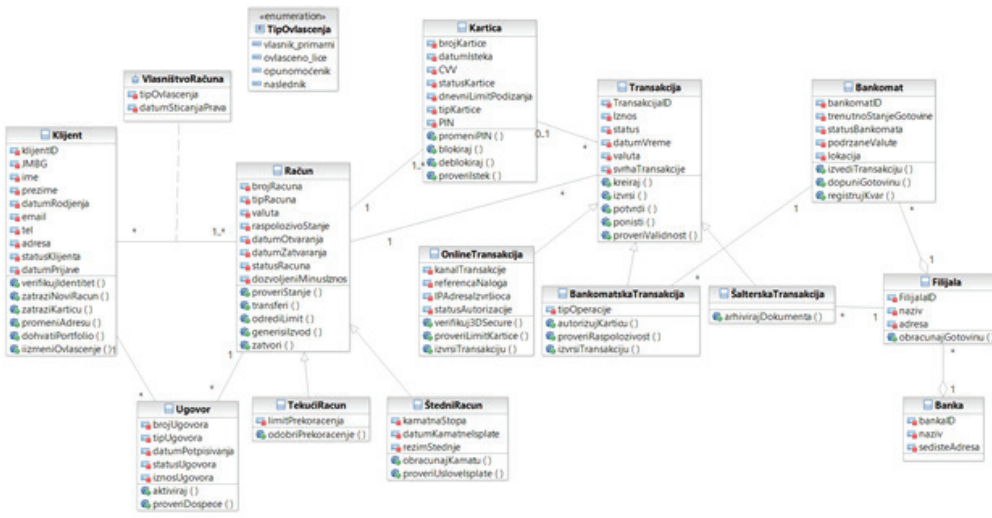
U kontekstu UML modelovanja:

1. **Konceptualni dijagram klasa** (*Conceptual Class Diagram*) fokusira se na poslovne domene. On prikazuje ključne entitete (klase) i njihove međusobne odnose (asocijacije i multiplicitet). U ovoj fazi nije potrebno prikazati **atribute** i **metode**.
2. **Logički dijagram klasa** (*Logical Class Diagram*) je korak dalje u detaljizaciji. On obuhvata:
   - Sve klase i veze definisane u konceptualnom modelu.
   - **Sve ključne atribute** (sa definisanim tipovima podataka, npr. String, Decimal, Date).
   - **Sve ključne metode** (sa definisanim ulaznim parametrima i povratnim vrednostima).

Drugim rečima, **Logički dijagram klasa** je **detaljna razrada** konceptualnog modela, gde se koncepti iz poslovnog sveta prevode u strukture koje su spremne za implementaciju (klase sa atributima i ponašanjem/metodama).

U zavisnosti od metodologije (npr. modelovanje baze podataka), ponekad se koristi sledeća podela:

- **Konceptualni model**: Samo entiteti i veze.
- **Logički model**: Entiteti, veze i svi atributi (sa tipovima podataka).
- **Fizički model**: Logički model prilagođen specifičnoj bazi podataka (tabele, indeksi, primarni/strani ključevi).



Slika 7.28. Prošireni konceptualni dijagram klasa bankarskog sistema

Zadatak 7.2. Kreirati UML dijagram klasa koji modeluje osnovne elemente onlajn prodavnice

Obavezne klase koje dijagram treba da sadrži su: Korisnik, Proizvod, Korpa, Porudžbina i Plaćanje. Po potrebi je dozvoljeno uvesti i dodatne klase. Pravila koja dijagram mora da ispuni su:

Korisnik i njegovi podaci

1. Korisnik i korpe – Jedan Korisnik ima jednu aktivnu Korpu, koja se koristi za pripremu kupovine, dok se prethodne kupovine čuvaju kroz Porudžbine.
2. Korisnik i porudžbine - Jedan Korisnik može imati više Porudžbina koje su ranije kreirane.
3. Korisnik i plaćanja - Svako Plaćanje mora biti povezano sa Korisnikom koji ga je pokrenuo, bez obzira da li je plaćanje uspešno ili neuspešno.

Korpa, porudžbina i proizvodi

1. Korpa i Porudžbina sadrže više različitih Proizvoda i uz svaki proizvod se čuva i količina.
2. Brisanje korpe i porudžbine - Ako se obriše Korpa → briše se i njen sadržaj i ako se obriše Porudžbina → briše se i njen sadržaj
3. Proizvodi postoje samostalno - Proizvod postoji nezavisno od Korpe i Porudžbine. Brisanje Korpe ili Porudžbine ne briše Proizvod.

Veza između korpe, porudžbine i plaćanja

1. Razlika između Korpe i Porudžbine - Korpa se može menjati (dodavanje, uklanjanje proizvoda), dok se Porudžbina više ne menja nakon što je kreirana.
2. Porudžbina se kreira na osnovu sadržaja Korpe - Na dijagramu klasa ne treba crtati direktnu vezu između klase Korpa i klase Porudžbina
3. Plaćanje se uvek odnosi na Porudžbinu, a ne na Korpu. Jedna Porudžbina može imati više pokušaja Plaćanja.

POGLAVLJE 8



OBJEKTNO-ORIJENTISANO PROJEKTOVANJE SOFTVERA

Objektno-orijentisano projektovanje softvera

8.1. Uvod u projektovanje softvera
8.2. Dijagram stanja
8.3. Dijagram klasa sa primenjenim obrascima projektovanja softvera
8.4. Dijagram komponenti
8.5. Dijagram raspoređivanja

Cilj poglavlja:

Cilj ovog poglavlja je transformacija apstraktnih funkcionalnih zahteva u precizan i formalizovan projektni model (*Design Model*), koji služi kao arhitektonska osnova za implementaciju. Fokus je na primeni objektno-orijentisane paradigme za specifikaciju statičke strukture (dijagram klasa), dinamičkog ponašanja (dijagram stanja) i fizičke topologije sistema (dijagrami komponenti i raspoređivanja). Poseban naglasak stavljen je na upotrebu UML standarda i obrazaca projektovanja (*Design Patterns*), čime se osigurava kreiranje skalabilnog, modularnog i održivog softverskog rešenja.

8. OBJEKTNO-ORIJENTISANO PROJEKTOVANJE SOFTVERA

8.1. UVOD U PROJEKTOVANJE SOFTVERA

Prelazak sa logičkih na fizičke modele koji se kreiraju u fazi projektovanja softvera je trenutak kada prestajemo da pričamo o "željama" i počinjemo da pričamo o **kodu**. U akademskoj literaturi ovaj proces se naziva **detaljni dizajn**. Dakle, faza projektovanja softvera (*Software Design*) predstavlja formalnu tranziciju od definisanja zahteva ka implementaciji sistema.

U kontekstu životnog ciklusa razvoja softvera, projektovanje premošćuje jaz između toga šta sistem treba da radi (faza analize) i kako će sistem to postići (faza implementacije). Dok je analiza softvera usmerena na domen problema (*Problem Domain*), rezultirajući konceptualnim i logičkim modelima koji odražavaju zahteve i poslovnu logiku korisnika, projektovanje softvera je usmereno na domen rešenja (*Solution Domain*). Cilj projektovanja je da se kreira *Design Model* – sveobuhvatni arhitektonski i detaljni nacrt sistema.

Ključna razlika između faza analize i projektovanja leži u nivou apstrakcije (Tabela 8.1):

1. Analiza: Modeli su tehnološki nezavisni i fokusiraju se na jasnoću poslovnih koncepata.
2. Projektovanje: Modeli se transformišu u tehnološki zavisne specifikacije (fizički modeli), koje služe kao direktna osnova za generisanje programskog koda. U ovoj fazi donose se ključne odluke o strukturi, arhitekturi, interfejsima i algoritmima, optimizujući sistem za skalabilnost, održivost i efikasnost.

Tabela 8.1. Uporedna analiza faza analize i projektovanja softvera

Karakteristika	Faza Analize Softvera (*Software Analysis*)	Faza Projektovanja Softvera (*Software Design*)
Primarni Fokus	**Domen problema** (*Problem Domain*). Razumevanje šta sistem treba da radi iz perspektive korisnika i poslovnih procesa.	**Domen rešenja** (*Solution Domain*). Utvrđivanje kako će se sistem implementirati da bi ispunio zahteve.
Glavni Cilj	Kreiranje potpunog, konzistentnog i nedvosmislenog modela funkcionalnosti sistema.	Utvrđivanje strukture, komponenti i interfejsa, obezbeđujući skalabilnost, modularnost i održivost.
Modeli	**Konceptualni/Logički modeli**. Modeli su bliski stvarnom svetu (npr. *use case* dijagrami, dijagrami aktivnosti, konceptualni/logički dijagram klasa).	**Fizički modeli**. Detaljni modeli strukture i ponašanja (Dijagram klasa sa implementacionim detaljima, Dijagram stanja, Dijagram komponenti, Dijagram raspoređivanja).

Karakteristika	Faza Analize Softvera (*Software Analysis*)	Faza Projektovanja Softvera (*Software Design*)
Nezavisnost od tehnologije	Model je apstraktan i nezavisan od konkretnog programskog jezika, baze podataka ili operativnog sistema.	Projektni model je usmeren ka implementaciji i uzima u obzir odabrane tehnologije (npr. stil mikroservisa, kontejnerizacija).
Izlazni artefakt	Specifikacija zahteva (SRS) i model domena (*Domain Model*).	Model projektovanja (*Design Model*) i projektna dokumentacija.

Kako u fazi analize već imate logičke modele, prelazak u fazu projektovanja (dizajna) nije puko crtanje novih dijagrama, već detaljnija razrada postojećih. U fazi detaljnog projektovanja sistema kreiraju se dijagram stanja, fizički dijagram sekvenci i fizički dijagram klasa kako bi se precizirala unutrašnja logika i struktura softvera. Uporedo sa definisanjem arhitekture, a pre same implementacije, izrađuju se dijagram komponenti, koji prikazuje organizaciju i zavisnosti softverskih modula, kao i dijagram raspoređivanja (*deployment*), kojim se modeluje fizička konfiguracija resursa i distribucija komponenti na hardverske čvorove.

Tabela 8.2. Pregled UML dijagrama prema fazama životnog ciklusa razvoja softvera

Faza razvoja	**Cilj** (Fokus)	**Dijagrami** (Artefakti)
Analiza	**Šta?** (Razumevanje problema i domena)	• **Use case dijagram** (Zahtevi korisnika) • **Dijagram aktivnosti** (Modelovanje procesa) • **Logički dijagram klasa** (Model domena) • **Logički dijagram sekvenci** (Interakcija korisnik-sistem)
Projektovanje (*Design*)	**Kako?** (Tehničko rešenje i arhitektura)	• **Dijagram stanja** (Životni ciklus kompleksnih objekata) • **Fizički dijagram sekvenci** • **Detaljni** (Fizički) **dijagram klasa** • **Dijagram paketa** (Organizacija podsistema)
Implementacija i razmeštaj	**Realizacija** (Kod i infrastruktura)	• **Izvorni kod** (*Source code*) • **Dijagram komponenti** (Fizički moduli, biblioteke, baze) • **Dijagram raspoređivanja** (*Deployment*) (Serveri, mreže, hardver)

U fazi projektovanja kreiraju se sledeći dijagrami:

1. **Dijagram stanja** - ukoliko ključne klase imaju kompleksno ponašanje, npr. klasa Porudžbina prolazi kroz mnogo faza (npr. Kreirana -> Plaćena -> Poslata -> Stornirana), tada se prvo kreira dijagram stanja, koji definiše **pravila poslovanja**. Ne može se krenuti dalje npr. u kreiranje fizičkog dijagrama sekvenci ako nisu definisana pravila npr. da li je dozvoljeno preći iz stanja "Kreirana" direktno u "Poslata" bez uplate. Dijagram stanja diktira logiku grananja (if/else ili State pattern) koju ćete kasnije ucrtati u fizički dijagram sekvenci.
2. **Fizički dijagrama klasa**: uvode se tehničke klase koje niste imali u analizi (npr. klase za pristup bazi podataka - DAO, kontrolere, interfejse, klase za bezbednost).
3. **Fizički dijagram sekvenci**: uzimate jedan use case i "razbijate" ga koristeći te nove tehničke klase. Ovde određujete tačne potpise metoda. Tokom rada na sekvencama, verovatno ćete shvatiti da vam treba još neka pomoćna klasa, pa se u tom slučaju opet vraćate na dijagram klasa.

U zavisnosti od vrste sistema koji se razvija, redosled kreiranja dijagama u fazi projektovanja bi bili:

- ukoliko je sistem "*Event-driven*" (upravljan događajima): prvo bi se kreirali dijagrami stanja, zatim fizički dijagram sekvenci, pa onda fizički dijagram klasa.
- za "*Data-centric*" sisteme gde je fokus na podacima i CRUD operacijama: prvo se kreira fizički dijagram klasa, pa onda fizički dijagram sekvenci. Dijagram stanja se ovde često i ne crta jer su stanja trivijalna.

U slučaju da sistem ima objekte čije je ponašanje uslovljeno statusima (što je najčešće slučaj u informacionim sistemima), dijagram stanja je logičan prvi korak u projektovanju. On postavlja "zakone" po kojima će se objekti ponašati, a koje ćete kasnije detaljno razraditi kroz fizičke sekvence i klase.

8.2. DIJAGRAM STANJA

Za razliku od dijagrama klasa, koji prikazuju statičku strukturu sistema (atribute, metode i relacije), dijagram stanja omogućava sagledavanje životnog ciklusa objekta, počev od njegovog kreiranja, preko svih relevantnih međustanja, pa sve do završnog stanja.

U praksi, dijagram stanja se primenjuje iz dva suštinska razloga:

- **Validacija dijagrama klase**: Služi kao kontrolni mehanizam za proveru potpunosti dijagrama klasa. Pomoću njega utvrđujemo da li su svi definisani atributi i metode dovoljni da podrže predviđeno ponašanje objekta. Ako određena promena stanja zahteva podatak koji nismo predvideli kao atribut ili operaciju koju nismo definisali kao metodu, dijagram stanja će ukazati na te propuste u ranoj fazi dizajna.
- **Modelovanje kompleksne poslovne logike**: Dijagram stanja je idealan alat za vizuelizaciju poslovnih pravila, npr. ukoliko klijent potroši više od 100e dati mu besplatnu uslugu ili ukoliko je korisnik u proteklih šest meseci kontinuirano trošio više od 100€ mesečno, objekat menja stanje iz "Standard" u "VIP klijent".

Bez dijagrama stanja, ovakva pravila ostaju skrivena duboko u programskom kodu (kroz niz komplikovanih uslovnih grananja), dok ih dijagram stanja čini transparentnim, proverljivim i lako razumljivim kako programerima, tako i poslovnim korisnicima.

Dakle, dijagram stanja se ne crta samo da bi se dokumentovao kod, već da bi se vizuelizovala logika koja upravlja ponašanjem sistema. U softverskom inženjerstvu, on služi kao most između poslovnih pravila i tehničke implementacije. Neki ključni razlozi zašto je ovaj dijagram nezaobilazan alat su:

- **Upravljanje složenošću** (logika "Događaj-Odziv") - Mnogi objekti u softveru ne reaguju uvek isto na istu komandu. Njihov odziv zavisi od toga **u kakvom su stanju trenutno.** Na primer, dugme "Play" na video plejeru radi jednu stvar ako je video zaustavljen (pokreće ga), a sasvim drugu ako je video već pokrenut (pauzira ga). Dijagram stanja nam u ovom slučaju pomaže da mapiramo te varijacije bez pisanja stotina linija komplikovanih if-else naredbi u glavi.
- **Definisanje dozvoljenih i nedozvoljenih tokova** (protokol ponašanja za klasu) - dijagram stanja jasno definiše šta sistem sme da uradi u kom trenutku, čime sprečava logičke greške poput: "Pokušaj slanja računa kupcu koji još nije ni kreirao porudžbinu". Protokol predstavlja **dozvoljeni** (i nedozvoljeni) **redosled poziva operacija** (metoda) nad objektom.
- **Precizna komunikacija sa klijentima** - Klijenti (vlasnici biznisa) retko razumeju kod, ali odlično razumeju procese. Umesto da im objašnjavate algoritam, pokažete im dijagram: "Vidite, ovde porudžbina prelazi iz 'Plaćeno' u 'Poslato' tek kada magacioner potvrdi bar-kod." To je najbolji način da se **pre programiranja** potvrdi da li je softver koji pravite zapravo ono što klijentu treba.
- **Osnova za automatizovano testiranje** - Kada imate nacrtan dijagram stanja, testiranje postaje matematički precizno. Proverava se da li smo prošli kroz svaku strelicu (tranziciju) i svako stanje. Ako softver uđe u stanje koje nije na dijagramu, odmah znamo da imamo "bug" u sistemu.

- **Projektovanje reaktivnih i real-time sistema** - u sistemima kao što su bankomati, liftovi, pejsmejkeri ili mikrotalasne pećnice, redosled događaja je pitanje života ili smrti (ili bar velike materijalne štete). Dijagram stanja je glavni plan jer ovi sistemi nemaju klasičan početak i kraj koda, već stalno čekaju na spoljne nadražaje (događaje).

Dijagram stanja crtamo da bismo **razmišljali pre kucanja koda**. On nam omogućava da uočimo logičke propuste u fazi dizajna, gde je popravka besplatna, umesto u fazi produkcije, gde je popravka veoma skupa.

Glavni cilj dijagrama stanja (state machine ili statechart diagram) je da **modeluje dinamičko ponašanje objekta** odnosno načina na koji se objekat tokom svog životnog ciklusa kreće kroz različita stanja kao odgovor na događaje iz okruženja ili unutrašnje promene. Njegova primena direktno doprinosi razvoju **pouzdanog, testabilnog i održivog softvera**, što je ključno za komercijalni uspeh svakog složenog sistema.

8.2.1. Osnovni elementi dijagrama stanja

Objekat, kao instanca klase, tokom svog životnog veka prolazi kroz više različitih stanja, koja se mogu uočiti i na dijagramima sekvenci. Životni vek objekta traje od trenutka njegovog kreiranja do trenutka uništavanja ili prestanka rada sistema. Promene stanja nastaju kao posledica događaja, a svako stanje predstavlja skup jedinstvenih informacija o objektu i definiše njegov odziv na naredne događaje.

Dijagram stanja prikazuje:

- sva moguća stanja u kojima se objekat može nalaziti
- događaje koji izazivaju prelaze između stanja
- uslove pod kojima se prelazi mogu izvršiti
- akcije i aktivnosti koje se izvršavaju tokom prelaza ili boravka u stanju.

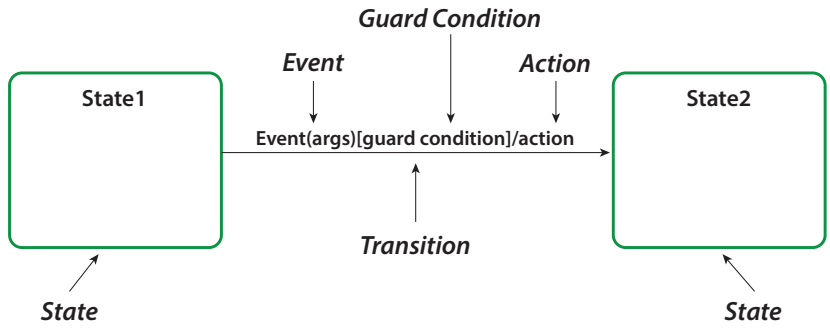
U praksi se dijagram stanja ne izrađuje za svaku klasu, već prvenstveno za **one klase koje imaju složeno ponašanje**, odnosno više od nekoliko značajnih stanja (uobičajeno više od tri).

Dijagram stanja (*statechart*) je **usmereni graf stanja** (čvorova) **povezanih vezama** (tranzicijama – usmerenim strelicama).

Osnovni elementi dijagrama stanja su:

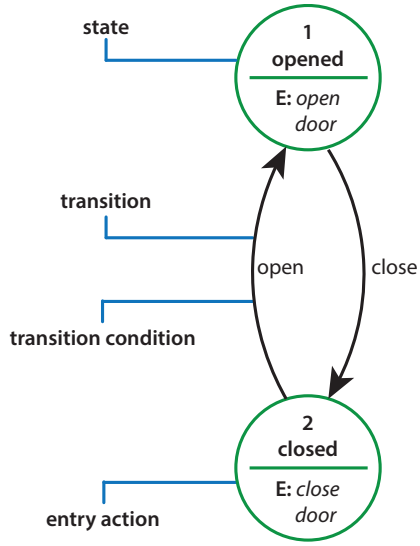
- **Stanje** (*State*) – predstavlja jedno stabilno stanje objekta u okviru njegovog životnog ciklusa i određuje kako objekat reaguje na događaje. Stanja se imenuju kao pridevi objekta, npr. ako se pravi dijagram stanja za klasu Klijent, onda bi nazivi stanja bili „Registrovan“, „Aktivan“, „Pasivan“, „VIP“, „Arhiviran“ i sl.
- **Tranzicija** (*Transition*) – prelaz iz jednog stanja u drugo, prikazan usmerenom strelicom.
- **Događaj** (*Event*) – okidač koji inicira tranziciju (npr. poziv metode, istekao vremenski interval, promena vrednosti atributa).

- **Uslov** (*Guard condition*) – logički (Boolean) izraz koji mora biti zadovoljen da bi se tranzicija realizovala.
- **Akcija** (*Action*) – kratka operacija koja se izvršava kao rezultat tranzicije.



Slika 8.1. Osnovni elementi dijagrama stanja

Na slici 8.2 se prikazuje pojednostavljen primer dijagrama stanja objekta „Vrata“ koji može imati dva stanja: otvorena (opened) i zatvorena (*closed*).



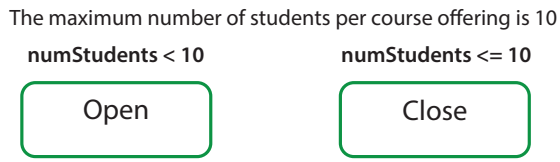
Slika 8.2. Jednostavan primer dijagrama stanja

8.2.2. Identifikovanje stanja, događaja i tranzicija

Proces izrade dijagrama stanja započinje identifikacijom stanja objekta.

Stanja se najčešće prepoznaju analizom:

- dinamičkih atributa klase (atributa čije vrednosti utiču na ponašanje objekta):



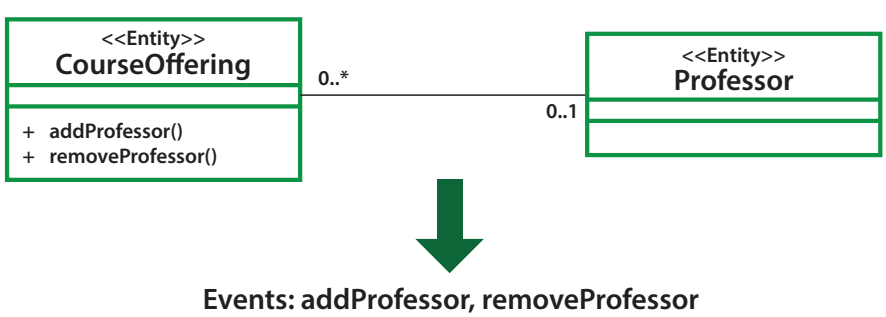
- postojanja ili nepostojanja veza sa drugim objektima:



- poslovnih pravila koja definišu dozvoljene faze u životnom ciklusu objekta.

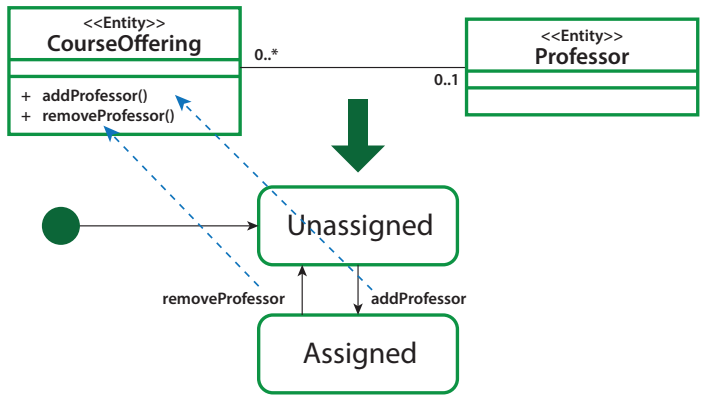
Događaji se identifikuju na osnovu interfejsa klase, odnosno metoda koje objekat izlaže drugim objektima, kao i na osnovu protokola komunikacije. Za svako stanje se zatim određuju:

- događaji koji su dozvoljeni u tom stanju,
- ciljna stanja do kojih ti događaji vode, eventualni uslovi (*guard conditions*) koji moraju biti ispunjeni.



Slika 8.3. Identifikovanje događaja (events) koji se mogu pronaći u interfejsima objekata ili protokolima

Tranzicije opisuju šta se dešava kao odziv na događaj.

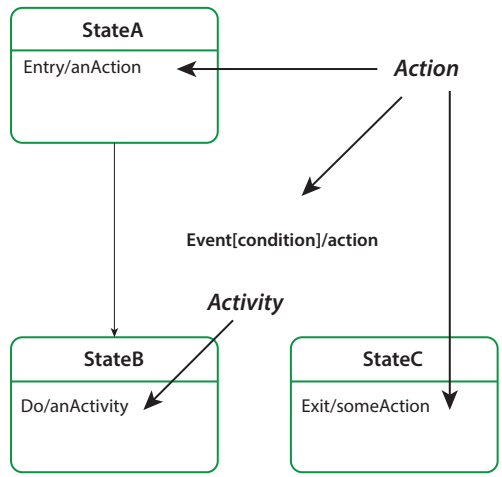


Slika 8.4. Uvidom u tranzicije na dijagramu stanja identifikuju se metode na dijagramu klasa i obratno

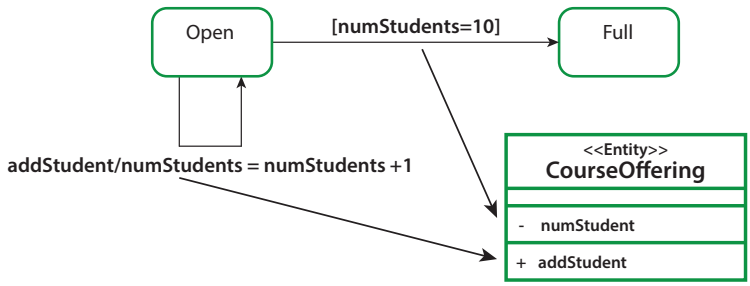
8.2.3. Aktivnosti i akcije

U UML u se razlikuju aktivnosti i akcije:

- **Aktivnosti** su vezane za stanje i izvršavaju se tokom boravka objekta u tom stanju. Mogu predstavljati složeniji proces koji se može prekinuti bez promene stanja.
- **Akcije** su vezane za tranzicije i izvršavaju se u trenutku prelaza iz jednog stanja u drugo.

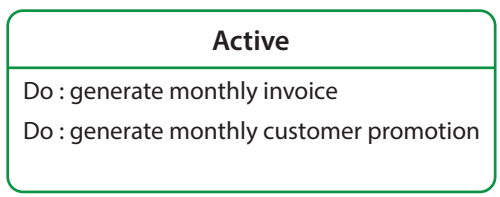


Slika 8.5. Razlika između aktivnosti i akcije



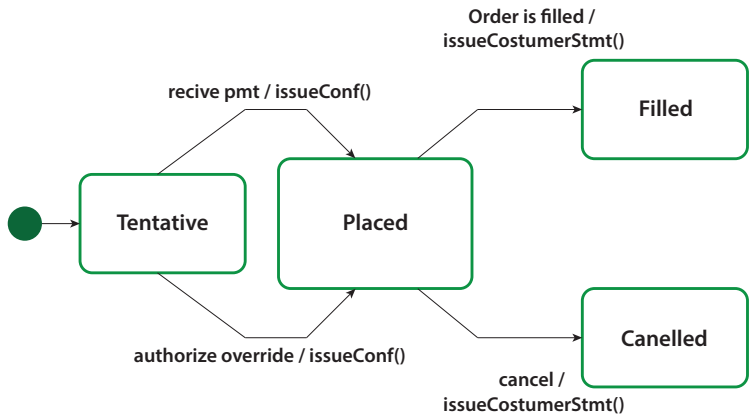
Slika 8.6. Primer aktivnosti i akcija u poređenju sa elementima klase

Interno stanje ukazuje na to šta objekat može da radi dok je u datom stanju. Da bi modelovali aktivnosti u stanju, koristi se ključna reč *Do*: koju prati jedna ili više aktivnosti. Ove aktivnosti će se izvršavati od trenutka kada objekat uđe u ovo stanje do trenutka kada ga napusti. Na primer, dok je klijent u VIP stanju, slati mu na email pogodnosti usluga.



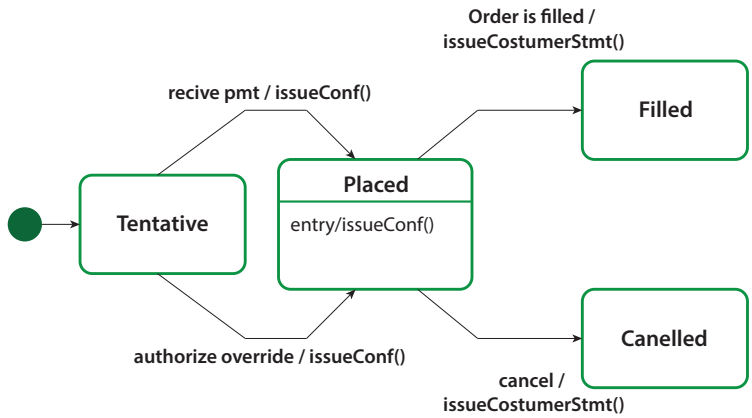
Slika 8.7. Primer internih aktivnosti objekta

Za modelovanje aktivnosti unutar stanja koristi se ključna reč do:, dok se za akcije na ulazu i izlazu iz stanja koriste ključne reči *entry* i *exit*. Ukoliko se ista akcija izvršava prilikom svakog ulaska u određeno stanje, preporučuje se njeno definisanje kao *entry* aktivnosti. Analogno tome, akcije koje se uvek izvršavaju prilikom napuštanja stanja mogu se definisati kao exit aktivnosti. Na primer, stanje porudžbine može da pređe iz porudžbine u pripremi (*Tentative*) u poslatu (*Placed*) i to bilo da se primi uplata (*receive pmt*) ili da se dobije odobrenje izuzetka (*authorize override*). Oba događaja zahtevaju istu akciju, a to je izdavanje potvrde porudžbine (*issueConf*()).



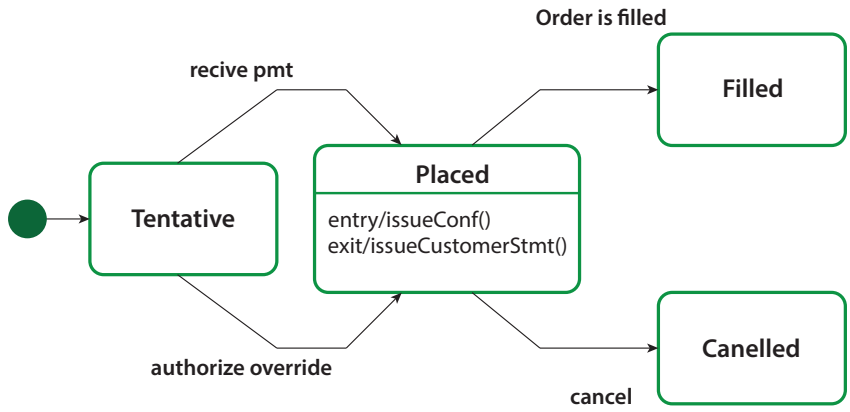
Slika 8.8. Primer dešavanja istih akcija prilikom ulaska u stanje „poslata porudzbina" (*Placed*) i izlaska iz tog stanja

S obzirom da se uvek dešava ista akcija *issueConf*() prilikom ulaskom u stanje „Placed", onda ćemo te akcije izbrisati sa tranzicija i prebaciti ih u stanje objekta pod *Entry* (ulazna) aktivnost.



Slika 8.9. Primer istih akcija nakon stanja „*Placed*"

Identičan postupak se može primeniti i za akcije koje su pridružene događajima koji napuštaju stanje tzv. izlazne (*exit*) akcije. S obzirom da se prilikom izlaska iz stanja „Placed" uvek događa ista akcija *issueCustomerStmt*() u tom slučaju ćemo je izbrisati sa tranzicija i dodai kao exit aktivnost unutar stanja „Placed".



Slika 8.10. Primer izlaznih aktinosti u stanju klase

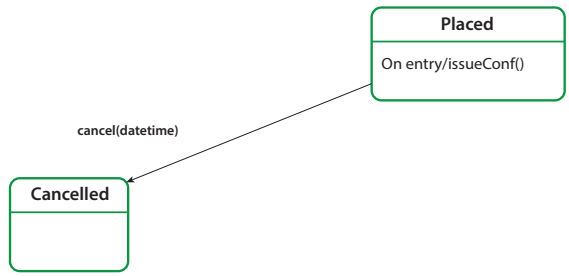
Napomena: Ukoliko postoji makar jedan izuzetak kada može da se desi da se ta akcija ne odigra onda je ne treba staviti kao ulaznu ili izlaznu aktivnost stanja.

8.2.4. Vrste događaja

U UML dijagramima stanja razlikuju se sledeće vrste događaja:

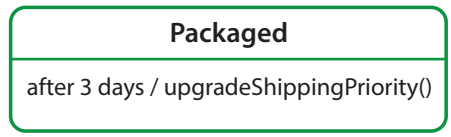
- *Call event* – nastaje kao rezultat poziva operacije objekta.
- *Time event* – aktivira se nakon isteka određenog vremenskog intervala (ključna reč *after*).
- *Change event* – aktivira se kada se ispuni određeni uslov nad stanjem ili atributima (ključna reč *when*).
- *Conditional event* – tranzicija se izvršava samo ukoliko je zadati uslov ispunjen.
- *Send event* – modeluje slanje poruke drugom objektu kao deo ponašanja objekta.

Događaji poziva (*Call event*) je najopštiji tip događaja u UML dijagramima stanja. Ovaj događaj predstavlja poziv operacije, odnosno izvršavanje jedne metode objekta. Na slici 8.9 je prikazana tranzicija iz stanja „Postavljena“ (*Placed*) u stanje „Otkazana“ (*Cancelled*). Tranzicija se aktivira (okida) događajem *cancel*(datetime), koji predstavlja jednu od operacija (metoda) klase Porudžbina (Order).



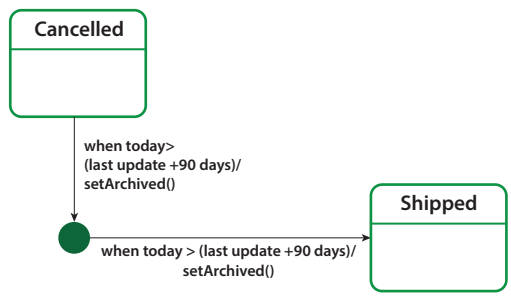
Slika 8.11. Primer događaja „*call event*"

Vremenski događaj (*Time event*) predstavlja događaj koji se aktivira nakon isteka određenog vremenskog intervala. U UML dijagramima stanja, vremenski događaji se definišu korišćenjem ključne reči *after*, čime se označava vremenski uslov za aktiviranje tranzicije. Na primer, porudžbina u stanju „Spakovana" čeka da bude isporučena. Ukoliko isporuka ne bude izvršena u roku od tri dana, aktivira se vremenski događaj *after*(3 days), nakon čega se povećava prioritet isporuke.



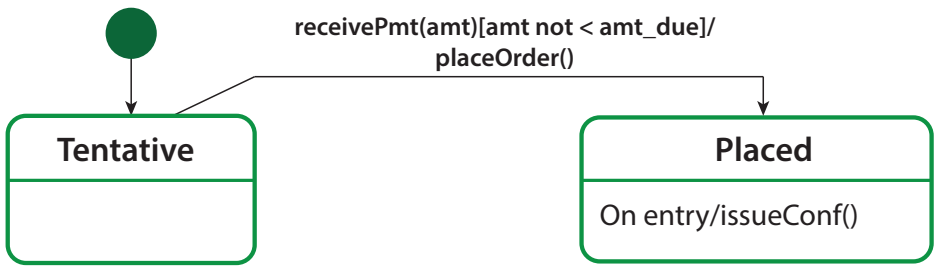
Slika 8.12. Primer *time event* događaja

Događaj promene (*change event*) predstavlja događaj koji se aktivira kada se ispuni određeni logički uslov nad stanjem objekta. U UML dijagramima stanja, ovaj tip događaja se definiše korišćenjem ključne reči **when**, kojoj se pridružuje odgovarajući izraz za proveru stanja. Na primer, stanje porudžbine može se proveravati u odnosu na vreme poslednje izmene. Ukoliko je proteklo više od 90 dana od poslednje izmene porudžbine, aktivira se događaj **when** (lastUpdate + 90 days < now).



Slika 8.13. Primer *change event* događaja

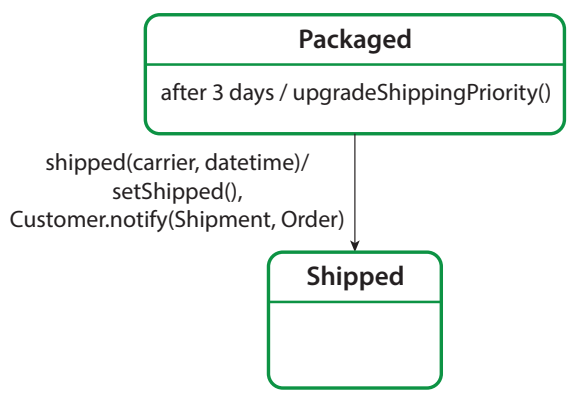
Uslovni događaji (*guard/conditional events*) koriste se za kontrolu izvršavanja tranzicija u UML dijagramima stanja. Kada se dogodi događaj koji okida tranziciju, proverava se pridruženi uslov. Ukoliko je uslov ispunjen (true), tranzicija se izvršava; u suprotnom, tranzicija se ne realizuje i događaj se zanemaruje. Na primer, porudžbina prelazi u stanje „Postavljena" samo ukoliko je plaćanje prihvaćeno i iznos uplate odgovara ukupnoj vrednosti porudžbine.



Slika 8.14. Primer *conditional event* događaja

Događaj slanja (*Send event*) se koristi za modelovanje situacija u kojima objekat inicira komunikaciju sa drugim objektom slanjem poruke. Ovim događajem se opisuje kada i zašto objekat obaveštava drugi objekat o određenoj akciji ili promeni stanja. Na primer, kada nastupi događaj **shipped**(carrier, datetime), izvršavaju se dve akcije:

1. 1poziv operacije **setShipped()**, kojom se ažurira stanje porudžbine i
2. slanje poruke objektu **Klijent**, čime se krajnji korisnik obaveštava da je njegova pošiljka poslata.



Slika 8.15. Primer *send event* događaja

8.2.5. Kompozitna i paralelna stanja

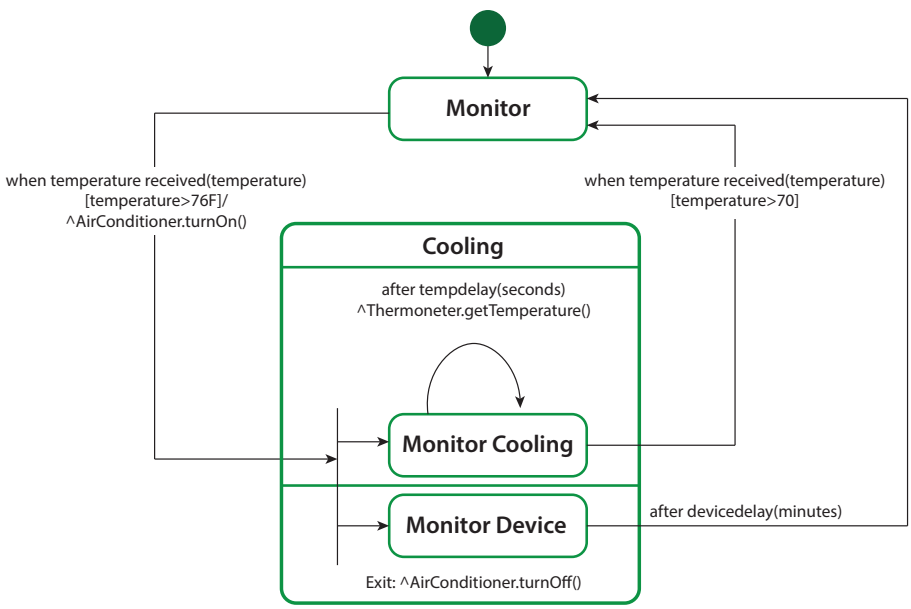
U UML dijagramima, stanja mogu biti:

- Prosta (jednostavna) stanja koja ne sadrže druga stanja.
- Kompozitna stanja koja u sebi sadrže podstanja i to:
  - sekvencijalna podstanja (jedna aktivna putanja) ili
  - paralelna (ortogonalna) podstanja raspoređena u regionima.

Termin nadstanje se koristi opisno i označava da se neko stanje nalazi iznad drugih u hijerarhiji, dok je kompozitno stanje zvaničan UML termin.

Modelovanje nadstanja i podstanja

Hijerarhijska organizacija stanja predstavlja standardan pristup za kontrolu kompleksnosti ponašanja objekta. **Nadstanja** (kompozitna stanja) grupišu povezana podstanja i omogućavaju definisanje zajedničkog ponašanja, dok **podstanja** opisuju detaljnije faze unutar jednog složenog stanja. Na primer, rad temorstata (Slika 8.16). Termostat započinje rad u stanju *Monitor*, gde neprekidno prati temperaturu okoline. Kada se detektuje temperatura viša od 76°F, termostat uključuje klima-uređaj i prelazi u stanje *Cooling*. Tokom hlađenja paralelno se odvijaju aktivnosti nadzora procesa hlađenja (Monitro Cooling) i rada uređaja (Monitor Device), uz periodično ponovno merenje temperature. Kada temperatura padne ispod 70°F, termostat isključuje klima-uređaj i vraća se u stanje *Monitor*.

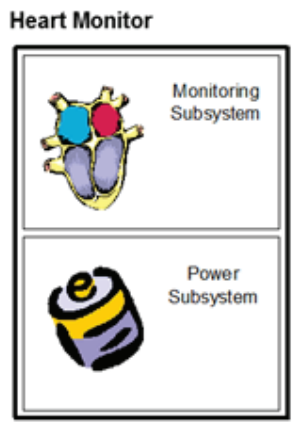


Slika 8.16. Dijagram stanja termostata sa paralelnim podstanjima u okviru kompozitnog stanja

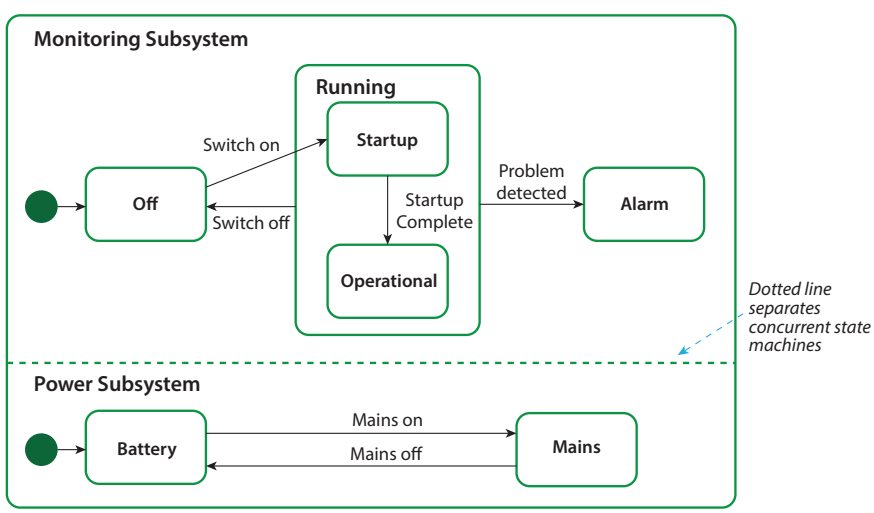
Paralelna stanja

Paralelna (ortogonalna) stanja omogućavaju istovremeno izvršavanje više međusobno nezavisnih podsistema unutar jednog objekta. Svaka regija kompozitnog stanja izvršava se paralelno, što je posebno značajno kod reaktivnih i ugrađenih sistema.

Složeni sistemi često sadrže paralelna stanja, jer se pojedini podsistemi mogu odvijati nezavisno jedan od drugog. Na primer, kod aparata za monitoring srca, jedno stanje može nadgledati rad srca, dok drugo istovremeno prati ispravno funkcionisanje samog aparata.



Detaljniji prikaz stanja:

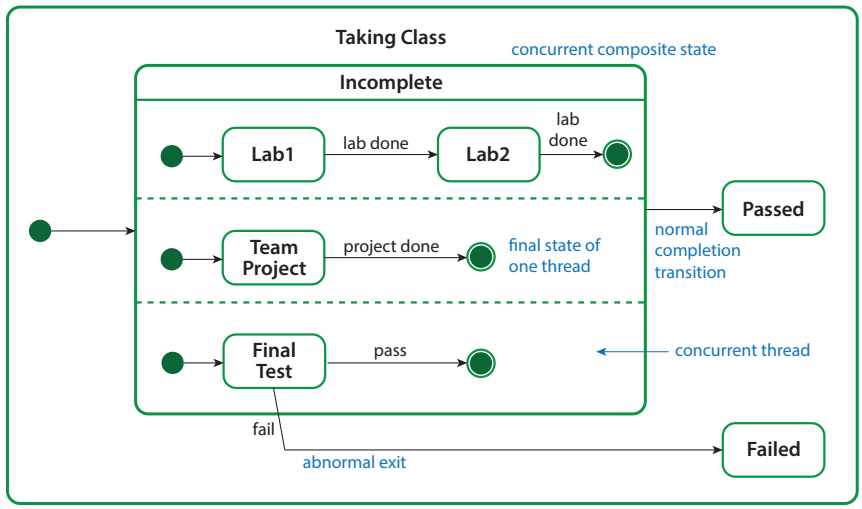


Slika 8.17. Podsistemi aparata za monitoring rada srca

Slika 8.18. prikazuje UML dijagram stanja za proces „*Taking Class*“ (pohađanje kursa). Dijagram prikazuje paralelno izvršavanje aktivnosti unutar kursa i moguće ishode (uspeh ili neuspeh). Kompozitno stanje “Incomplete” predstavlja trenutni status dok student još nije završio sve aktivnosti. Unutar “Incomplete” postoje tri paralelne (*concurrent*) niti:

- Lab1 → Lab2: student prvo radi Lab1, pa Lab2. Kada su oba urađena, ta nit se završava.
- Term Project: student radi projekat; završetak projekta označava kraj te niti.
- Final Test: student polaže završni test. Ako prođe, nit se završava; ako ne, aktivira se abnormal exit i stanje prelazi u “Failed”.

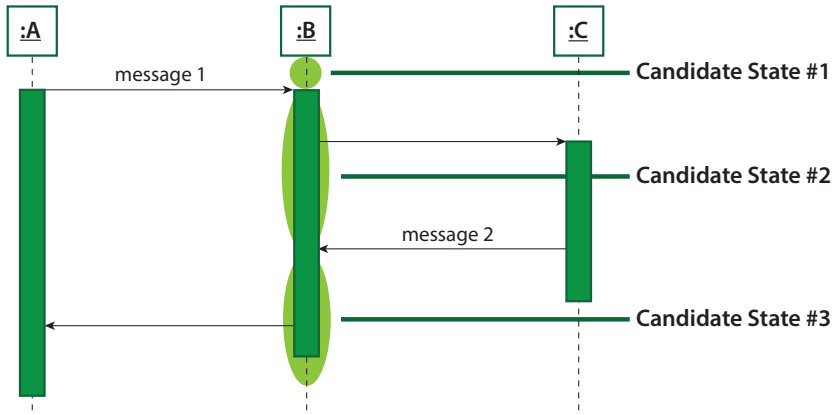
Kada sve niti završe normalno, događa se normal completion transition i stanje prelazi u Passed. Ako bilo koja nit završi neuspehom (npr. fail na Final Test), proces odmah prelazi u Failed.



Slika 8.18. Dijagram stanja za proces „*Taking Class*“ sa prikazom paralelenih izvršavanja aktivnosti

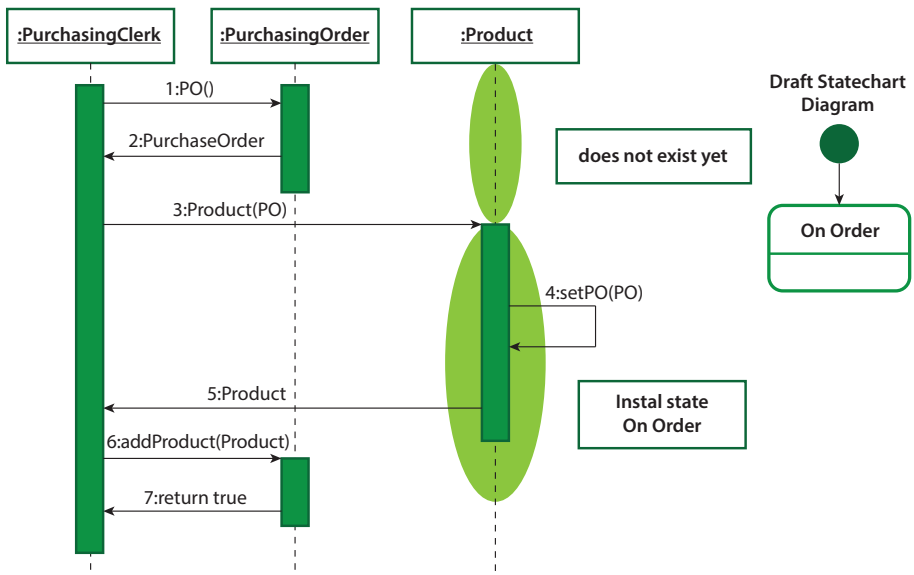
8.2.6. Povezanost dijagrama stanja i dijagrama sekvenci

Dijagrami stanja se često mogu izvesti iz dijagrama sekvenci. Životna linija objekta na dijagramu sekvenci predstavlja njegov životni vek, dok poruke upućene tom objektu predstavljaju događaje koji mogu izazvati promenu njegovog stanja. Period između dve uzastopne poruke na životnoj liniji odgovara vremenskom intervalu tokom kojeg se objekat nalazi u istom stanju. Na taj način, dijagram sekvenci pruža osnovu za identifikaciju stanja objekta i prelaza između njih, koji se zatim formalno modeluju dijagramom stanja.

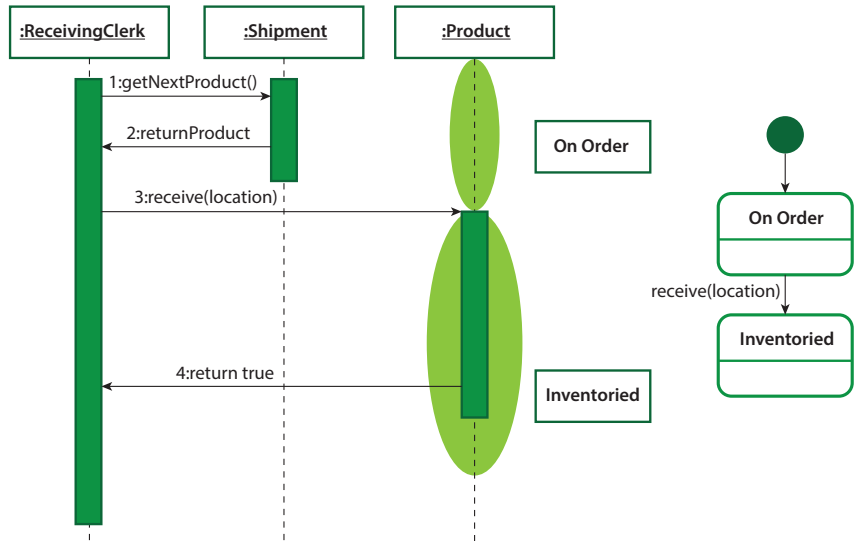


Slika 8.19. Primer prepoznavanja stanja objekta sa dijagrama sekvenci

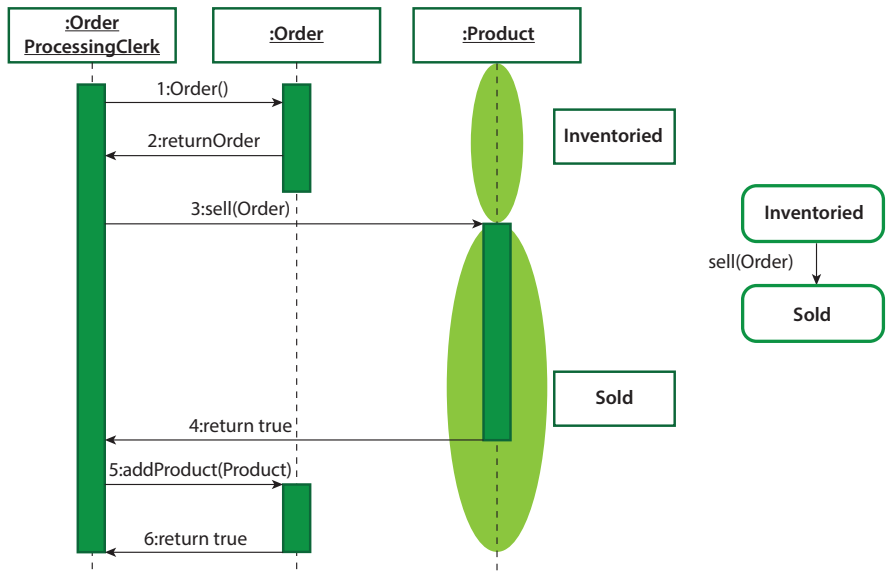
Na narednim slikama se prikazuju koraci kreiranja dijagrama stanja za objekat Proizvod na osnovu dijagrama sekvenci.



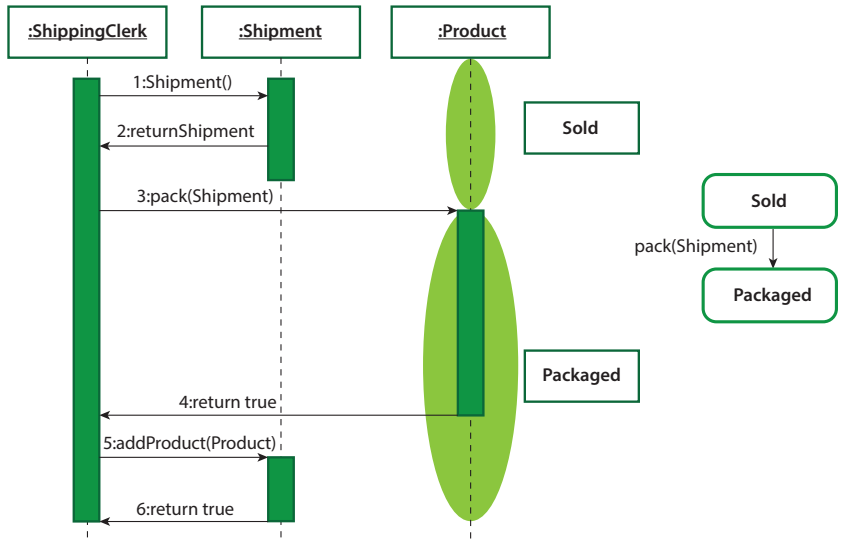
Slika 8.20. Scenario kreiranja porudžbine prozvoda na dijagramu klasa i dodavanja stanja proizvoda „On Order“



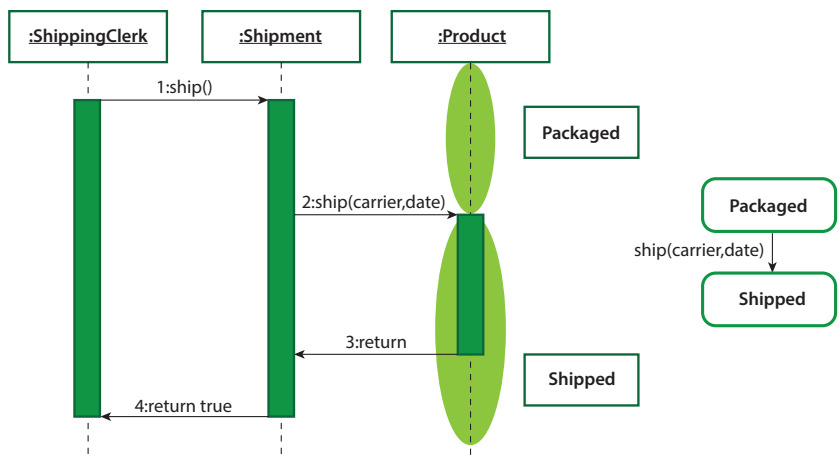
Slika 8.21. Scenario 2: Poruka na dijagramu sekvenci predstaljva tranziciju u naredno stanje na dijagramu klasa



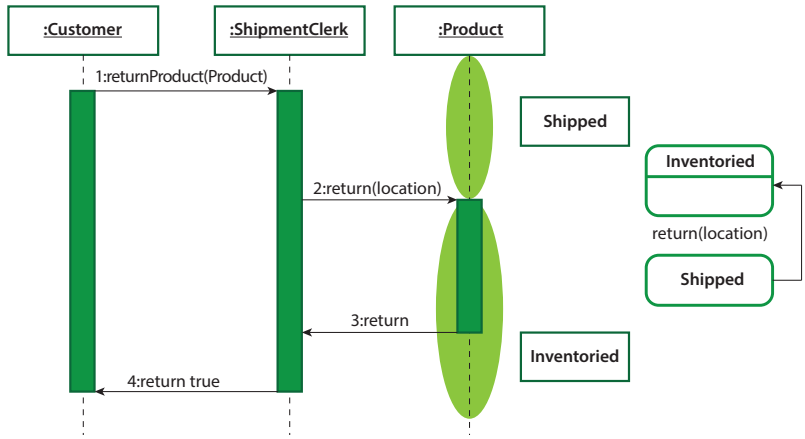
Slika 8.22. Scenario 3: poruka *sell*(*Order*) na dijagramu sekvenci prouzrokuje novu tranziciju i prelazak u stanje „Sold".



Slika 8.23. Scenario 4: Slanjem poruke *pack*(Shipment) u okviru dijagrama sekvenci za isporuku uzrokuje novu tranziciju i prelazak u stanje spakovana (*Packaged*)

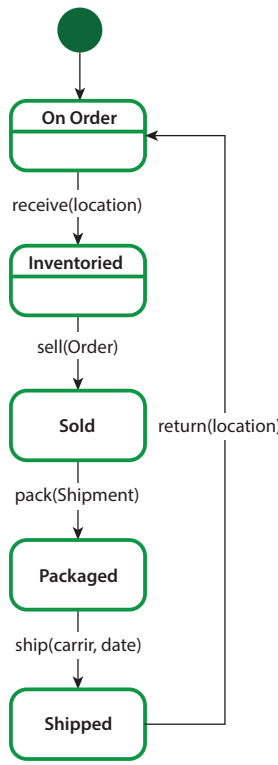


Slika 8.24. Scenario 5: Slanjem poruke *ship*(*carrier, date*) kreira se novo stanje objekta Proizvod „Shipped“



Slika 8.25. Poruka return(*location*) će prouzrokovati vraćanje na stanje uskladišten „Invetoried"

Na osnvou prethonih dijagrama sekvenci može se rekonstruisati sledeći dijagram stanja za klasu Proizvod.



Slika 8.26. Dijagram stanja za objekat Proizvod

8.2.7. Tipične greške u kreiranju dijagrama stanja

Prilikom izrade UML dijagrama stanja, u praksi se često javljaju sledeće greške:

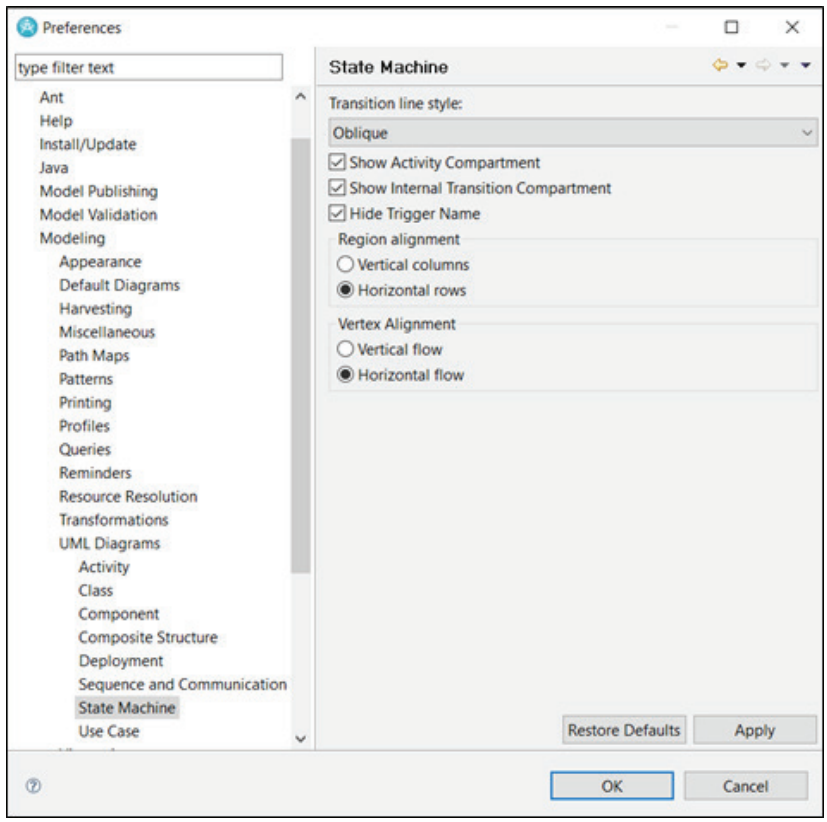
- Dijagram stanja se crta za celu aplikaciju tj. za sve klase – dok je potrebno samo za jednu konkretnu klasu koja ima više stanja.
- Crtanje dijagrama stanja za klase koje nemaju značajno ponašanje ili imaju samo jedno ili dva trivijalna stanja;
- Mešanje pojmova stanja i akcija (npr. korišćenje glagola kao naziva stanja umesto prideva);
- Izostavljanje uslova (*guard expressions*) kod tranzicija koje zavise od vrednosti atributa;
- Definisanje metoda klase koje se ne mogu mapirati ni na jedan događaj ili tranziciju u dijagramu stanja;
- Postojanje stanja iz kojih ne postoji nijedna izlazna tranzicija, iako objekat po poslovnoj logici mora nastaviti životni ciklus.

Vežba 8.1. Kreirati jednostavan dijagram stanja nekog uređaja koji ima samo dva stanja uključen (*On*) i isključen (*Off*) korišćenjem IBM RSA alata.

Korak 1. Podešavanje prikaza stanja

Prilikom kreiranja dijagrama stanja može da se desi da vam se samo prikazuje stanje bez dela za prikaz internih aktivnosti. Koraci podešavanja su:

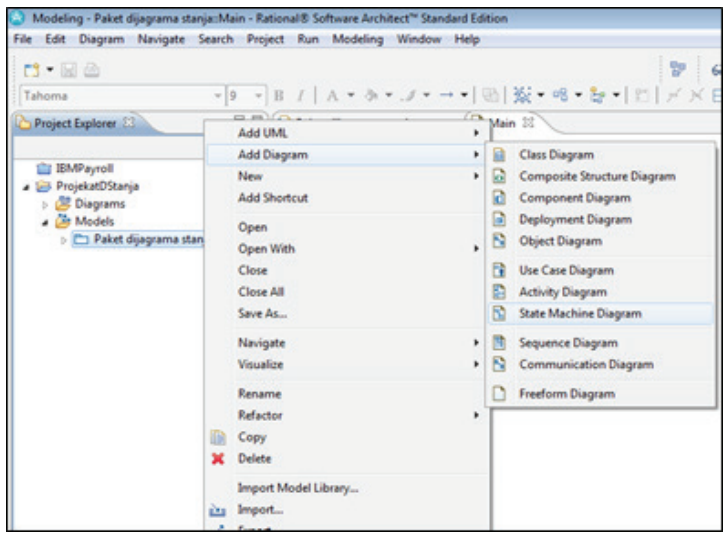
- Sa menija odaberite Window zatim Preferences (Slika 8.27)
- Proširiti *Modeling*
- Proširiti UML *Diagrams*
- Selektovati *State Machine*
- Sa desne strane prozora, selektovati *Show Activity Compartment* (za mogućnost prikaza do, entry i exit aktivnosti)
- Možete selektovati i *Show Internal Transition Compartment* za prikaz kompozitnih stanja.



Slika 8.27. Podešavanja prikaza stanja na dijagramu stanja

Korak 2. Kreirati dijagram stanja

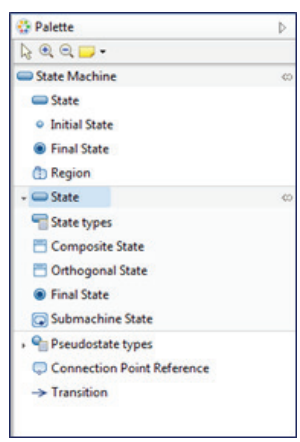
Nakon što ste kreirali projekat, desnim klikom na paket selektujte *Add Diagram*, a zatim *State Machine Diagram*:



Slika 8.28. Kreiranje dijagrama stanja

Korak 3: Modelovanje dijagrama stanja i rad sa paletom alata

Pre samog iscrtavanja dijagrama, neophodno je upoznati se sa elementima na paleti alata (Slika 8.29). Ovi elementi nam omogućavaju da precizno modelujemo ponašanje uređaja, od najjednostavnijih do veoma složenih procesa.



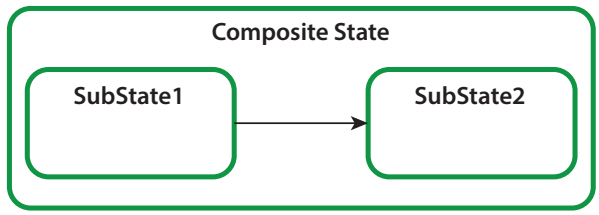
Slika 8.29. Paleta elemenata za crtanje dijagrama stanja

1. Tipovi stanja

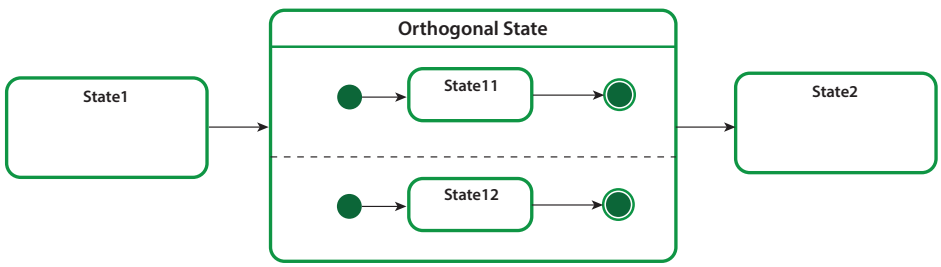
- State (Stanje): Predstavlja osnovno, stabilno stanje objekta koje nema unutrašnju podstrukturu.



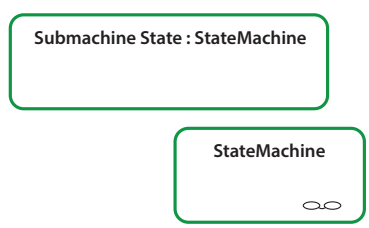
- Composite State (Kompozitno stanje): Složeno stanje koje sadrži jednu ili više regija sa podstanjima. Omogućava hijerarhijsku organizaciju modela.



- Orthogonal State (Ortogonalno stanje): Specijalna vrsta kompozitnog stanja gde se podstanja unutar različitih regija izvršavaju paralelno (istovremeno).



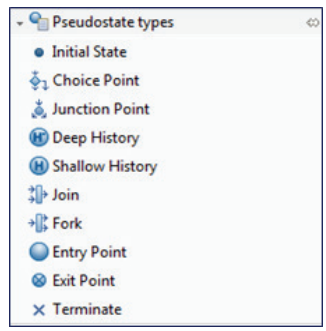
- **Submachine State**: Omogućava ponovnu upotrebu već definisanog dijagrama stanja unutar drugog dijagrama, što olakšava modelovanje velikih sistema.



2. Pseudostanja (čvorovi za kontrolu toka)

Pseudostanja nisu prava stanja u kojima objekt boravi, već služe za upravljanje tranzicijama:

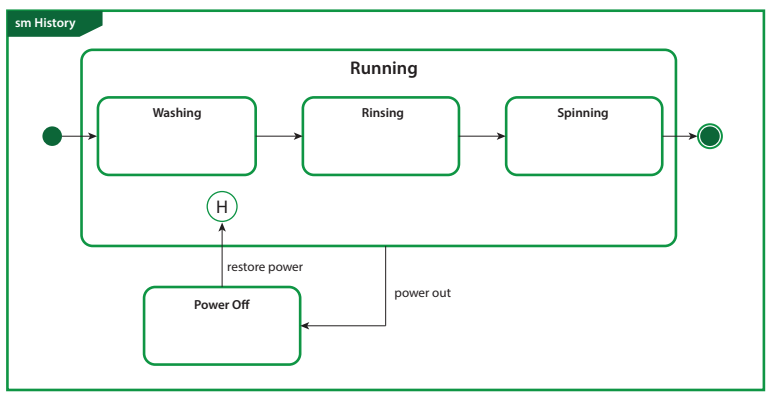
- *Initial State* (Inicijalno stanje): Početna tačka životnog ciklusa.
- *Choice / Junction Point*: Tačke odluke gde se tok granja na osnovu uslova (*Guard*).
- *Join / Fork*: Koriste se za spajanje ili razdvajanje paralelnih tokova izvršavanja.
- *Entry / Exit Point*: Definišu formalne tačke ulaska u kompozitno stanje ili izlaska iz njega.
- *Terminate*: Označava trenutni prestanak rada mašine stanja (objekat se uništava).



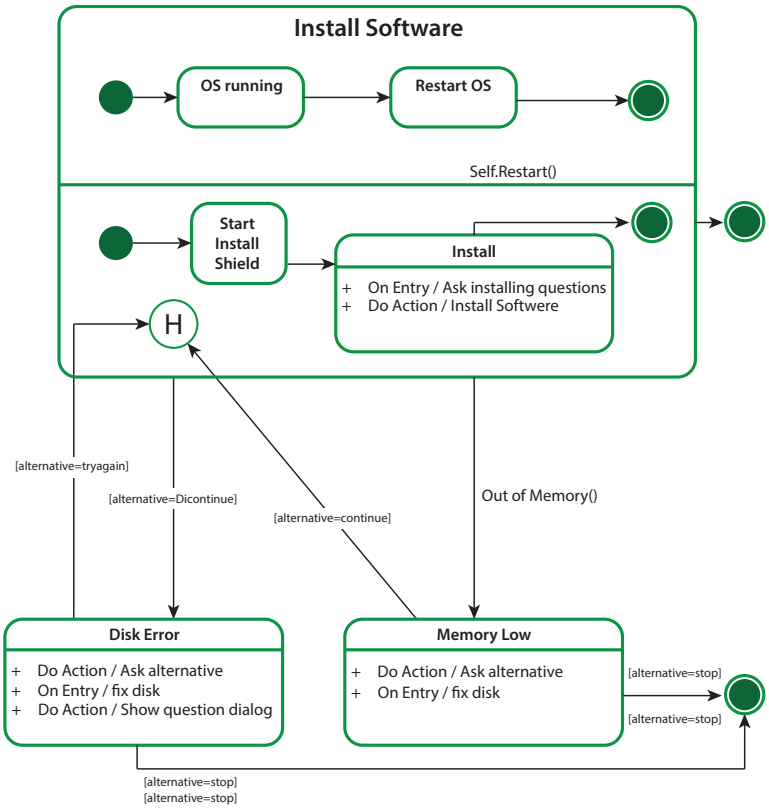
Slika 8.30. Tipovi pseudostanja u IBM RSA alatu

3. Mehanizmi memorije (*History States*)

- *Shallow History* (Površna istorija): Pamti poslednje aktivno stanje na trenutnom nivou hijerarhije. Tranzicija u ovaj čvor vraća objekat u to stanje bez ulaska u njegove dublje nivoe. Označava se na dijagramu sa oznakom H. Na donjoj slici je dat primer dijagrama stanja veš mašine. Kada dođe struja, mašina će nastaviti od poslednjeg aktivnog stanja pranja.



- *Deep History* (Duboka istorija): Pamti kompletnu konfiguraciju (sva podstanja na svim nivoima) koja je bila aktivna pre izlaska iz kompozitnog stanja. Oznaka je H*.

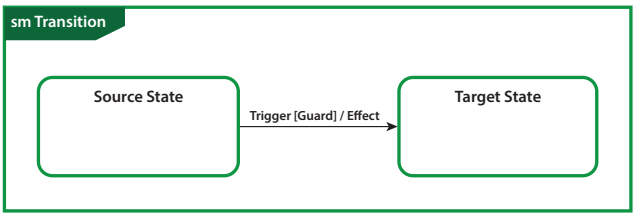


Slika 8.31. Primer duboke istorije (*Deep Hisory*)

4. Osobine tranzicije (*Properties*)

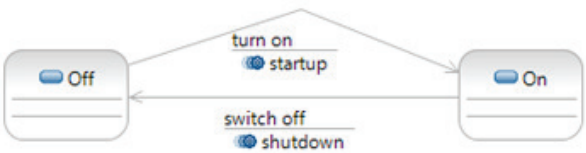
Svaka tranzicija (strelica) na dijagramu ima precizno definisane parametre u *Properties* panelu:

- *Name*: Logički naziv prelaza.
- *Kind* (Vrsta): Može biti *External* (standardni prelaz između dva stanja), *Internal* (izvršava se unutar stanja bez izlaska iz njega) ili *Local*.
- *Trigger* (Okidač): Događaj koji pokreće tranziciju (npr. pritisak tastera, istekao tajmer).
- *Guard* (Uslov): Logički izraz (u zagradama [ ]) koji mora biti istinit da bi se tranzicija uopšte izvršila.
- *Effect* (Efekat): Kratka aktivnost ili operacija koja se izvršava tokom samog prelaska iz jednog stanja u drugo.



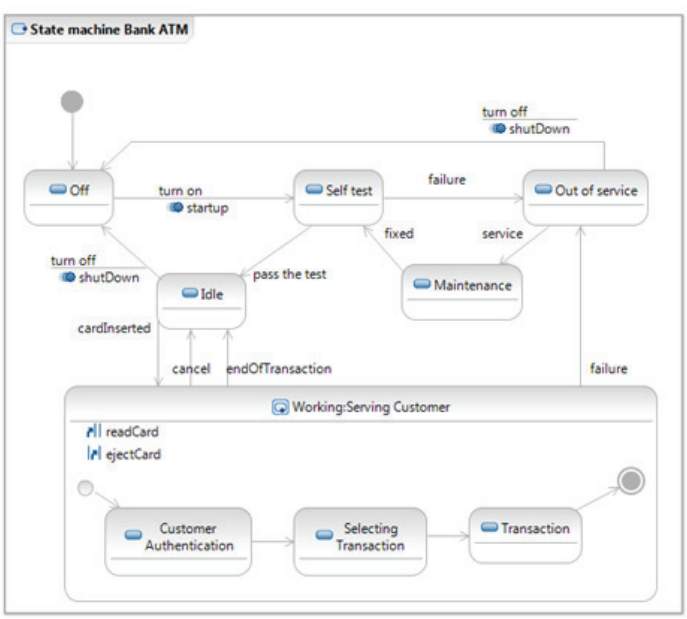
Slika 8.32. Grafički prikaz elemnata tranzicije

Rešenje vežbe



Slika 8.33. Jednostavan primer dijagrama stanja nekog uređaja

Vežba 8.2. U IBM RSA alatu nacrtati sledeći dijagram stanja ATM mašine



Slika 8.34. Dijagram stanja ATM mašine

Vežba 8.3. Kreirati dijagram stanja za stanja klase Student

Razmotrimo klasu Student, koja ima sledeća stanja tokom svog životnog ciklusa: Upisan, Obučavan, Testiran i Sertifikovan.

Povezanost sa dijagramom klasa

Na dijagramu klasa, klasa *Student* može imati sledeće atribute i metode:

- atributi: brojPoena, status, datumUpisa
- metode: započniSemestar(), završiSemestar(), polaganjeTesta(), izdajSertifikat()

Iscrtavanjem dijagrama stanja proverava se da li:

- svaka metoda ima odgovarajući događaj koji izaziva tranziciju,
- atribut brojPoena utiče na ponašanje kroz uslov [brojPoena > 50],
- ne postoji metoda koja nema jasno definisano ponašanje u određenom stanju.

Opis tranzicija:

- Regulisanjem svih obaveza vezano za upis na kurs, početno stanje studenta je Upisan.
- Kada počne semestar, student prelazi u stanje Obučavan.
- Po završetku semestra (završiSemestar()), student prelazi u stanje *Testiran*.
- Nakon polaganja testa (polaganjeTesta()):
  - ako je uslov [brojPoena > 50] ispunjen, student je položio test i prelazi u stanje *Sertifikovan*;
  - u suprotnom je pao i vraća se u stanje *Upisan*.
- Pri izlasku iz stanja *Sertifikovan* izvršava se izlazna akcija izdajSertifikat(). Nakon čega sistem nema potrebe da prati studenta.

Rešenje:



Zadatak 8.1. Kreirati dijagram stanja klase „ŠopingKorpa“

Šoping korpa predstavlja privremeni objekat u sistemu elektronske trgovine koji korisniku omogućava prikupljanje proizvoda pre završetka kupovine. Model treba da obuhvati sledeću logiku:

- **Inicijalizacija i početno stanje** – Objekat korpe nastaje onog trenutka kada korisnik pokrene sesiju na veb sajtu. Prilikom instanciranja (newShoppingCart), korpa prelazi u stanje **Prazna**.
  - Pri ulasku u stanje vrši se akcija **PonudiPopust()**.
  - Dok je u ovom stanju sistem treba da vrši pozadinsku aktivnost **šaljiReklamni Popust()** korisniku kako bi ga podstakao na kupovinu.
  - Prilikom izlaska iz stanja vrši se **ukloniReklamniBaner()**.
- **Upravljanje sadržajem** (Aktivna korpa) – Događajem **dodavanjeArtikla / dodaj** (ArtikalID), korpa prelazi u stanje **Aktivna**.
  - Pri ulasku u ovo stanje vrši se **inicijalizujKorpu()**.
  - Tokom boravka u stanju sistem **ažurira ukupnu cenu.**
  - Prilikom izlaska iz ovog stanja obavezno se vrši **provera dostupnosti artikala** u magacinu.
  - Ukoliko korisnik pokrene akciju **ukloniArtikal** uz uslov da je korpa ostala bez stavki **[brojStavki == 0]**, poziva se metod **isprazniKorpu()** i korpa se vraća u stanje **Prazna**.
- **Proces naplate i zaključavanje** – Kada korisnik **potvrdi narudžbinu**, korpa tranzicijom prelazi u stanje **Zaključana**.
  - Pri ulasku u stanje sadržaj korpe se **zaključava** za dalje izmene.
  - U ovom stanju sistem vrši pozadinsku **pripremu za dalju obradu** (obuhvata proveru plaćanja i generisanje računa).
- **Arhiviranje i kraj životnog ciklusa** – Nakon uspešne potvrde tranzicijom **obrada Završena()**, korpa automatski prelazi u stanje **Arhivirana**.
  - Pri ulasku u ovo stanje (entry), vrši se trajno beleženje podataka u bazu (arhiviranje).
  - Ovo je finalno stanje nakon kojeg se životni ciklus objekta završava.

8.3. DIJAGRAM KLASA SA PRIMENJENIM OBRASCIMA PROJEKTOVANJA SOFTVERA

U fazi analize softvera identifikuju se osnovne klase sistema na osnovu funkcionalnih zahteva i poslovnih procesa. Međutim, ove klase ne predstavljaju konačno rešenje. Tokom faze projektovanja softvera, klase se dodatno razrađuju, prilagođavaju i po potrebi proširuju ili restrukturišu. Na primer, ukoliko se uoči da jedna klasa ima preveliki broj odgovornosti ili metoda, ona se može podeliti na dve ili više kohezivnijih klasa, čime se poboljšavaju čitljivost, održivost i ponovna upotrebljivost koda.

U fazi projektovanja softvera primenjuju se i ustaljeni obrasci projektovanja (*design patterns*), koji predstavljaju proverena i opšteprihvaćena rešenja za često ponavljane probleme u razvoju softvera. Dizajn paterni pomažu projektantima da nađu balans između fleksibilnosti i jednostavnosti sistema, omogućavaju bolju organizaciju klasa i jasno razdvajanje odgovornosti, kao i lakšu komunikaciju među članovima razvojnog tima, jer koriste zajednički, standardizovan jezik dizajna.

Primena dizajn paterna na dijagramu klasa omogućava da se već u ranoj fazi projektovanja jasno prikažu odnosi između klasa, obrasci saradnje i način na koji se sistem može prilagoditi budućim promenama. Na taj način se smanjuje rizik od loših arhitektonskih odluka i povećava kvalitet dizajna pre same implementacije.

Tokom projektovanja, logički dizajn klasa postepeno prelazi u fizički dizajn. Logički dizajn se fokusira na strukturu sistema, odgovornosti klasa i njihove međusobne odnose, dok fizički dizajn razrađuje konkretne implementacione detalje, kao što su programski jezici, tipovi podataka i konkretne strukture. Na osnovu fizičkog dizajna moguće je generisati programski kod, čime se obezbeđuje jasan i sistematičan prelaz iz modela u implementaciju.

8.3.1. Primena obrazaca projektovanja softvera

Primena obrazaca projektovanja softvera (*design patterns*) u dijagramima klasa omogućava razvoj fleksibilnih, proširivih, održivih i ponovo upotrebljivih softverskih sistema. Obrasci projektovanja predstavljaju proverena i dokumentovana rešenja za tipične probleme koji se ponavljaju u objektno-orijentisanom razvoju softvera. Umesto ad-hoc pristupa dizajnu, obrasci nude standardizovan način razmišljanja i komunikacije među developerima.

Design paterni ne predstavljaju gotova rešenja u vidu koda, već apstraktne šablone koji opisuju strukturu klasa, njihove odgovornosti i međusobne odnose. Upravo zbog toga se često i prirodno modeluju pomoću UML dijagrama klasa, gde se jasno mogu uočiti relacije kao što su nasleđivanje, agregacija, kompozicija i zavisnosti.

U akademskoj i industrijskoj literaturi, obrasci projektovanja se posmatraju kao deo najboljih praksi softverskog inženjeringa, jer doprinose:

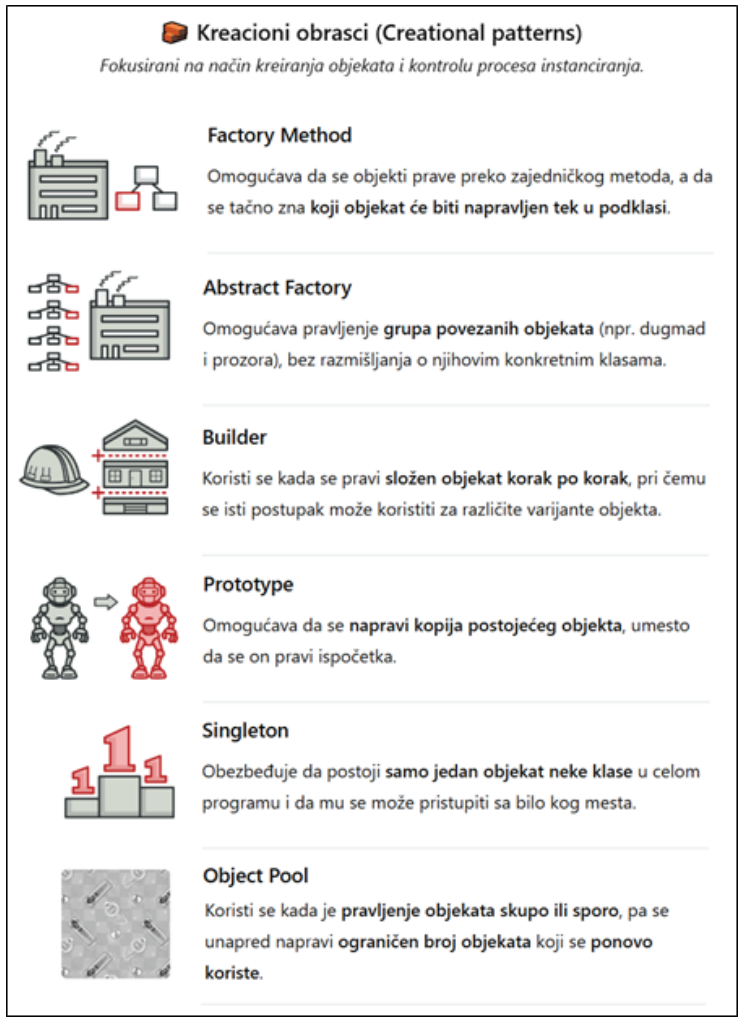
- boljoj čitljivosti i razumljivosti koda,
- smanjenju međuzavisnosti između klasa (*loose coupling*),
- lakšem održavanju i testiranju sistema,
- jednostavnijem proširivanju funkcionalnosti bez izmene postojećeg koda,
- boljoj raspodeli odgovornosti između klasa.

Design paterni su stekli široku popularnost nakon objavljivanja knjige „*Design Patterns: Elements of Reusable Object-Oriented Software*“ (Gamma, Helm, Johnson, Vlissides, 1995), čiji su autori poznati kao *Gang of Four* (GoF). U toj knjizi ukupno 23 obrazaca je sistematizovano i klasifikovano u tri osnovne grupe:

- **Kreacioni obrasci** (*Creational patterns*) – bave se načinom kreiranja objekata i apstrahuju proces instanciranja, čime se smanjuje zavisnost sistema od konkretnih klasa. Na primer, Singleton, Factory Method, Abstract Factory (Slika 8.35).
- **Strukturni obrasci** (*Structural patterns*) – fokusiraju se na organizaciju i povezivanje klasa i objekata u veće strukture, uz očuvanje fleksibilnosti i efikasnosti sistema. Na primer, Adapter, Composite, Decorator (Slika 8.36).
- **Obrasci ponašanja** (*Behavioral patterns*) – opisuju komunikaciju i raspodelu odgovornosti između objekata. Na primer, Observer, Strategy, State (Slika 8.37).

Za developera je važno da razume kada i zašto primeniti određeni obrazac, a ne samo kako ga implementirati. Neadekvatna ili prekomerna upotreba obrazaca može dovesti do nepotrebne složenosti dizajna (*overengineering*). Zbog toga se u savremenoj praksi naglašava balans između jednostavnosti dizajna i primene obrazaca samo tamo gde oni donose jasnu korist.

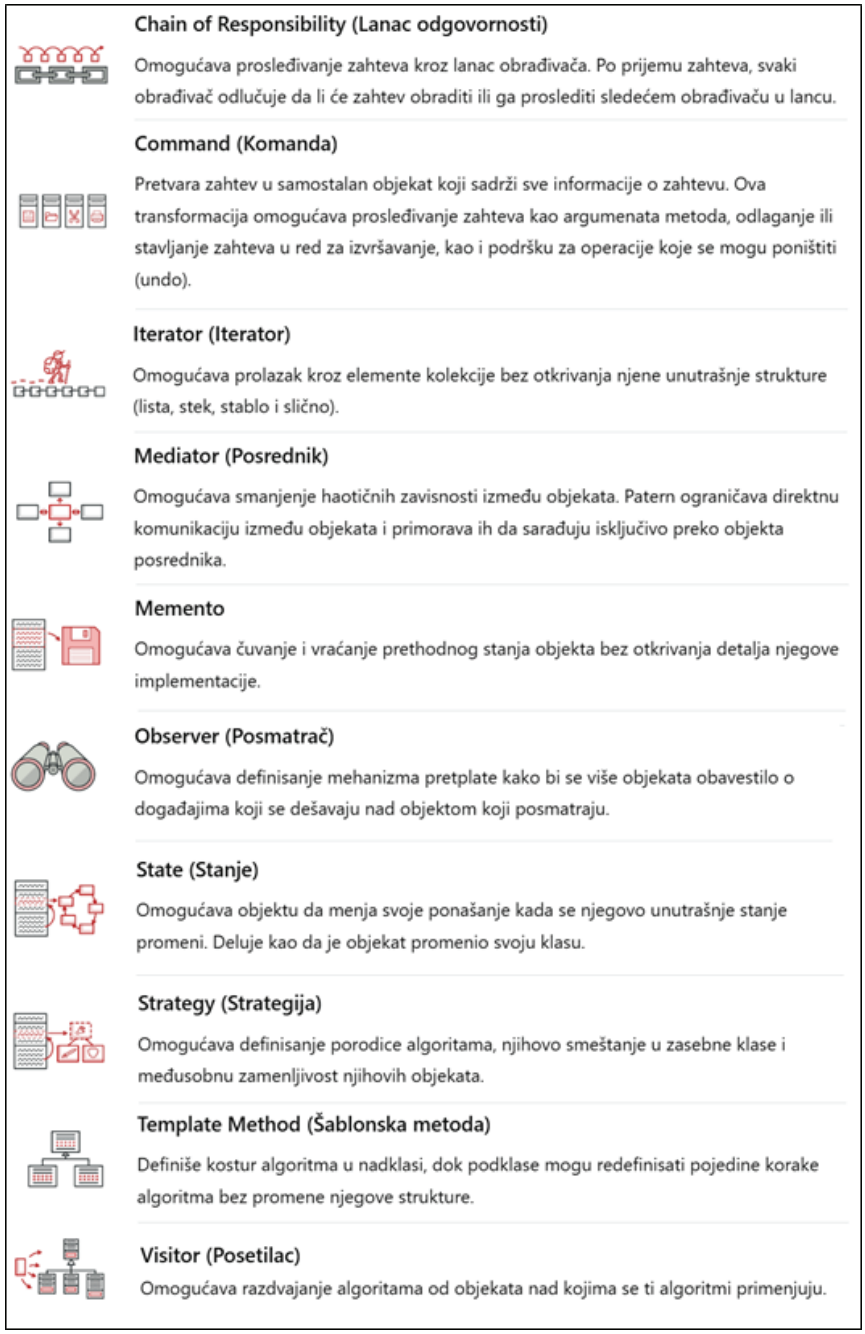
U kontekstu dijagrama klasa, obrasci projektovanja omogućavaju da se arhitektonske odluke jasno dokumentuju, lakše razumeju i dosledno primenjuju tokom celog životnog ciklusa softvera, što ih čini ključnim alatom u profesionalnom razvoju softverskih sistema.



Slika 8.35. Kratak opis kreacionih obrazaca



Slika 8.36. Strukturni obrasci



Slika 8.37. Obrasci ponašanja

Razumevanje obrazaca projektovanja

Da bismo razumeli šta su obrasci projektovanja (*design patterns*), posmatrajmo jedan svakodnevni primer koji svi prepoznajemo - koncept kuhinje. Kada kažemo reč "kuhinja", mi zapravo imenujemo rešenje za jedan vrlo specifičan, učestao problem: „Kako organizovati prostor za bezbedno čuvanje i efikasnu pripremu hrane?“ Svaki put kada arhitekta projektuje stan, on se ne pita iznova čemu služi taj prostor; on koristi već usvojeni obrazac koji taj problem rešava. Takođe, svima su poznati i osnovni resursi kuhinje: frižider, šporet, radna površina i sudopera. Međutim, da li su sve kuhinje implementirane na isti način? U malom stanu, kuhinja može biti svedena na jednu liniju elemenata. U restoranu, to je ogroman sistem sa profesionalnim pećima i hladnjačama. Iako su implementacije (izgled, veličina, cena materijala) potpuno različite, obrazac (suština i raspored resursa) ostaje isti. Vi ne morate videti konkretnu kuhinju da biste znali kako da se u njoj ponašate – znate da se hrana uzima iz frižidera, a termički obrađuje na šporetu.

U svetu softvera, obrasci projektovanja funkcionišu identično:

- Prepoznavanje problema: Programeri su primetili da se određeni problemi (npr. kreiranje objekata, komunikacija između klasa) stalno ponavljaju.
- Standardizovano rešenje: Umesto da svaki put "izmišljaju točak", oni koriste obrazac koji im kaže: "Za ovaj problem, najbolje je da tvoji objekti komuniciraju na ovaj način."
- Sloboda u kodu: Obrazac vam daje nacrt (*blueprint*), ali vi ga implementirate u svom programskom jeziku (Java, C++, Python) i prilagođavate specifičnim potrebama vašeg projekta.

Dakle, obrazac projektovanja nije gotov kod koji prekopirate u svoj projekat. To je zajednički jezik programera i konceptualni alat koji vam omogućava da složene sisteme gradite na temeljima proverenih i efikasnih rešenja.

U inženjerskoj praksi, dokumentacija svakog obrasca obavezno sadrži sledeće ključne segmente:

- Problem: Precizan opis specifičnog problema u dizajnu koji obrazac nastoji da reši, uz navođenje konteksta u kojem je njegova primena opravdana.
- Struktura i resursi (*Participants*): Definisanje klasa i objekata koji učestvuju u obrascu, njihove uloge i odgovornosti. U alatima poput IBM RSA, ovo se vizuelizuje kroz dijagram klasa koji prikazuje interakcije između apstrakcija i konkretnih implementacija.
- Kolaboracija: Opis načina na koji definisani resursi (objekti) međusobno komuniciraju kako bi izvršili zadatak.
- Posledice: Analiza rezultata i kompromisa (*trade-offs*) koji nastaju primenom obrasca. Ovo uključuje uticaj na fleksibilnost, performanse i kompleksnost sistema.
- Pravila i smernice: Praktična uputstva za implementaciju obrasca, uključujući specifičnosti vezane za programski jezik ili razvojno okruženje.

U tabeli 8.3 je prikazan opis ovih elemenata na primeru kuhinje i značenje u softverskom inženjerstvu.

Tabela 8.3. Razumevanje elemenata obrazaca kroz analogiju "kuhinje"

Element obrasca	Opis kroz analogiju (kuhinja)	Opis obrazaca
Problem	Potreba za organizovanim mestom za čuvanje i pripremu hrane u zatvorenom prostoru.	Opis specifične situacije u dizajnu (npr. prevelika zavisnost klasa) u kojoj se obrazac primenjuje.
Resursi (Učesnici)	Uređaji (šporet, frižider), namirnice i posuđe.	Klase i objekti koji učestvuju u obrascu, njihove uloge i međusobne veze (npr. Context, State, ConcreteState).
Pravila (Saradnja)	Frižider isključivo skladišti, rerna isključivo peče.	Protokol interakcije između objekata — kako resursi razmenjuju poruke da bi postigli cilj.
Smernice (Kontekst)	Ovaj patern dobro funkcioniše kada se koristi u kući, dok nije preporuka da se koristi van kuće (npr. kod kampovanja)	Opis situacija u kojima je obrazac primenljiv, kao i prednosti i mane (posledice) njegove upotrebe.

Primena *State* obrasca na dijagramu klasa u IBM RSA alatu

Prelazak iz faze analize u fazu projektovanja softvera predstavlja ključan korak u životnom ciklusu razvoja sistema. Dok dijagram klasa u fazi analize prikazuje domenske koncepte i njihove međusobne odnose, fizički dijagram klasa u fazi projektovanja mora precizno definisati strukturu koja će biti direktno implementirana u izvornom kodu.

U procesu projektovanja inženjeri se često suočavaju sa izazovima kao što su:

- Kruta zavisnost (*Tight Coupling*): Klase su previše međusobno povezane, što otežava izmene.
- Slaba kohezija: Klase preuzimaju previše odgovornosti koje im logički ne pripadaju.
- Složenost uslovne logike: Nagomilavanje if-then-else ili switch-case struktura koje postaju teške za održavanje i mnogi drugi.

Uzmimo za primer sistem za upravljanje porudžbinama, u kome se klasa Proizvod može nalaziti u različitim stanjima (npr. Poručen, Plaćen, Spakovan). Česta greška u dizajnu ovakvih sistema jeste korišćenje jedne klase koja putem brojnih uslovnih grananja proverava trenutno stanje proizvoda. Ovakav pristup često dovodi do tzv. „špageti koda", gde svaka promena poslovnog pravila zahteva izmene u velikom broju delova iste klase. Na primer, ukoliko je proizvod u stanju Plaćen, sistem treba da obavesti magacionera radi pripreme isporuke, kao i finansijsku službu radi kreiranja fakture.

Direktna implementacija ovakvih provera unutar osnovne klase čini sistem nefleksibilnim, otežava održavanje i komplikuje uvođenje novih stanja ili promenu redosleda poslovnih koraka.

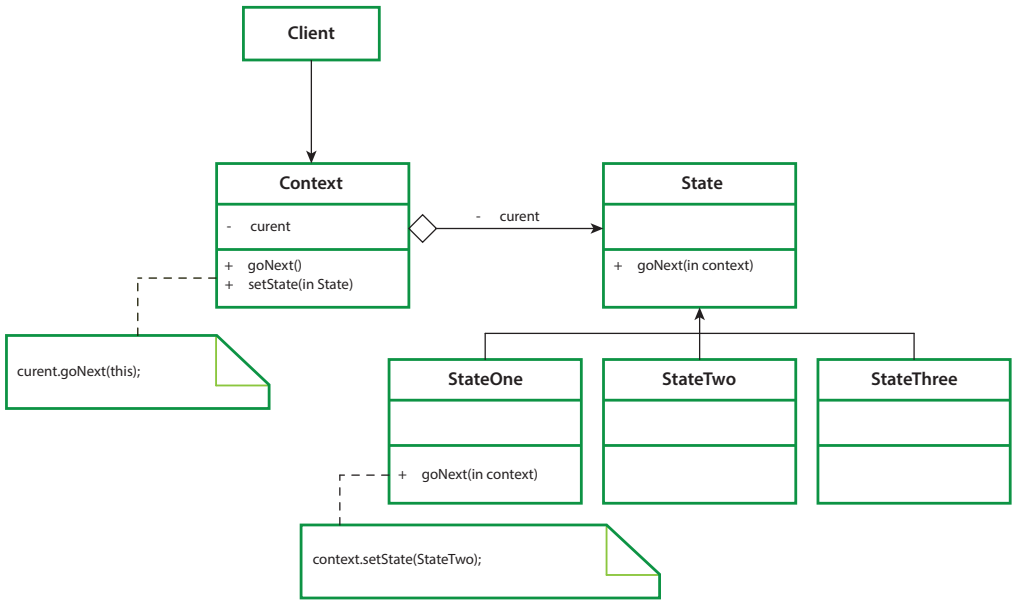
Kao odgovor na opisani problem primenjuje se **obrazac Stanja** (*State Design Pattern*). Ovaj obrazac spada u grupu bihejvioralnih dizajn paterna i koristi se u situacijama kada ponašanje objekta zavisi od njegovog unutrašnjeg stanja, pri čemu se to ponašanje menja tokom izvršavanja sistema. Umesto upotrebe složenih uslovnih struktura (if–else ili switch), svako stanje se modeluje kao poseban objekat koji inkapsulira ponašanje karakteristično za to stanje.

U kontekstu dijagrama klasa, to znači da se pored klase Proizvod **uvodi interfejs** (ili apstraktna klasa) ***ProductState***, kao **i skup konkretnih klasa koje predstavljaju različita stanja proizvoda**, na primer Poručen, Plaćen, Isporučen i slično. Na ovaj način se ponašanje proizvoda delegira objektima stanja, čime se postiže fleksibilniji i lakše proširiv dizajn.

Objašnjenje *State* obrasca kroz opis osnovnih elemenata:

- **Problem**: Proizvod se tokom svog životnog ciklusa može nalaziti u različitim stanjima (npr. Dostupan, Nema na stanju, Rezervisan, Povučen iz prodaje), pri čemu se njegovo ponašanje razlikuje u zavisnosti od trenutnog stanja.
- **Resursi** koji rešavaju dati problem:
  - ***Context*: Proizvod** - Klasa Proizvod predstavlja kontekst u *State* paternu. Ona čuva referencu na trenutno stanje (ProductState), prima zahteve od klijenata sistema i delegira obradu zahteva objektu koji predstavlja trenutno stanje.
  - ***State: Stanje*** proizvoda (ProductState) - *ProductState* definiše zajednički interfejs za sva moguća stanja proizvoda. Obično je realizovan kao interfejs ili apstraktna klasa, koja deklariše metode koje zavise od stanja (npr. poruči(), rezerviši(), povucilzProdaje()).
- **Pravila** primene *State* dizajn paterna:
  - **Generalizacija stanja proizvoda** - *ProductState* predstavlja generalizaciju (interfejs ili apstraktnu klasu) svih konkretnih stanja proizvoda. Svako konkretno stanje (npr. DostupanState, NemaNaStanjuState) implementira ovaj interfejs i sadrži sopstvenu implementaciju ponašanja za definisane metode. Na taj način, ponašanje koje zavisi od stanja nije deo klase Proizvod, već je inkapsulirano u klasama stanja.
  - **Delegiranje zahteva iz konteksta** - Klasa Proizvod prihvata zahteve spolja (npr. zahtev za poručivanje ili rezervaciju proizvoda), ali ne odlučuje sama kako će se zahtev obraditi.Umesto toga Proizvod delegira svaki zahtev instanci *ProductState*, a instanca *ProductState* predstavlja trenutno stanje proizvoda i zna kako da se ponaša u toj situaciji. Ovim se eliminiše potreba za grananjem logike unutar klase Proizvod.
  - **Izvršavanje ponašanja i povratak kontrole** - Instanca *ProductState* izvršava odgovarajuću implementaciju metode u skladu sa stanjem, po potrebi može promeniti stanje proizvoda (npr. iz Dostupan u Nema na stanju) i nakon izvršenja vraća kontrolu objektu Proizvod. Na ovaj način se omogućava da se ponašanje objekta menja dinamički, bez promene njegove klase.

Osnovna struktura ovog obrasca je prikazana na slici 8.36.



Slika 8.38. Osnovna struktura *State* dizajn paterna

Ključni učesnici i uloge *State* obrasca prema osnovnoj strukturi su:

Client (Klijent) - Predstavlja korisnika sistema koji inicira akciju. Klijent ne mora poznavati unutrašnju logiku prelaza niti trenutno stanje objekta; on isključivo komunicira sa *Context* objektom pozivajući definisane operacije (npr. „izvrši sledeći korak").

Context (Kontekst) - Konkretna klasa (u našem primeru klasa Proizvod) koja održava funkcionalnost sistema. Sadrži:

- Atribut *current* - referenca na interfejs State koja čuva informaciju o trenutnom aktivnom stanju.
- Metodu *goNext*() - operacija koju poziva klijent. Kontekst ne implementira logiku prelaza, već samo prosleđuje poziv trenutnom stanju: *current.goNext*(this);.
- Metodu *setState*(in State) - služi za ažuriranje atributa *current*. Ovu metodu pozivaju konkretna stanja, a ne klijent, čime se obezbeđuje kontrolisana promena unutrašnjeg statusa.

State (Apstraktno stanje) - Interfejs ili apstraktna klasa koja definiše zajednički protokol za sva stanja (npr. metodu goNext(in context)).

ConcreteStates (Konkretna stanja: StateOne, StateTwo...) - Klase koje implementiraju specifično ponašanje za svaku fazu životnog ciklusa objekta. Svako konkretno stanje „zna" šta sledi nakon njega i na kraju svoje operacije vrši promenu stanja u kontekstu pozivom *context.setState*(StateNext).

Dinamika izvršavanja (Korak po korak):

1. Klijent poziva metodu *context.goNext*().
2. Delegacija: Objekt *Context* prosleđuje zahtev trenutnom objektu stanja pozivom *current.goNext*(this).
3. Izvršavanje logike: Trenutno stanje (npr. *StateOne*) izvršava poslovnu logiku specifičnu za tu fazu.
4. Tranzicija: Nakon izvršene logike, trenutno stanje donosi odluku o sledećem koraku i poziva *context.setState*(*StateTwo*).
5. Ažuriranje: *Context* menja svoju referencu, čime je objekat spreman za sledeći poziv klijenta u okviru novog stanja.

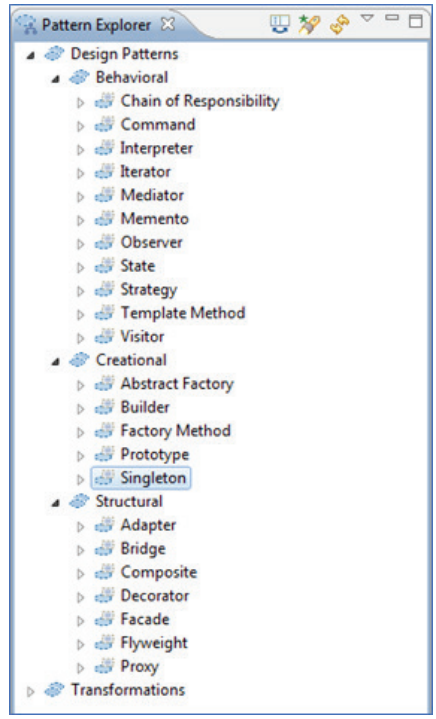
Ovim dizajnom postižemo da *Context* ostaje „neznalica" po pitanju specifičnih ponašanja u različitim fazama – on služi samo kao stabilan interfejs ka klijentu, dok je sva inteligencija sistema distribuirana u objekte stanja.

Prednosti ovakve strukture su:

- jasno razdvajanje stanja i ponašanja,
- lakše dodavanje novih stanja,
- veća proširivost i održivost sistema.

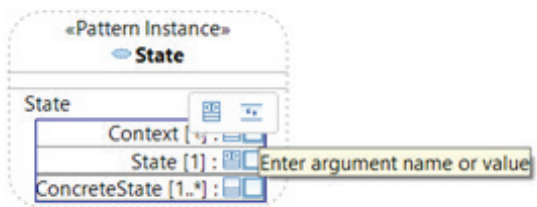
Koraci primene State obrasca u IBM RSA alatu su:

1. **Aktivacija i korišćenje kataloga obrazaca u IBM RSA alatu** - Da bi se obrasci projektovanja mogli primeniti na dijagramu klasa, neophodno je prvo aktivirati *Pattern Explorer*, koji služi kao centralni repozitorijum svih dostupnih rešenja. Postupak aktivacije *Pattern Explorer* panela:
   - Na glavnoj traci menija izaberite opciju *Window*, zatim *Show View*, i na kraju kliknite na *Other*.
   - U novootvorenom prozoru za pretragu, pronađite i proširite folder pod nazivom *Modeling*.
   - Selektujte stavku *Pattern Explorer* i potvrdite izbor klikom na dugme *OK*.
   - Podrazumevano, *Pattern Explorer* će se otvoriti u donjem delu ekrana. Radi bolje preglednosti prilikom projektovanja, preporučuje se da ga prevučete (opcija drag and drop) na desnu ivicu radnog prostora.
   - Proširite *Design Patterns* da biste dobili uvid u standardne kategorije (Kreacioni, Strukturni i Obrasci ponašanja).
   - Svaka kategorija se može dodatno proširiti kako bi se prikazali pojedinačni obrasci, poput *State, Observer* ili *Strategy* obrazaca.



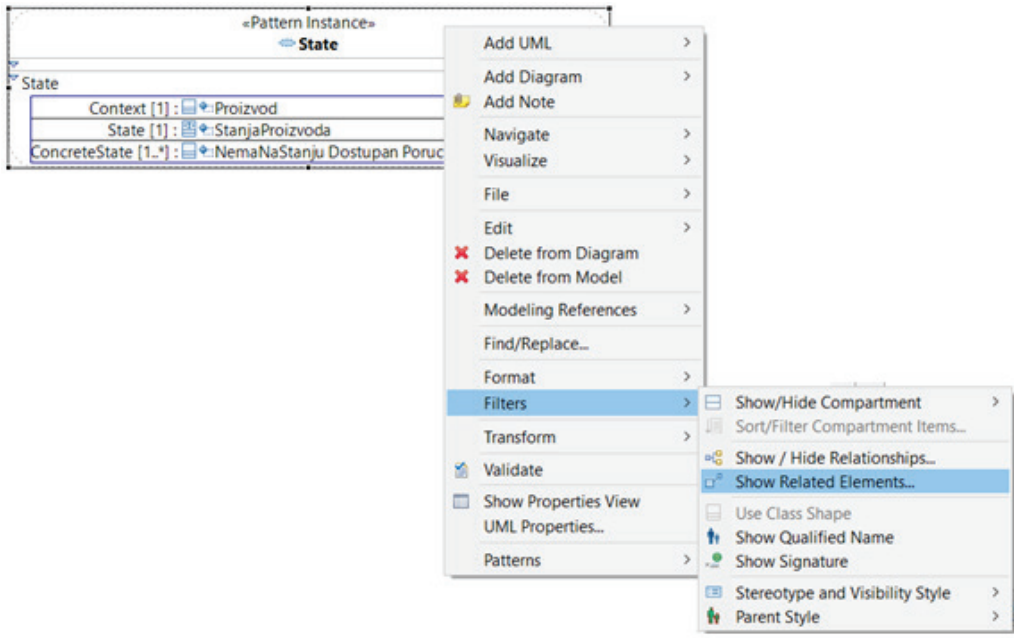
Slika 8.39. Dostupni paterni u IBM RSA alatu

2. Pronađite **obrazac *State*** unutar kategorije *Behavioral* u *Pattern Explorer*-u i jednostavno ga prevucite na vaš aktivni dijagram klasa.
3. Ukoliko već imate **klasu Proizvod** samo je **prevucite u deo za Context** ili je možete ručno dodati, tako što pređete mišem preko *Context* u okviru koga će vam se pojaviti prečice za unos. Klikom na drugi element „*Enter argument nae or value*" unestie naziv klase – Proizvod:

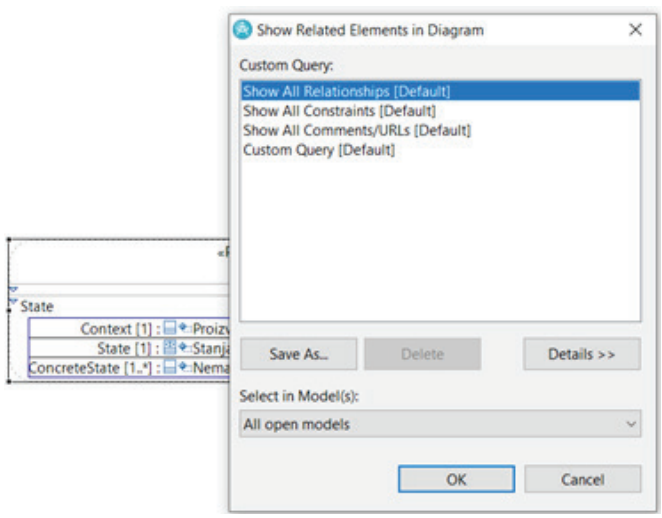


4. Nastavite sa unosom i drugih elemenata obrazaca istim postupkom. Pod *State* unesite naziv interfejsa StanjeProizvoda, a pod *ConcreteState* pojedinačno dodajte stanje: NemaNaStanju, Dostupan, Poručen, Plaćen, Isporučen.

5. Da bi se svi elementi prikazali na dijagramu klasa, prvo selektujte ceo obrazac, a zatim desnim klikom odaberite *Filters* pa *Show Related* elements. Na novom prozoru odaberite *Show All Relationships* [*Default*]:

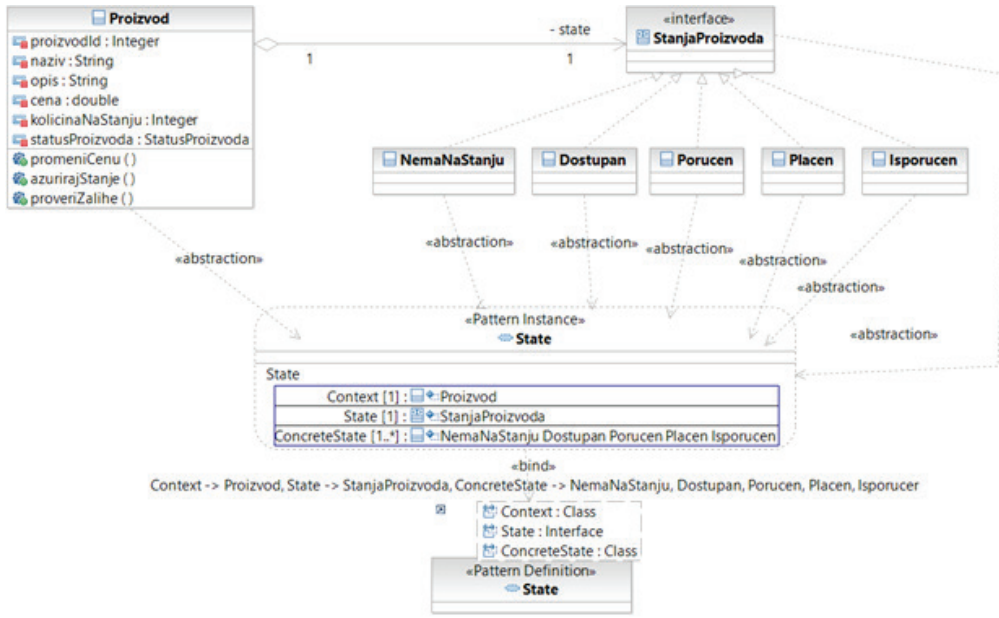


Slika 8.40. Koraci prikaza svih klasa i relacija na dijagramu klasa



Slika 8.41. Prikaz svih relacija na dijagramu klasa

6. Izgenersani dijagram klasa možete doterati, kao što je prikazano na sledećoj slici 8.42:



Slika 8.42. Dijagram klasa sa primenjenim *State* obrascem

Kada bi generisali programski kod, videli bi da alat automatski:

- U klasi *Context* se kreira atribut *current* koji referencira tip *State*.
- Metode u *Context* klasi (poput goNext()) biće unapred pripremljene da poziv proslede trenutnom stanju: *current.goNext*(this).
- Svako konkretno stanje dobija svoju verziju metode u kojoj vi kasnije dopisujete specifičnu poslovnu logiku i poziv za promenu stanja: *context.setState*(NextState).

Za detaljan prikaz procesa generisanja Java programskog koda na osnovu kreiranih modela u IBM RSA okruženju, kao i za sveobuhvatnu analizu ostalih kategorija obrazaca projektovanja, studentima se preporučuje konsultovanje stručne literature: Njeguš, A. (2023). Obrasci projektovanja softvera.

Vežba 8.4. Dijagram klasa „Onlajn prodavnice“ sa primenom obrazaca

U ovoj vežbi se prikazuje sagledavanje primene obrazaca nad klasama sistema „Onlajn prodavnice“:

1. **ŠopingKorpa** (*ShoppingCart*)
   - Atributi: artikli: List<Artikal>, status: StatusKorpe
   - Metode: dodajArtikal(), ukloniArtikal(), zaključiKupovinu()
   - **Obrazac**: *State Pattern* – za upravljanje stanjima korpe (Prazna, Aktivna, Zaključana, Arhivirana)
2. **Artikal** (*Product*)
   - Atributi: naziv, cena, kolicina
   - Metode: prikaziDetalje()
3. **Korisnik** (*User*)
   - Atributi: ime, email, korpa: ShoppingCart
   - Metode: kreirajKorpu(), završiKupovinu()
4. ***Singleton*: BazaPodataka** (*Database*)
   - Atributi: instanca: Database
   - Metode: getInstance(), sacuvajPorudzbinu()
   - **Obrazac**: Singleton Pattern – postoji samo jedna instanca baze podataka u sistemu.
5. **Porudzbina** (*Order*)
   - Atributi: idPorudzbine, artikli, ukupnaCena
   - Metode: kreiraj(), posalji()
   - **Obrazac**: *Factory Method* – kreira porudžbinu iz korpe preko metode kreiraj Porudzbinu().
6. ***Observer*: Obavestenje** (*Notification*)
   - Interfejs Observer
   - Klase koje implementiraju: EmailNotification, SMSNotification
   - **Obrazac**: *Observer Pattern* – kada se porudžbina kreira, korisnici se obaveštavaju.

8.4. DIJAGRAM KOMPONENTI

Dijagram komponenti modeluje strukturu softvera na nivou implementacije i često se posmatra zajedno sa dijagramom uvođenja (deployment diagram). Dok dijagram uvođenja opisuje raspodelu softverskih artefakata na hardverske čvorove, dijagram komponenti se fokusira na same softverske module i njihove međusobne zavisnosti. Na ovaj način, dijagram komponenti predstavlja most između logičkog dizajna sistema i njegove fizičke realizacije, omogućavajući jasniji uvid u to kako se delovi sistema povezuju, zavise jedni od drugih i zajednički realizuju ukupnu funkcionalnost.

U savremenom softverskom inženjerstvu, dijagrami komponenti imaju važnu ulogu u razumevanju arhitekture sistema, planiranju razvoja, kao i u komunikaciji između različitih učesnika u projektu (arhitekata, programera, administratora sistema). Njegov značaj se ogleda kroz različite faze životnog ciklusa softvera:

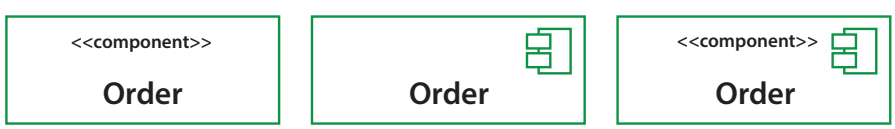
- **Faza projektovanja**: Koristi se za definisanje strukturnih modula pre samog kôdiranja. On predstavlja „nacrt" koji timovima jasno ukazuje na to koji moduli moraju biti razvijeni i koje interfejse (protokole komunikacije) moraju da poštuju.
- **Faza implementacije i održavanja**: Služi kao dokumentacija fizičkih artefakata (npr. .jar, .dll, .exe fajlova, baze podataka ili mikrosevisa) i njihovih međusobnih zavisnosti.

Finalne komponente vidimo u fazi implementacije, ali njihovu strukturu planiramo znatno ranije.

Dijagram komponenti predstavlja jedan od strukturnih dijagrama u UML u i koristi se za modelovanje fizičke strukture softverskog sistema. Njegova osnovna namena jeste da prikaže kako je sistem razložen na veće, logički zaokružene celine – komponente, kao i kakve zavisnosti i oblike komunikacije te komponente imaju međusobno. Za razliku od dijagrama klasa, koji se fokusira na logičku strukturu i odnose između klasa, dijagram komponenti se bavi implementacionim aspektima sistema i povezuje dizajn sa stvarnom realizacijom softvera.

8.4.1. Pojam komponente

Komponenta predstavlja modularnu jedinicu softverskog sistema koja enkapsulira (skriva) unutrašnju implementaciju i nudi svoje usluge putem definisanih interfejsa. Notacija komponente prema UML standardu koristi pravougaonu reprezentaciju sa specifičnom ikonom u uglu (Slika 8.43).



Slika 8.43. Različite notacije komponenti

Glavne karakteristike komponente su:

- **Autonomnost**: Predstavlja nezavisan deo sistema ili podsistem.
- **Kohezija**: Sadrži grupu povezanih klasa koje zajedno ostvaruju određenu funkciju.
- **Ponovna upotrebljivost**: Može se koristiti u različitim delovima sistema ili čak u različitim projektima.
- **Zamenljivost**: Zahvaljujući jasnim interfejsima, jedna komponenta se može zameniti drugom (npr. boljom verzijom) bez narušavanja rada ostatka sistema.

Na primer, ključne komponente sistema za onlajn prodavnicu su: upravljanje korisnicima, korpa za kupovinu (*ShoppingCart*), sistem za plaćanje i dr. (Tabela 8.4)

Tabela 8.4. Opis ključnih komponenata sistema onlajn prodavnice

Komponenta	Opis i odgovornost
Katalog proizvoda	Upravlja podacima o artiklima, kategorijama, cenama i zalihama. Komunicira sa bazom podataka.
Upravljanje korisnicima	Zadužena za registraciju, autentifikaciju (*login*) i profile kupaca.
Korpa za kupovinu	Privremeno čuva odabrane artikle korisnika i obračunava trenutni iznos.
Sistem za plaćanje	Komponenta koja se obično povezuje sa spoljnim servisima (npr. PayPal, Stripe) za sigurnu obradu transakcija.
Upravljanje porudžbinama	Generiše fakture, prati status isporuke i čuva istoriju kupovine.
Skladišni sistem (*Inventory*)	Prati stanje na lageru i automatski obaveštava katalog kada proizvoda nestane.
Email/Notifikacioni servis	Šalje potvrde o kupovini i obaveštenja o akcijama korisnicima.

Komponente softvera se mogu grupisati prema tipu i vremenu njihovog nastanka unutar životnog ciklusa razvoja: tokom prevođenja (*compile-time*), povezivanja (*link-time*) ili izvršavanja (*run-time*). One predstavljaju fizičke artefakte softvera, kao što su izvršne datoteke, biblioteke, moduli, servisi, baze podataka, pa čak i dokumenti ili skripte. Uobičajeni stereotipi preciziraju njihovu ulogu:

- «executable»: Predstavlja izvršnu jedinicu namenjenu direktnom pokretanju na procesoru ili unutar izvršnog okruženja (npr. .exe ili .jar fajl).
- «library»: Označava statičku ili dinamičku biblioteku resursa (npr. .dll ili .so) namenjenu ponovnoj upotrebi.
- «source»: Fajl koji sadrži izvorni kod pre procesa kompajliranja u izvršni oblik.
- «file»: Opšti fizički fajl koji može sadržati konfiguracione parametre, podatke ili druge resurse.
- «table»: Specifičan artefakt koji modeluje tabelu unutar baze podataka kao ključni deo sistema za upravljanje podacima.

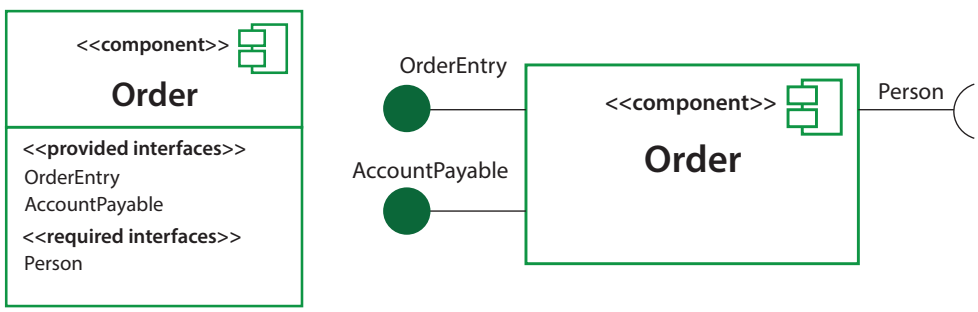


Slika 8.44. Prikaz komponenti sistema kroz UML stereotipe

Komponente definišu svoje ponašanje putem interfejsa. UML razlikuje dve osnovne vrste interfejsa:

- **Obezbeđeni interfejs** (*Provided Interface*): opisuju skup servisa koje komponenta nudi svom okruženju (simbol kruga).
- **Zahtevani interfejs** (*Required Interface*): Usluge koje su komponenti neophodne od drugih delova sistema za ispravan rad (simbol polukruga).

Interfejsi mogu biti prikazani na dva načina (Slika 8.45). Slika levo prikazuje specifikaciju interfejsa unutar strukture komponente, dok slika desno koristi grafičke simbole za eksternu vizuelizaciju ponuđenih i zahtevanih usluga, olakšavajući razumevanje interakcije komponente sa okruženjem.



Slika 8.45. Unutrašnja i spoljašnja (*external view*) notacija interfejsa komponente Order

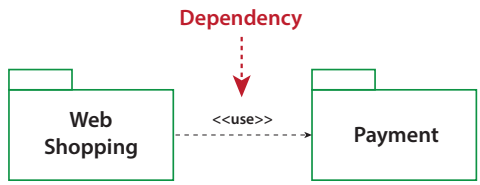
Interfejs predstavlja ugovor između komponente i njenog okruženja. Zahvaljujući tom ugovoru, komponente mogu biti razvijane i menjane nezavisno, sve dok se poštuje definicija interfejsa. Upravo zbog toga, zamena jedne komponente drugom moguća je samo ukoliko obe komponente podržavaju iste interfejse.

Danas, u eri mikroservisne arhitekture, pojam komponente je evoluirao. Jedan mikroservis se na dijagramu komponenti tretira kao samostalna komponenta. Svaki mikroservis poseduje svoje obezbeđene interfejse (npr. REST API) i zahtevane interfejse (npr. pozivi drugim servisima ili bazama podataka). Dijagram komponenti je postao primarni alat za vizuelizaciju složenih distribuiranih sistema, gde svaka komponenta predstavlja nezavisno razvijen i isporučen servis.

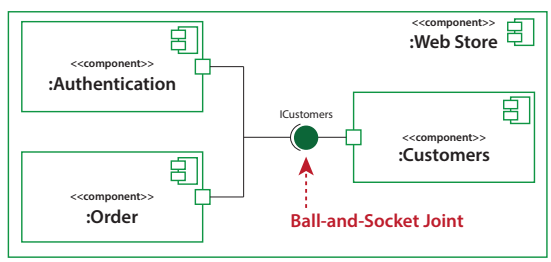
8.4.2. Relacije između komponenti

U UML dijagramu komponenti, relacije opisuju kako komponente sarađuju kako bi ostvarile funkcionalnost sistema. Razumevanje ovih veza je ključno za analizu zavisnosti i procenu uticaja promena u sistemu. Glavni tipovi relacija i konektora su:

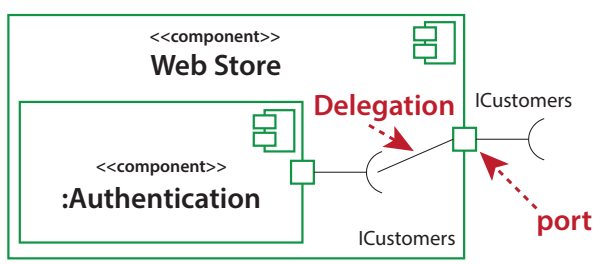
- **Zavisnost** (*Dependency*): Najčešća relacija koja ukazuje da jedna komponenta (klijent) zahteva prisustvo ili usluge druge komponente (dobavljač) za svoje pravilno funkcionisanje. Na dijagramu se prikazuje kao isprekidana strelica usmerena ka komponenti od koje se zavisi.



- **Sastavljanje** (*Assembly Connector*): Ova relacija direktno povezuje obezbeđeni i zahtevani interfejs dveju komponenti. Poznata je i kao "loptica i utičnica" (*ball-and-socket*) konekcija. Ona grafički prikazuje da usluga koju jedna komponenta nudi savršeno odgovara potrebama druge.



- **Delegacija** (*Delegation Connector*): Koristi se za povezivanje spoljašnjeg interfejsa komponente sa njenom unutrašnjom realizacijom (npr. konkretnim klasama ili podkomponentama). On "delegira" poruku sa granice komponente do unutrašnjeg dela koji zapravo obavlja posao.



- **Portovi** (*Ports*): Predstavljaju tačku interakcije između komponente i njenog okruženja ili između komponente i njenih internih delova. Portovi omogućavaju preciznije definisanje komunikacionih tačaka i često se koriste kod složenih ili kompozitnih komponenti.

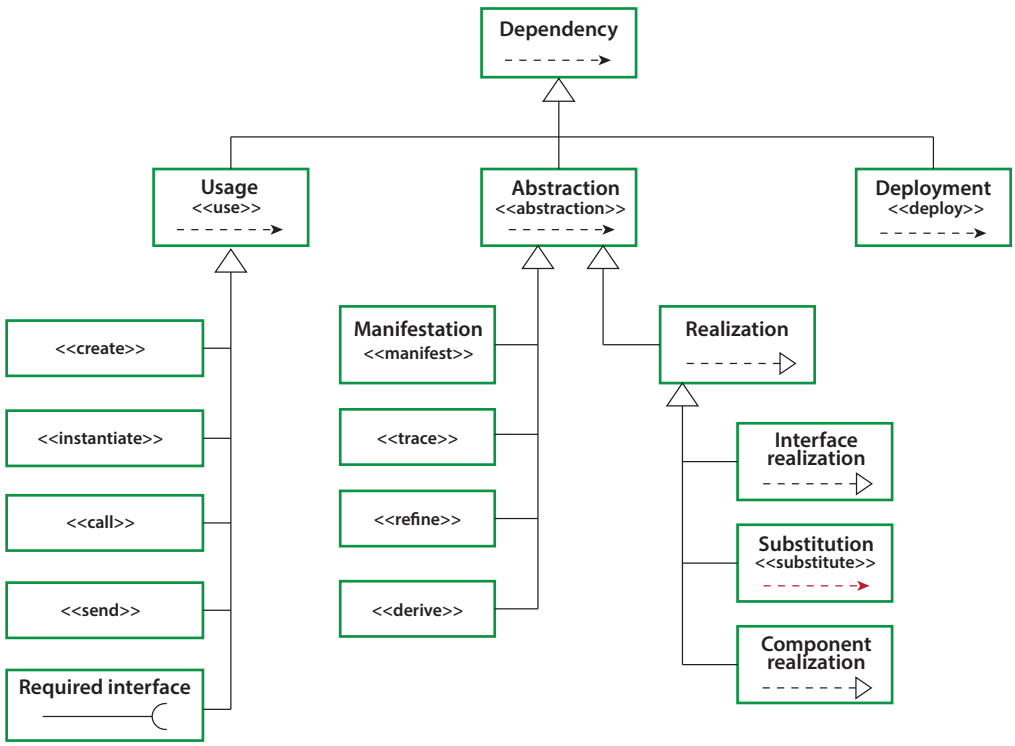
Relacija zavisnosti

Osnovna relacija koja povezuje komponente je **zavisnost** (*dependency*). Ona ukazuje na to da jedna komponenta (klijent) zahteva prisustvo ili usluge druge komponente (dobavljač/supplier) kako bi ispravno funkcionisala. Primeri zavisnosti u realnim sistemima:

- **Izvršni fajlovi**: Izvršna komponenta (npr. .exe) može zahtevati pristup dinamičkim bibliotekama (npr. .dll) radi korišćenja specifičnih funkcija.
- **Klijent-server arhitektura**: Klijentska aplikacija zavisi od serverske aplikacije, koja dalje može zavisiti od interfejsa baze podataka.

Zavisnosti se u UML-u detaljnije specifikuju korišćenjem standardnih stereotipa, koji se mogu podeliti u nekoliko ključnih kategorija (Slika 8.46):

1. **Usage** («use»): Ovo je najčešći tip zavisnosti gde klijent zahteva drugi element (*supplier*) kako bi se potpuno definisao ili implementirao. Podtipovi uključuju:
   - **«create» i «instantiate»**: Kada jedna komponenta kreira instance druge.
   - **«call» i «send»**: Označavaju pozivanje operacija ili slanje signala/poruka.
   - **Zahtevani interfejs**: Specifičan oblik upotrebe gde komponenta komunicira preko definisanog "polukruga".
2. **Abstraction** («abstraction»): Povezuje dva elementa koji predstavljaju isti koncept, ali na različitim nivoima apstrakcije ili fazama razvoja.
   - **Realization**: Ključni podtip apstrakcije koji povezuje dizajn (npr. interfejs) sa njegovom fizičkom implementacijom (komponentom).
   - **Manifestation**: Pokazuje kako se modelni element (poput klase) manifestuje u fizičkom artefaktu.
3. **Deployment** («deploy»): Specifična zavisnost koja pokazuje lokaciju nekog elementa, odnosno na koji je hardverski čvor ili izvršno okruženje komponenta raspoređena.



Slika 8.46. Pregled relacija zavisnosti

Dijagram komponenti ne služi samo za crtanje strukture, već kao alat za upravljanje složenošću. Jasnim definisanjem relacija i interfejsa, razvojni timovi mogu raditi paralelno na različitim modulima, osiguravajući da će se oni, jednom kada budu gotovi, besprekorno uklopiti u jedinstvenu celinu.

Primer relacija između komponenti sistema onlajn prodavnice

Kako bi se komponente sistema onlajn prodavnice (Katalog, Korpa, Plaćanje) povezale u smislenu celinu, na dijagramu kompnenti bismo definisali sledeće veze:

- Komponenta "Korpa za kupovinu" ima zavisnost ka komponenti "Katalog proizvoda" jer mora proveriti dostupnost i cenu artikla pre dodavanja.
- Komponenta "Upravljanje porudžbinama" ostvaruje *assembly* relaciju sa "Sistemom za plaćanje". "Sistem za plaćanje" nudi (*Provided Interface*) uslugu autorizacije, dok je "Upravljanje porudžbinama" zahteva (*Required Interface*) kako bi finalizovala prodaju.
- Email/Notifikacioni servis deluje kao nezavisna komponenta koju koriste gotovo sve ostale komponente (Korisnici, Porudžbine, Skladište) putem zajedničkog interfejsa za slanje poruka.

8.4.3. Osnovni elementi dijagrama komponenti

Dijagram komponenti modeluje strukturu softvera, ukazujući na zavisnosti između softverskih komponenata. Crta se kao graf komonenti, ponekad grupisanih u pakete ili podsisteme, na kojem se prikazuju zavisnosti između njih.

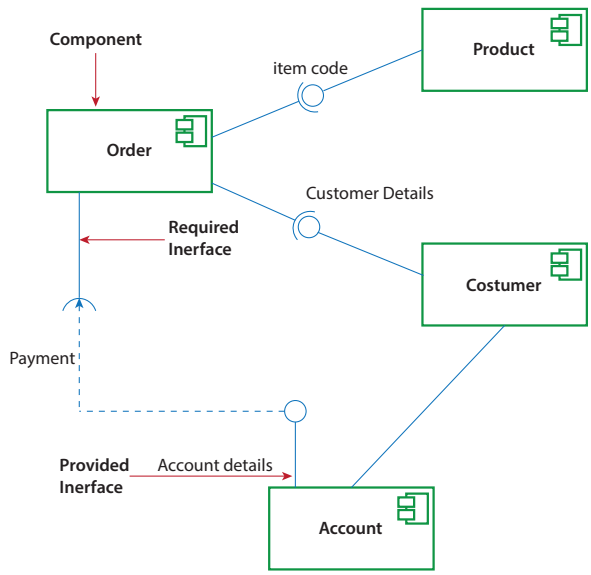
Namena dijagrama komponenti je da definiše softverske module i njihove međusobne relacije. Komponenta je autonomna i enkapsulirana jedinica unutar sistema ili podsistema koja mora da definiše interfejs koji omogućava drugim komponenatama da komuniciraju sa njom. Svaka komponenta softvera je deo koda koji se nalazi u memoriji na nekom delu hardvera. Navodeći ključne elemente kao što su komponente, interfejsi komponenti i zavisnosti može se opisati fizička implementacija sistema u terminima softverskih modula i njihovih međusobnih odnosa.

Prema standardnoj specifikaciji, dijagram se sastoji od sledećih elemenata:

- **Komponenta** (*Component*): Modularni deo sistema koji enkapsulira sadržaj i čije je manifestovanje u okviru izvršnog okruženja zamenljivo. Na dijagramu se prikazuje kao pravougaonik sa ikonom komponente u gornjem desnom uglu ili stereotipom «*component*».
- **Interfejs** (*Interface*): Definiše skup operacija koje komponenta nudi ili zahteva.
  - **Obezbeđeni interfejs** (*Provided Interface*): predstavlja usluge koje komponenta nudi okruženju.
  - **Zahtevani interfejs** (*Required Interface*): predstavlja usluge koje su komponenti neophodne za rad.
- **Zavisnost** (*Dependency*): Usmerena linija koja pokazuje da jedna komponenta zavisi od druge ili koristi njene usluge radi svoje potpune implementacije.
- **Port**: Tačka interakcije između komponente i njenog okruženja, koja omogućava delegiranje zahteva unutrašnjim strukturama. Predstavljen je malim kvadratom na obodu složene komponente (koja sadrži u sebi i druge komponente).
- **Konektori** (*Connector*): Definiše vezu između komponenti ili njihovih delova, a javlja se u dva osnovna tipa:
  - ***Assembly*** konektor: Povezuje zahtevani interfejs jedne komponente sa ponuđenim interfejsom druge, čime se formira funkcionalna celina.
  - ***Delegation*** konektor: Povezuje port komponente sa njenom unutrašnjom strukturom, omogućavajući prenos zahteva od spoljašnjeg interfejsa do konkretnog realizatora funkcionalnosti.

Dijagram komponenti na slici 8.45, ilustruje međuzavisnost i povezivanje četiri ključne komponente: Order, Product, Customer i Account. Centralna komponenta *Order* funkcioniše kao klijent koji putem zahtevanih interfejsa (simbol polukruga ili "utičnice") potražuje podatke poput koda artikla (*Item code*), detalja o kupcu (*Customer Details*) i plaćanja (*Payment*). Sa druge strane, komponente poput *Account* nude svoje funkcionalnosti kroz ponuđene interfejse (simbol kruga), kao što je *Account details*. Ova linija označava da komponenta *Order* zavisi od interfejsa koji nudi *Account*. Strelica pokazuje smer zavisnosti – ako se interfejs *Account details* promeni, to će direktno uticati na komponentu *Order*.

Veza između komponenti *Customer* i *Account* je prikazana kao obična puna linija (asocijacija). Korišćenje obične linije (asocijacije) umesto assembly konektora smatra se manje formalnim. Dok je asocijacija standardna na dijagramu klasa, na dijagramu komponenti ona često služi kao "prečica" da se pokaže da komponente komuniciraju, ali bez preciziranja koji interfejsi se koriste. Takva relacija sugeriše jaču spregu (*tight coupling*). Dok interfejsi (*Payment, item code*) jasno odvajaju šta se radi od toga kako se radi, obična linija može sugerisati da komponente direktno pristupaju jedna drugoj, što narušava princip enkapsulacije. Kod tehničke dokumentacije, bilo bi bolje da umesto relacije asocijacije stoji assembly konektor. To bi jasno definisalo koji tačno set usluga (npr. *AccountLookup*) komponenta *Customer* zahteva od komponente *Account*.



Slika 8.47. Osnovni elementi dijagrama komponenti

Razlike između relacije *assembly connector* i *depedency* je prikazana u tabeli 8.5.

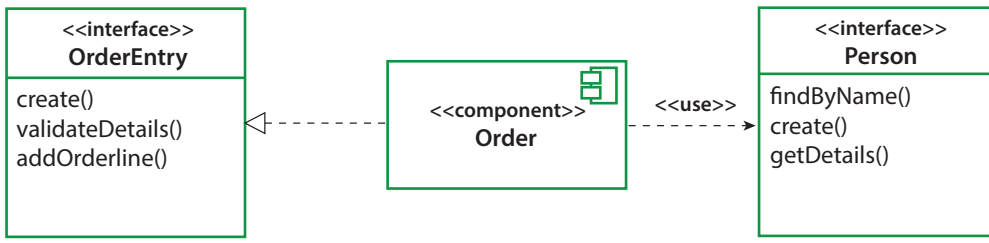
Tabela 8.5. Poređenje direktnog povezivanja (*Assembly*) i logičke zavisnosti (*Dependency*)

Karakteristika	*Assembly connector*	*Dependency*
Tip linije	Puna linija (direktan spoj)	Isprekidana linija sa strelicom
Fokus	**Sklapanje**: Prikazuje da su delovi fizički povezani.	**Zavisnost**: Prikazuje da jedna strana ne može raditi bez definicije druge.
Stepen sprege	Nešto jača (direktno uparivanje). Npr. Order direktno konzumira uslugu Customer Details koju nudi kompnenta Customer.	Maksimalno labava (fokus na ugovoru/ interfejsu). Npr. komponenta Order je dizajnirana tako da joj je potreban bilo koij izvor koji ispunjava Account details interfejs.

UML omogućava dva osnovna načina posmatranja komponenti:

- ***Black box*** (crna kutija) **prikaz** – interna struktura komponente nije vidljiva; prikazuju se samo interfejsi koje komponenta obezbeđuje ili zahteva;
- ***White box*** (bela kutija) **prikaz** – prikazuje se interna struktura komponente, uključujući klase, interne komponente i način realizacije interfejsa.

Black box prikaz je pogodan za nivo modelovanja arhitekture sistema, dok se *white box* prikaz koristi kada je potrebno detaljnije objasniti implementaciju i unutrašnju organizaciju komponente.

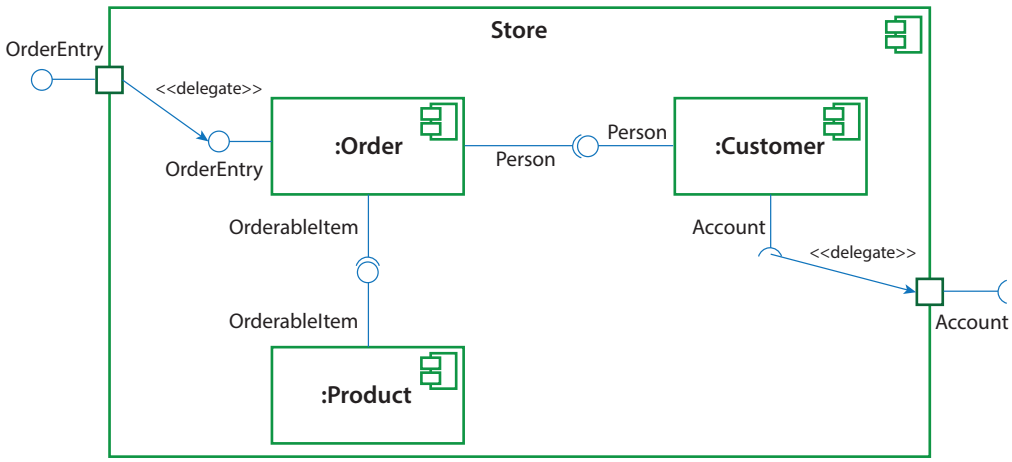


Slika 8.48. Prošireni "black-box" prikaz sa detaljnom specifikacijom interfejsa kompnente Order

Dijagram na slici 8.47, ilustruje hijerarhijsku strukturu složene komponente (podsistema) Store i načine na koje njene unutrašnje komponente komuniciraju međusobno i sa spoljašnjim okruženjem putem dve vrste konektora i to:

- ***Delegation*** **konektor** (Delegacija): Prikazan strelicom sa oznakom «delegate». Ovaj konektor povezuje spoljašnju granicu sistema sa unutrašnjim strukturama. Port za ulazni interfejs OrderEntry delegira zahteve direktno unutrašnjoj komponenti Order, dok komponenta Customer delegira svoje potrebe ka spoljašnjem portu za Account.
- ***Assembly*** **konektor** (Sklapanje): Predstavljen je spojem "kruga" (ponuđeni interfejs) i "polukruga" (zahtevani interfejs). Ovaj konektor povezuje komponente na istom nivou apstrakcije. Veza između komponenti *Order* i *Product* putem interfejsa *OrderableItem*, omogućava da komponenta *Order* koristi funkcionalnosti komponente Product bez poznavanja njene unutrašnje implementacije.

Ovakav prikaz demonstrira princip enkapsulacije, gde podsistem Store funkcioniše kao "crna kutija" za spoljašnje klijente, dok se unutrašnja logika precizno distribuira između specijalizovanih modula putem strogo definisanih tačaka interakcije.



Slika 8.49. Dijagram podsistema *Store*

Dakle, dijagrami komponenti imaju višestruku ulogu u razvoju softvera, oni pomažu u:

- definisanju softverskih modula i njihovih odgovornosti;
- razumevanju zavisnosti između delova sistema;
- planiranju razvoja i raspodele rada;
- dokumentovanju arhitekture sistema.

Zahvaljujući svojoj jasnoći i standardizovanoj notaciji, dijagram komponenti predstavlja važan alat za opisivanje strukture sistema i čini sastavni deo savremene UML dokumentacije.

8.4.4. Dijagram kompozitne strukture

Dijagram kompozitne strukture (*Composite Structure Diagram*) je strukturni UML dijagram koji se koristi za prikaz unutrašnje arhitekture jednog složenog objekta (kao što su klasa, komponenta ili ceo podsistem). On detaljno prikazuje njegove delove i njihove međusobne interakcije. Dok standardni dijagram klasa prikazuje šta sistem poseduje (statički plan), dijagram kompozitne strukture prikazuje kako ti delovi zajedno funkcionišu u realnom vremenu kako bi ostvarili konkretnu funkciju.

Namena ovog dijagrama je:

- **Detaljna dekompozicija**: Omogućava projektantu da „zaviri unutar" jedne klase ili komponente i vidi od čega se ona tačno sastoji.
- **Enkapsulacija putem portova**: Definiše precizne tačke (kapije) kroz koje sistem komunicira sa spoljnim svetom, čime se sakriva unutrašnja složenost.
- **Modelovanje uloga** (*Roles*): Ne fokusira se na tipove podataka, već na uloge koje instance igraju u konkretnoj saradnji.

Osnovni elementi dijagrama su:

- **Okvir** (*Boundary*): Veliki pravougaonik koji predstavlja složeni objekat koji modelujemo (npr. podsistem ili klasu).
- **Deo** (*Part*): Predstavlja ulogu koju instanca igra unutar sistema. Crta se kao pravougaonik unutar okvira. Na primer, *webStore* je deo (*part*) jer on predstavlja konkretnu komponentu koja se koristi unutar strukture sistema za prodaju.
- **Port**: Tačka interakcije na ivici okvira. Omogućava slanje poruka bez otkrivanja unutrašnje strukture.
- **Konektor**: Linija koja omogućava komunikaciju između delova ili između portova i delova.
- **Interfejsi** (zahtevani i ponuđeni): Definišu ugovor o komunikaciji (šta deo nudi drugima, a šta mu je potrebno za rad).

Jedna od najvažnijih primena ovog dijagrama je prikazivanje kolaboracija. Kolaboracija opisuje mehanizam koji definiše kako skup delova sarađuje da bi izvršio određeni zadatak, bez potrebe da se opisuje cela klasa. Za razliku od diajgrama klasa koji prikazuje statičke veze (npr. nasleđivanje) koje su fiksne i nepromenljive, dijagram kompozitne strukture prikazuje dinamičnu konfiguraciju delova u specifičnom trenutku ili kontekstu.

Na slici ispod je ilustrovan princip labave povezanosti (*loose coupling*) između dva dela sistema skladišta (*warehouse*) i internet prodavnice (*webStore*):

- **Povezivanje putem interfejsa**: Umesto da klasa WebStore direktno poziva metode klase Warehouse, one su povezane preko portova i interfejsa SearchInventory. Ovo je funkcija pretrage zaliha koju skladište stavlja na raspolaganje, dok WebStore koristi tu uslugu kako bi prikazao korisnicima dostupne proizvode.
- **Uloga konektora**: Konektor koji spaja krug (pruženi interfejs) i polukrug (zahtevani interfejs) predstavlja uspostavljenu vezu u realnom vremenu. On omogućava da zahtev iz prodavnice stigne do konkretne instance skladišta.
- **Prednost enkapsulacije**: Ovakva struktura znači da WebStore ne mora da poznaje unutrašnju logiku baze podataka ili unutrašnju strukturu skladišta, već komunicira isključivo preko definisanog porta.
- **Fleksibilnost sistema**: Ovakav pristup omogućava da se kasnije modul Warehouse zameni nekim drugim modulom (npr. eksternim dobavljačem), a da se pritom ne menja kôd unutar modula WebStore, sve dok novi modul poštuje isti port i interfejs.



Vežba 8.5. Nacrtati dijagram komponenti za sistem za rezervaciju i prodaju karata u IBM RSA alatu

Sistem za rezervaciju i prodaju karata podržava sledeće kanale:

- **Kiosk** (Self-Service Terminal): Namenjen kupcima na licu mesta. Ovaj kanal zahteva (requires) interfejse *IEventLookup* za pregled repertoara, *IReservation* za proces rezervacije i *ISecureAuth* za opcionu prijavu korisnika.
- **ClerkAdminPOS**: Radna stanica za službenike. Pored *IReservation* i *ICancelation*, ovaj kanal zahteva i napredne interfejse kao što su *ITicketAdmin* za storniranje karata i *IReportGen* za uvid u statistiku prodaje.
- **Onlajn korisnik** (UserAccount): Omogućava registrovanim korisnicima pristup personalizovanim uslugama. Ova komponenta zahteva *IDataStorage* za čuvanje profila, dok sistemu nudi interfejse *IUserProfile* i *IUserContract* koje koriste drugi moduli, poput *SecurityManagement*-a i *Notification*-a.

Centralna komponenta ***TicketReservation*** koordinira proces kupovine i otkazivanja karata. Ona nudi (provides) interfejse *IReservation* i *ICancelation* koje koriste ulazni terminali (Kiosk i Clerk). Za realizaciju ovih funkcija, ona zahteva (requires):

- *IEventLookup* od komponente *EventCatalog* (za izbor filma/događaja).
- *ISeatAvailability* od komponente *SeatMap* (za proveru slobodnih mesta).
- Prilikom obrade, sistem se oslanja na *PriceEngine* putem interfejsa *IPriceCalculation* za dobijanje tačne cene.

Nakon obračuna, TicketReservation koristi:

- *IProcessPayment* kako bi prosledila zahtev za naplatu komponenti *PaymentProcess*.
- *INotify* kako bi obavestila korisnika o statusu rezervacije.
- *ISecureAuth* (iz SecurityManagement) kako bi osigurala da je korisnik verifikovan.

Komponente su:

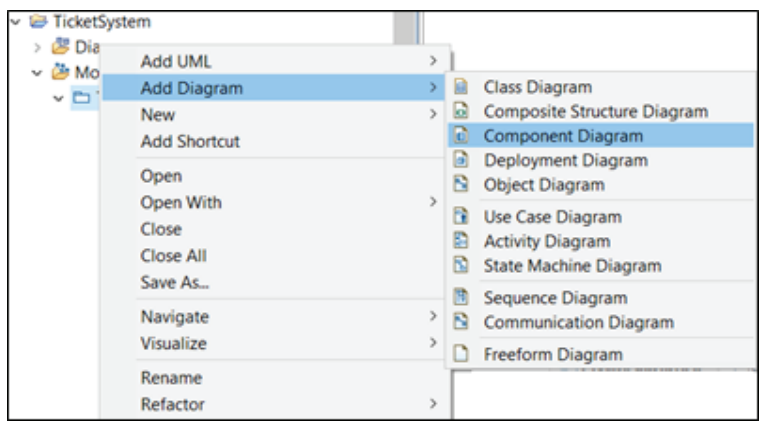
- ***EventCatalog*** (Katalog događaja): upravlja podacima o predstavama, utakmicama ili koncertima. TicketReservation koristi ovu komponentu za pretragu dostupnih događaja (IEventLookup).
- ***SeatMap*** (Mapa sedišta): Poseban modul koji vodi računa o fizičkom rasporedu mesta u sali i njihovoj dostupnosti u realnom vremenu (kako bi se izbeglo da dva korisnika kupe istu kartu). Nudi proveru dostupnosti mesta (ISeatAvailability) kojeg zahteva TicketReservation.
- ***PricingEngine*** (Modul za cene): Komponenta koja obračunava popuste (npr. grupni popusti, pretplata) pre nego što se pređe na *Payment*. TicketReservation zahteva *IPriceCalculation*.
- ***NotificationService*** (Servis za obaveštenja): Odgovoran za generisanje i slanje elektronskih karata (e-mail ili SMS) nakon uspešne uplate. Ova komponenta nudi interfejs INotify, a zahteva *IemailService* (za povezivanje sa eksternim Email

Serverom) i *ISMSService* (za povezivanje sa eksternim SMS Gateway-om), kao i IUserContract kako bi od komponente *UserAccount* dobila neophodne kontakt informacije korisnika za isporuku generisane karte ili obaveštenja.

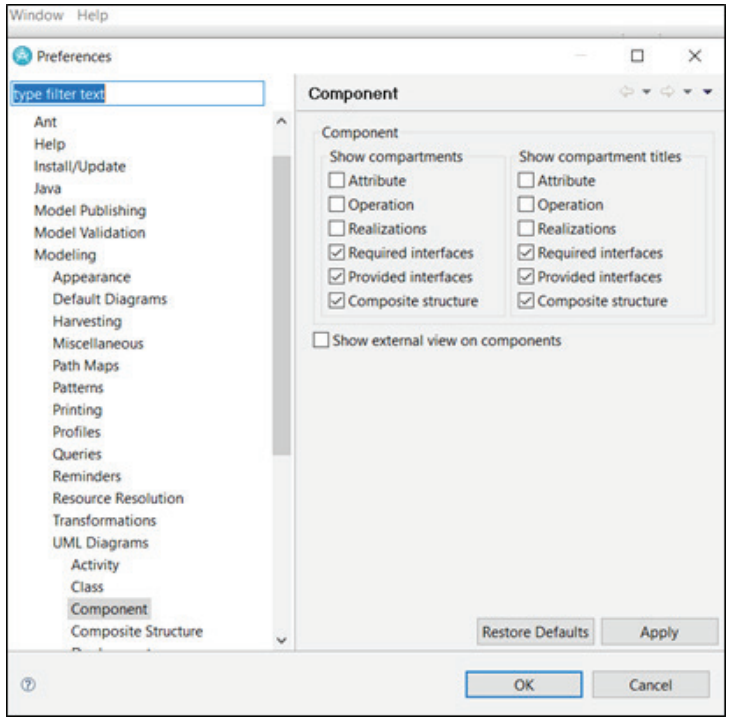
- ***PaymentProcess***: Ova komponenta je odgovorna za realizaciju finansijskih transakcija. Ona nudi interfejs *IProcessPayment* koji koriste komponente za rezervaciju, a zahteva *IPaymentGateway* za komunikaciju sa spoljnim sistemom naplate. Takođe, zahteva INotify kako bi, po uspešnoj uplati, signalizirala *NotificationService*-u da pošalje potvrdu i kartu korisniku.
- ***TicketDB*** (Baza podataka): Predstavlja centralno skladište sistema. Pruža interfejse: IDataStorage za osnovno čuvanje objekata (karte, korisnici), IDataAccess za bezbednosne provere i autorizaciju, i IDataQuery koji je optimizovan za kompleksno izvlačenje podataka potrebnih modulu za izveštavanje (Reporting).
- ***SecurityManagement***: Nudi *ISecureAuth* i *ISession*, a zahteva *IDataAccess* (iz TicketDB) kako bi uporedio lozinke i prava pristupa.
- ***Reporting***: Ova komponenta je zadužena za obradu podataka i vizuelni prikaz poslovnih rezultata. Ona nudi interfejs *IReportGen* koji koristi *ClerkAdminPOS* za generisanje dnevnih izveštaja, a zahteva *IDataQuery* od baze podataka kako bi izvršila složene analitičke upite nad statistikom prodaje.

Koraci izrade vežbe:

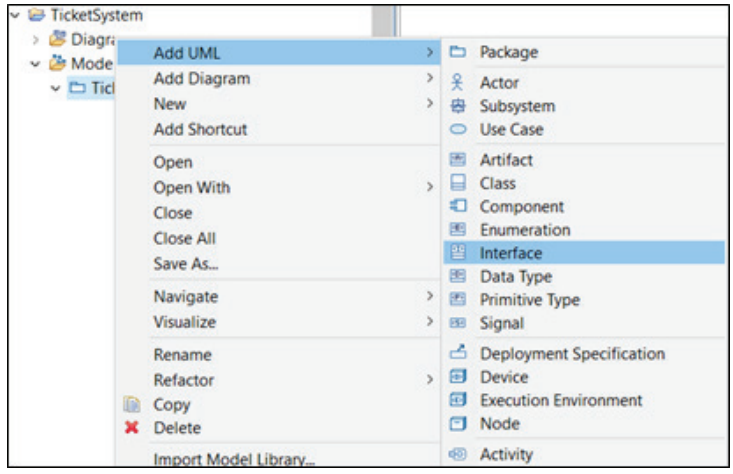
1. **Kreiranje dijagrama komponenti u IBM RSA alatu**. Kada ste kreirali projekat, desnim klikom na paket i klikom na *Add Diagram*, odaberite *Component Diagram*:



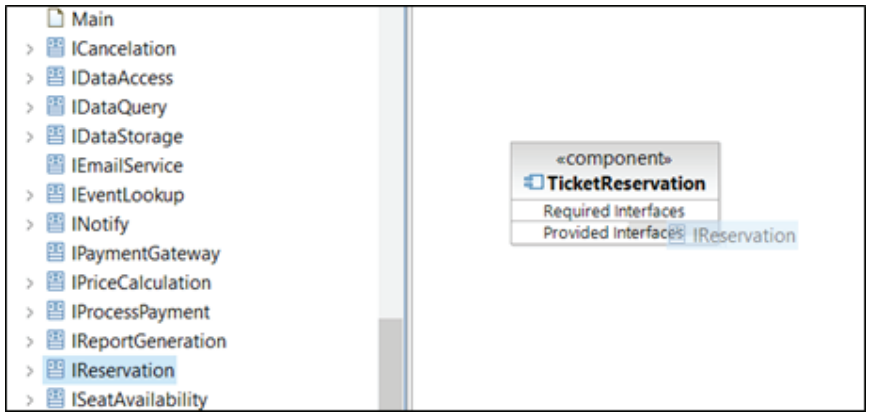
2. **Podešavanje vidljivosti pregrada u komponenti**. U meniju pod *Window*, odaberite *Preferences*, zatim proširite *Modeling*, potom *UML Diagrams* i odaberite *Component*. Sa desne strane čekriati opcije pod *Show compartments: Required interfaces* i *Provided interfaces*. Ukoliko želite da vam se prikažu i nazivi odeljaka tada pod *Show compartment* titles čekirati isto *Required* i *Provided* interfejse. Ukoliko kreirate složenije komponente, tada ćete želeti da uključite i *composite structure* deo komponente.



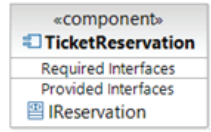
3. **Kreiranje interfejsa koje će komponente nuditi ili zahtevati.** Ukoliko već nemate kreirane interfejse prilikom crtanja prethodnih dijagrama, na dijagramu komponenti ih možete ručno dodati na više načina. Najjednostavniji način je da se desnim klikom na paket odabere opcija *Add UML* i selektuje *Interface* i imenujte interfejse kako se traže u zadatku.



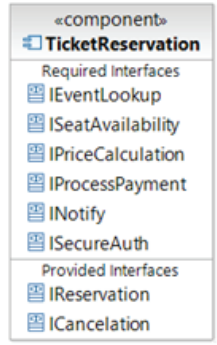
4. **Dodati komponente na dijagramu komponenti i prevući interfejse**. Sa palete prevući *Component* na centralni deo i imenovati kao *TicketReservation*. Sa leve strane pronaći interfejs *IReservation* i prevući pod *Provided Interface* u okviru komponente:



Rezultat prevlačenja interfejsa:

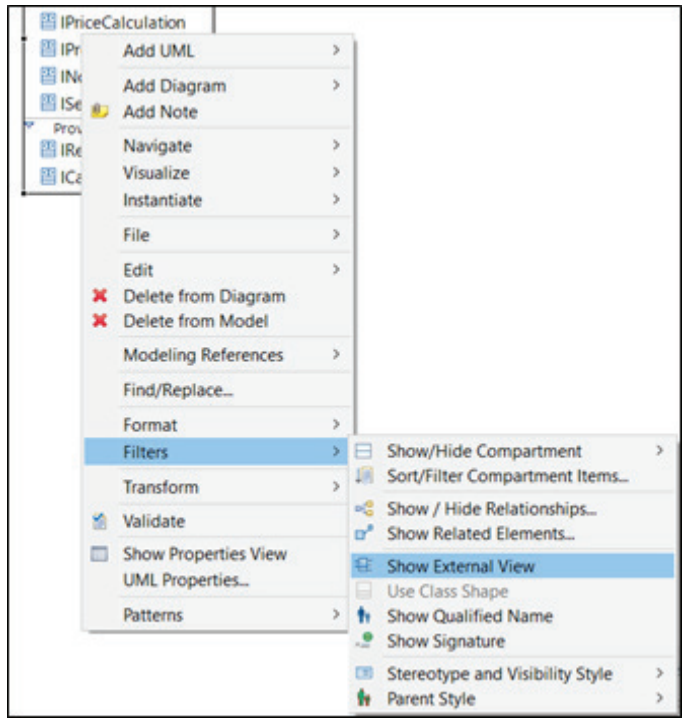


Na identičan način prevući i ostale interfejse u provided ili required u zavisnosti od zahteva:

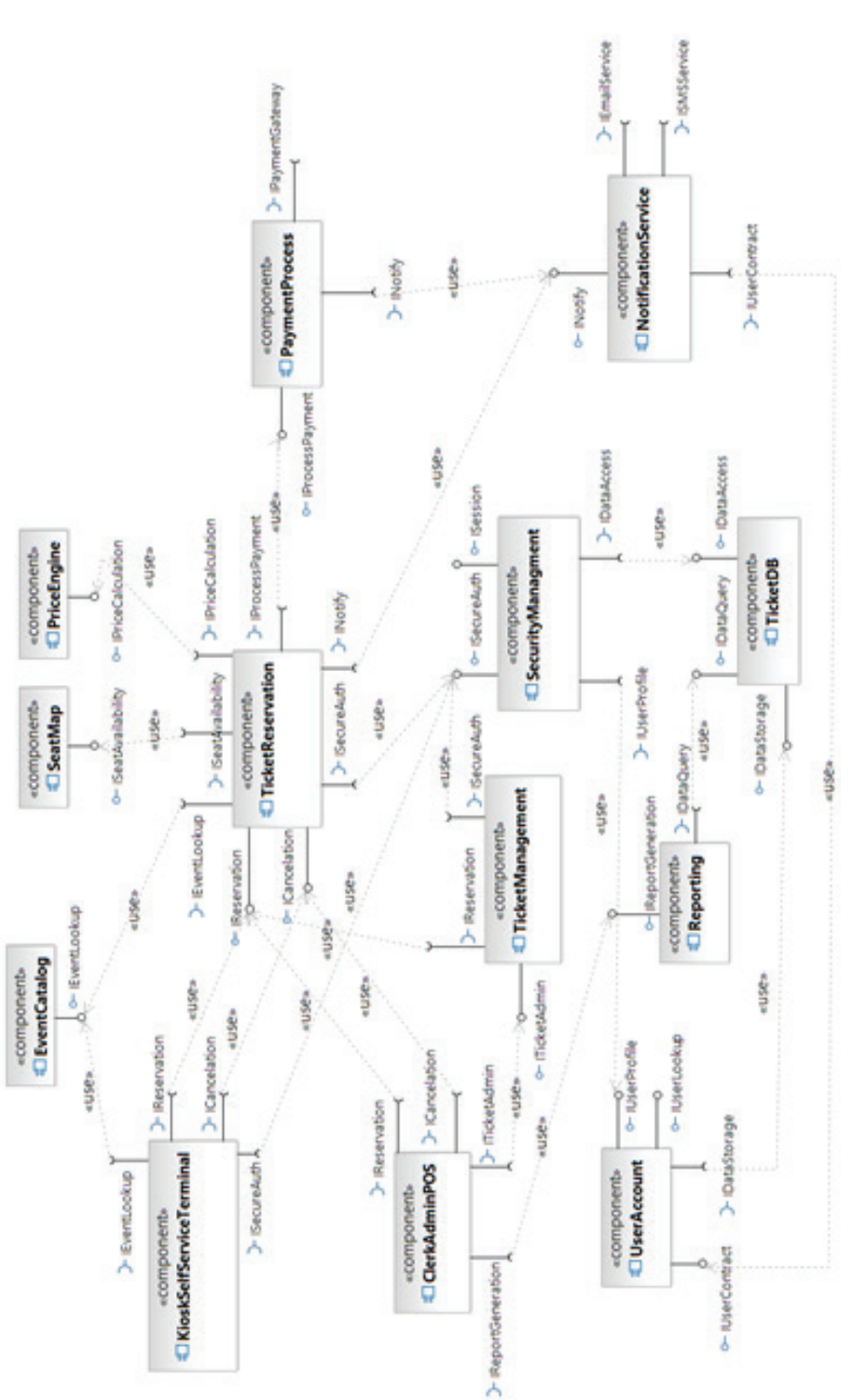


Dodati sve komponente i njihove interfejse.

5. ***External View* prikaz i povezivanje komponenti**. Da bi se interfejsi prikazali izvan komponente (bilo uvidu kruga ili polukruga) neophodno je selektovati sve komponente, zatim desnim klikom odabrati *Filters* i kliknuti na *Show External View*:



Sa palete odabrati relaciju *Usage* i prevući od polukruga (required interface) ka punom krugu (provided interface). Na ovaj način će se prikazati relacije zavisnosti između komponenata. Finalni dijagram komponenti je prikazan na sledećoj slici 8.48:



Slika 8.50. Dijagram komponenti sistema za rezervaciju i prodaju karata

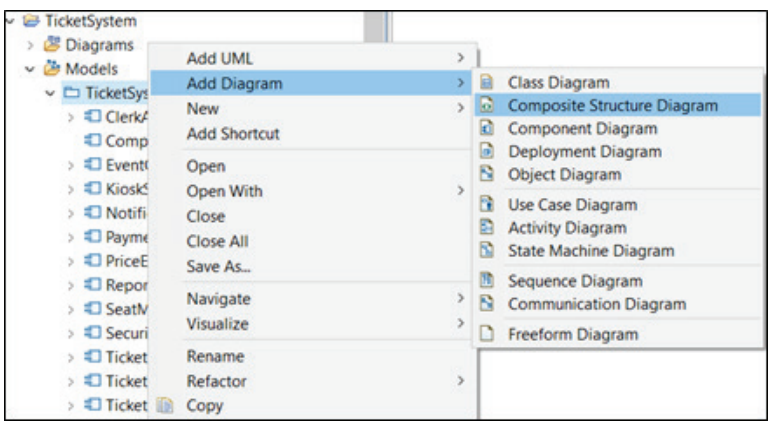
Vežba 8.6. Kreirati kompozitni strukturni dijagram sistema *TicketingSystem*.

Potrebno je modelovati arhitekturu podsistema ***TicketingSystem*** i definisati njegove interakcije sa spoljnim okruženjem. Podsistem funkcioniše kao centralni deo koji se oslanja na nekoliko eksternih servisa kako bi izvršio svoje operacije. Zahtevi za povezivanje su:

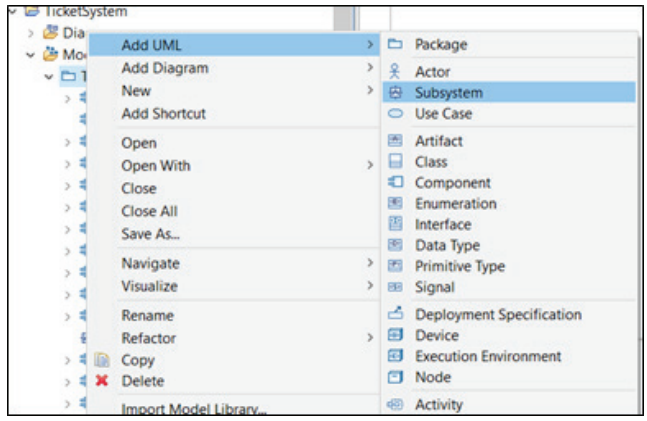
1. **Bankarski sistem** (*BankingSystem*): Podsistem zahteva interfejs IPaymentGateway za realizaciju transakcija.
2. Sistemi za obaveštavanje: Od **SMS Gateway** sistema zahteva se servis ISMSService. Od **Email servera** zahteva se servis IEmailService.
3. **Geolokacioni servisi** (*Maps/Location Service*): Za potrebe prikaza lokacije dvorane na mapi i izračunavanja udaljenosti korisnika od bioskopa, podsistem mora imati portove za interfejse ILocationService i IMapIntegration.
4. **Katalog događaja** (*EventCatalog*): Za sinhronizaciju podataka o događajima sa eksternim izvorima, zahteva se interfejs IExternalEventSync.

Koraci rešenja zadatka:

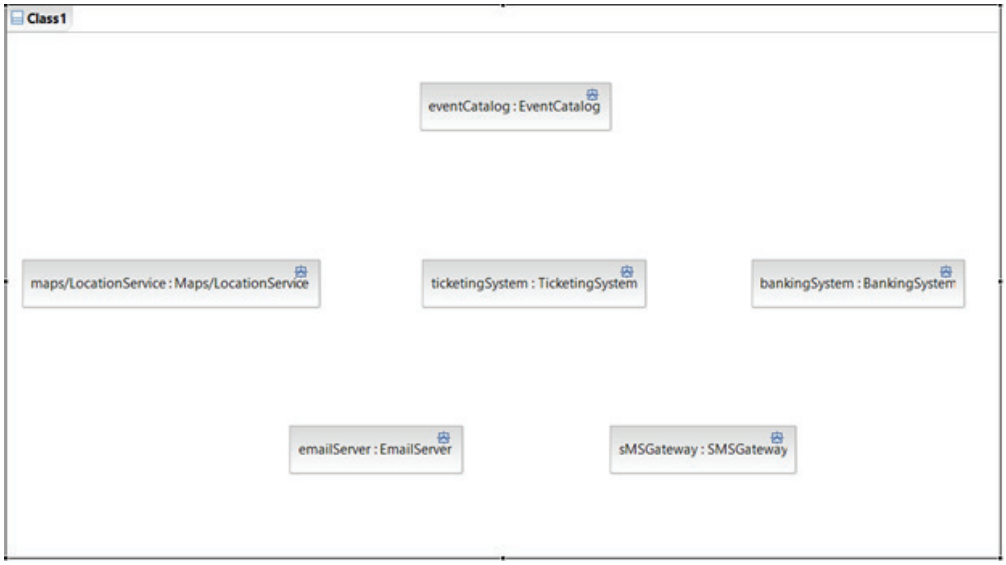
Korak 1. Kreiranje dijagrama kompozitne strukture. Prvi korak je otvaranje novog UML dijagrama kompozitne strukture unutar radnog okruženja. Ovaj dijagram će služiti kao okvir za prikaz unutrašnje organizacije našeg sistema.



Korak 2. Dodavanje podsistema (*Subsystems*). Podsisteme dodajemo desnim klikom na paket u stablu projekta (Add UML -> Subsystem). Podsisteme koje je potrebno kreirati su: TicketingSystem, EventCatalog, BankingSystem, SMSGateway, EmailServer i Maps/LocationService.

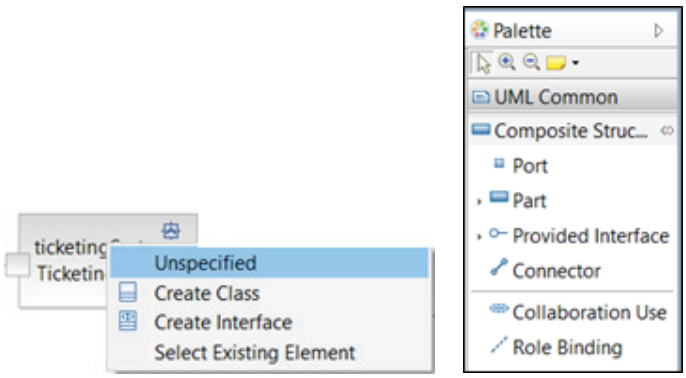


Kreirane podsisteme prevući na dijagram. *TicketingSystem* postaviti kao centralni, uokvireni deo (*Boundary*), jer on predstavlja sistem koji projektujemo. Ostali podsistemi se postavljaju oko njega kao spoljašnji saradnici. Okvir sistema omogućava enkapsulaciju: sve što se dešava unutar njega je skriveno, a komunikacija sa spoljnim svetom (ostalim podsistemima) odvija se isključivo preko portova na njegovim granicama.

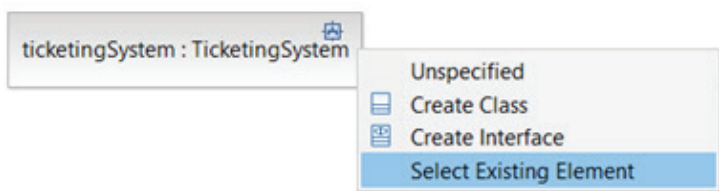


Korak 3. Kreiranje portova i definisanje interfejsa.

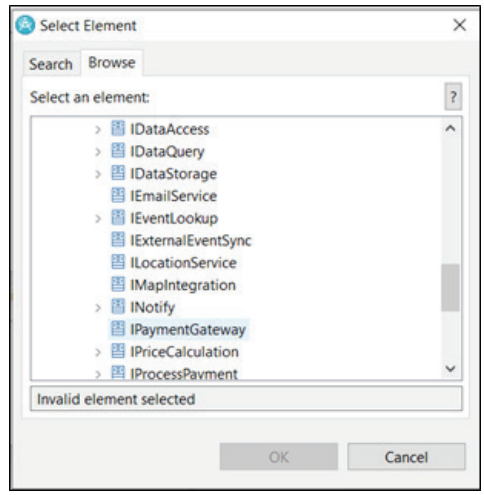
Sa palete odabrati „port" i kliknuti na podsistem, npr. *TicketingSystem*. Iz ponuđenog menija odabrati *Unspecified*, jer ćemo kasnije specificirati da li se radi o *provided* ili *required* interfejsu koji će se kroz ovaj port nuditi ili zahtevati. Isti postupak ponoviti i na druge podsisteme.



Nakon dodavanja portova, selektovati sa palete *provided interface* i prevući do porta podsistema koji nude neku uslugu. Na primer, BankingSystem nudi interfejs *IPaymentGateway*. S obzirom da je ovaj interfejs već kreiran, selektovaćemo opciju *Select Existing Element*.



U polju za pretragu (*Search*) ili klikom na drugi tab *Browse*, ukucati ili pronaći željeni interfejs.

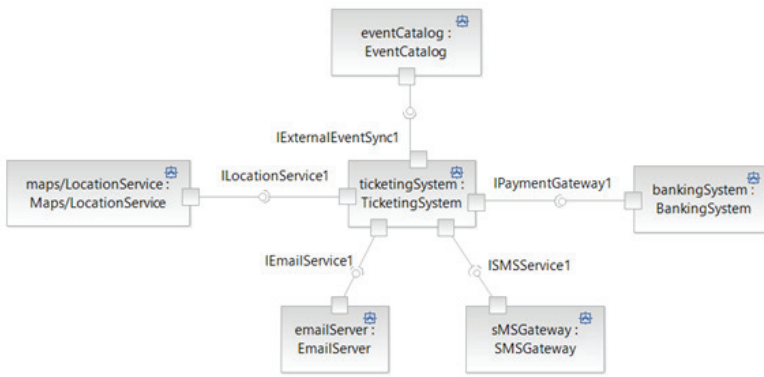


Ovaj postupak prevlačenja bilo *provided* ili *required* interfejsa na portove podsistema, ponoviti za svaki podsistem. Ukoliko, nemate kreiran neki interfejs, možete izabrati i opciju *Create interface* i naknadno ga dodati.

Korak 4. Povezivanje portova podsistema i finaliziranje dijagrama.

Povezivanje portova se vrši tako što se sa palete odabere element *Connector* i prevuče od porta koji predstalja *required interface* ka portu koji nudi (*provided*) interfejs. Kao rezultat kreiraće se ball and socket notacija, tj. notacija konektora sklapanja (*assembly*). Ovaj konektor ne označava fizičku vezu u kodu, već logičku zavisnost između delova sistema. Dakle, konektor sklapanja (*Assembly Connector*) označava da su podsistemi uspešno integrisani i spremni za razmenu poruka u fazi izvršavanja programa.

Finalni dijagram je prikazan na slici 8.49.



Slika 8.51. Kompozitni strukturni dijagram za *TicketingSystem*

Zadatak 8.2. Kreirati dijagram komponenti za sistem onlajn prodavnice

Sistem onlajn prodavnice je organizovan u nekoliko glavnih komponenti koje komuniciraju međusobno kroz jasno definisane interfejse:

Portal kupca (*CustomerPortal*) – korisnički interfejs namenjen krajnjim korisnicima (kupcima) koji omogućava pregled kataloga proizvoda, upravljanje šoping korpom i kreiranje porudžbine. Ova komponenta koristi sledeće usluge sistema:

- Interfejs za prikaz i pretragu proizvoda *IProductView*
- Interfejs za upravljanje stavkama u korpi *ICartManagement* (za dodavanje i uklanjanje proizvoda iz korpe)
- Interfejs za kreiranje i slanje porudžbine *IOrderProcessing*
- Interfejs za prijavu korisnika *ISecureAuth* (za prijavu i proveru identiteta korisnika)
- Interfejs za prikazivanje i izmenu podataka o korisniku *IUserAccount*.

Administratorski portal (*AdminPortal*) – koriste zaposleni za upravljanje sadržajem prodavnice. Putem ovog kanala moguće je ažurirati proizvode i pratiti porudžbine. Ova komponenta koristi:

- Interfejs za upravljanje proizvodima IProductAdmin
- Interfejs za administriranje porudžbina IOrderAdmin

Upravljanje korisnicima (*UserManagement*) - zaduženo za autentifikaciju i autorizaciju korisnika, kao i upravljanje osnovnim korisničkim podacima. Ova komponenta nudi:

- IUserAccount – pristup osnovnim podacima o korisniku
- ISecureAuth – autentifikacija i autorizacija korisnika

Katalog proizvoda (*Product*) - Odgovorna za čuvanje i prikaz informacija o proizvodima. Omogućava pretragu i administrativno upravljanje proizvodima.

- IProductInfo – prikaz detalja o proizvodima (naziv, opis, cena, stanje na lageru i sl.)
- IProductView – pregled i pretraga proizvoda (filtriranje, sortiranje i sl.)
- IProductAdmin – administracija proizvoda (dodavanje, izmena, brisanje proizvoda)

Šoping korpa (*ShoppingCart*) - Upravljanje privremenim podacima o proizvodima koje je korisnik dodao u šoping korpu tokom kupovine. Nudi *ICartManagement* za dodavanje, uklanjanje i izmenu stavki u korpi, a zahteva *IProductInfo* radi pribavljanja informacija o proizvodima.

Porudžbina (*Order*) - Upravlja procesom kreiranja porudžbine. Tokom ovog procesa proverava stanje korpe, obračunava ukupnu cenu i inicira plaćanje. Koristi:

- *ISecureAuth* – provera da li je korisnik prijavljen
- *IUserAccount* – pristup podacima radi fakturisanja ili isporuke porudžbine
- *ICartManagement* – pristup sadržaju šoping korpe
- *IProductInfo*
- *IPaymentProcessing* – iniciranje procesa plaćanja
- *INotification* – slanje informacije o statusu porudžbine.

Omogućava:

- IOrderProcessing – kreiranje i obradu porudžbina
- IOrderAdmin – administraciju porudžbina

Plaćanje - Obrada plaćanja i komunikacija sa eksternim platnim servisom. Nudi *IPaymentProcessing* za iniciranje i obradu plaćanja, a zahteva *IExternalPayment* za komunikaciju sa spoljnim platnim servisom.

Obaveštenja (*Notification*) - Slanje obaveštenja korisnicima (npr. email o potvrdi kupovine). Nudi *INotification* za slanje sistemskih obaveštenja, a zahteva:

- *IEmailService* za slanje email poruka putem eksternog servisa
- *IUserAccount* za pribavljanje kontakt podataka korisnika.

8.5. DIJAGRAM RASPOREĐIVANJA

Dijagram raspoređivanja (*Deployment Diagram*) predstavlja jedan od strukturnih dijagrama u UML u i koristi se za modelovanje fizičkog aspekta softverskog sistema. Njegova osnovna svrha jeste da prikaže kako su softverski artefakti raspoređeni na fizičke ili logičke resurse obrade, kao i kako su ti resursi međusobno povezani. Dijagram raspoređivanja daje statički pogled na okruženje implementacije i omogućava razumevanje distribuiranosti informacionog sistema.

U savremenim softverskim sistemima, naročito u kontekstu distribuiranih, cloud i mobilnih aplikacija, razdvajanje funkcionalnosti sistema na više fizičkih čvorova predstavlja standardnu praksu. Dijagram raspoređivanja omogućava projektantima i razvojnim timovima da jasno sagledaju gde se pojedini delovi sistema izvršavaju, koje tehnologije se koriste, kao i koje komunikacione veze postoje između čvorova.

Osnovni ciljevi dijagrama raspoređivanja su:

- objašnjavanje načina na koji je funkcionalnost sistema distribuirana na fizičke čvorove;
- vizuelizacija hardverske topologije sistema;
- opis izvršne arhitekture sistema;
- identifikacija komunikacionih puteva između čvorova;
- podrška donošenju arhitektonskih odluka u vezi sa performansama, pouzdanošću i skalabilnošću.

Dijagram raspoređivanja se često koristi u fazama projektovanja i implementacije, ali ima značajnu ulogu i u dokumentovanju sistema, kao i u komunikaciji između različitih zainteresovanih strana (arhitekata, developera, sistem administratora i menadžmenta).

Dijagram raspoređivanja ima značajnu ulogu u mapiranju softverskih komponenti na fizičku arhitekturu sistema. On omogućava analizu migracije komponenti tokom izvršavanja, zavisnosti između različitih delova sistema i potencijalnih uskih grla. U tom kontekstu mogu se koristiti i specijalni stereotipi, poput <<becomes>>, za označavanje runtime migracije komponenti između čvorova.

Objektni dijagram raspoređivanja (*Object level Deployment Diagram*) predstavlja instancu dijagrama raspoređivanja i modeluje konkretne primerke čvorova u sistemu. Ovaj tip dijagrama omogućava detaljan prikaz konfiguracije svakog čvora, uključujući njegove atribute, memorijske i procesorske zahteve, kao i softverske komponente koje se na njemu izvršavaju.

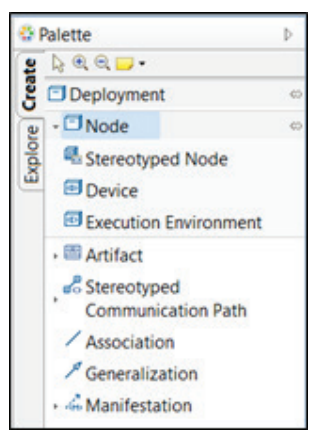
8.5.1. Osnovni elementi dijagrama raspoređivanja

Primenom dijagrama raspoređivanja (*Deployment Diagram*) moguće je jasno sagledati fizičku distribuciju softverskih artefakata, strukturu izvršnih okruženja i komunikacione puteve između čvorova. On pruža arhitektonsku osnovu za projektovanje, implementaciju i održavanje savremenih distribuiranih informacionih sistema.

Osnovni elementi dijagrama raspoređivanja su:

Čvor

Čvor (*Node*) predstavlja osnovni element koji označava računarski resurs na kome se izvršavaju softverski elementi sistema. Čvor može biti fizički uređaj ili logičko izvršno okruženje. **Fizički čvorovi** obuhvataju resurse poput servera, klijentskih računara, mobilnih uređaja ili mrežne opreme (rutera, svičeva). **Logički čvorovi** predstavljaju softverske slojeve unutar hardvera koji omogućavaju rad aplikacija (npr. virtuelne mašine ili baze podataka).



Slika 8.52. Vrste čvorova i pratećih elemenata u IBM RSA okruženju

Stereotipi čvorova omogućavaju preciznije opisivanje tipa čvora i njegove uloge. Standardni UML stereotipi koji se koriste u alatima za modelovanje su:

- **«device»** – Fizički hardverski uređaj koji poseduje procesorsku snagu i memoriju (npr. DB Server, Smartphone).
- **«execution environment»** – Softverski sistem unutar čvora koji obezbeđuje okruženje za izvršavanje specifičnih tipova artefakata (npr. Java Virtual Machine, Web Browser, Node.js Runtime).
- **Namenski stereotipi** – Često se koriste i specifični nazivi koji sugerišu ulogu čvora u sistemu, kao što su: «application server», «database system», «firewall» ili «client workstation».



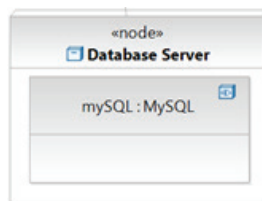
Slika 8.53. Primer čvora tipa «device» (Smartphone)

Čvorovi mogu biti ugnježdeni, čime se omogućava detaljan prikaz složenih konfiguracija. Na ovaj način se vizuelno prikazuje da se određeno **izvršno okruženje** nalazi unutar konkretnog **fizičkog uređaja**.

Izvršno okruženje

Izvršno okruženje (*Execution Environment*) je specijalan tip čvora, označen stereotipom <<execution environment>>, koji predstavlja softversku platformu ili okruženje unutar kog se izvršavaju određeni tipovi artefakata. Primeri izvršnih okruženja uključuju operativne sisteme, virtuelne mašine, aplikativne servere, DBMS sisteme i slične platforme.

Jedan fizički čvor može sadržati više izvršnih okruženja. Na primer, Java virtuelna mašina (JVM) predstavlja izvršno okruženje za Java aplikacije i može biti instalirana unutar operativnog sistema koji se izvršava na serverskom hardveru.



Slika 8.54. Primer izvršnog okruženja <<EE>>

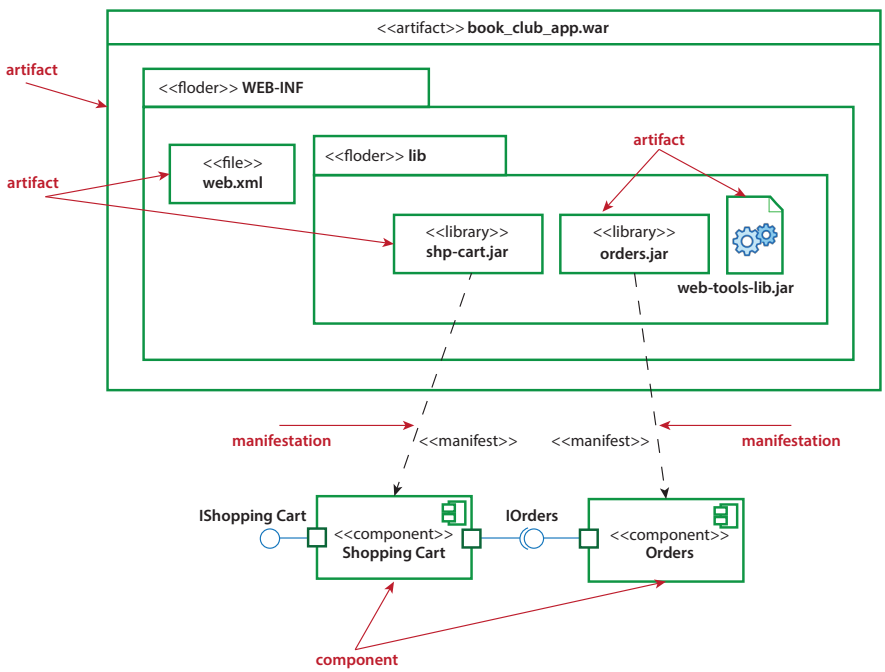
Slika 8.54 prikazuje primer **ugnježdavanja**, gde je unutar čvora koji predstavlja bazu podataka (Database Server) specifikovano izvršno okruženje (MySQL). Ovakav prikaz jasno razdvaja infrastrukturu od softverske platforme koja omogućava izvršavanje baze podataka.

Artefakt

Artefakti (*Artifact*) predstavljaju fizičke jedinice informacija koje nastaju kao rezultat procesa razvoja softverskog sistema. Oni su konkretni nosioci podataka ili koda koje sistem koristi ili proizvodi tokom svog rada. U UML-u se vizuelno predstavljaju pravougaonikom sa ključnom reči «**artifact**», često uz prateći simbol dokumenta u uglu. Tipični primeri artefakata su:

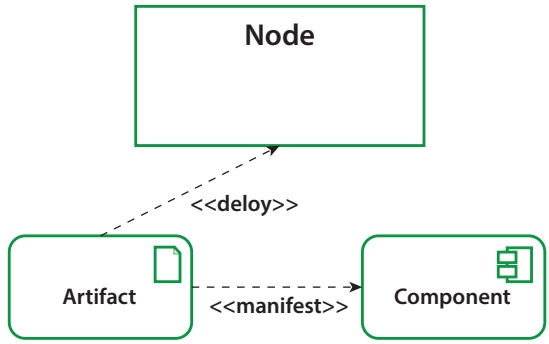
- Izvršni fajlovi i biblioteke (npr. .exe, .jar, .dll),
- Konfiguracioni fajlovi (npr. .xml, .properties),
- Arhive i skripte (npr. .war, .sql),
- Fizičke baze podataka i dokumentacija.

Artefakti se raspoređuju (*deploy*) na čvorove, čime se definiše gde se koji deo softvera fizički nalazi u trenutku izvršavanja sistema (Slika 8.55).



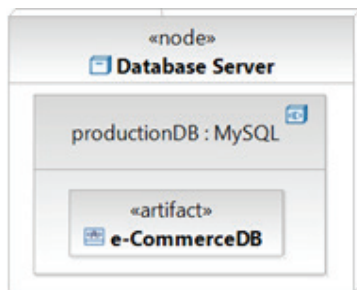
Slika 8.55. Primer artifakata u sistemu „BookClub“

U UML-u se artefakti modeluju kao autonomni elementi koji se sa čvorovima povezuju putem relacije raspoređivanja («deploy»). Važno je istaći da postoje dva ravnopravna načina vizuelizacije ove veze: eksplicitnim crtanjem zavisnosti između artefakta i čvora ili, što je češći slučaj u praksi, ugnježdavanjem (engl. nesting) artefakta unutar samog čvora. U ovom poglavlju koristiće se metod ugnježdavanja radi preglednijeg prikaza strukture sistema.



Slika 8.56. Primer zavisnosti između osnovnih elemenata dijagrama raspoređivanja

Na slici 8.55 prikazan je primer ugnježdavanja elemenata. Čvor *Database Server* služi kao domaćin izvršnom okruženju *productionDB : MySQL*. Ovo znači da je productionDB konkretan primerak (instanca) softvera koji radi na tom serveru. MySQL (desno od dvotačke) je klasa ili tip koja govori sistemu da je to izvršno okruženje tipa MySQL (a ne npr. Oracle ili PostgreSQL). Ovakav zapis je koristan u složenim sistemima gde na jednom serveru može biti više različitih instanci istog softvera (npr. testBaza : MySQL i produkcionaBaza : MySQL). Unutar tog okruženja razmešten je artefakt *e-CommerceDB*, koji predstavlja fizičku realizaciju same baze podataka (npr. skup fajlova na disku ili šemu) koju sistem koristi. Ovakva granulacija omogućava precizan uvid u to koji softverski entiteti zavise od kojih platformi i na kojoj se infrastrukturi oni nalaze.



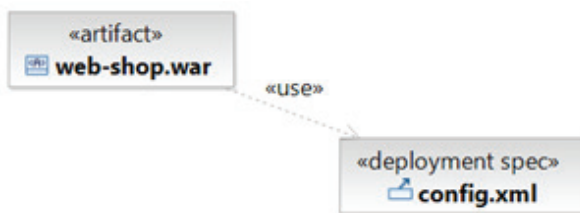
Slika 8.57. Primer ugnježdenih elemenata unutar čvora baze podataka *e-Commerce* sistema na dijagramu raspoređivanja

Specifikacija razmeštanja

Specifikacija razmeštanja (*Deployment Specification*) je poseban tip artefakta koji detaljno opisuje kako je primarni artefakt konfigurisan i uveden na čvor. Dok artefakt predstavlja samu komponentu (npr. aplikaciju), specifikacija određuje parametre njenog rada u konkretnom okruženju. Najčešće je u pitanju konfiguracioni fajl (npr. .xml, .properties ili .yaml) koji definiše:

- Mrežne parametre (npr. IP adrese, portovi),
- Bezbednosne opcije (npr. prava pristupa, sertifikati),
- Putanje i resurse (npr. lokacije log fajlova, konekcioni stringovi za bazu podataka),
- Performansne parametre (npr. maksimalni broj niti ili veličina memorijskog bafera).

UML dijagram prikazuje zavisnost artefakta od njegove specifikacije razmeštanja. Isprekidana linija sa stereotipom «**use**» pokazuje ka konfiguracionom fajlu (npr. config.xml), jer aplikacija od njega zavisi kako bi preuzela parametre rada prilikom podizanja na čvoru (Slika 8.58). U realnom prikazu, oba elementa se nalaze unutar istog čvora (servera).



Slika 8.58. Primer specifikacije razmeštanja artefakta *web-shop.war* na čvoru

U praksi, specifikacija razmeštanja omogućava **ponovnu upotrebu istog artefakta** (npr. identične verzije softvera) u različitim okruženjima. Zahvaljujući tome, ista aplikacija se može različito ponašati na testnom i produkcionom serveru bez izmene samog kôda. Na primer, aplikacija web prodavnice upakovana u **web-shop.war** fajl predstavlja primarni artefakt koji ne može funkcionisati samostalno. Njemu su neophodni specifični parametri za rad, kao što su mrežni portovi i adrese baza podataka. Kroz specifikaciju razmeštanja u YAML formatu, definišemo ove razlike za testno i produkciono okruženje:

1. **Verzija za TEST okruženje** - Ova specifikacija kaže aplikaciji da koristi testnu bazu i da bude "opuštenija" sa resursima.

```yaml
environment: "testing"
network:
  port: 8080
  host: "test.server.local"
database:
  connection_string: "jdbc:mysql://test-db:3306/test_db"
  max_connections: 5
security:
  ssl_enabled: false
logging:
  path: "/var/logs/test_app.log"
  level: "DEBUG"
```

2. **Verzija za PRODUKCIONO okruženje** - Isti softver, ali specifikacija sada nalaže strogu bezbednost i visoke performanse.

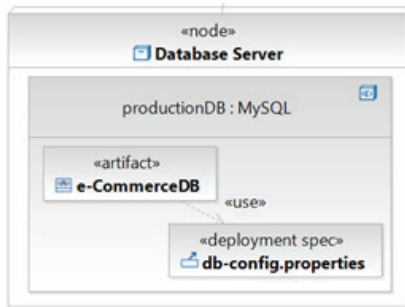
```yaml
environment: "production"
network:
  port: 443
  host: "www.prodavnica.rs"
database:
  connection_string: "jdbc:mysql://prod-db-cluster:3306/real_db"
  max_connections: 100
security:
  ssl_enabled: true
  certificate_path: "/etc/ssl/certs/prod_cert.pem"
logging:
  path: "/var/logs/prod_app.log"
  level: "ERROR"
```

Ključne razlike iz primera:

- Mrežni parametri: U testu je port 8080, u produkciji 443 (HTTPS).
- Resursi: Test koristi test-db, produkcija koristi prod-db-cluster.
- Performanse: Broj konekcija sa bazom je ograničen na 5 u testu, dok je u produkciji 100.
- Bezbednost: U produkciji je obavezan SSL sertifikat, dok u testu nije.

Zahvaljujući specifikaciji razmeštanja, programeri ne moraju da menjaju kod aplikacije svaki put kada je sele sa jednog servera na drugi; oni samo promene ovaj konfiguracioni fajl.

Dijagram na slici 8.57 ilustruje fizičku arhitekturu gde čvor («node» Database Server) udomljuje specifično izvršno okruženje (MySQL). Unutar njega, primarni artefakt (e-CommerceDB) koristi relaciju zavisnosti «use» ka specifikaciji razmeštanja (db-config.properties). Ova specifikacija, u formi .properties fajla (format ključ-vrednost), definiše konkretne parametre rada baze podataka — poput mrežnih portova i konekcionih stringova — omogućavajući da se isti artefakt prilagodi specifičnim zahtevima produkcionog okruženja bez izmene samog koda.



Slika 8.59. Prikaz specifikacije razmeštanja artefakta unutar izvršnog okruženja na čvoru

Primer *db-config.properties* fajla koji koristi jednostavnu strukturu ključ=vrednost:

Properties

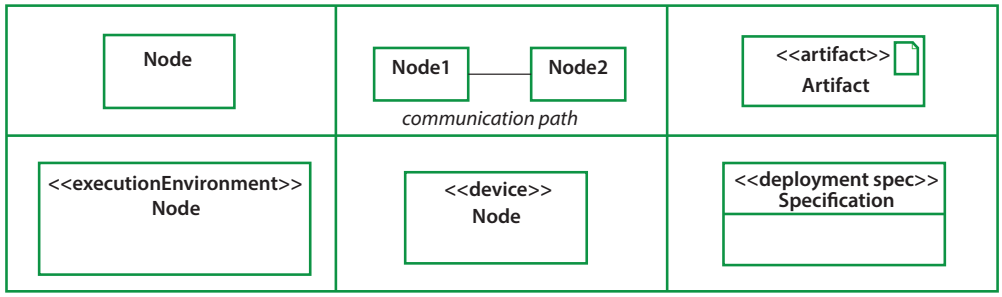
```
# Mrežni parametri
db.port=3306
db.host=192.168.1.50

# Bezbednosne opcije
db.user=prod_admin
db.ssl.enabled=true

# Putanje i resursi
db.storage.path=/var/lib/mysql/data
db.log.file=/var/log/mysql/error.log

# Performansni parametri
db.max_connections=150
db.buffer.size=512MB
```

Osnovni gradivni elementi dijagrama raspoređivanja su prikazani na sledećoj slici.



Slika 8.60. Notacija osnovnih gradivnih elemenata dijagrama raspoređivanja

8.5.2. Konekcije i relacije na dijagramu raspoređivanja

Da bi dijagram raspoređivanja pružio potpunu sliku arhitekture sistema, neophodno je precizno definisati način na koji su njegovi elementi povezani. Dok čvorovi i artefakti predstavljaju statične delove infrastrukture, **konekcije i relacije** unose dinamiku i kontekst, opisujući kako se softver fizički instalira, na koji način komponente komuniciraju i kako se logički modeli prevode u izvršne fajlove. U nastavku su detaljno opisani ključni mehanizmi za modelovanje ovih veza: komunikacione putanje, kao i relacije raspoređivanja i manifestacije.

Komunikaciona putanja (*Communication Path*) - Predstavlja asocijaciju između dva čvora koja definiše mehanizam razmene poruka i signala. One mogu biti opisane fizičkim medijumom (npr. Ethernet, optički kabl) ili komunikacionim protokolom (npr. TCP/IP, HTTP, SOAP), čime se omogućava analiza propusne moći, latencije i pouzdanosti mreže.

Relacija raspoređivanja (<<deploy>>) - Ukazuje na to da je određeni artefakt raspoređen na konkretnom čvoru. U dijagramu se ovo može prikazati na dva načina i to ugnježđivanjem artefakta unutar čvora ili eksplicitnom relacijom <<deploy>> koja povezuje artefakt i čvor.

Relacija manifestacije (<<manifest>>) – Označava da artefakt implementira (manifestuje) jednu ili više logičkih komponenti. Ova relacija je ključna za održavanje **sledljivosti** (*traceability*), jer objašnjava kako se apstraktni elementi dizajna (komponente) transformišu u fizičke entitete (artefakte) spremne za izvršavanje.

8.5.3. Koraci u modelovanju distribuiranosti sistema

Modelovanje distribuiranosti sistema pomoću dijagrama raspoređivanja predstavlja proces prevođenja logičke arhitekture u fizičku implementaciju. Ovaj proces obično obuhvata sledeće ključne korake:

- **Sagledavanje fizičkog okruženja mreže**: Analiza geografskih lokacija i mrežne topologije kako bi se razumeli fizički okviri sistema.
- **Identifikacija čvorova na mreži**: Definisanje uređaja («device») i njihovih hardverskih konfiguracija.
- **Definisanje izvršnih okruženja**: Preciziranje tehnologija, alata i platformi («executionEnvironment») koje omogućavaju rad softvera.
- **Raspoređivanje elemenata** (Deployment): Proces alokacije softverskih procesa i artifakata na čvorove koristeći relaciju «**deploy**», kao i definisanje njihove veze sa logičkim komponentama putem relacije «manifest».
- **Definisanje komunikacionih puteva**: Uspostavljanje kanala komunikacije (Communication Path) i izbor odgovarajućih protokola (npr. HTTP, TCP/IP).

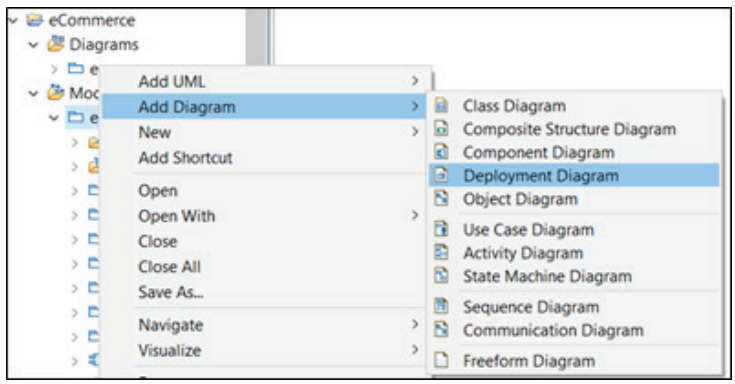
Prilikom analize fizičkog okruženja, od suštinske je važnosti uzeti u obzir faktore **kvaliteta usluge** (QoS), kao što su širina mrežnog opsega, postojanje redundantnih puteva za visoku dostupnost, kao i kritične hardverske karakteristike čvorova poput procesorske snage, memorije i kapaciteta skladištenja.

Vežba 8.7. Kreiranje dijagrama raspoređivanja *e-commerce* aplikacije u IBM RSA alatu

Sistem e-commercei se sastoji od:

- Klijentskog sloja: PC i mobilni uređaji koji koriste veb pregledače.
- Serverskog sloja: Apache veb server (kao ulazna tačka) i Node.js aplikativni server (poslovna logika).
- Sloja podataka: Lokalna MySQL baza podataka i eksterni AWS S3 cloud servis za multimediju.

Korak 1. Kreiranje dijagrama raspoređivanja



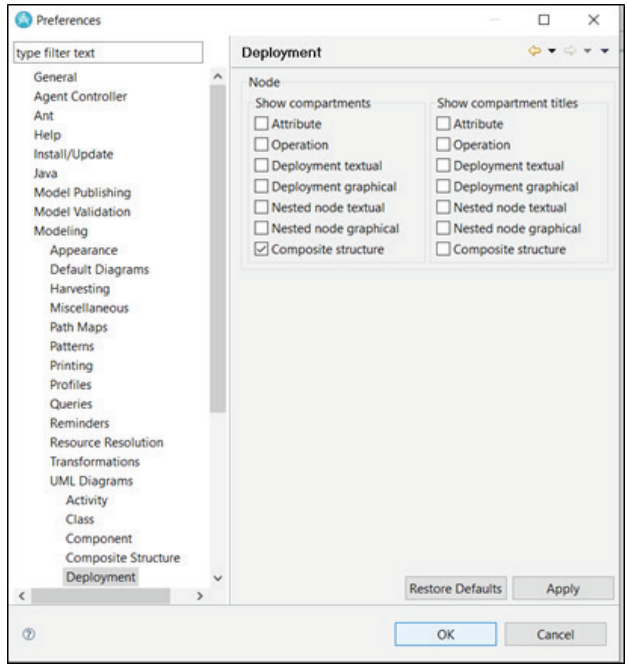
Slika 8.61. Kreiranje *Deployment* dijagrama

Korak 2. Konfiguracija prikaza i vidljivosti čvora

Kako bi dijagram ostao pregledan i fokusiran na suštinu razmeštanja, potrebno je prilagoditi vidljivost elemenata unutar čvora. Za potrebe ove vežbe, primarni fokus je na kompozitnoj strukturi (Composite Structure), koja omogućava vizuelni prikaz (gnježđenje) izvršnog okruženja, komponenti i pripadajućih artifakata unutar samog čvora. Da biste aktivirali ovaj prikaz u alatu, pratite sledeću putanju:

- Idite na Window -> Preferences.
- Proširite stavku Modeling, a zatim izaberite UML Diagram.
- Selektujte tip dijagrama *Deployment*.
- Sa desne strane, u okviru panela *Show Compartments*, označite isključivo polje *Composite Structure*.

Nakon potvrde na dugme *Apply*, svaki novi čvor koji prevučete na radnu površinu automatski će imati otvoren prostor za prikaz unutrašnjih elemenata, što značajno olakšava dalje modelovanje hijerarhije sistema.



Slika 8.62. Podešavanje vidljivosti delova čvorova

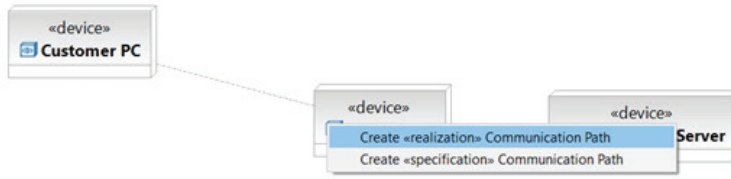
Korak 3. Modelovanje mrežne topologije

U ovoj fazi se definiše fizička infrastruktura sistema mapiranjem hardverskih uređaja i njihovih međusobnih veza. Fokus je na postavljanju čvorova koji predstavljaju klijentske uređaje (poput računara i mobilnih telefona) i serverske resurse (web, aplikacioni i database serveri). Za svaki čvor se definiše odgovarajući stereotip:

- «device»: Koristi se za precizno označavanje fizičkog hardvera (npr. serverske mašine).
- «node»: Može se koristiti kao opštiji termin za bilo koji procesorski resurs u mreži.

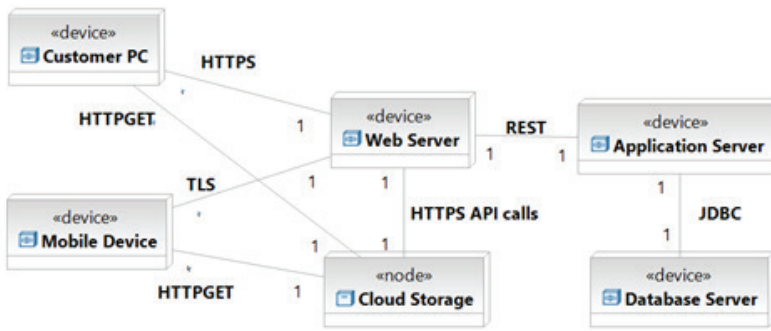
Povezivanje identifikovanih čvorova vrši se putem komunikacionih putanja (Communication Paths), koje predstavljaju fizičke ili logičke asocijacije između resursa. Na ovim vezama se eksplicitno navode protokoli (npr. HTTPS, REST, JDBC) koji omogućavaju bezbednu i strukturiranu razmenu podataka, čime se dobija jasan uvid u mrežnu arhitekturu i hijerarhiju sistema. Prilikom povezivanja dva uređaja («device») u alatu IBM RSA, koristi se opcija Stereotyped Communication Path. Alat tada nudi dve mogućnosti:

- Create «realization» Communication Path: Ovu opciju je potrebno odabrati jer ona označava konkretnu implementaciju veze u realnom sistemu.
- Create «specification» Communication Path: Ova opcija se ređe koristi u ovoj fazi jer služi za definisanje opštih pravila veze, a ne njene direktne primene. U UML-u, *Specialization* (specijalizacija/generalizacija) se koristi kada jedan element nasleđuje osobine drugog (npr. "Laptop" je specijalizacija "Računara"), što ovde nije slučaj jer PC i Server ne nasleđuju jedan drugog, već komuniciraju.



Slika 8.63. Kreiranje <<*realization*>> *Communication Path*

Nakon kreiranja realizacije, vrši se imenovanje tipa komunikacije (unos naziva protokola) i podešavanje multiplikativnosti (npr. 1, * ili 2), čime se definiše broj instanci čvorova koji učestvuju u interakciji.



Slika 8.64. Arhitektura mrežne topologije *e-commerce* aplikacije na nivou čvorova

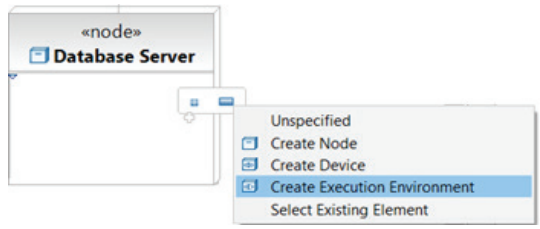
Dijagram na slici 8.64 prikazuje kompletnu mrežnu topologiju *e-commerce* sistema. Na dijagramu se jasno uočava podela na klijentski sloj (uređaji korisnika), serverski sloj (infrastruktura aplikacije) i eksterni sloj za skladištenje podataka u oblaku. Upotreba specifičnih protokola i definisana multiplikativnost između čvorova omogućavaju precizno planiranje mrežnih resursa i bezbednosnih polisa sistema.

Korak 4. Definisanje izvršnih okruženja unutar čvorova

Nakon definisanja fizičke topologije, potrebno je specifikovati softverske platforme koje omogućavaju rad aplikacije. Izvršno okruženje («executionEnvironment») predstavlja softverski čvor koji nudi okruženje za izvršavanje specifičnih tipova artifakata (npr. operativni sistem, J2EE kontejner ili baza podataka).

U alatu IBM RSA, proces dodavanja izvršnog okruženja unutar postojećeg čvora (npr. Database Server) vrši se na sledeći način:

- Potrebno je osigurati da je proširen strukturni deo čvora.
- Prelaskom miša preko unutrašnjosti čvora pojavljuju se ikonice za brzi unos. Potrebno je odabrati opciju *Parts* (delovi).
- Iz padajućeg menija koji se pojavi, odaberite stavku *Create Execution Environment*.



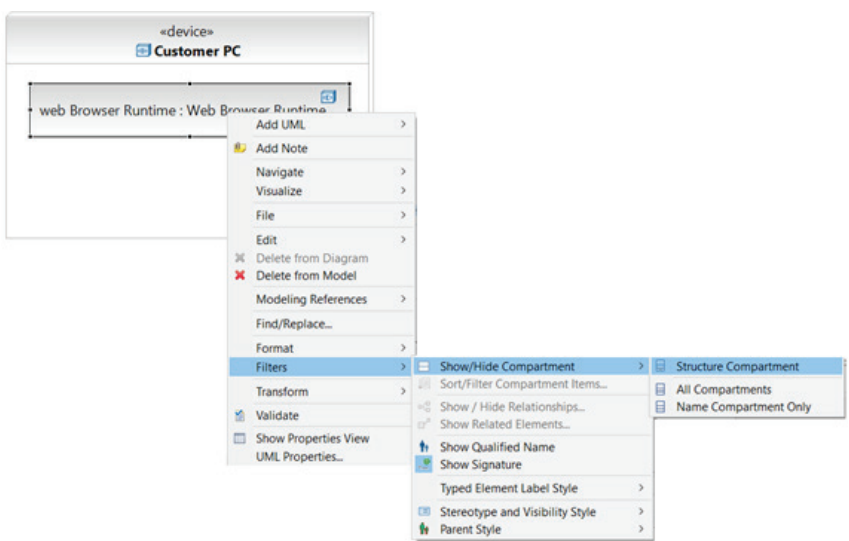
Slika 8.65. Dodavanje izvršnih okruženja

Nakon ovog koraka, unutar čvora će se pojaviti novi pravougaonik sa stereotipom «executionEnvironment», koji potom možete imenovati kao u ovom primeru *Web Browser Runtime*.

Nakon što kreirate izvršno okruženje ono je inicijalno prikazano kao prazan element. Da biste unutar njega mogli da prikažete konkretne artefakte koji se u njemu izvršavaju, potrebno je omogućiti vidljivost njegovog unutrašnjeg sadržaja. Kao što je prikazano na slici, postupak u alatu je sledeći:

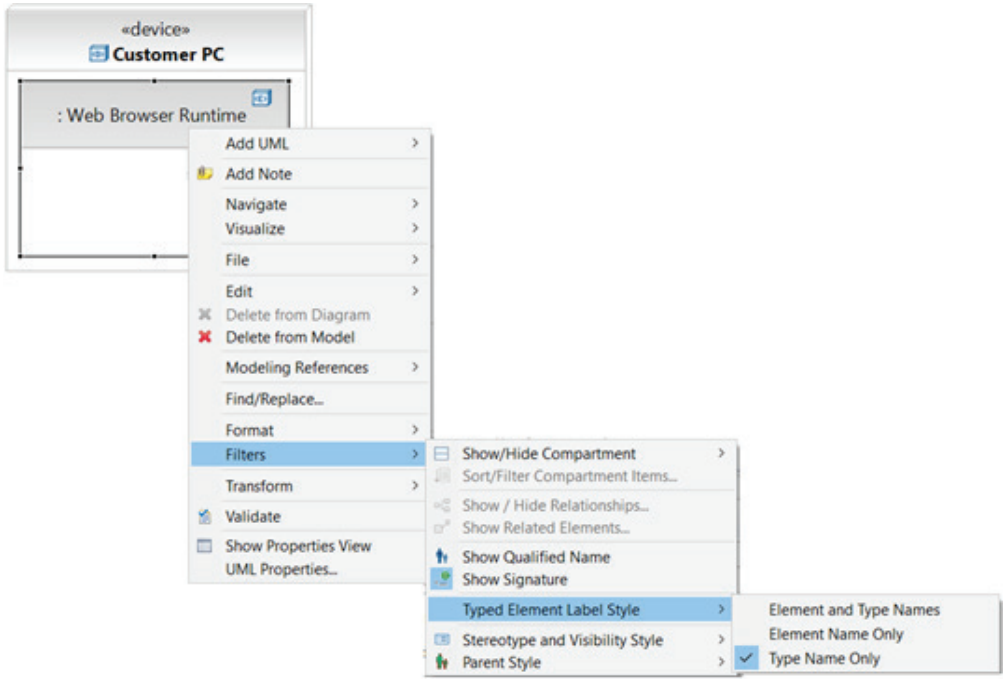
- Desni klik na izvršno okruženje.
- Odaberite opciju *Filters* iz kontekstnog menija.
- U podmeniju *Show/Hide Compartment*, selektujte stavku *Structure Compartment*.

Aktivacijom ove opcije, izvršno okruženje se vizuelno proširuje, stvarajući prostor u koji možete prevući ili kreirati artefakte (npr. .js fajlove, klijentske skripte ili .html stranice). Ovo omogućava precizno modelovanje arhitekture gde se jasno vidi koji softverski entitet "živi" u kom radnom okruženju.



Slika 8.66. Podešavanje vidljivosti strukturnog odeljka u okviru izvršnog okruženja

U IBM RSA alatu, prikazaće vam se dugačak naziv (npr. web Browser Runtime : Web Browser Runtime) jer alat podrazumevano prikazuje **ime instance** i **ime tipa** (klase) razdvojene dvotačkom. Da biste to skratili i ostavili samo naziv okruženja, pratite korake prikazane na sledećoj slici:

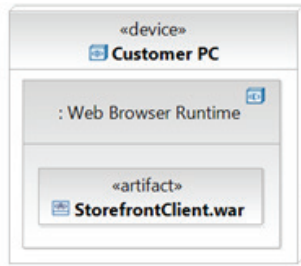


Slika 8.67. Podešavanje prikaza naziva elementa

Korak 5. Raspoređivanje softverskih artefakata na čvorove sistema

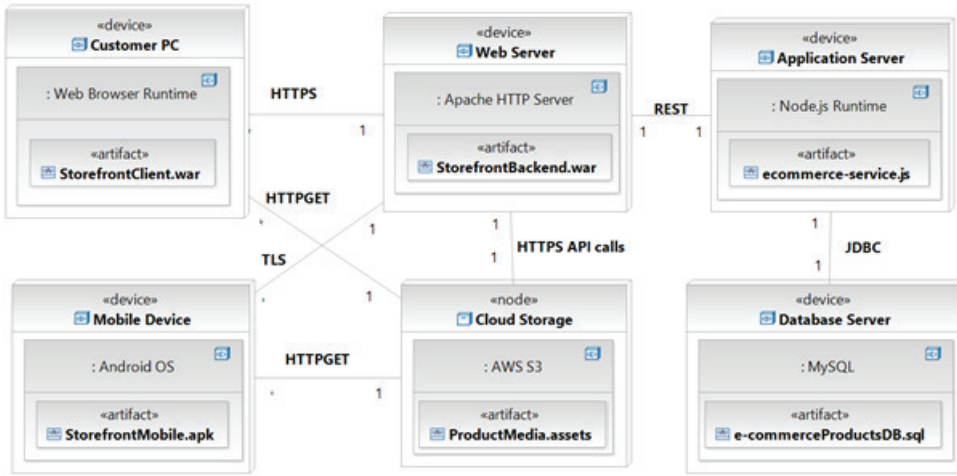
U ovom koraku vrši se povezivanje softverskih komponenti sa hardverskom infrastrukturom. Artefakti se dodaju tako što se sa palete alata selektuje opcija **Artifact**, definiše se njegov naziv (npr. sa ekstenzijom .war ili .jar) i prevlači se unutar odgovarajućeg **izvršnog okruženja** na čvoru.

Ovim postupkom se vizuelno i logički potvrđuje da će se određeni deo aplikacije (npr. klijentski interfejs) izvršavati u specifičnom okruženju (npr. web pregledaču) na ciljanom uređaju.



Slika 8.68. Dodavanje *StorefrontClient.war* dela aplikacije u okviru izvršnog okruženja

Model na slici, prikazuje distribuciju softverskih artefakata kroz složenu mrežnu infrastrukturu. Sistem je projektovan tako da klijentski uređaji (PC i mobilni) pristupaju Apache web serveru putem HTTPS-a, koji dalje komunicira sa Node.js aplikativnim servisom. Podaci su perzistirani u MySQL bazi podataka, dok se statički multimedijalni resursi skladište na AWS S3 cloud platformi. Ovakva arhitektura osigurava skalabilnost, sigurnost i jasno razdvajanje odgovornosti između slojeva sistema.



Slika 8.69. Raspoređivanje softverskih artefakata na čvorove sistema

Korak 6. Konfiguracija parametara putem specifikacije razmeštanja (Opciono)

Iako je ovaj korak opcioni, on je ključan za sisteme koji zahtevaju specifična podešavanja prilikom instalacije na različita okruženja. Putem specifikacije razmeštanja (Deployment Specification), unutar čvora se definišu parametri poput *db-config.properties*, koji sadrže podatke o konekciji sa bazom, portovima ili sigurnosnim ključevima.

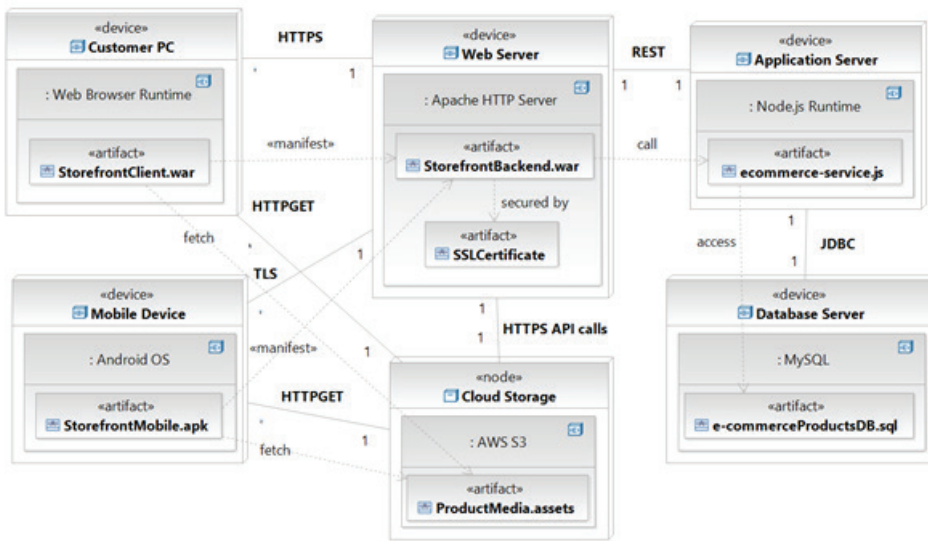
Korak 7. Povezivanje artefakata i drugih komponenti

Nakon što su svi artefakti raspoređeni unutar svojih izvršnih okruženja, poslednji korak u izradi dijagrama je definisanje zavisnosti i interakcija između samih softverskih komponenti. U ovom koraku se vrši sledeće:

- Logičko povezivanje: Artefakti se povezuju sa drugim komponentama sistema ili eksternim servisima (poput Cloud Storage-a) kako bi se prikazao protok podataka na aplikativnom nivou.
- Definisanje zavisnosti: Koriste se relacije zavisnosti (*Dependency*) ili asocijacije kako bi se naglasilo da jedan artefakt (npr. klijentska aplikacija) zahteva resurse ili usluge drugog čvora.
- Validacija komunikacije: Proverava se da li su protokoli definisani u ranijim koracima (HTTPS, REST, JDBC) usklađeni sa artefaktima koji ih koriste za razmenu informacija.

Ovim korakom dijagram prestaje da bude samo prikaz hardvera i postaje potpuna mapa funkcionisanja e-commerce sistema, gde se vidi tačna putanja od korisničkog klika u browseru do upisa u bazu podataka.

Finalna verzija dijagrama razmeštanja integriše fizičku infrastrukturu sa logičkim zavisnostima softverskih artefakata. Upotrebom stereotipa poput «**manifest**», **call** i **fetch**, jasno je definisan put podataka od korisničkog interfejsa, preko sigurnosnih sertifikata i aplikativne logike, pa sve do perzistentnih podataka u bazi i cloud resursima.



Slika 8.70. Finalni dijagram razmeštanja e-commerce sistema sa definisanim zavisnostima između artefakata

Vežba 8.8. Kreirati dijagram raspoređivanja za mobilnu aplikaciju za upoznavanje „MeetU“

Nacrtati četvoroslojnu arhitekturu i povezati sledeće komponente:

1. **Klijentski čvor** (Mobile Device)
   - EE (Execution Environment): Android operativni sistem koji pokreće aplikaciju.
   - Artefakt: *MeetU_Mobile_Client.apk* – izvršna verzija aplikacije.
2. **Sloj prezentacije** (Web Server)
   - EE: Nginx kao web server i reverse proxy.
   - Artefakt: *MeetU_Web_Frontend* – statički resursi (HTML/JS/CSS).
3. **Sloj poslovne logike** (Application Server)
   - EE: Apache Tomcat
   - Artefakt: *MeetU_Backend.war* – arhivirani fajl sa Java servisima koji obrađuju zahteve korisnika.
4. **Sloj podataka** (Database Server)
   - EE: MS SQL Server.
   - Artefakt: *MeetU_Schema* – relacioni model podataka gde se čuvaju profili korisnika i njihove interakcije.

Postaviti sledeće komunikacione puteve:

Veza između čvorova	Protokol	Multiplicitet
Mobile Device --- Web Server	HTTPS	* (kod mob.): 1 (kod web)
Web Server --- Application Server	AJP	1: 1
Application Server --- Database Server	JDBC	1: 1

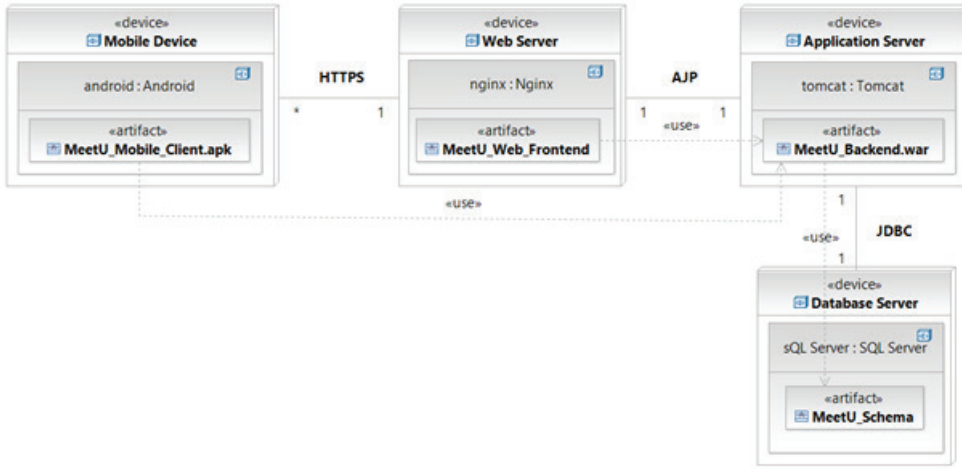
Rešenje vežbe

Sistem „MeetU“ je dizajniran kao moderna četvoroslojna arhitektura koja razdvaja klijentski interfejs, sloj prezentacije, poslovnu logiku i skladištenje podataka. Ovakav pristup omogućava visoku bezbednost i lakše održavanje sistema.

Ključne karakteristike prikazanog rešenja:

- Bezbedna komunikacija: Klijenti pristupaju sistemu isključivo putem HTTPS protokola koji završava na Nginx serveru.
- Separacija klijentskih strana: Sistem istovremeno podržava mobilne korisnike putem Android aplikacije (.apk) i veb korisnike putem statičkog frontenda.
- Optimizovan *Backend*: Veza između veb servera i aplikativnog servera (Tomcat) ostvarena je preko AJP protokola radi boljih performansi.
- Centralizovana baza: Podaci su izolovani na namenskom SQL serveru kojem se pristupa isključivo putem JDBC drajvera iz sloja poslovne logike.

Finalni dijagram raspoređivanja:



Slika 8.71. Dijagram raspoređivanja sistema „MeetU"

Zadatak 8.3. Kreirati dijagram raspoređivanja za Cloud-native arhitekturu sistema „QuickBite"

Kreirati **dijagram raspoređivanja** (*Deployment Diagram*) koji prikazuje fizičku i logičku arhitekturu platforme za dostavu hrane. Fokus je na primeni kontejnerizacije i cloud servisa za distribuciju sadržaja. Platforma „QuickBite" omogućava krajnjim korisnicima naručivanje hrane putem mobilnih uređaja, dok vlasnici restorana upravljaju svojim menijima i narudžbinama putem veb-baziranog administrativnog portala.

Tehničke specifikacije po slojevima:

1. **Klijentski sloj** (Customer Device):
   - **EE**: Android operativni sistem.
   - **Artefakt**: QuickBite_User.apk (mobilna aplikacija).
2. **Sloj prezentacije** (AWS Cloud): Umesto klasičnog servera, koristiti kombinaciju dva cloud servisa:
   - **CloudFront** (CDN): Služi kao ulazna tačka za sav HTTPS saobraćaj i kešira statički sadržaj.
   - **Amazon S3** (Object Storage): Čuva artefakt QuickBite_Restaurant_Portal (skup statičkih HTML/JS resursa).
3. **Sloj poslovne logike** (Application Server):

- **EE**: Docker Engine unutar kojeg se pokreće Node.js Runtime okruženje.
- **Artefakt**: QuickBite_API_Service.js (JavaScript fajl koji sadrži poslovnu logiku i API rute).

4. **Sloj podataka** (Database Server):
   - **EE**: MongoDB (NoSQL baza podataka).
   - **Artefakt**: QuickBite_Collections (dokumentacioni model podataka).

Zahtevi za komunikacione puteve i zavisnosti:

- **Protokoli**:
  - Između klijenta i CloudFront-a koristiti **HTTPS** (multiplicitet * : 1).
  - Između sloja prezentacije i aplikativnog servera koristiti **HTTP/2** protokol (multiplicitet 1 : 1).
  - Komunikaciju sa bazom podataka definisati putem **Wire Protocol** standarda (ili JDBC).
- **Zavisnosti**:
  - Obavezno ucrtati logičke zavisnosti («use») koje pokazuju da klijentski artefakti pozivaju API servis, a API servis pristupa bazi podataka.

POGLAVLJE 9



SERVISNO-ORIJENTISANO MODELOVANJE SOFTVERA

Servisno-orijentisano modelovanje softvera

9.1. Uvod u servisno-orijentisane koncepte i tehnologije
9.2. Servisno-orijentisano modelovanje softvera
9.3. Modelovanje servisno-orijentisane integracije softvera
9.4. Mikroservisna arhitektura softvera

Cilj poglavlja:

Cilj ovog poglavlja je prelazak sa tradicionalnih monolitnih struktura na savremene arhitektonske paradigme koje omogućavaju razvoj kompleksnih, visoko dostupnih sistema. Fokus je na savladavanju principa servisno-orijentisane arhitekture (SOA) i mikroservisnog pristupa, gde se softverski sistem dekomponuje na niz nezavisnih, labavo povezanih komponenti (servisa). Kroz ovo poglavlje, čitalac će naučiti kako da projektuje softver koji je otporan na greške, lako skalabilan i pogodan za kontinuiranu integraciju i isporuku (CI/CD), čime se postiže fleksibilnost neophodna za funkcionisanje sistema na nivou velikih digitalnih platformi.

9. SERVISNO-ORIJENTISANO MODELOVANJE SOFTVERA

9.1. UVOD U SERVISNO-ORIJENTISANE KONCEPTE I TEHNOLOGIJE

Savremeni razvoj softvera obeležen je prelaskom sa rigidnih, izolovanih sistema ka otvorenim, distribuiranim arhitekturama. Razumevanje ovog prelaska ključno je za projektovanje sistema koji mogu da odgovore na dinamične zahteve modernog poslovanja.

9.1.1. Evolucija od monolitne do servisno orijentisane arhitekture

Tradicionalni razvoj softvera decenijama se oslanjao na **monolitnu arhitekturu**. U ovom modelu, aplikacija se gradi kao jedinstvena, nerazdvojna logička celina. Tipična troslojna monolitna aplikacija obuhvata korisnički interfejs (UI), serversku logiku i bazu podataka, gde je serverski deo jedan izvršni fajl koji obrađuje sve zahteve. Prelazak sa monolitne na mikroservisnu arhitekturu nije samo tehnička promena, već evolutivni proces dekompozicije sistema (Newman, 2019).

Iako je monolitna arhitektura prirodna za manje projekte, ona ispoljava značajne nedostatke u velikim sistemima:

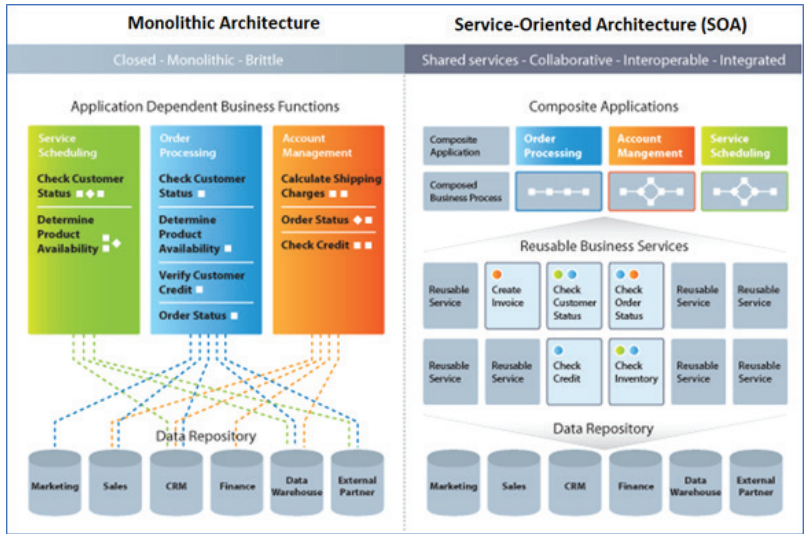
- **Otežano skaliranje**: Nije moguće skalirati samo kritični deo aplikacije; mora se replicirati ceo monolit, što troši resurse.
- **Niska agilnost**: Svaka izmena, čak i u malom segmentu koda, zahteva ponovno prevođenje (re-build) i postavljanje (deployment) celokupnog sistema.
- **Tehnološka zavisnost**: Ceo sistem je obično vezan za jedan tehnološki stek.

Kao odgovor na ove izazove nastala je **servisno-orijentisana arhitektura** (SOA), koja softverske komponente izdvaja u nezavisne, autonomne jedinice – servise. Za razliku od monolitnog pristupa, servisno-orijentisani (SO) pristup obuhvata tehnologije koje ističu:

- labavo povezane softverske komponente
- ponovnu upotrebljivost
- otvorene standarde
- kompozitnu implementaciju
- brzu primenu na tržištu i dr.

SOA je arhitekturalni stil koji omogućava da se poslovne funkcionalnosti predstave kao skup labavo povezanih (loose coupling) servisa, u cilju dostizanja maksimalne poslovne fleksibilnosti na interoperabilan i tehnološki nezavisan način.

Na slici 9.1 se prikazuje poređenje između **monolitne** i **servisno-orijentisane arhitekture** (SOA) kroz tri nivoa: aplikativne funkcije, način njihove organizacije i pristup podacima.



Slika 9.1. Poređenje monolitne i SOA arhitekture

Na slici 9.1 su slikovito opisane ključne razlike između ova dva stila arhitekture i to:

- **Monolitna arhitektura**:
  - **Struktura**: Aplikativne funkcije (zakazivanje servisa, obrada porudžbina, upravljanje nalozima) su **zatvorene i krute** (Closed - Brittle). One čine jednu neodvojivu celinu.
  - **Zavisnost**: Funkcije su direktno zavisne jedna od druge unutar samog koda aplikacije.
  - **Pristup podacima**: Postoji direktna, isprepletena veza između svake funkcije i različitih repozitorijuma podataka (Marketing, Sales, CRM...). Svaka promena u bazi podataka zahteva izmene u samom jezgru aplikacije.
- **Servisno-orijentisana arhitektura:**
  - **Struktura**: Arhitektura se zasniva na deljenim, kolaborativnim i integrisanim servisima (Shared - Collaborative - Integrated).
  - **Kompozitne aplikacije**: Umesto jednog bloka, aplikacije su „sastavljene" od različitih poslovnih procesa koji koriste zajedničke resurse.
  - **Ponovna upotrebljivost**: Uvode se ponovno upotrebljivi **poslovni servisi** (npr. Check Credit, Create Invoice). Jedan isti servis može koristiti više različitih aplikacija, što smanjuje dupliranje koda.
  - **Pristup podacima**: Servisi služe kao posrednik (apstrakcija) između aplikacije i podataka. Aplikacija ne pristupa bazama direktno, već to čini preko standardizovanih servisa, što povećava fleksibilnost i olakšava održavanje sistema.

Dakle, dok je monolitni pristup težak za menjanje i skaliranje zbog isprepletenih veza, SOA omogućava agilnost jer se servisi mogu nezavisno razvijati, menjati i kombinovati u nove poslovne procese.

9.1.2. Uloge u servisno-orijentisanom modelu

Da bi sistem zasnovan na servisima funkcionisao, neophodna je koordinacija između tri strane: Provajdera, Registra i Korisnika. Svaka uloga ima specifične odgovornosti u životnom ciklusu servisa. Sa perspektive arhitekture, SOA se zasniva na interakciji ova tri ključna aktera:

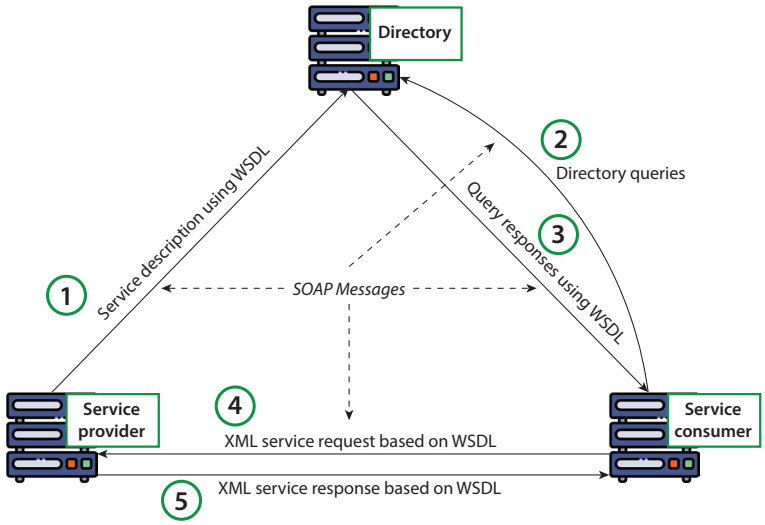
1. **Provajder servisa** (Service Provider) – Provajder je mrežni entitet koji poseduje poslovnu logiku, implementira servis i čini ga dostupnim na određenoj mrežnoj adresi (endpoint). Njegova odgovornost je da prihvati dolazne zahteve, obradi ih i vrati odgovor korisniku, uz prethodno objavljivanje (publikovanje) informacija o servisu u registru. Na primer, u Cloud okruženju provajderi su specifični servisi ili funkcije poput **AWS Lambda** (Inventory) servisa koji proverava dostupnost proizvoda na stanju.
2. **Korisnik servisa** (Service Consumer) – Korisnik je bilo koja aplikacija, softverski modul ili klijentski uređaj koji ima potrebu za određenom funkcionalnošću. On inicira proces tako što pronalazi servis u registru, povezuje se na adresu provajdera i poziva njegove operacije. Na primer, to može biti mobilna aplikacija koja slanjem zahteva „Kupac je kliknuo na KUPI“ poziva eksterni **Stripe API** servis radi obrade naplate.
3. **Registar servisa** (Service Registry) – Predstavlja centralno skladište koje omogućava otkrivanje i pronalaženje servisa. Kada je SOA koncept nastajao, vizija je bila postojanje globalnog „telefonskog imenika“ pod nazivom **UDDI** (Universal Description, Discovery, and Integration), zamišljenog kao XML registar gde bi kompanije širom sveta objavljivale svoje servise. Iako se u praksi globalni UDDI pokazao kao previše složen, princip je opstao kroz mehanizme **Service Discovery** (pronalaženje servisa). Primer modernog registra je **AWS Service Catalog**, koji nudi spisak svih dostupnih servisa, njihovih interfejsa i mrežnih adresa (ARN) neophodnih za integraciju.

Tabela 9.1. Uporedni prikaz uloga u SOA

Karakteristika	Provajder servisa	Registar servisa	Korisnik servisa
Opis	Entitet koji nudi i izvršava uslugu.	Centralni posrednik (imenik) servisa.	Aplikacija koja zahteva uslugu.
Glavna aktivnost	**Publikovanje**: Šalje informacije o sebi registru.	**Skladištenje**: Čuva podatke o dostupnim servisima.	Pronalaženje: Traži lokaciju servisa u registru.
Faza u procesu	Kreira i postavlja (deploy) servis.	Katalogizuje i prati status servisa.	Povezuje se (bind) i koristi servis.
Primer (Interni)	*Inventory Service* (Skladište)	**Netflix Eureka**.	*Order Service* (Narudžbine)
Primer (Eksterni)	Banka (API za plaćanje).	**Swagger Hub** / Developer Portal.	Web prodavnica (npr. Amazon).

Na slici 9.2 se prikazuje dinamički ciklus saradnje između tri ključna aktera, koji se odvija u pet osnovnih koraka:

1. ***Publish*** (Objavljivanje): Provajder šalje opis servisa (najčešće u WSDL formatu) u registar kako bi ga učinio vidljivim.
2. ***Find*** (Pronalaženje): Korisnik šalje upit registru (*Directory*) tražeći specifičnu funkcionalnost.
3. ***Response***: Registar vraća korisniku informaciju o tome gde se servis nalazi i kako da ga pozove.
4. ***Bind / Request***: Korisnik, koristeći dobijene instrukcije, šalje direktan zahtev provajderu (npr. putem SOAP poruke).
5. ***Response***: Provajder izvršava uslugu i vraća rezultat korisniku.



Slika 9.2. Konceptualni model interakcije aktera u SOA arhitekturi

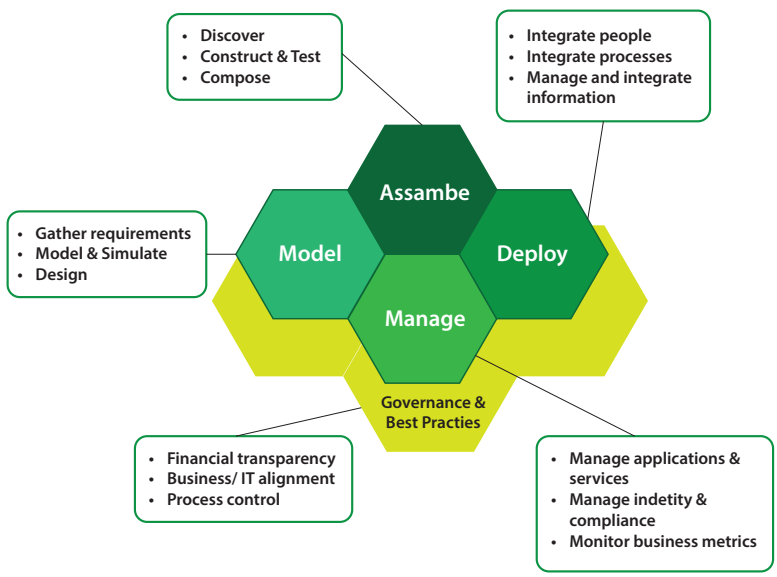
U današnjoj softverskoj arhitekturi, Registar servisa predstavlja ključnu komponentu koja omogućava lociranje resursa. U praksi se razlikuju dva osnovna modela primene, zavisno od toga da li se koriste unutar zatvorenog sistema ili za eksternu komunikaciju:

- **Interni registar servisa** (Service Registry) – Ovaj model se koristi unutar infrastrukture kompanije (naročito u mikroservisnim arhitekturama) za automatsko upravljanje lokacijama. Najčešće se koriste alati kao što su **Netflix Eureka, Consul** ili **Apache Zookeeper**.
  - Princip rada: Proces je potpuno automatizovan. Kada se određena instanca servisa (npr. Servis za procesiranje plaćanja) pokrene, ona vrši samoregistraciju slanjem svojih mrežnih parametara (IP adresa i port) centralnom registru. Na taj način, ostali servisi u sistemu mogu dinamički saznati lokaciju ovog servisa bez potrebe za ručnim konfigurisanjem adresa.

- **Eksterni API portali** (Developer Portals) – Kada su servisi namenjeni spoljnim korisnicima ili trećim stranama (npr. javni API-ji banaka ili kompanije Google), ulogu registra preuzimaju portali za programere. Na primer, platforme su: **Swagger Hub, MuleSoft Anypoint** ili **Apigee**.
  - Princip rada: Za razliku od internog registra, ovde je fokus na dokumentaciji i integraciji. Programer posećuje portal kako bi pristupio specifikacijama (kao što su OpenAPI za REST servise ili WSDL za SOAP servise). Portal služi kao centralno mesto gde se korisnik informiše o dostupnim funkcijama, formatima podataka i načinu autorizacije. Ovi portali često omogućavaju korisnicima da testiraju servise u realnom vremenu pre nego što napišu prvu liniju koda za integraciju.

9.1.3. SOA životni ciklus razvoja sistema

Razvoj sistema zasnovanih na servisima prati specifičan iterativni ciklus koji povezuje poslovne zahteve sa tehnološkom implementacijom. Na slici 9.3 prikazan je klasičan IBM-ov model SOA životnog ciklusa koji se sastoji od četiri ključne faze: modelovanja, sklapanja, implementacije i upravljanja, uz kontinuirani nadzor i upravljanje (Governance).



Slika 9.3. Faze SOA životnog ciklusa razvoja sistema

Slika 9.3 predstavlja klasični SOA životni ciklus, čiji su principi postavili osnovu za današnje moderne metodologije poput mikroservisa i Agile/DevOps pristupa. Iako su alati evoluirali, logika razdvajanja modelovanja, sklapanja i upravljanja ostaje ključna za skalabilne sisteme.

Četiri ključne faze SOA životnog ciklusa razvoja sistema:

1. ***Model*** (Modelovanje) - Faza poslovne analize i dizajna. Pre pisanja samog koda, vrši se identifikacija poslovnih zahteva i simulacija procesa. Aktivnosti ove faze su: prikupljanje zahteva, dizajniranje arhitekture i definisanje interakcije između servisa. U modernom kontekstu, ovde nastaju *korisničke priče* (User Stories).
2. ***Assemble*** (Sklapanje) - Pronalaženje postojećih servisa i kreiranje novih koji se "sklapaju" u celinu. Naglasak je ponovnoj upotrebi (reusability) servisa. Aktivnosti ove faze su: pronalaženje postojećih servisa u registru, kreiranje novih komponenti i njihovo povezivanje ("komponovanje") u funkcionalnu aplikaciju.
3. ***Deploy*** (Implementacija) - Puštanje rešenja u rad i njegova integracija u realno okruženje. Aktivnosti su: konfigurisanje *runtime* okruženja, povezivanje ljudi sa procesima i osiguravanje da su svi podaci (informacije) dostupni kroz integrisane sisteme.
4. ***Manage*** (Upravljanje) - Kontinuirano praćenje performansi i bezbednosti. Ova faza prikuplja podatke koji se vraćaju u prvu fazu (Model) radi daljih optimizacija. Aktivnosti su: praćenje performansi (monitoring), upravljanje identitetima, bezbednost i praćenje ključnih poslovnih metrika (KPI).

Governance & Best Practices (Upravljanje i najbolje prakse) predstavlja temelj koja osigurava da sve faze prate ista pravila, finansijsku isplativost i usklađenost sa biznis ciljevima.



Slika 9.4. Servisi SOA Governance

SOA *Governance* nije pojedinačna alatka, već okvir koji osigurava da svi servisi unutar organizacije rade usklađeno, bezbedno i kvalitetno. Na slici 9.4 su prikazani njegovi osnovni stubovi:

- ***Organizational Change*** (Organizacione promene): SOA zahteva promenu načina na koji timovi sarađuju. Upravljanje pomaže ljudima da pređu sa izolovanog razvoja na model deljenja i ponovne upotrebe servisa.
- ***Service Lifecycle Management*** (Upravljanje životnim ciklusom usluga): Kontrola nad servisom od njegovog kreiranja, preko aktivne upotrebe, pa sve do povlačenja iz rada.
- ***Policy*** (Politike): Definisanje pravila ko može da koristi koji servis, pod kojim uslovima i na koji način.
- ***Quality*** (Kvalitet): Osiguravanje da servisi ispunjavaju određene standarde performansi i pouzdanosti pre nego što postanu dostupni drugima.
- ***Security*** (Bezbednost): Centralizovano upravljanje pristupom i zaštita podataka unutar celokupne mreže servisa.
- ***Enforcement*** (Sprovođenje): Mehanizmi koji osiguravaju da se definisana pravila i standardi zaista poštuju u praksi.

9.1.4. Tehnološki stubovi za implementaciju servisno-orijentisanih sistema

Uspešna implementacija SOA životnog ciklusa počiva na četiri ključna tehnološka stuba koji omogućavaju transformaciju poslovnih zahteva u funkcionalna IT rešenja:

1. **Upravljanje poslovnim procesima** (BPM): Alati i metodologije za modelovanje i optimizaciju tokova aktivnosti (workflow) koji definišu potrebu za servisima.
2. **Servisno-orijentisana arhitektura** (SOA): Arhitekturni stil koji osigurava agilnost poslovanja kroz kreiranje labavo povezanih i ponovo upotrebljivih servisa.
3. **Veb servisi**: Osnovna tehnologija za interakciju i razmenu podataka između različitih softverskih komponenti unutar sistema, bez obzira na njihovu tehnološku osnovu.
4. **Otvoreni standardi** (XML, JSON): Formati koji osiguravaju univerzalnu razmenu podataka i neometanu komunikaciju između heterogenih platformi.

S obzirom na to da su koncepti BPM-a i SOA-e detaljno obrađeni u prethodnim poglavljima, u nastavku se fokusiramo na veb servise kao ključnu tehnologiju za operativno opsluživanje savremenog poslovanja.

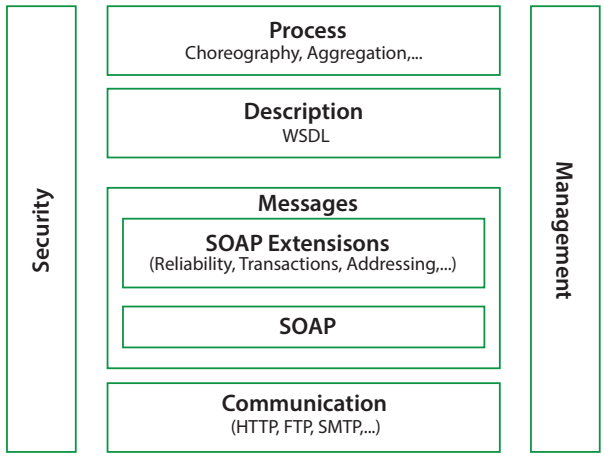
Veb servisi: definicija i karakteristike

Veb servis je softverska aplikacija dizajnirana da obavlja jednu specifičnu i ponavljajuću aktivnost (npr. otvaranje korisničkog računa), koja je univerzalno dostupna i može se koristiti unutar različitih informacionih sistema. Veb servis se definiše kao modularna, samoopisujuća softverska aplikacija koja je dostupna putem mreže i omogućava interoperabilnost između različitih sistema u distribuiranom okruženju. Prema širim akademskim definicijama i standardima (poput W3C i O'Reilly), veb servisi poseduju skup karakteristika koje ih čine osnovom savremenih distribuiranih sistema:

- **Tehnološka nezavisnost i interoperabilnost**: Servisi su potpuno nezavisni od operativnih sistema i programskih jezika. To omogućava heterogenim aplikacijama (npr. sistemu pisanom u Javi i klijentu u Python-u) da neometano komuniciraju koristeći zajedničke veb standarde.
- **Standardizovana razmena poruka**: Komunikacija se zasniva na otvorenim protokolima i formatima (poput XML-a ili JSON-a). Poruke su često **samoopisujuće**, što znači da definicija formata i struktura podataka dolaze u samom zaglavlju ili telu poruke.
- **Labava povezanost** (Loose Coupling): Ovo je jedna od najvažnijih odlika. Izmene u unutrašnjoj logici servisa ne zahtevaju promene na strani korisnika, sve dok definisani javni interfejs (ugovor) ostaje nepromenjen.
- **Modularnost i isporučljivost**: Veb servisi su dizajnirani kao samostalne, modularne jedinice softvera koje se mogu lako distribuirati, publikovati u registrima i pozivati putem mreže.
- **Minimalni klijentski zahtevi**: Servisi su samostalni u smislu da na klijentskoj strani nije potreban specifičan softver ili dodatne biblioteke; dovoljna je osnovna podrška programskog jezika za HTTP protokol i obradu formata poruka (XML/JSON).
- **Dostupnost putem otvorenih standarda**: Veb servisi su kreirani tako da budu lako locirani (putem registara) i pokrenuti preko interneta ili intraneta, oslanjajući se isključivo na globalno prihvaćene internet standarde.

Strukturna složenost veb servisa najbolje se može uočiti kroz njihov tehnološki stek (Slika 9.5):

- ***Communications*** (Transport): Mrežni protokoli (HTTP, SMTP) koji se koriste kao transportni mehanizam za prenos poruka (npr. SOAP ili XML/JSON) između različitih servisa. Dok su se ranije definicije isključivo fokusirale na SOAP protokol i XML, savremena literatura (npr. Erl, 2016) pod veb servisima podrazumeva i RESTful servise koji koriste HTTP metode kao primarni mehanizam za manipulaciju resursima, čineći sisteme još lakšim i bržim za razvoj.
- ***Messages*** (Razmena poruka): Ovaj sloj definiše format i pravila razmene podataka. Ovde se koriste XML ili JSON kao formati za strukturiranje podataka, dok se putem SOAP protokola ili REST arhitekturnog stila definiše semantika i način na koji se te poruke prenose između klijenta i servera.
- ***Descriptions*** (Opis): Standardi kao što je WSDL (Web Services Description Language) koji služe kao "uputstvo" za korišćenje servisa.
- ***Processes***: Najviši nivo gde se pojedinačni servisi kombinuju u složene poslovne tokove.
- ***Security & Management***: Vertikalni stubovi koji osiguravaju bezbednost i upravljanje kroz sve slojeve steka.



Slika 9.5. Tehnološki stek veb servisa: Prikaz slojeva od mrežnog transporta do poslovnih procesa

Tehnički standardi i protokoli u SOA arhitekturi

Razvoj servisno-orijentisanih (SO) aplikacija neraskidivo je povezan sa upotrebom **meta-podataka** koji precizno opisuju karakteristike servisa i strukturu podataka. Dok se za sam opis podataka najčešće koriste **XML** ili **JSON**, komunikacija i orkestracija se oslanjaju na sledeće standarde:

1. **Opis i orkestracija servisa** (WSDL i BPEL)
   - **WSDL** (Web Service Definition Language): Jezik zasnovan na XML-u koji služi kao "lična karta" servisa. On opisuje šta servis radi, gde se nalazi i na koji način mu se može pristupiti.
   - **BPEL** (Business Process Execution Language): Poznat i kao WS-BPEL, ovaj standard služi za **orkestraciju**[5]. On definiše složene poslovne procese koji upravljaju interakcijom između više različitih servisa (definisanih preko WSDL-a).
   - **Orkestracija**: Proces u kojem se SO aplikacije razvijaju povezivanjem nezavisnih servisa koji komuniciraju putem poruka kako bi izvršili složen poslovni zadatak.

2. **Modelovanje poslovnih procesa** (BPMN)

Za vizuelno i logičko modelovanje poslovnih procesa, koji kasnije postaju osnova za razvoj servisa, koristi se **BPMN** (Business Process Modeling Notation) standard. On omogućava da se poslovna logika predstavi na način koji je razumljiv i biznis analitičarima i programerima.

5 Orkestracija je proces upravljanja i povezivanja više pojedinačnih servisa u jednu logičnu celinu kako bi se izvršio složeni poslovni proces. Kao dirigent u orkestru koji određuje kojim redosledom se servisi pozivaju, kako se razmenjuju podaci između njih i šta se dešava ako dođe do greške.

3. Protokoli za pristup servisima: SOAP vs. REST

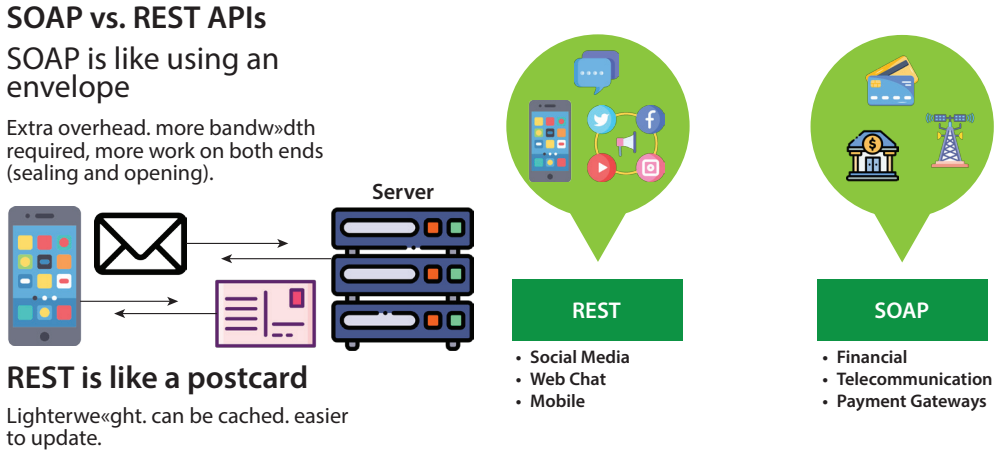
Pristup veb servisima se danas realizuje kroz dva dominantna pristupa (Tabela 9.2). Zbog svoje jednostavnosti i brzine, REST je postao dominantan izbor u razvoju modernih mikroservisnih aplikacija, dok se SOAP i dalje često sreće u kompleksnim bankarskim i korporativnim sistemima koji zahtevaju stroge bezbednosne protokole.

Tabela 9.2. Poređenje karakteristika SOAP i REST

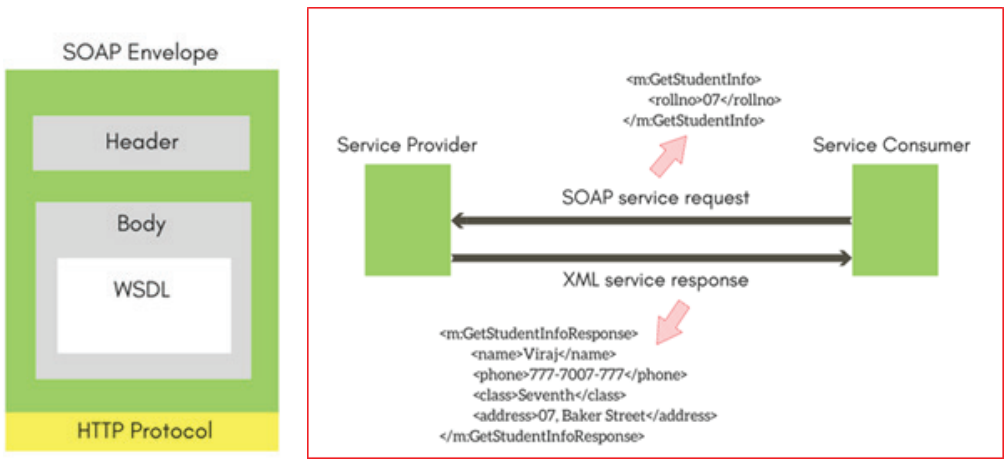
Karakteristika	SOAP (Simple Object Access Protocol)	REST (Representational State Transfer)
Priroda	Strogo definisan protokol (W3C standard).	Arhitekturni stil (nije fiksni standard).
Komunikacija	Koristi specifične, prilagođene metode (npr. getCustomerName).	Koristi standardne HTTP metode (GET, POST, PUT, DELETE).
Format	Razmena strukturiranih informacija (isključivo XML).	Fokus na resursima (URI) i razmeni njihovih reprezentacija.
Efikasnost	Zahteva parsiranje XML-a (može biti sporiji).	Često koristi JSON, lakši za implementaciju i brži za izvršavanje.

Iako su tehničke specifikacije jasne, razliku u načinu rada i opterećenju sistema koje ovi protokoli proizvode najlakše je razumeti kroz analogiju sa poštanskim pošiljkama (Slika 9.6):

- **SOAP kao koverta**: Baš kao što pismo u koverti zahteva dodatni trud (pečaćenje, otvaranje) i zauzima više mesta, SOAP ima visok *overhead*. Zahteva više propusnog opsega jer svaka poruka mora biti upakovana u složenu XML strukturu sa zaglavljima (tzv. envelope). SOAP se koristi tamo gde je integritet podataka važniji od brzine. Zbog svoje stroge strukture i ugrađenih bezbednosnih standarda, on je nezamenljiv u finansijskom sektoru (Financial), telekomunikacijama i sistemima za procesuiranje plaćanja (Payment Gateways).
- **REST kao razglednica**: On je lagan i direktan. Informacije su "vidljive" (često u samom URL-u ili laganom JSON formatu), lakše se keširaju i zahtevaju manje resursa za obradu, što ga čini idealnim za brze moderne veb aplikacije. REST se koristi tamo gde je bitno opslužiti ogroman broj korisnika uz minimalno kašnjenje. Tipični primeri su društvene mreže, aplikacije za dopisivanje (Web Chat) i sve moderne mobilne aplikacije.



Slika 9.6. Vizuelna analogija: Poređenje SOAP protokola i REST arhitekturnog stila

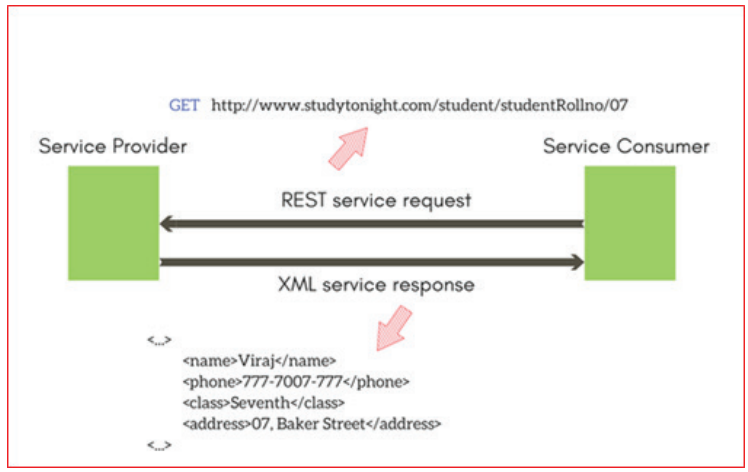


Slika 9.7. Struktura SOAP poruke (SOAP Envelope)

U SOAP modelu, klijent ne traži samo podatak, već poziva specifičnu funkciju koja je definisana unutar XML koverte:

- **SOAP *Service Request*** (Zahtev): Umesto običnog linka, klijent (Service Consumer) šalje XML blok podataka. Primetite tag <m:GetStudentInfo> – to je specifična metoda koja je unapred definisana u WSDL dokumentu, što SOAP čini strožim i formalnijim od REST-a.
- **Identifikacija**: Parametar *rollno* sa vrednošću 07 nije deo URL-a, već je "upakovan" unutar XML poruke.

- **XML *Service Response*** (Odgovor): Server vraća odgovor koji je takođe uokviren XML tagovima (npr. <m:GetStudentInfoResponse>).
- **Složenost** (Overhead): Vidljivo je da poruka sadrži mnogo više teksta (tagova) nego samih podataka. Upravo to je onaj "overhead" (dodatni teret) o kojem govori analogija sa kovertom – za prenos jedne informacije (ime: Viraj), troši se znatno više bajtova na definisanje strukture.



Slika 9.8. Primer REST zahteva i odgovora u realnom vremenu

Slika 9.8 vizuelno potvrđuje zašto je REST nazvan "razglednicom":

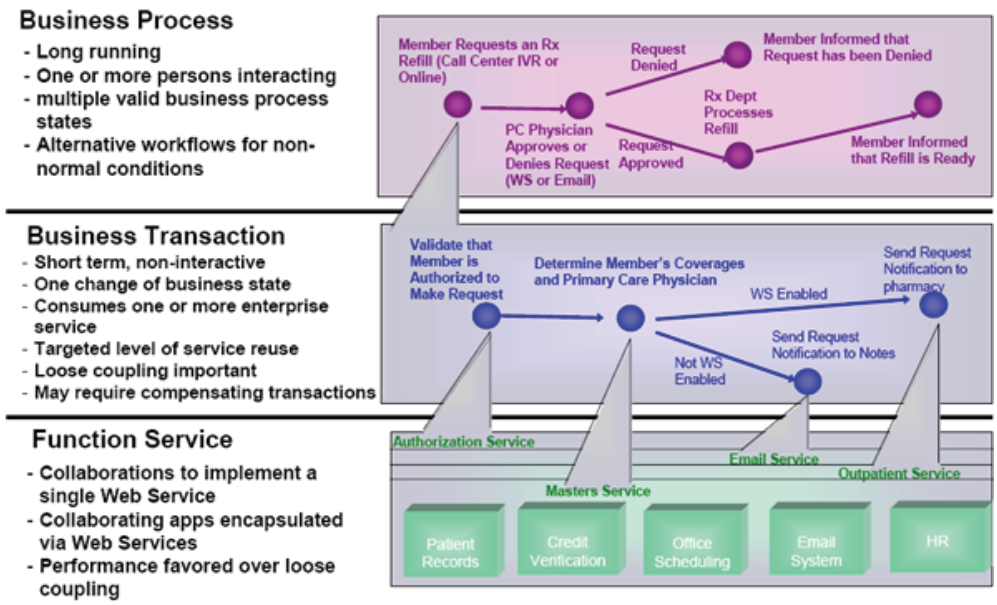
- **REST *Service Request*** (Zahtev): Vidimo da klijent (Service Consumer) šalje jednostavnu **GET** metodu putem URL-a. U samom URL-u se nalazi identifikator resursa (npr. broj studenta 07), što je suština REST-a – adresiranje resursa putem URI-ja.
- ***Service Provider*** (Server): Prima zahtev, pronalazi resurs u svojoj bazi i priprema odgovor.
- **XML *Service Response*** (Odgovor): Server vraća tražene podatke u strukturiranom formatu. Iako slika prikazuje **XML**, može biti vraćen i **JSON**, koji je danas češći zbog svoje lakoće.
- **Transparentnost**: Podaci poput imena, telefona i adrese su jasno definisani unutar tagova, što omogućava **sintaksnu interoperabilnost**.

Taksonomija i klasifikacija servisa u SOA sistemima

Radi efikasnijeg modelovanja i implementacije, servisi se unutar servisno-orijentisanih sistema klasifikuju prema njihovom **nivou apstrakcije** i **granularnosti**. Ova podela omogućava povezivanje tehničkih komponenti sa poslovnim ciljevima (Slika 9.9):

1. **Tehnički i funkcionalni servisi** (*Technical & Function Services*) - Najniži nivo koji obuhvata atomske, infrastrukturne operacije (npr. provera lozinke, logovanje grešaka). Ovde je fokus na performansama.
   - **Nivo**: Aktivnosti i metode.
   - **Opis**: Predstavljaju najniži nivo apstrakcije i često su "atomski" (ne mogu se dalje deliti). Oni inkapsuliraju specifične aplikativne funkcije ili podatke.
   - **Karakteristike**: Fokus je na visokim performansama. Mogu biti čisto tehnički (npr. checkUserPassword, auditEvent) ili jednostavne poslovne funkcije koje ne menjaju stanje sistema, već vrše proračune (npr. calculateTax ili getPrice).
2. **Servisi poslovnih transakcija** (*Business Transaction Services*) - Srednji sloj koji kombinuje funkcionalne servise radi izvršenja konkretne aktivnosti koja menja stanje sistema (npr. "Rezerviši let" ili "Kreiraj fakturu").
   - **Nivo**: Klase i komponente.
   - **Opis**: Ovaj srednji sloj kombinuje jedan ili više funkcionalnih servisa kako bi izvršio konkretnu poslovnu aktivnost koja menja stanje sistema.
   - **Karakteristike**: Naglasak je na **ponovnoj upotrebi** (reuse) i labavom povezivanju. Primeri uključuju operacije kao što su "Nabavka materijala" ili "Snimanje nove transakcije klijenta".
3. **Servisi poslovnih procesa** (*Business Process Services*) - Najviši nivo koji predstavlja kompleksne, dugotrajne tokove rada (npr. "Obrada kreditnog zahteva"). Ovi servisi upravljaju logikom celokupnog poslovnog cilja.
   - **Nivo**: *Use-case* i nivo aplikacije.
   - **Opis**: Najviši nivo apstrakcije koji predstavlja orkestraciju (povezivanje) više servisa u dugotrajne tokove aktivnosti (workflows). Ovi procesi ispunjavaju jedan širi poslovni cilj.
   - **Karakteristike**: Često uključuju interakciju između ljudi i sistema, upravljaju složenim stanjima i različitim scenarijima. Primeri su "Obrada zahteva za kredit" ili "Kreiranje i upravljanje porudžbinom".

Razumevanje ovih nivoa je ključno za uspešnu SOA integraciju, jer omogućava da se izmene u nižim slojevima (npr. promena načina provere lozinke) vrše bez remećenja celokupnog poslovnog procesa, dokle god su interfejsi servisa stabilni. Slika 9.9, ne prikazuje samo različite servise, već kako se oni grupišu od najjednostavnijih tehničkih funkcija do kompleksnih poslovnih procesa.



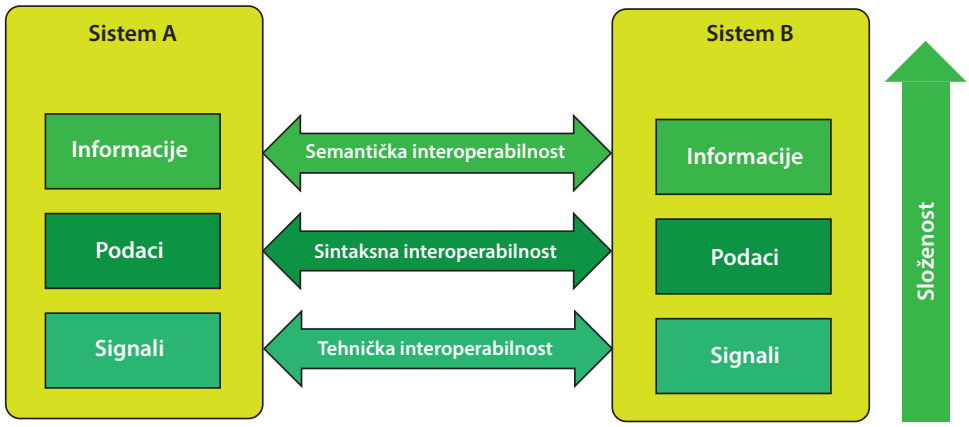
Slika 9.9. Klasifikacija servisa prema nivou apstrakcije

Razlozi za primenu i značaj veb servisa u poslovanju

Primarni cilj primene veb servisa je dostizanje pune **interoperabilnosti**, koja se prema IEEE standardu definiše kao sposobnost dva ili više sistema ili komponenti da razmenjuju informacije i da razmenjene informacije ispravno koriste. U širem smislu, interoperabilnost predstavlja sposobnost heterogenih softverskih jedinica da pružaju ili prihvataju usluge drugih sistema, omogućavajući efikasan zajednički rad bez narušavanja integriteta ili degradacije podataka. U arhitekturi veb servisa, ovaj koncept se realizuje kroz dva komplementarna nivoa:

- **Sintaksna interoperabilnost**: Fokusira se na tehničku mogućnost razmene podataka. Nju omogućava korišćenje standardizovanih formata poput **XML**-a ili **JSON**-a, koji definišu gramatiku i strukturu same poruke.
- **Semantička interoperabilnost**: Ide korak dalje i osigurava da oba sistema na identičan način interpretiraju značenje razmenjenih podataka. Ovaj nivo se postiže upotrebom zajedničkih protokola i specifikacija (kao što su SOAP i WSDL za strogu definiciju, ili REST u kombinaciji sa OpenAPI/Swagger dokumentacijom), koji precizno definišu kontekst i poslovnu logiku interakcije.

Nivoi interoperabilnosti između sistema su prikazani na slici 9.10.



Slika 9.10. Nivoi interoperabilnosti u komunikaciji između sistema

Osnovni razlog zbog kojeg kompanije usvajaju veb servise leži u rešavanju problema **heterogenosti IT okruženja**. Glavne prednosti uključuju:

- **Prevazilaženje nekompatibilnosti**: Rešavaju kritičan problem komunikacije između različitih formata podataka i operativnih sistema. Bez njih, prenos podataka od tačke A do tačke B zahtevao bi skupu i kompleksnu prilagodbu svakog pojedinačnog sistema.
- **Smanjenje troškova integracije**: Korišćenjem standardizovanih pristupa, drastično se smanjuju resursi potrebni za povezivanje novih aplikacija sa postojećim (*legacy*) sistemima.
- **Standardizacija putem protokola**: Veb servisi svoju efikasnost crpe iz široko prihvaćenih softverskih standarda i tehnologija:
  - **Formati podataka**: XML i JSON (omogućavaju struktuiranu razmenu).
  - **Komunikacioni protokoli:** SOAP i REST (definišu pravila slanja poruka).
  - **Opis i orkestracija**: WSDL (za opis interfejsa servisa) i BPEL (za definisanje redosleda izvršavanja poslovnih procesa).

9.2. SERVISNO-ORIJENTISANO MODELOVANJE SOFTVERA

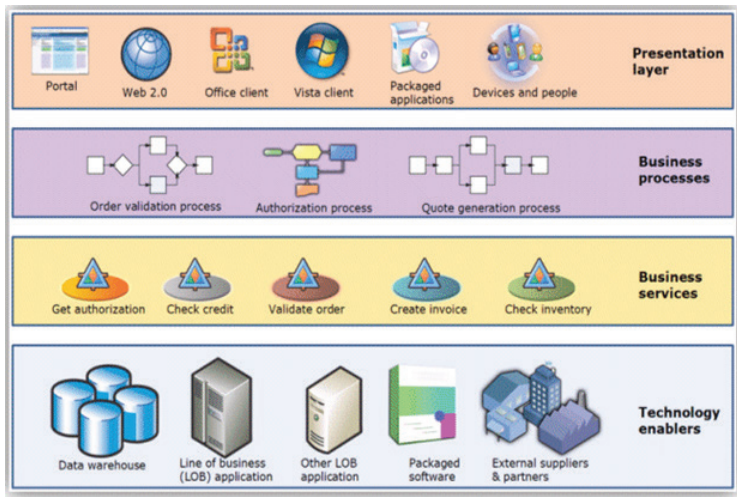
Servisno-orijentisano modelovanje softvera predstavlja sistematičan postupak projektovanja aplikacije kao skupa **slabo povezanih, autonomnih i ponovo upotrebljivih servisa**, koji zajednički realizuju poslovne procese organizacije. Za razliku od tradicionalnog objektno-orijentisanog pristupa, gde je fokus na klasama i objektima, servisno-orijentisani pristup polazi od **poslovnih sposobnosti** (*business capabilities*) i njihove realizacije kroz jasno definisane servise.

Približavanjem programskog koda poslovnim zahtevima dolazi do zbližavanja dve tradicionalno razdvojene oblasti: **softverskog inženjerstva** i **upravljanja poslovnim procesima**. U tom kontekstu, softverski inženjeri sve češće učestvuju i u modelovanju i unapređivanju poslovnih procesa, dok poslovni akteri aktivno utiču na dizajn informacionih sistema.

Upravljanje poslovnim procesima (*Business Process Management*, BPM) predstavlja integrisani pristup identifikaciji, modelovanju, analizi, automatizaciji i kontinuiranom unapređenju ključnih procesa u organizaciji, sa ciljem povećanja efikasnosti, smanjenja troškova i prilagođavanja promenljivom poslovnom okruženju. Kao disciplina, BPM obuhvata kako organizacione, tako i tehničke aspekte poslovanja.

Sa aspekta servisno-orijentisane arhitekture, **poslovni proces** se može posmatrati kao skup povezanih i strukturiranih aktivnosti ili zadataka koji se realizuju kroz **poslovne servise** (*Business Services*). Ovi servisi predstavljaju gradivne blokove sistema i omogućavaju uspostavljanje veze između poslovnog modela i njegove softverske realizacije.

Uvođenjem **sloja poslovnih servisa** postiže se razdvajanje poslovne logike od implementacionih tehnologija, čime se obezbeđuju veća fleksibilnost sistema, ponovna upotrebljivost funkcionalnosti i interoperabilnost između heterogenih aplikacija (Slika 9.11).



Slika 9.11. Evolucija ka servisno-orijentisanoj arhitekturi: uvođenje sloja poslovnih servisa

9.2.1. Standardizacija modelovanja poslovnih procesa

Razvoj i primena standarda za modelovanje i automatizaciju poslovnih procesa rezultat su aktivnosti brojnih međunarodnih organizacija, među kojima se posebno izdvajaju:

- **OMG** (*Object Management Group*) – razvija i održava standarde za modelovanje i softversku arhitekturu, uključujući BPMN, UML i druge specifikacije relevantne za BPM i SOA.
- **OASIS** (*Organization for the Advancement of Structured Information Standards*) – razvija standarde za interoperabilnost informacionih sistema, uključujući BPEL, ebXML i druge specifikacije za integraciju i orkestraciju servisa.
- **UN/CEFACT** (*United Nations Centre for Trade Facilitation and Electronic Business*) – razvija međunarodne standarde i preporuke za elektronsko poslovanje i razmenu poslovnih dokumenata u globalnoj trgovini.
- **W3C** (*World Wide Web Consortium*) – definiše i održava osnovne web standarde (npr. XML, WSDL, SOAP) koji čine tehnološku osnovu za web i servisno-orijentisane sisteme.
- **Akademske i istraživačke institucije** – doprinose razvoju teorijskih modela, metodologija i alata u oblasti BPM-a i servisno-orijentisanih sistema kroz naučna istraživanja i obrazovanje.

Organizacije koje su imale značajnu istorijsku ulogu u razvoju BPM standarda, ali danas ne deluju kao samostalne standardizacione inicijative, uključuju:

- **BPMI** (*Business Process Management Initiative*) – inicijativa zaslužna za rani razvoj BPMN-a (pripojena OMG-u 2005. godine).
- **WfMC** (*Workflow Management Coalition*) – organizacija koja je razvila standarde u oblasti workflow sistema, uključujući XPDL (prestala sa radom kao organizacija, ali su njeni standardi i dalje prisutni u praksi).

Standardi u oblasti BPM-a mogu se klasifikovati prema svojoj nameni na sledeće kategorije:

- **Grafički standardi** – omogućavaju vizuelno predstavljanje poslovnih procesa, tokova i tranzicija putem dijagrama, čime se olakšava komunikacija između poslovnih i tehničkih aktera. Najznačajniji primer je **BPMN** (*Business Process Model and Notation*), koji je danas standard za modelovanje poslovnih procesa. BPMN obezbeđuje bogatu notaciju za prikaz sekvencijalnih i paralelnih tokova, događaja, uloga i poruka i podržan je u savremenim BPM alatima kao što su *Camunda Modeler, Signavio, Bizagi* i *IBM Blueworks Live*.
- **Standardi razmene** – omogućavaju prenosivost i razmenu modela poslovnih procesa između različitih alata za modelovanje i izvršavanje, čime se smanjuje zavisnost od konkretnog proizvođača. Primeri uključuju **XPDL** (*XML Process Definition Language*), koji je razvijen od strane WfMC i omogućava razmenu BPMN modela između alata, kao i **BPMN XML** format, koji je sastavni deo BPMN specifikacije i danas se najčešće koristi u modernim BPM platformama.

- **Izvršni standardi** – podržavaju automatizaciju i izvršavanje poslovnih procesa kroz procesne engine-e, čime se modeli prelaze iz dizajna u operativnu upotrebu. Klasičan primer je **BPEL** (*Business Process Execution Language*), koji je namenjen orkestraciji veb servisa u SOA okruženju. U savremenim sistemima, izvršavanje BPMN modela realizuje se putem BPM *engine*-a kao što su *Camunda, Flowable* i *jBPM*, koji omogućavaju direktno izvršavanje BPMN procesa bez potrebe za dodatnim transformacijama.
- **Dijagnostički standardi** – omogućavaju praćenje, analizu i monitoring poslovnih procesa, kao i upravljanje njihovim performansama. Ovi standardi su osnova za ***Business Activity Monitoring*** (BAM) i ***Process Analytics***, a uključuju mehanizme za prikupljanje metrika kao što su vreme trajanja procesa, zadržavanja na aktivnostima i iskorišćenost resursa. U praksi se realizuju kroz kombinaciju standarda i alata, npr. *BPMN event logove,* ***process mining*** tehnike (kao što su Celonis ili ProM) i integraciju sa sistemima za nadzor i izveštavanje.

BPMN (*Business Process Model and Notation*) razvijen je sa ciljem da obezbedi standardizovanu notaciju za definisanje poslovnih procesa, koja je **lako razumljiva svim zainteresovanim stranama**, od poslovnih korisnika do tehničkih stručnjaka. BPMN omogućava prevazilaženje jaza između dizajna poslovnih procesa i njihove softverske implementacije.

BPMN razlikuje sledeće osnovne tipove modela:

- **Orkestracija procesa** – opisuju centralno kontrolisan tok izvršavanja procesa putem procesnog "engine"-a.
- **Koreografija** – opisuje proces kroz razmenu poruka između učesnika, bez centralne kontrole toka izvršavanja.
- **Kolaboracije poslovnih partnera** – proširuju koreografiju uključivanjem:
  - B2B peer-to-peer odnosa,
  - komunikacije isključivo putem poruka,
  - ugovornih obaveza između učesnika,
  - razmene preko Interneta,
  - usaglašavanja semantike poslovnih poruka.

Servisno-orijentisano modelovanje softvera predstavlja most između poslovnih procesa i softverske arhitekture. Korišćenjem BPMN-a kao ulaznog modela, a servisnih modela kao izlaza, obezbeđuje se:

- jasno razdvajanje odgovornosti,
- lakša evolucija sistema,
- bolja usklađenost IT sistema sa poslovnim potrebama.

9.2.2. Koraci servisno-orijenisanog modelovanja sistema

Postupak servisno-orijentisanog modelovanja obuhvata sledeće korake:

1. Identifikacija i analiza poslovnih procesa

Polazna tačka servisno-orijentisanog modelovanja jeste **poslovni domen**. Poslovni procesi se identifikuju i modeluju pomoću **BPMN dijagrama**. Cilj ovog koraka nije tehnička realizacija, već razumevanje:

- *šta* sistem radi,
- *ko* učestvuje u procesu,
- *kako* se aktivnosti sekvencijalno i paralelno izvršavaju.

BPMN modeli služe kao konceptualni temelj za dalje izdvajanje servisa.

2. Identifikacija kandidata za servise

Na osnovu BPMN modela vrši se identifikacija **servisnih kandidata**. Tipično se izdvajaju:

- **Poslovni servisi** – predstavljaju ključne poslovne funkcije (npr. Upravljanje porudžbinom, Obrada plaćanja)
- **Aplikacioni servisi** – podržavaju poslovne procese (npr. Autentifikacija korisnika, Slanje notifikacija)
- **Tehnički/infrastrukturni servisi** – zajedničke tehničke funkcije (npr. Logovanje, Monitoring)

Servisni kandidat se obično prepoznaje kao **grupa povezanih aktivnosti** u BPMN dijagramu ili kao samostalna poslovna odgovornost.

3. Definisanje granica i odgovornosti servisa

Za svaki identifikovani servis definišu se:

- jasne granice (*service boundary*),
- odgovornosti koje servis obuhvata,
- podaci kojima servis upravlja.

Servis treba da bude: **autonoman** (ima sopstvenu logiku i podatke), **slabo povezan** sa ostalim servisima, **visoko kohezivan** (fokusiran na jednu poslovnu svrhu). U ovom koraku se izbegava deljenje baze podataka između servisa i preklapanje odgovornosti.

4. Modelovanje servisnih interfejsa i ugovora

Nakon definisanja servisa, modeluje se njihova komunikacija putem **servisnih ugovora**. Za svaki servis se definiše:

- koje **operacije** pruža,
- ulazni i izlazni podaci i
- pravila komunikacije.

Na konceptualnom nivou, ovo se može prikazati:

- UML dijagramima komponenti,
- UML dijagramima sekvenci (fokus na razmenu poruka),
- opisima servisnih ugovora (bez tehnologije).

U ovom koraku se ne vezujemo za REST, SOAP ili konkretnu platformu, već je fokus na šta *servis nudi*, a ne kako je implementiran.

5. Modelovanje orkestracije i koreografije servisa

Servisi retko deluju izolovano, oni sarađuju u realizaciji poslovnih procesa.

- **Orkestracija servisa** opisuje centralno kontrolisan tok poziva servisa (često mapiran iz BPMN dijagrama)
- **Koreografija servisa** opisuje raspodeljenu interakciju bez centralne kontrole

U modelovanju:

- BPMN proces može direktno da se mapira na orkestraciju,
- sekvence poziva servisa se prikazuju UML sekvencnim dijagramima.

Ovim se povezuje **poslovni pogled** sa **servisnim pogledom sistema**.

6. Mapiranje na arhitekturu sistema

U završnom koraku servisni modeli se povezuju sa arhitektonskim slojevima: klijentske aplikacije (*frontend*), servisni sloj (SOA / mikroservisi) i sloj podataka (po servisima). Ovaj korak omogućava prelazak sa modela na implementaciju, analizu skalabilnosti, bezbednosti i performansi.

Vežba 9.1. Modelovanje servisno orijentisane aplikacije za obradu zahteva za bankarski kredit korišćenjem BPMN notacije

Cilj ove vežbe je da se demonstrira **modelovanje servisno orijentisane aplikacije** (SOA) primenom **BPMN notacije**, pri čemu se BPMN dijagram koristi kao sredstvo za **orkestraciju nezavisnih servisa** i **koordinaciju** aktivnosti različitih organizacionih uloga unutar banke. Potrebno je jasno razlikovati:

- modelovanje *poslovnog procesa* i
- modelovanje *servisno orijentisane aplikacije*,

kao i automatizovane servisne aktivnosti od aktivnosti koje obavljaju ljudski akteri.

Opis sistema

Proces obrade zahteva za bankarski kredit realizovan je kao **servisno orijentisana aplikacija**, sastavljena od više autonomnih servisa koji međusobno komuniciraju razmenom poruka. Glavni poslovni proces unutar banke deluje kao **orkestrator servisa**, pozivajući odgovarajuće servise u različitim fazama obrade zahteva. Interakcija sa klijentom odvija se preko klijentskog portala, dok se interna obrada zahteva sprovodi kroz saradnju više sektora banke i automatizovanih servisa.

Učesnici u procesu

Proces se modeluje korišćenjem **jedne particije** (Pool) **za Banku**, podeljenog na sledeće staze (Lanes):

1. **Klijentski portal** (Client Portal) – automatizovana interakcija sa klijentom
2. **Predstavnik korisničke službe** (Customer Service Representative)
3. **Službenik za obradu kredita** (Loan Processing Clerk)
4. **Menadžer za kredite** (Loan Manager)

Pored toga, u modelu je potrebno prikazati **eksterne servise** (kao posebne pool-ove ili jasno označene Service Task-ove), kao što su:

- Servis za proveru dokumentacije
- Eksterni servis za proveru kreditnog rejtinga
- Eksterni servis za slanje notifikacija
- Servis za primenu poslovnih pravila

Klijent se posmatra kao **eksterni učesnik** („crna kutija"), bez detaljnog modelovanja njegove interne logike.

Opis procesa

1. Faza inicijacije – Klijentski portal

- Proces započinje **porukom klijenta** koji šalje upit o podobnosti za kredit.
- Sistem automatski pruža informacije o uslovima kredita (poziv odgovarajućeg informacionog servisa).
- Proces zatim čeka zvaničan zahtev za kredit. Ukoliko zahtev ne stigne u roku od **mesec dana**, aktivira se **tajmerski događaj** koji pokreće slanje podsetnika klijentu sa pitanjem da li je i dalje zainteresovan.
- Prijem zvaničnog zahteva za kredit pokreće narednu fazu procesa.

2. Provera dokumentacije – Korisnička služba

- Predstavnik korisničke službe inicira proveru dokumentacije.
- Provera kompletiranosti dokumentacije realizuje se pozivom **servisa za validaciju dokumentacije**.
- Ukoliko dokumentacija nije potpuna, klijentu se automatski šalje zahtev za dopunu dokumenata, nakon čega se proces vraća u stanje čekanja.
- Kada je dokumentacija kompletna, zahtev se prosleđuje na analizu kreditne sposobnosti.

3. Analiza kreditne sposobnosti – Službenik za obradu kredita

- Analiza kreditne sposobnosti realizuje se pozivom **servisa za proveru kreditnog rejtinga**, uz primenu definisanih poslovnih pravila.
- Ukoliko kreditni rejting ne ispunjava uslove, proces se završava odbijanjem zahteva ili se prosleđuje na dopunsku proveru (po izboru).
- Ako je kreditni rejting zadovoljavajući, proverava se iznos traženog kredita:

- Za iznose **ispod definisanog praga** (threshold), sistem automatski šalje pismo o odobrenju klijentu.
- Za iznose **iznad praga**, zahtev se prosleđuje menadžeru za kredite.

4. Finalno odobrenje – Menadžer za kredite

- Menadžer vrši naprednu proveru kreditnog rejtinga i donosi finalnu odluku.
- Odluka se može realizovati kao kombinacija **User Task**-a i poziva odgovarajućeg servisnog modula.
- U slučaju odobrenja, sistem automatski šalje pismo o odobrenju klijentu.
- Proces se uspešno završava i prelazi u narednu fazu isplate sredstava (Payout), koja nije predmet ovog zadatka.

Potrebno je da BPMN dijagram ispuni sledeće zahteve:

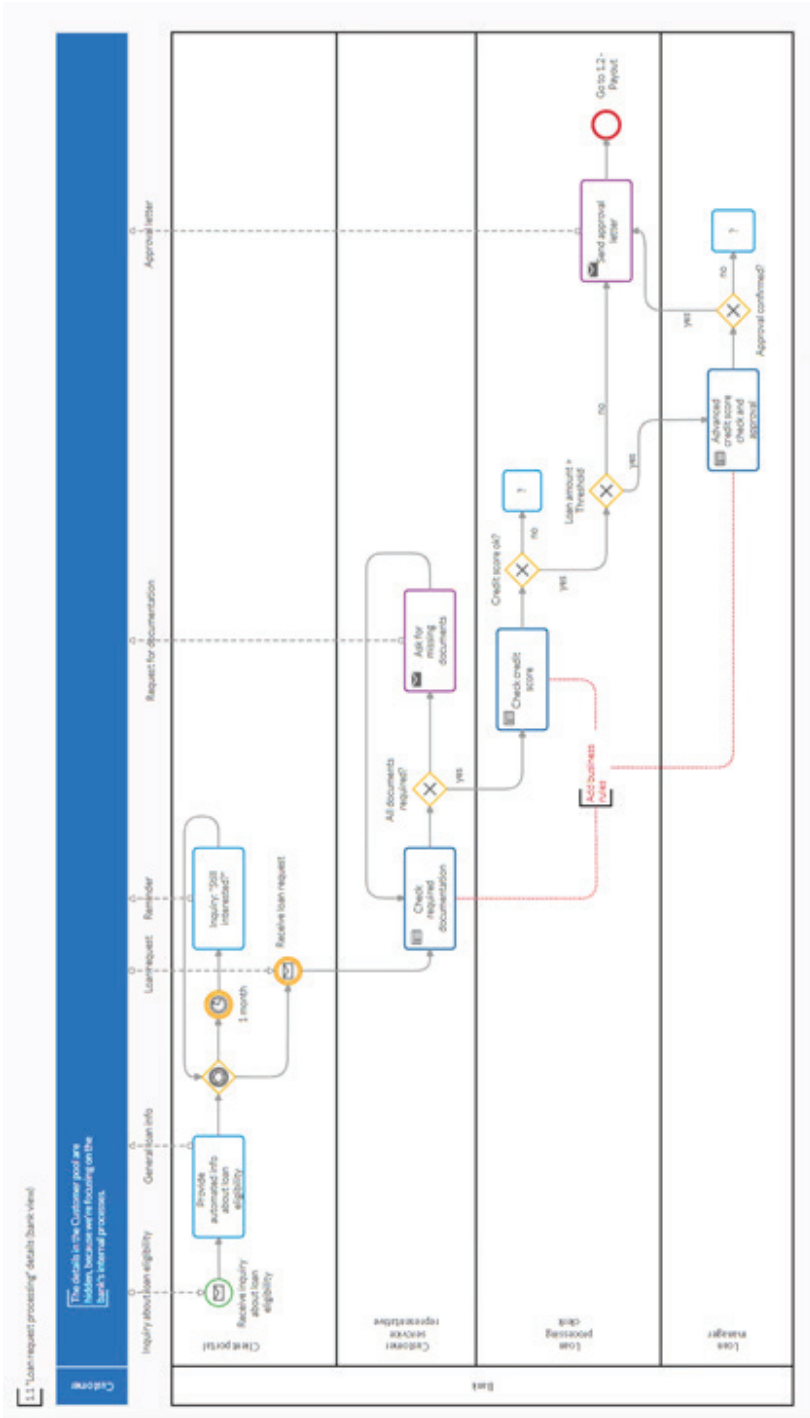
1. **Organizacija**: Jedan *Pool* za Banku sa jasno definisanim *lanes*.
2. **Događaji**: Pravilna upotreba početnih (Message Start), prelaznih (Timer, Message Intermediate) i završnih događaja.
3. **Aktivnosti**: Jasno razlikovati **Service Task, User Task** i **Manual Task**.
4. **Kapije** (Gateways): Koristiti ekskluzivne (XOR) kapije za donošenje odluka na osnovu uslova.
5. **Servisna orijentacija**: Identifikovati i eksplicitno označiti servise koji učestvuju u procesu i prikazati orkestraciju servisa iz glavnog procesa.
6. **Komunikacija**: Prikazati tokove poruka između banke, klijenta i eksternih servisa.
7. **Poslovna pravila**: Označiti gde se primenjuju interna poslovna pravila (npr. kreditni prag, pravila rejtinga).

Očekuje se da rešenje pokaže sposobnost da se:

- modeluje servisno orijentisana aplikacija korišćenjem BPMN notacije,
- prikaže BPMN kao mehanizam orkestracije servisa,
- pravilno primene BPMN elementi u realnom bankarskom scenariju,
- jasno i konzistentno obrazlože doneta modelarska rešenja.

Rešenje:

Koristeći bpmn.io alat kreirati model.



Izvor: https://www.cardanit.com/blog/banking-business-process-mapping/

9.3. MODELOVANJE SERVISNO-ORIJENTISANE INTEGRACIJE SOFTVERA

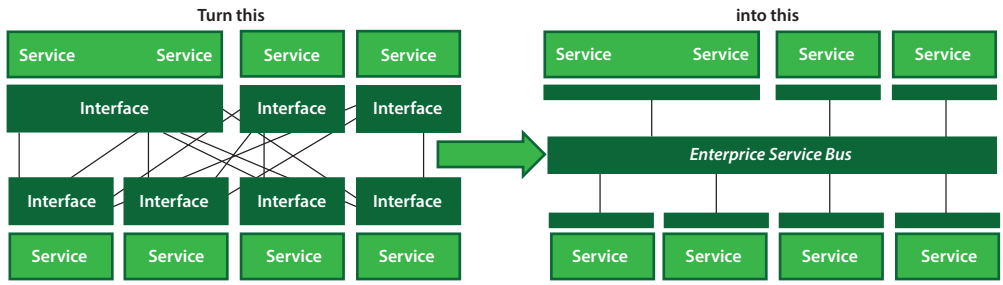
9.3.1. Uvod u servisno orijentisanu integraciju

Savremene organizacije u svom poslovanju koriste veliki broj različitih informacionih sistema (aplikacija) koji su najčešće razvijani u različitim vremenskim periodima, korišćenjem različitih tehnologija i za potrebe različitih organizacionih celina. Kao posledica toga, ovi sistemi često funkcionišu kao izolovane celine koje ne komuniciraju međusobno na adekvatan način, što otežava razmenu podataka i koordinaciju poslovnih procesa. Ovakva fragmentacija informacionog sistema predstavlja značajan izazov za organizacije koje teže efikasnom upravljanju procesima i donošenju odluka zasnovanih na pouzdanim i ažurnim podacima.

U praksi, organizacije istovremeno koriste više vrsta informacionih sistema. Na primer, CRM sistemi se koriste za upravljanje odnosima sa klijentima i najčešće imaju sopstvenu bazu podataka koju prvenstveno koristi marketing ili prodajni sektor. ERP sistemi, sa druge strane, automatizuju interne poslovne procese kao što su finansije, nabavka, logistika i ljudski resursi i takođe poseduju sopstvene baze podataka. Pored toga, organizacije često koriste veb aplikacije i portale putem kojih krajnji korisnici unose svoje podatke, podnose zahteve ili iniciraju poslovne procese. Uz ove savremene sisteme, neretko su prisutni i *legacy* sistemi koji su razvijeni ranije i koji su i dalje kritični za poslovanje. Iako svaki od navedenih sistema uspešno ispunjava svoju osnovnu namenu, glavni problem nastaje u trenutku kada je potrebno obezbediti razmenu podataka i koordinaciju aktivnosti između njih. Zbog nedostatka integracije, isti podaci se često unose više puta, javljaju se nekonzistentnosti u informacijama, a poslovni procesi postaju spori i nepouzdani.

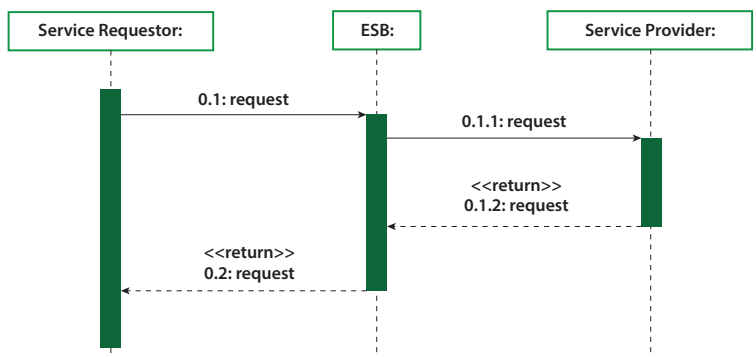
Najčešći i intuitivni pristup rešavanju problema integracije jeste primena tzv. *point-to-point* rešenja. Ovim pristupom se posmatraju dva konkretna sistema i implementira se njihova direktna integracija, najčešće putem prilagođenih interfejsa ili razmene poruka. Nakon toga se na sličan način integrišu naredni parovi sistema, u zavisnosti od poslovnih potreba. Iako *point-to-point* integracija može biti prihvatljivo rešenje u okruženjima sa malim brojem sistema, ovaj pristup brzo pokazuje svoja ograničenja kako sistem raste. Sa povećanjem broja aplikacija, broj integracionih veza eksponencijalno raste, što dovodi do složenih i teško održivih integracionih struktura. Svaka promena u jednom sistemu može zahtevati izmene u više integracionih veza, čime se povećavaju troškovi održavanja i rizik od grešaka.

Kao odgovor na ograničenja *point-to-point* pristupa, razvijen je koncept servisno orijentisane integracije. Servisno orijentisana integracija (SOI) nastoji da reši jedan od ključnih problema savremenih informacionih sistema, a to je problem interoperabilnosti između heterogenih sistema. Heterogeni sistemi se razlikuju po tehnologijama, platformama, protokolima komunikacije i formatima podataka koje koriste, što otežava njihovu direktnu međusobnu integraciju. SOI ovaj problem rešava uvođenjem posrednog integracionog sloja, čija je centralna komponenta *Enterprise Service Bus* (ESB).



Slika 9.12. Transformacija iz point-to-point integracije do ESB arhitekture

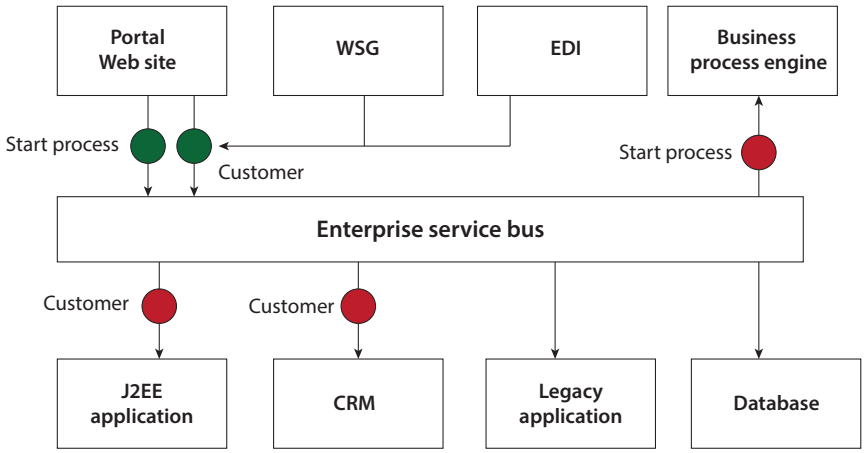
ESB predstavlja infrastrukturnu komponentu koja omogućava da sistemi komuniciraju posredno, bez direktne zavisnosti jedni od drugih. Umesto da svaki sistem mora da poznaje detalje implementacije drugih sistema, komunikacija se odvija putem ESB-a, koji preuzima ulogu posrednika. Na ovaj način se postiže razdvajanje aplikacija i smanjenje međusobne zavisnosti, što doprinosi većoj fleksibilnosti i održivosti integracionog rešenja.



Slika 9.13. Posrednička uloga ESB-a

Jedna od najvažnijih uloga ESB-a jeste obezbeđivanje posredničkih (*mediation*) servisa koji rešavaju nekompatibilnosti između sistema. Ovi servisi omogućavaju transformaciju formata podataka (npr. XML, JSON, CSV), mapiranje struktura podataka između različitih modela, kao i prilagođavanje komunikacionih protokola koje sistemi koriste. Na taj način, svaki sistem može da koristi sopstveni interni format i tehnologiju, dok ESB obezbeđuje da razmena podataka bude konzistentna i razumljiva svim učesnicima.

Pored transformacije podataka, ESB često obezbeđuje i dodatne funkcionalnosti kao što su rutiranje poruka na osnovu poslovnih pravila, orkestracija jednostavnijih tokova komunikacije, kontrola pristupa, kao i nadzor i evidentiranje razmene poruka. Ove funkcionalnosti omogućavaju centralizovano upravljanje integracijom i olakšavaju dijagnostiku i održavanje sistema.



Slika 9.14. Položaj ESB

U kontekstu servisno orijentisane arhitekture, ESB se može posmatrati kao komunikaciona kičma koja povezuje servise i aplikacije u jedinstveno integraciono okruženje. Time se stvara osnova za dalji razvoj složenijih integracionih scenarija, kao što su orkestracija poslovnih procesa pomoću BPM alata i njihovo modelovanje BPMN notacijom. Na ovaj način, ESB ne predstavlja samo tehničko rešenje za razmenu poruka, već i ključni element koji povezuje tehničku infrastrukturu sa poslovnim procesima organizacije.

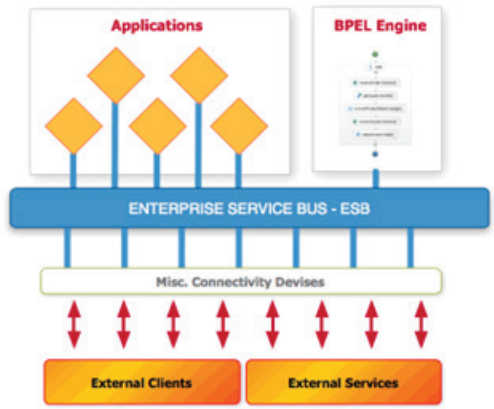
9.3.2.Arhitektura i osnovne funkcije magistrale poslovnih servisa - ESB

SOI uključuje integraciju višestrukih internih i spoljnih (eksternih) sistema kako bi se zadovoljili poslovni ciljevi:

- **Integracija informacija** – uključuje podatke kompanije i dr. izvora
- **Integracija procesa** – podrazumeva integraciju kroz koreografiju procesa i servisa sa višestrukim interfejsima
- **Integracija aplikacija** – uključuje razvoj novih interfejsa
- **Integracija sistema** – podrazumeva povezanost heterogenih i *legacy* sistema

Integraciju heterogenih sistema i servise za posredovanje između njih, omogućava *magistrala poslovnih servisa* (*Enterprise Service Bus*, ESB). ESB nudi posredničke servise koji obezbeđuju:

- Integraciju baza podataka i *legacy* sistema
- Mapiranje servisa
- Pretvaranje komunikacionih protokola
- Mapiranja između jezika interfejsa kod poziva servisa
- Inteligentno rutiranje
- Provera poruka, QoS, logovanje, merenje, monitoring i druge funkcionalnosti

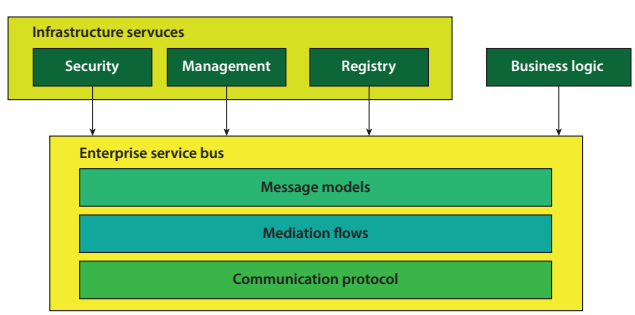


Slika 9.15. Realizacija SOI pomoću ESB

ESB funkcioniše kao posrednik koji povezuje različite aplikacije, servise i baze podataka, rešavajući nekompatibilnosti u formatima podataka, protokolima i komunikacionim modelima.

Arhitektura ESB-a obično se sastoji od sledećih ključnih komponenti:

1. **Konektori i adapteri** – omogućavaju povezivanje različitih sistema i aplikacija, transformišući podatke iz jednog formata u drugi i prilagođavajući protokole komunikacije.
2. **Posrednički servisi** (*Mediators*) – implementiraju logiku transformacije i rutiranja poruka između servisa, čime se rešavaju razlike u strukturi i semantici podataka.
3. **Ruter poruka** (*Message Router*) – upravlja tokovima poruka i omogućava dinamičko usmeravanje poruka ka odgovarajućim servisima na osnovu definisanih pravila.
4. **Transformatori podataka** (*Data Transformers*) – konvertuju podatke između različitih formata i standarda kako bi različiti sistemi mogli efikasno da komuniciraju.
5. **Servisi za upravljanje greškama i sigurnošću** – osiguravaju pouzdanu isporuku poruka, praćenje grešaka i kontrolu pristupa unutar integrisanog okruženja.



Slika 9.16. Osnovna arhitektura ESB-a

Osnovne funkcije ESB-a uključuju:

- **Integracija heterogenih sistema**: omogućava povezivanje aplikacija koje koriste različite tehnologije i protokole.
- **Transformacija i normalizacija podataka**: osigurava da podaci budu razumljivi i kompatibilni između različitih servisa.
- **Rutiranje poruka**: određuje kojim redosledom i kojim servisima će poruke prolaziti.
- **Centralizovana kontrola i praćenje**: omogućava nadzor i logovanje interakcija između servisa, olakšavajući održavanje i dijagnostiku.
- **Podrška za skalabilnost i fleksibilnost**: olakšava dodavanje novih servisa i promena u poslovnim procesima bez značajnih promena u postojećoj infrastrukturi.

Kroz ove funkcionalnosti, ESB omogućava organizacijama da smanje složenost integracije, poboljšaju interoperabilnost i ubrzaju adaptaciju poslovnih procesa na promene tržišnih zahteva.

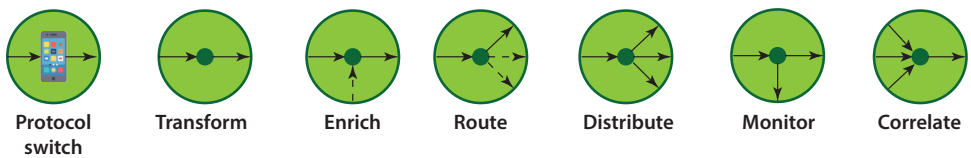
9.3.3. Obrasci posredovanja u ESB arhitekturi

U okviru ESB-a **posredovanje** (*mediation*) predstavlja ključni mehanizam koji omogućava labavu povezanost (*loose coupling*) između heterogenih sistema. Prema standardnim obrascima integracije preduzeća (*Enterprise Integration Patterns*), ovi mehanizmi rešavaju probleme nekompatibilnosti protokola, formata podataka i logike adresiranja.

Glavni obrasci posredovanja obuhvataju:

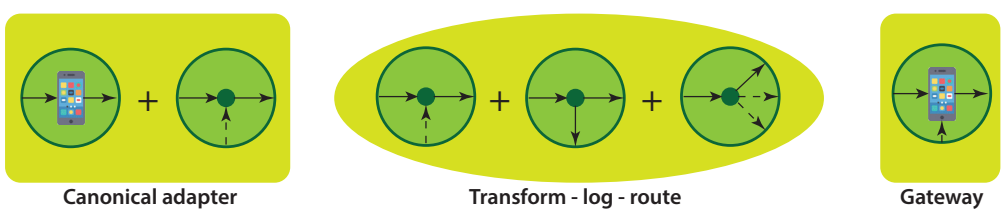
- **Konverzija protokola** (*Protocol Switch*): Sposobnost prihvatanja jednog tipa protokola i komunikacije kroz drugi protokol sa provajderom servisa. Ovo omogućava korisniku da pošalje poruku koristeći različite API-je (npr. SOAP/HTTP, JMS ili MQI), dok ESB vrši prevođenje u protokol koji provajder podržava (npr. RMI/IIOP). Na primer, konverzija iz **MQTT** protokola (koji koriste IoT senzori) u **REST/HTTP** pozive ka cloud servisima ili prevođenje **gRPC** saobraćaja u nasleđeni **SOAP** format za komunikaciju sa bankarskim mainframe sistemima.
- **Transformacija** (*Transform*): Prevođenje sadržaja poruke iz šeme korisnika u šemu provajdera. ESB konvertuje strukturu i format zahteva kako bi osigurao semantičku usaglašenost između različitih tehnoloških stakala. Na primer, automatsko mapiranje **JSON** objekata sa mobilnih aplikacija u **XML** format (npr. za COBOL aplikacije) ili transformacija binarnih podataka sa industrijskih mašina u čitljiv tekstualni format za potrebe analitike.
- **Obogaćivanje poruke** (*Enrich*): Povećavanje sadržaja poruke dodavanjem informacija iz spoljnih izvora podataka. Posrednik presreće dolaznu poruku i dopunjuje je parametrima definisanim putem upita baze podataka ili eksternih API-ja. Na primer, **geolokacijsko obogaćivanje**, gde sistem presreće ID korisnika i, pozivajući eksterni GPS servis, dodaje koordinate u poruku pre nego što je prosledi službi za logistiku.

- **Rutiranje** (*Route*): Dinamičko upravljanje putanjom poruke biranjem između više provajdera servisa koji podržavaju traženu funkcionalnost. Odluka se donosi na osnovu sadržaja same poruke (Content-Based Routing) ili predefinisanih poslovnih pravila. Na primer, **A/B testiranje**, gde se određeni procenat saobraćaja rutira ka novoj verziji mikroservisa, dok ostatak ostaje na stabilnoj verziji, bez potrebe za rekonfiguracijom klijentskih aplikacija.
- **Distribucija** (*Distribute*): Distribuiranje poruke ka svim zainteresovanim stranama. Ovaj obrazac omogućava da jedna informacija bude isporučena većem broju podsistema istovremeno (često kroz Publish-Subscribe model).
- **Monitoring** (*Monitor*): Posmatranje poruka koje prolaze kroz posredničke servise bez menjanja njihovog sadržaja. Služi za reviziju (audit), merenje performansi sistema i osiguranje ugovorenog nivoa kvaliteta usluge (SLA).



Slika 9.17. Posrednički servisi

Posrednički servisi se mogu kombinovati u složene sekvence kako bi se realizovala kompleksna posredovanja.



Slika 9.18. Kombinovanje posredničkih servisa

Uvođenje ovih obrazaca eliminiše "špageti" arhitekturu (*Point-to-Point*) i zamenjuje je uređenim sistemom, čime se postiže:

1. **Dekaplovanje** (*Decoupling*): Razdvajanje direktnih veza između servisa i interfejsa.
2. **Agilnost**: Dinamička zamena i skaliranje servisa bez prekida rada celog sistema.
3. **Ponovna upotrebljivost**: Lakše pronalaženje i korišćenje postojećih interfejsa za nove poslovne procese.

Rezultat je veća poslovna odzivnost i značajno smanjenje troškova održavanja kompleksnih IT ekosistema.

Kako bi integraciona infrastruktura bila robusna, posrednički servisi moraju zadovoljiti stroge kriterijume **kvaliteta usluge** (*Quality of Service*, QoS):

1. **Bezbednost podataka**: Implementacija enkripcije i dekripcije podataka u tranzitu, uz strogu kontrolu pristupa i politike autentifikacije.
2. **Pouzdanost** (*Reliable Messaging*): Garantovanje isporuke poruka čak i u uslovima nestabilne mrežne veze putem mehanizama potvrde i redova čekanja.
3. **Upravljanje greškama** (*Fault Handling*): Standardizovano rukovanje izuzecima i detekcija nepravilnosti u komunikaciji.

9.3.4. Od modela do izvršenja: Uloga orkestracije

U kontekstu upravljanja poslovnim procesima (BPM), SOI obezbeđuje tehničku osnovu za realizaciju poslovnih procesa kroz kompoziciju servisa. Poslovni procesi se ne posmatraju kao monolitne aplikacije, već kao koordinisani skup interoperabilnih servisa koji zajedno ostvaruju poslovni cilj.

Modelovanje servisno-orijentisane integracije započinje definisanjem orkestracije servisa. Orkestracija predstavlja mehanizam kojim se konceptualni, vizuelni modeli poslovnih procesa – najčešće izraženi pomoću BPMN notacije – transformišu u formalni, izvršni opis koji može biti automatski interpretiran od strane softverskog sistema. U tom smislu, orkestracija predstavlja ključnu sponu između faze dizajna i faze izvršavanja poslovnih procesa.

Ključni aspekti modelovanja integracije putem orkestracije obuhvataju:

- **Upravljanje redosledom** (*Sequencing*): Definisanje preciznog niza aktivnosti i njihovog međusobnog odnosa (na primer, prvo pribavljanje izveštaja kreditnog biroa, a zatim donošenje odluke o odobrenju kredita).
- **Upravljanje stanjem** (*State Management*): Sposobnost sistema da čuva i prati podatke o toku izvršavanja procesa, što je posebno važno kod dugotrajnih procesa koji mogu trajati satima, danima ili duže.
- **Korelacija**: Mehanizam kojim se obezbeđuje da se dolazne poruke i odgovori eksternih servisa povežu sa tačno određenom instancom procesa koja je u toku.

Razlikuju se dva osnovna modela kompozicije servisa: orkestracija i koreografija:

- **Orkestracija** (*Orchestration*) fokusira se na interne, privatne poslovne procese koji se odvijaju unutar jedne particije (Pool). U ovom pristupu postoji centralni proces, odnosno orkestrator, koji upravlja celokupnim tokom izvršavanja, poziva eksterne servise i koordinira razmenu podataka. Orkestracija se realizuje pomoću izvršnih jezika, kao što je BPEL, zbog čega se često poistovećuje sa izvršnim procesima.
- **Koreografija** (*Choreography*) opisuje saradnju između više autonomnih učesnika u okviru globalnih, kolaborativnih procesa u kojima učestvuje više nezavisnih partnera. Fokus koreografije nije na centralnoj kontroli, već na definisanju koordinisanih interakcija i razmene poruka između dve ili više strana. Koreografija se često primenjuje u B2B (*Business to Business*) scenarijima i predstavlja opis saradnje između različitih organizacija ili informacionih sistema.

Jezik za izvršavanje poslovnih procesa

U okviru servisno-orijentisane arhitekture, poslovni procesi se realizuju kroz kompoziciju servisa. Da bi se takvi procesi mogli automatski izvršavati, neophodan je standardizovan izvršni jezik koji precizno definiše redosled, uslove i način pozivanja servisa. Taj standard je WS-BPEL (*Web Services Business Process Execution Language*).

BPEL (*Business Process Execution Language*) predstavlja standardizovani izvršni jezik za definisanje i automatizaciju poslovnih procesa zasnovanih na veb servisima. BPEL je prvobitno razvijen u saradnji kompanija BEA, IBM, Microsoft, SAP i Siebel Systems, dok se danas nalazi pod okriljem OASIS konzorcijuma, koji razvija i održava niz relevantnih industrijskih standarda.

BPEL je XML-zasnovan jezik orkestracije, čija je osnovna namena da opiše način povezivanja i koordinacije veb servisa radi implementacije složenijih poslovnih funkcionalnosti. On omogućava integraciju eksternih servisa, kao i uključivanje interakcija sa ljudima, pri čemu je fokus stavljen na definisanje kontrolne logike poslovnog procesa. BPEL modeli se interpretiraju i izvršavaju pomoću specijalizovanih izvršnih okruženja, tzv. BPEL *engine*-a.

Ključni elementi BPEL modela

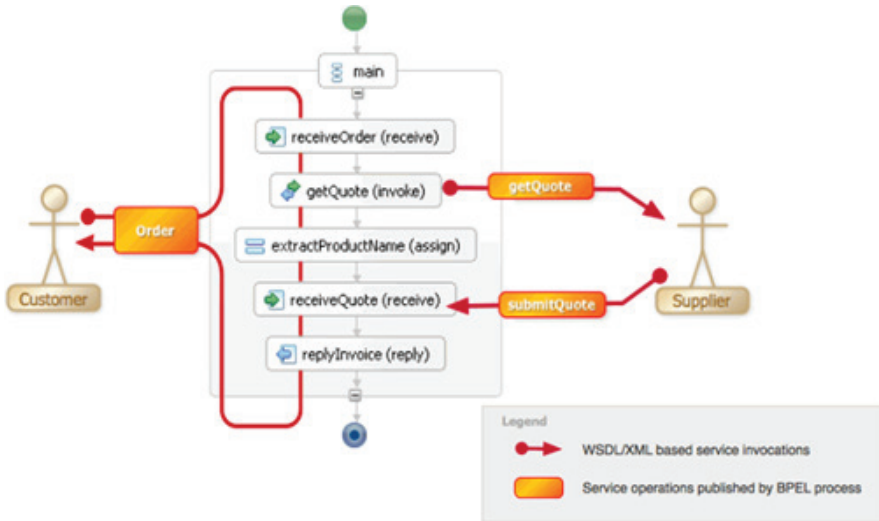
BPEL model se zasniva na nekoliko ključnih koncepata koji omogućavaju servisno-orijentisanu integraciju:

- ***Partner Links*** (partnerske veze): Definišu uloge i interfejse u interakciji između poslovnog procesa i spoljnih entiteta, kao što su eksterni servisi ili aplikacije (npr. proces obrade zahteva za kredit i servis kreditnog biroa).
- ***Variables*** (varijable): Služe za čuvanje podataka o stanju procesa i poruka koje se razmenjuju između servisa. Njihova struktura je definisana pomoću XML *Schema* specifikacija.
- **Osnovne aktivnosti**: Obuhvataju prijem poruka (receive), pozivanje eksternih servisa (invoke), obradu i dodelu podataka (assign) i slanje odgovora (reply).
- **Strukturne aktivnosti**: Omogućavaju definisanje složenijeg toka izvršavanja i direktno mapiranje BPMN elemenata u izvršnu logiku, uključujući sekvencijalni tok (sequence), paralelno izvršavanje (flow) i uslovno grananje (if–then–else).

U okviru definisanja poslovnih procesa pomoću BPEL jezika, koriste se specifične aktivnosti koje omogućavaju komunikaciju sa eksternim servisima i upravljanje podacima unutar samog procesa. Ključne operacije obuhvataju:

- **Aktivnost *Receive*** (Prijem): Predstavlja ulaznu tačku procesa koja blokira dalje izvršavanje dok ne stigne odgovarajuća poruka. Ova aktivnost je ključna za pokretanje nove instance poslovnog procesa nakon prihvatanja eksternog zahteva.
- **Aktivnost *Invoke*** (Poziv): Koristi se za sinhrono ili asinhrono pozivanje operacija koje obezbeđuju poslovni partneri. Kroz ovu aktivnost, proces komunicira sa drugim WSDL servisima, šalje potrebne podatke i, u slučaju sinhronog poziva, prikuplja povratne informacije (odzivne poruke).

- **Aktivnost *Assign*** (Dodela): Služi za manipulaciju podacima unutar procesa. Ona omogućava ažuriranje vrednosti varijabli, kao i kopiranje podataka iz jedne varijable u drugu (npr. iz ulazne poruke u format potreban za poziv sledećeg servisa).
- **Aktivnost *Reply*** (Odgovor): Označava završetak interakcije slanjem odzivne poruke klijentu koji je inicirao proces. Koristi se isključivo u kombinaciji sa aktivnošću *Receive* kako bi se zatvorio ciklus zahtev-odgovor (request-response).



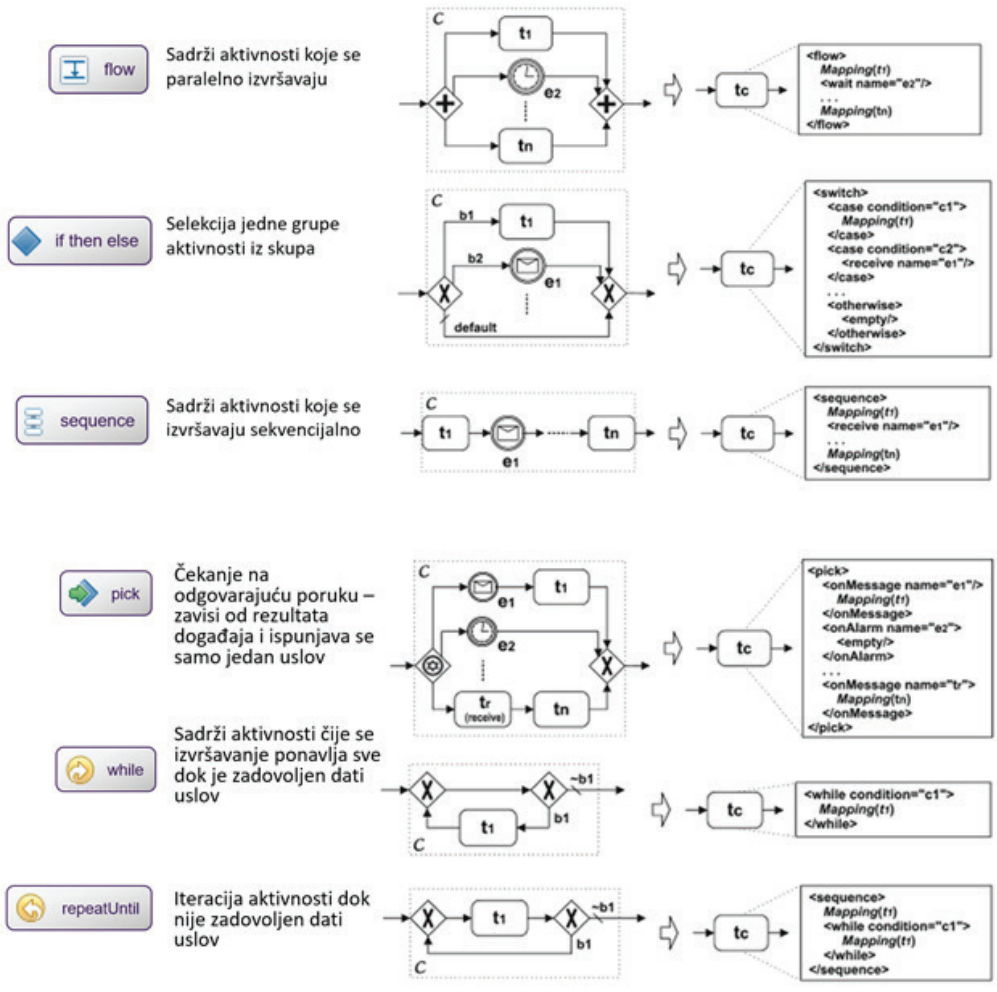
Slika 9.19. Osnovne aktivnosti BPEL jezika

Dok osnovne aktivnosti definišu pojedinačne korake u razmeni poruka, strukturne aktivnosti u BPEL jeziku omogućavaju definisanje složene logike procesa. One služe kao "lepak" koji povezuje osnovne aktivnosti u smislene celine, kontrolišući redosled i način njihovog izvršavanja. Razlikujemo tri ključne strukturne aktivnosti koje definišu kontrolu toka unutar orkestracije servisa:

- *Flow* (Paralelno izvršavanje): Ova aktivnost se koristi za definisanje skupa operacija koje se mogu izvršavati istovremeno. Ona omogućava paralelizam u procesu, što je od ključnog značaja za performanse sistema kada koraci nisu međusobno zavisni. Na dijagramu je to predstavljeno grananjem koje se kasnije sinhronizuje.
- *If-Then-Else* (Uslovno grananje): Omogućava selekciju jedne specifične grupe aktivnosti iz skupa ponuđenih, na osnovu definisanog uslova (tzv. switch logika u XML kodu). Ovo je osnovni mehanizam donošenja odluka unutar poslovnog procesa, gde se tok menja u zavisnosti od ulaznih podataka ili rezultata prethodnih servisa.
- *Sequence* (Sekvencijalno izvršavanje): Najjednostavnija, ali i najučestalija struktura koja osigurava da se aktivnosti izvršavaju strogo jedna za drugom, definisanim redosledom. Svaka aktivnost mora biti uspešno završena pre nego što sledeća započne, što garantuje integritet logičkog toka.

Kao što se vidi na slici 9.22 svaka grafička reprezentacija strukturne aktivnosti ima svoj direktni ekvivalent u XML/BPEL kodu. Ova preslikavanja omogućavaju da se vizuelno modelovan poslovni proces automatski prevede u izvršni kod koji ESB infrastruktura može da interpretira i sprovede.

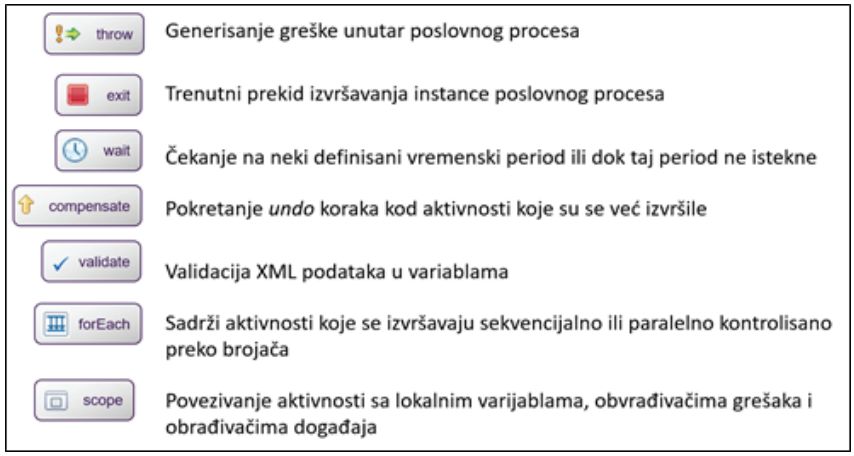
Pravilnom kombinacijom ovih struktura postiže se visoka fleksibilnost u dizajniranju servisa, što direktno doprinosi većoj poslovnoj responzivnosti (Greater business responsiveness), jer omogućava brzu adaptaciju procesa na nove zahteve tržišta.



Slika 9.20. Strukturne aktivnosti BPEL jezika

Pored osnovnih i strukturnih elemenata, BPEL nudi skup specijalizovanih aktivnosti za precizno upravljanje izvršavanjem:

- **Throw i Exit**: Služe za generisanje grešaka unutar procesa ili trenutni prekid instance.
- **Wait**: Omogućava pauziranje procesa na definisani vremenski period.
- **Compensate**: Pokreće *undo* korake za aktivnosti koje su već uspešno izvršene, što je ključno za transakcioni integritet.
- **Validate**: Vrši proveru XML podataka unutar varijabli procesa.
- **ForEach i Scope**: Omogućavaju kontrolisano ponavljanje aktivnosti preko brojača, odnosno logičko grupisanje aktivnosti sa lokalnim varijablama i obrađivačima grešaka.



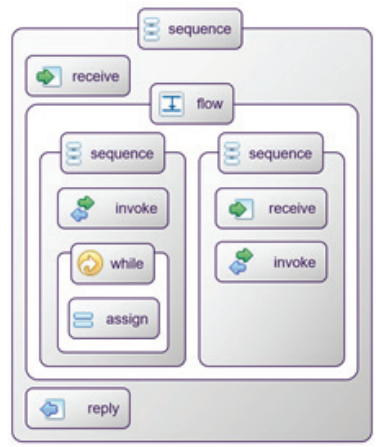
Slika 9.21. Pregled pomoćnih i kontrolnih BPEL aktivnosti za upravljanje procesima

Složenost savremenih poslovnih procesa zahteva mogućnost kombinovanja različitih logičkih celina unutar jedinstvene orkestracije. BPEL jezik omogućava **gneždenje struktura**, gde jedna strukturna aktivnost (poput sekvence) može unutar sebe sadržati druge kompleksne aktivnosti (poput paralelnih tokova ili petlji). Slika 9.23 ilustruje kako se apstraktni model procesa prevodi u izvršnu XML strukturu:

- **Hijerarhija elemenata**: Na dijagramu vidimo korensku aktivnost <sequence> koja definiše strogi redosled izvršavanja. Unutar nje se nalazi aktivnost <flow>, koja omogućava da se dva različita pod-procesa (leve i desne sekvence) izvršavaju paralelno.
- **Upotreba petlji**: Unutar leve grane paralelnog toka primenjena je aktivnost <while>, koja omogućava ponavljanje operacije dodele vrednosti (<assign>) sve dok je određeni uslov ispunjen.
- **Logički integritet**: Ovakav način modelovanja osigurava da se krajnji odgovor klijentu (<reply>) pošalje tek nakon što se sve unutrašnje, paralelne i sekvencijalne aktivnosti uspešno završe.

Ovaj nivo struktuiranja direktno doprinosi **većoj poslovnoj responzivnosti** (Greater business responsiveness), jer omogućava programerima i poslovnim analitičarima da precizno modeluju kompleksne scenarije realnog poslovanja, uz zadržavanje jasne preglednosti koda i logike.

```
<sequence>
   <receive .../>
   <flow>
      <sequence>
         <invoke .../>
         <while ... >
            <assign>...</assign>
         </while>
      </sequence>
      <sequence>
         <receive .../>
         <invoke ... >
      </sequence>
   </flow>
   <reply>
</sequence>
```



Slika 9.22. Ugnježdena struktura

Modelovanje poslovnih procesa u BPEL-u ne svodi se isključivo na definisanje redosleda aktivnosti, već uključuje i obezbeđivanje odgovarajućeg kvaliteta servisa (Quality of Service, QoS). To podrazumeva bezbednosne mehanizme za autentifikaciju i autorizaciju, pouzdanu razmenu poruka, kao i očuvanje transakcionog integriteta procesa. Poseban značaj ima korelacija poruka, koja omogućava pravilno povezivanje dolaznih poruka sa odgovarajućim instancama procesa koje su u toku.

BPMN i BPEL imaju komplementarne uloge u savremenim BPM okruženjima. BPMN je prvenstveno namenjen vizuelnom modelovanju poslovnih procesa na višem nivou apstrakcije, sa fokusom na razumevanje procesa od strane poslovnih analitičara i drugih zainteresovanih strana. Nasuprot tome, BPEL predstavlja izvršni jezik koji omogućava preciznu, formalnu specifikaciju i automatizaciju procesa.

U praksi se često primenjuje transformacija BPMN modela u BPEL modele, čime se premošćuje jaz između konceptualnog dizajna i tehničkog izvršavanja poslovnih procesa. Zbog toga mnogi softverski alati razdvajaju okruženja za modelovanje i okruženja za izvršavanje, omogućavajući simulaciju i analizu poslovnih tokova pre njihove implementacije u produkcionom sistemu.

Savremeni alati i platforme za orkestraciju procesa

Iako je WS-BPEL postavio temelje za automatizaciju servisa, današnje tržište se podelilo na evoluirana "enterprise" rešenja i moderna "cloud-native" rešenja koja često kombinuju BPEL sa BPMN 2.0 standardom.

1. **Vodeća komercijalna rešenja** (*Enterprise*) - Ovi alati predstavljaju direktnu evoluciju softvera navedenih u ranijoj literaturi:
   - **Oracle SOA Suite** (*BPEL Process Manager*): I dalje najrelevantniji alat za čistu BPEL orkestraciju. Danas je integrisan sa *Oracle Integration Cloud* (OIC), omogućavajući povezivanje lokalnih (*on-premise*) sistema sa cloud aplikacijama.
   - **IBM Business Automation Workflow**: Naslednik *WebSphere Process Servera*. IBM je integrisao BPEL motor u širu platformu koja podržava i složene ljudske interakcije i automatizovane servise.
   - **Microsoft Azure Logic Apps**: Moderni naslednik *BizTalk Servera*. Iako se BizTalk i dalje koristi, Microsoft primarno promoviše *Logic Apps* kao cloud servis za grafičko modelovanje procesa i integraciju hiljada različitih konektora.
2. **Vodeća rešenja otvorenog koda** (*Open Source*) - Savremeni razvoj se u velikoj meri oslanja na fleksibilne alate koji podržavaju mikroservisnu arhitekturu:
   - **Camunda**: Trenutno vodeća platforma za orkestraciju koja favorizuje BPMN 2.0 standard. Veoma je popularna u modernom razvoju jer omogućava programerima da upravljaju procesima kroz kod (Java, Spring Boot).
   - **WSO2 Enterprise Integrator**: Robustno rešenje koje sadrži namenski BPEL motor. Često se koristi u sistemima gde je neophodna visoka performansa i pouzdana razmena poruka između heterogenih servisa.
   - **Apache ODE** (*Orchestration Director Engine*): Najznačajniji preživeli čisti BPEL engine otvorenog koda koji služi kao osnova za mnoge druge integracione alate.
3. **Cloud-Native i "*Serverless*" orkestracija** - Najnoviji trend u industriji su alati koji apstrahuju XML i fokusiraju se na skalabilnost u oblaku:
   - **AWS Step Functions**: Servis kompanije Amazon koji omogućava orkestraciju Lambda funkcija i drugih AWS servisa koristeći JSON baziranu logiku, što je funkcionalni ekvivalent BPEL-u u serverless svetu.
   - **Temporal.io**: Moderni alat koji rešava probleme dugotrajnih procesa i rukovanja greškama direktno kroz programski kod (Java, Go, Python), eliminišući potrebu za XML vizuelnim modelovanjem.

9.4. MIKROSERVISNA ARHITEKTURA SOFTVERA

9.4.1. Uvod u mikroservisnu arhitekturu

Mikroservisna arhitektura predstavlja savremeni stil arhitekture razvoja softverskih rešenja u kome se aplikacija gradi kao skup malih, autonomnih i labavo povezanih servisa (Fowler & Lewis, 2014). Ovaj pristup evoluirao je iz potrebe za bržim razvojem i lakšim skaliranjem u cloud okruženju (Dragoni et al., 2017).

Za razliku od monolitnih aplikacija, gde je kompletna funkcionalnost objedinjena u jednoj izvršnoj celini, kao i klasičnih SOA rešenja zasnovanih na centralizovanoj orkestraciji, mikroservisi promovišu decentralizaciju, nezavisnost i jasnu podelu odgovornosti. Ovaj pristup se razvio kao odgovor na potrebe distribuiranih i *cloud*-orijentisanih sistema koji zahtevaju visoku skalabilnost, otpornost na greške i brzu isporuku softvera.

U mikroservisnoj arhitekturi svaki servis implementira jednu jasno definisanu poslovnu funkciju i može se razvijati, testirati i skalirati nezavisno od ostalih. Servisi komuniciraju preko mreže koristeći lagane protokole, pri čemu se koriste dva osnovna modela interakcije:

- **Sinhrona komunikacija** (*Request/Response*): Klijent šalje zahtev i čeka odgovor (najčešće putem REST ili gRPC protokola). Ovaj model je intuitivan, ali stvara vremensku zavisnost – ako ciljni servis nije dostupan, pozivajući servis ne može da završi operaciju.
- **Asinhrona komunikacija** (*Event-driven*): Servisi razmenjuju podatke putem poruka ili događaja koristeći posrednika (*Message Broker*) kao što su Apache Kafka ili RabbitMQ. Ovo obezbeđuje maksimalnu labavu povezanost (*loose coupling*), jer sistem funkcioniše čak i ako su određeni servisi privremeno nedostupni.

Mikroservisi obezbeđuju enkapsulaciju funkcionalnosti i podataka, pri čemu spoljni svet vidi servis kao "crnu kutiju". U tom kontekstu, formalna specifikacija ima centralnu ulogu i obuhvata:

- **Ugovor interfejsa** (API *contract*) – precizan opis operacija i parametara.
- **Preuslove i postuslove** – opis stanja pre i nakon izvršavanja operacije.
- **Ograničenja** (*constraints*) – pravila komunikacije.

Sa perspektive implementacije, svaki mikroservis je projektovan kao samostalna jedinica koja se tipično sastoji od tri komponente:

1. **Servisna logika** (*Service Logic*): Obuhvata poslovna pravila i procesiranje podataka.
2. **Servisni interfejs** (*Service Interface*): Definiše kanale komunikacije (npr. REST API), precizirajući protokole i formate (OpenAPI/JSON).
3. **Upravljanje podacima** (*Data Management*): Ključna karakteristika gde svaki servis poseduje sopstvenu bazu podataka (Database-per-Service), čime se osigurava potpuna autonomija.

Mikroservisi se najčešće izvršavaju u kontejnerskim okruženjima (npr. Docker), što predstavlja tehnologiju„pakovanja“ aplikacije i njenih zavisnosti (biblioteka i konfiguracija) u jedinstvenu, izolovanu jedinicu — kontejner. Ovakav pristup osigurava potpunu konzistentnost, garantujući da će softver raditi identično na računaru programera, u fazi testiranja i na produkcionom serveru. Pored toga, kontejnerizacija značajno olakšava automatizovano skaliranje i distribuciju mikroservisa na različite cloud platforme (Pahl, 2015).

9.4.2. Od klasičnih veb servisa do mikroservisa

Mikroservisna arhitektura se može posmatrati kao savremena varijanta servisno-orijentisane arhitekture (SOA), ali sa značajnim konceptualnim razlikama. Dok je klasična SOA često bila zasnovana na "krupnozrnim" (*coarse-grained*) servisima, složenim standardima i centralizovanoj orkestraciji, mikroservisi naglašavaju sitnu granulaciju, lagane protokole i decentralizovano upravljanje.

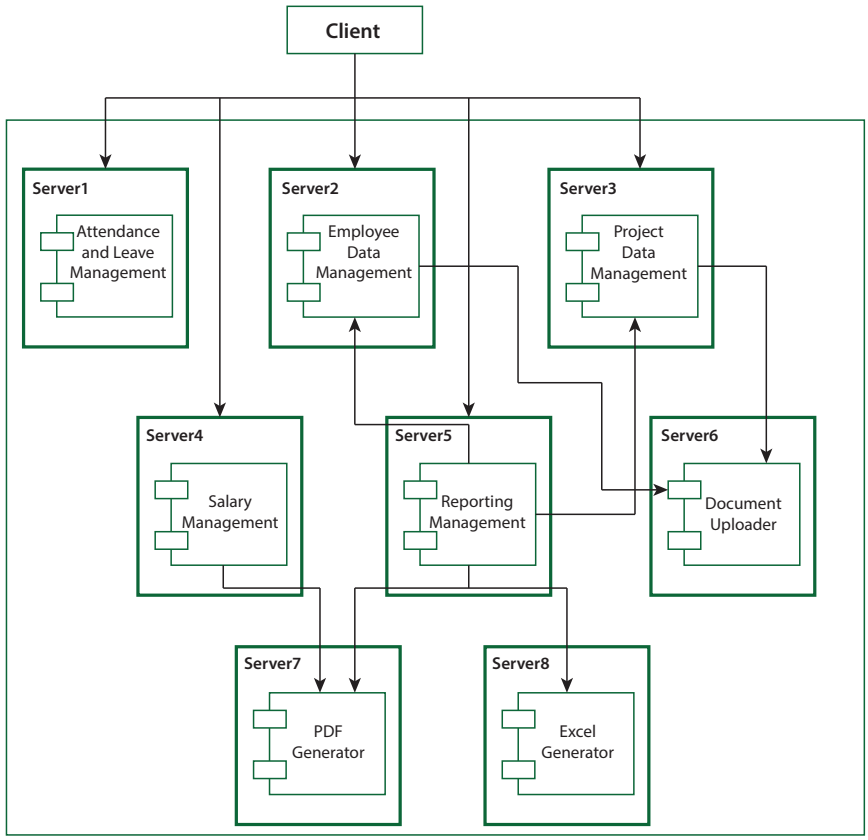
Tehnološka implementacija ovih servisa evoluirala je kroz dve glavne faze:

1. **Klasični veb servisi** (SOA koren): Oslanjaju se na stroge W3C standarde, koriste **SOAP** protokol i **XML** format poruka, uz **WSDL** za formalni opis interfejsa. Upravljanje se često vrši putem centralizovane orkestracije (npr. BPEL). Ovi sistemi su i danas dominantni u bankarskim i finansijskim sistemima gde su rigidna sigurnost i formalni ugovori prioritet.
2. **Mikroservisna arhitektura** (Savremeni pristup): Naglašava ekstremnu granulaciju gde je svaki servis mala, autonomna aplikacija fokusirana na jednu poslovnu funkciju. Komunikacija je bazirana na **REST** arhitekturalnom stilu i **JSON** formatu, a servisi su optimizovani za kontejnerska okruženja (npr. Docker) i DevOps prakse.

Poređenje dve dominantne tehnologije komunikacije u ovim fazama prikazano je u tabeli 9.3.

Tabela 9.3. Poređenje SOAP i REST pristupa u softverskoj arhitekturi

Karakteristika	**SOAP** (Klasični veb servisi)	**REST** (Mikroservisi)
Protokol	Strogi standard (W3C)	Arhitekturalni stil (HTTP)
Format poruke	Isključivo XML	Primarno JSON (lakši i brži)
Složenost	Visoka (zahteva specifične biblioteke)	Niska (koristi standardne HTTP metode)
Primena	Bankarski sistemi, visoka sigurnost	Web i mobilne aplikacije, cloud sistemi



Slika 9.23. Primer aplikacije bazirane na mikroservisnoj arhitekturi

9.4.3. Evolucija od SOA životnog ciklusa do DevOps ciklusa

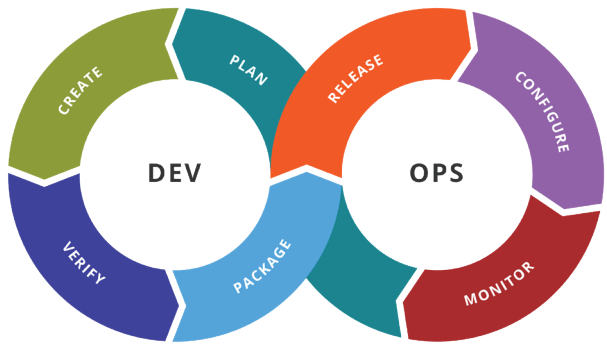
Prelazak sa monolitnih i SOA arhitektura na mikroservisnu arhitekturu zahtevao je promenu ne samo u tehnologiji, već i u načinu na koji razvojni i operativni timovi sarađuju. Ova promena je dovela do nastanka **DevOps** (skraćeno od *Development* i *Operations*) okvira. DevOps predstavlja skup praksi koji spaja razvoj softvera i IT operacije, sa ciljem skraćivanja životnog ciklusa razvoja sistema uz obezbeđivanje visoke učestalosti isporuka i visokog kvaliteta (Kim et al., 2016).

U svetu mikroservisa, gde sistem nije jedan blok koda već skup desetina nezavisnih jedinica koje komuniciraju preko mreže, DevOps postaje neophodnost iz sledećih razloga (Humble & Farley, 2010):

- **Kontinuirana integracija i isporuka** (CI/CD): Svaka izmena koju programer napravi u jednom mikroservisu automatski prolazi kroz seriju testova i, ako su uspešni, automatski se isporučuje u produkciju. To omogućava onaj "fleksibilan IT" o kojem smo govorili u uvodnom delu.

- **Automatizacija infrastrukture**: Mikroservisi se oslanjaju na *Cloud* tehnologije. DevOps timovi koriste "Infrastrukturu kao kod"[6] (Infrastructure as Code) kako bi brzo i bez grešaka podigli stotine novih instanci servisa.
- **Monitoring i opservabilnost**: Pošto je sistem distribuiran, DevOps uvodi alate koji u realnom vremenu prate rad svakog mikroservisa, omogućavajući timovima da otkriju grešku pre nego što je korisnik primeti (Bass et al., 2015).

Dok je tradicionalni SOA životni ciklus bio fokusiran na dugoročno planiranje i orkestraciju servisa, DevOps ciklus je kružan i beskonačan. On podrazumeva stalnu povratnu spregu: planiranje, kodiranje, izgradnju, testiranje, isporuku, rad, monitoring i ponovno planiranje na osnovu podataka iz realnog rada.



Slika 9.24. Životni ciklus DevOps-a: Kontinuirana petlja saradnje između razvojnog tima (Development) i tima za operacije (Operations) koja omogućava brzu i pouzdanu isporuku softvera

Tabela 9.4. Pregled faza i aktivnosti u okviru DevOps životnog ciklusa

Faza razvoja (Dev)	**Faza operacija** (Ops)
Planiranje (Plan): Definisanje zahteva i ciljeva na osnovu povratnih informacija.	**Puštanje u rad** (Release): Priprema stabilne verzije softvera za isporuku.
Kodiranje (Code): Pisanje izvornog koda i pregled (review) od strane kolega.	**Implementacija** (Deploy): Automatsko postavljanje koda na servere ili Cloud.
Izgradnja (Build): Kompajliranje koda i kreiranje izvršnih fajlova (npr. Docker image).	**Operativni rad** (Operate): Upravljanje softverom u realnom radnom okruženju.
Testiranje (Test): Automatska provera funkcionalnosti i kvaliteta koda.	**Monitoring** (Monitor): Praćenje performansi i detekcija grešaka u realnom vremenu.

6 Infrastruktura kao kod (*Infrastructure as Code - IaC*) je pristup u kojem se serverski resursi (*procesori, memorija, mreže*) definišu putem koda umesto ručnog podešavanja. Ovo omogućava DevOps timovima da automatski i za nekoliko minuta podignu identično okruženje za rad stotina mikroservisa.

Kao što je SOA imala svoj definisani ciklus (koji je bio postupan, faza po faza), moderna industrija je razvila životni ciklus razvoja mikroservisa koji je kontinuiran i koji se poistovećuje sa **DevOps petljom** (beskonačna petlja). Glavne razlike u fazama su:

- **Plan & Design** (kod SOA: Model): Umesto modelovanja celokupnog poslovnog procesa odjednom, fokus je na identifikaciji granica mikroservisa (primenom DDD-a). Dizajniraju se API ugovori (API-first design) pre pisanja koda, što omogućava timovima da rade paralelno i nezavisno.
- **Develop & Build** (kod SOA: Assemble): Komponente se više ne sklapaju na jedan centralni server. Svaki mikroservis se razvija u sopstvenom ritmu, koristi zasebnu bazu podataka i pakuje se u kontejner (npr. Docker).
- **CI/CD & Release** (kod SOA: Deploy): Ovo predstavlja najveću promenu. Dok je *Deploy* faza u SOA sistemima bila događaj koji se planirao nedeljama, danas je to potpuno automatizovan **CI/CD *pipeline***[7] (Continuous Integration/Continuous Deployment) koji omogućava trenutno testiranje i puštanje koda u rad.
- **Observe & Feedback** (kod SOA: Manage): U SOA sistemu se pratio jedan centralni procesni server. U mikroservisima je neophodno pratiti stotine distribuiranih jedinica. Fokus je na *Observability* alatima kao što su Prometheus i Grafana, koji u realnom vremenu šalju povratne informacije timu.

Zbog velike dinamike sistema, u ovoj fazi ključnu ulogu igra i **Service Discovery** – mehanizam koji omogućava servisima da automatski pronađu mrežne adrese drugih servisa u promenljivom cloud okruženju.

Tabela 9.5. Poređenje faza SOA i mikroservisnog ciklusa

SOA ciklus	Mikroservisni ciklus	Ključna promena
Model	*Plan & Design*	Fokus na male, ograničene kontekste (Bounded Contexts).
Assemble	*Build & Test*	Automatizovana izgradnja i kontejnerizacija (Docker).
Deploy	*Continuous Deployment*	Puštanje koda u rad na dnevnom nivou (CI/CD).
Manage	*Observe & Monitor*	Praćenje distribuiranog sistema i automatsko pronalaženje servisa.

7 CI/CD tok ili *pipeline* je automatizovani niz koraka kroz koji kod prolazi od trenutka kada ga programer završi do trenutka kada postane dostupan korisniku. On obuhvata faze *Build, Test* i *Deploy* kod DevOps okvira.

Danas DevOps nastavlja da se razvija kao kritična disciplina u okviru softverskog inženjerstva, prvenstveno podstaknut ubrzanim napretkom u oblastima automatizacije, bezbednosti i tehnologija izvornih za oblak (cloud-native). Integracija veštačke inteligencije (AI) i mašinskog učenja (ML) iz korena menja tradicionalne procese, omogućavajući primenu prediktivne analitike za efikasnije upravljanje incidentima. Savremeni sistemi postaju sposobni za automatsko generisanje testova i samostalno otklanjanje grešaka (self-healing), čime se vreme zastoja svodi na minimum. Ovaj prelaz ka DevOps-u zasnovanom na veštačkoj inteligenciji značajno unapređuje monitoring, optimizaciju resursa i rano otkrivanje bezbednosnih pretnji.

Bezbednost podataka i sistema ostaje prioritetni cilj, zbog čega DevSecOps dobija sve veći zamah kroz primenu strategije "pomeranja ulevo" (shift left security). Ova praksa podrazumeva identifikaciju ranjivosti već u najranijim fazama životnog ciklusa razvoja softvera, uz korišćenje automatizovanih bezbednosnih testova integrisanih direktno u CI/CD cevovode. Koncept "bezbednosti kao koda" (Security as Code) postao je standard u industriji, što znači da se bezbednosne polise tretiraju kao kôd pod verzijom, čime se garantuje konzistentnost i stroga usklađenost sa propisima. Dodatno, savremeni alati poput GitHub Advanced Security proširuju mogućnosti kroz poboljšanu vidljivost upozorenja i bolju integraciju sa radnim zadacima, što značajno ubrzava radne tokove.

U domenu upravljanja infrastrukturom, GitOps se profiliše kao vodeća paradigma koja koristi Git kao jedini izvor istine za celokupnu implementaciju. Ovakav pristup omogućava deklarativno upravljanje resursima, smanjuje rizik od nepredviđenih promena u konfiguraciji i značajno unapređuje saradnju između razvojnih i operativnih timova. Uporedo sa tim, primena serverless računarstva i cloud-native arhitektura omogućava inženjerima da se fokusiraju isključivo na logiku aplikacije, dok platforme vrše automatsko skaliranje resursa u zavisnosti od potreba.

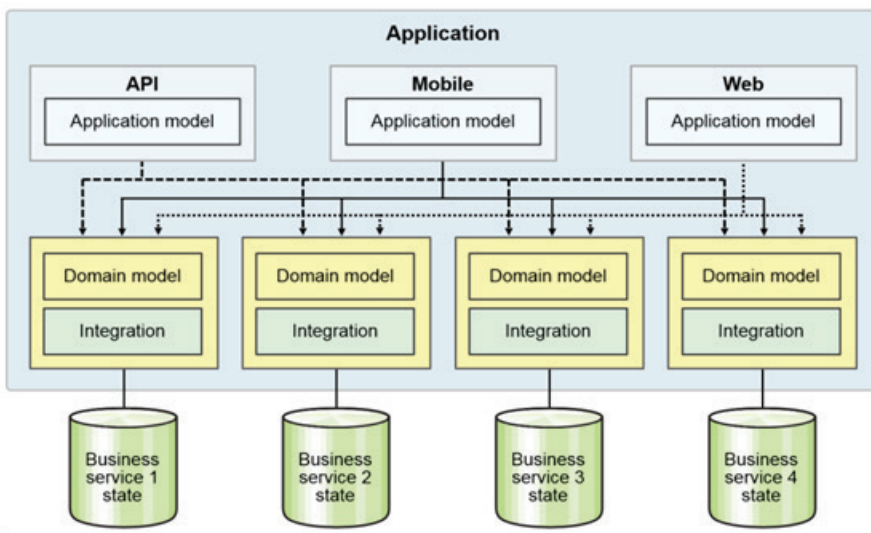
Iako se tehnologije menjaju, alati za infrastrukturu kao kôd (IaC) poput Terraform-a, Ansible-a i AWS CloudFormation-a ostaju temelj za kreiranje ponovljivih i stabilnih okruženja. Kontejnerizacija putem Docker-a i orkestracija pomoću Kubernetes-a[8] i dalje predstavljaju standard za mikroservisne aplikacije, dok platforme za opservabilnost, kao što su Prometheus i Grafana, postaju neophodne za nadzor u realnom vremenu. Od DevOps profesionalaca u 2025. godini očekuje se duboko poznavanje vodećih cloud platformi, automatizacije kroz GitHub Actions ili Jenkins, kao i vešto upravljanje podacima u SQL i NoSQL bazama. Uprkos povremenim debatama o zasićenosti tržišta, potražnja za stručnjacima koji podržavaju AI infrastrukturu ostaje izuzetno visoka, nudeći stabilne karijerne prilike fokusirane na kontinuirano unapređenje procesa.

8 Orkestracija pomoću Kubernetes alata služi kao "dirigent" za kontejnere. On automatski upravlja njihovim pokretanjem, skaliranjem (*povećavanjem broja instanci usled velikog saobraćaja*) i zamenom u slučaju kvara.

9.4.4. Metodološki pristup modelovanju mikroservisne arhitekture

Mikroservisna arhitektura se ne posmatra primarno kao implementacioni problem, već kao arhitektonski koncept koji se mora precizno modelovati pre same realizacije. Ključni izazov je identifikacija granica servisa i definisanje njihovih odgovornosti kako bi se izbegla tesna povezanost.

Modelovanje započinje na konceptualnom nivou primenom *Domain-Driven Design* (DDD) principa, gde se poslovni domen deli na granične kontekste (Bounded Contexts). Na logičkom nivou, ovi konteksti se transformišu u komponente sa precizno definisanim interfejsima, dok se na fizičkom nivou planira njihovo izvršavanje u distribuiranom okruženju (Slika 9.25).



Slika 9.25. Konceptualni model distribucije odgovornosti između klijentskih aplikacija i autonomnih biznis servisa (de la Torre et al, 2022)

U okviru **UML notacije**, mikroservisi se specifikuju kroz sledeće modele:

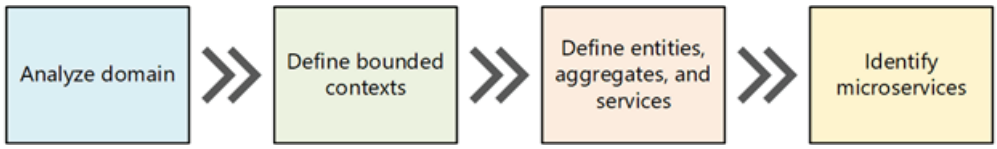
- **Dijagrami komponenti**: Prikazuju mikroservise kao nezavisne jedinice, njihove "ponuđene" i "zahtevane" interfejse.
- **Dijagrami sekvenci**: Ključni su za modelovanje interakcije (sinhrone ili asinhrone) između više servisa tokom jednog poslovnog procesa.
- **Dijagrami raspoređivanja** (Deployment diagrams): Mapiraju mikroservise na kontejnerske čvorove i cloud infrastrukturu.
- **Dijagrami stanja**: Koriste se za modelovanje životnog ciklusa ključnih objekata (npr. status porudžbine) unutar jednog servisa.

Poseban značaj ima modelovanje **API ugovora** (npr. putem OpenAPI specifikacije). Interfejs postaje centralni deo specifikacije, dok se detalji implementacije svesno izostavljaju kako bi se omogućio nezavisan razvoj timova. Za razliku od monolitnih sistema, modelovanje ovde mora predvideti i scenarije grešaka, kašnjenja mreže i delimičnu dostupnost.

Tipičan iterativni proces modelovanja obuhvata sledeće korake:

1. **Analiza zahteva i poslovnog domena**: Identifikacija ključnih procesa i aktera.
2. **Definisanje graničnih konteksta**: Podela domena na logičke celine sa specifičnom terminologijom.
3. **Mapiranje na mikroservise**: Određivanje koji kontekst postaje samostalan servis.
4. **Specifikacija interfejsa** (API): Precizno definisanje ulaza, izlaza i protokola komunikacije.
5. **Modelovanje ponašanja**: Upotreba dijagrama sekvenci za opisivanje scenarija interakcije.

Na primer, sistem za elektronsku trgovinu se može modelovati kroz servise: *Korisnici, Katalog, Porudžbine* i *Plaćanja*. Dok dijagram komponenti prikazuje njihovu strukturu, dijagram sekvenci može opisati tok gde servis *Porudžbine* proverava stanje u *Katalogu* i inicira *Plaćanje*.



Slika 9.26. Faze projektovanja mikroservisa primenom DDD metodologije

Uobičajene greške u modelovanju su:

- **Preuranjena dekompozicija**: Razbijanje sistema pre nego što se razume poslovni domen.
- **Mešanje modela i implementacije**: Unošenje konkretnih tehnologija u fazu arhitektonske specifikacije.
- **Loša granulacija**: Kreiranje servisa koji su previše krupni (skriveni monolit) ili previše sitni (preveliki mrežni trošak).
- **Zanemarivanje grešaka**: Pretpostavka da će mrežna komunikacija uvek biti savršeno pouzdana.

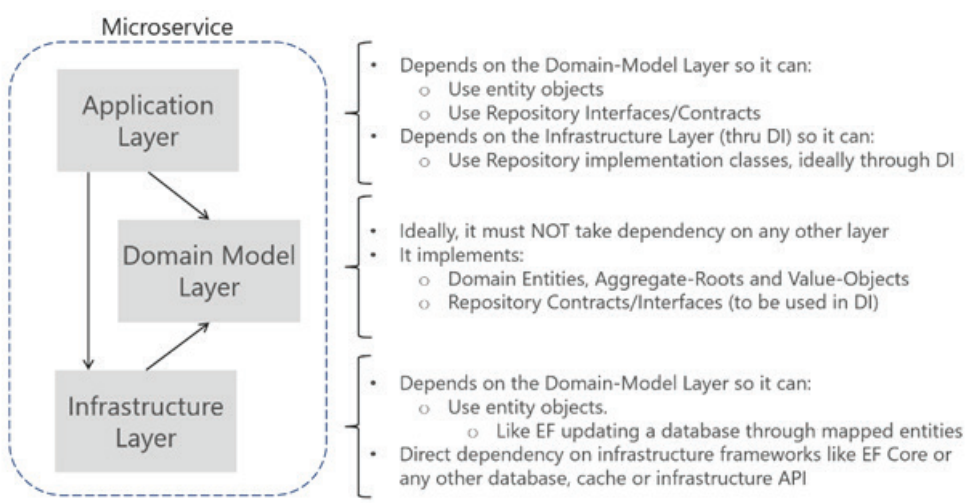
9.4.5. Proces projektovanja mikroservisa

Uspeh mikroservisne arhitekture direktno zavisi od pravilne dekompozicije sistema. Kako bi se izbeglo stvaranje „distribuiranih monolita" (sistema koji su fizički razdvojeni, ali logički neraskidivo vezani), primenjuje se **Dizajn vođen domenom** (*Domain-Driven Design*, DDD). Ovaj okvir osigurava da softverska struktura verno oslikava stvarne poslovne potrebe. Dizajn vođen domenom (DDD) postavlja fokus na razumevanje poslovnih procesa pre pisanja koda (Evans, 2003).

Proces projektovanja primenom DDD-a deli se na dve ključne ravni:

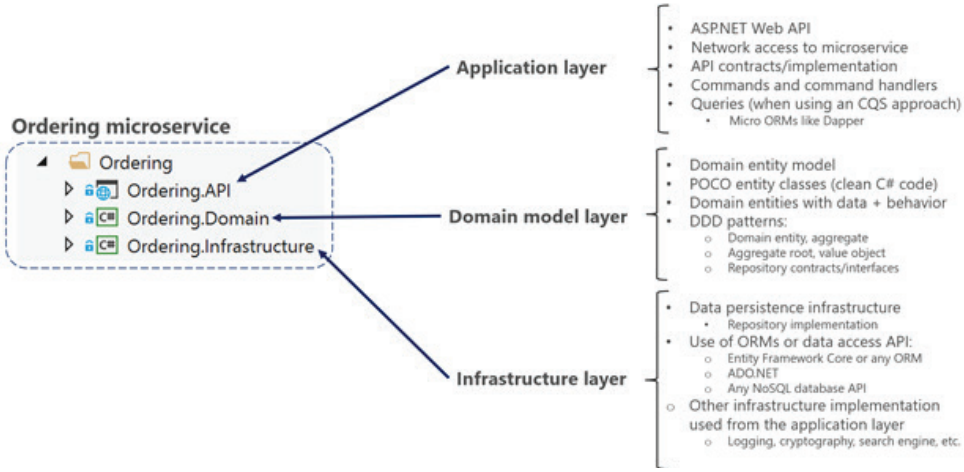
- **Strateški DDD**: Fokusira se na razumevanje šire slike, identifikaciju poslovnih mogućnosti i postavljanje granica sistema. Odgovara na pitanje: *Šta sistem radi i gde su mu granice?*
- **Taktički DDD**: Pruža konkretne dizajnerske obrasce za unutrašnje modelovanje svakog mikroservisa (Entiteti, Agregati, Domenski servisi). Odgovara na pitanje: *Kako ćemo unutar tih granica organizovati kod i podatke?*

Slika 9.27 prikazuje kako *Domain Model Layer* predstavlja centar mikroservisa, dok Application i Infrastructure slojevi zavise od njega, čime se postiže izolacija poslovne logike od tehničkih detalja implementacije.



Slika 9.27. Zavisnosti između slojeva unutar mikroservisa dizajniranog prema DDD principima (de la Torre et al, 2022)

Praktična primena slojevite arhitekture unutar jednog mikroservisa može se videti na primeru servisa za naručivanje (Slika 9.28). Ovde se jasno vidi kako se logička podela na slojeve preslikava na fizičku strukturu projekata u razvojnom okruženju, gde svaki sloj ima specifičnu odgovornost — od definisanja API ugovora do upravljanja perzistencijom podataka putem ORM (Object-Relational Mapping) alata.



Slika 9.28. Primer strukture projekta Ordering mikroservisa sa jasno razgraničenim slojevima: Application, Domain, Infrastructure (de la Torre et al, 2022)

Operativne faze projektovanja

Projektovanje mikroservisa se odvija kroz četiri sukcesivna koraka koji transformišu apstraktne poslovne zahteve u konkretnu arhitekturu:

1. Analiza domena (Problem Space)

Ovo je temelj procesa u kojem inženjeri i poslovni eksperti (Domain Experts) zajednički dekomponuju kompleksne procese. Cilj je uspostavljanje zajedničkog jezika (Ubiquitous Language) koji eliminiše nesporazume između programera i menadžera. Često se koriste tehnike poput Event Storming-a kako bi se vizuelno mapirali poslovni događaji hronološkim redom.

2. Definisanje graničnih konteksta (Bounded Contexts)

Ovo je ključni korak strateškog DDD-a. Granični kontekst predstavlja logičku granicu unutar koje model domena ima puno značenje. Na primer, pojam "Proizvod" u kontekstu Kataloga definisan je cenom i opisom, dok je u kontekstu Isporuke definisan težinom i dimenzijama. Umesto jednog glomaznog modela, kreiraju se dva odvojena konteksta, što je osnovni preduslov za nezavisnost servisa.

3. Definisanje entiteta, agregata i servisa (Taktička faza)

U ovoj fazi modeluje se unutrašnja anatomija svakog konteksta:

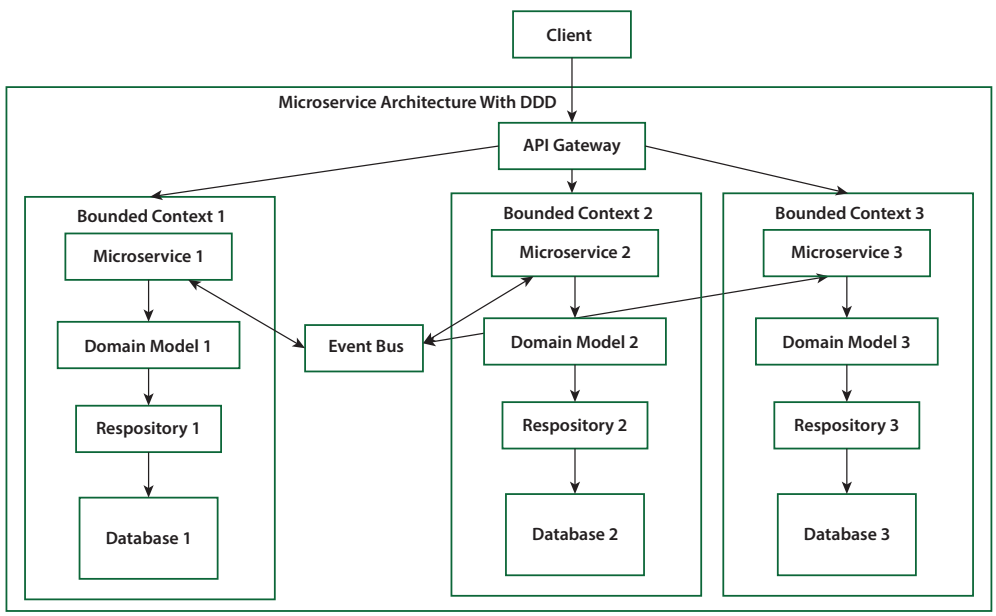
- **Entiteti**: Objekti sa jedinstvenim identitetom (npr. Korisnik sa ID brojem).
- **Agregati**: Skupovi povezanih objekata koji se tretiraju kao jedna jedinica za promenu podataka (npr. Narudžbenica sa svojim stavkama). Svaki agregat ima svoj koren (Aggregate Root) preko kojeg se vrši sva komunikacija.
- **Domenski servisi:** Poslovna logika koja prirodno ne pripada nijednom objektu (npr. složeni obračun poreza).

4. Identifikacija mikroservisa (Finalno mapiranje)

Teorijski modeli se pretvaraju u softversku arhitekturu. Najbolja praksa je mapiranje jedan kontekst = jedan mikroservis. Svaki identifikovani mikroservis dobija sopstveni sloj logike, API i sopstvenu bazu podataka (Database-per-service).

Svaki mikroservis treba da poseduje sopstvenu bazu podataka kako bi se osigurala labava povezanost (Richardson, 2018). Ovakav pristup zahteva poznavanje fundamentalnih obrazaca softverske arhitekture kako bi se izbegla prevelika kompleksnost sistema (Richards & Ford, 2020).

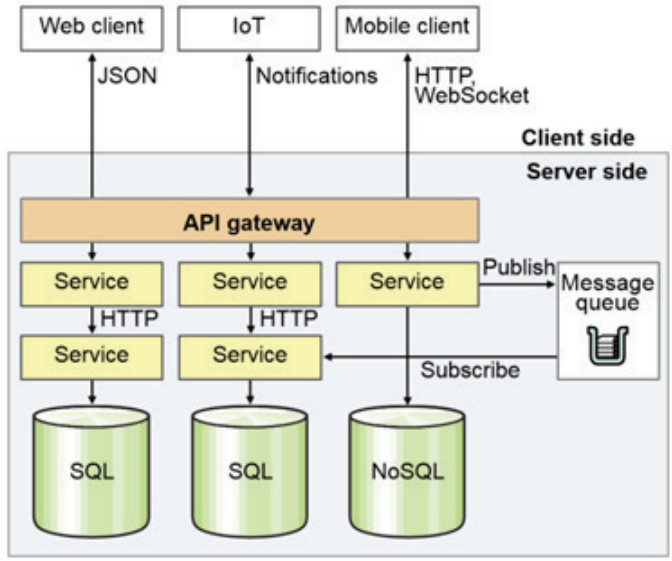
Konačna struktura sistema, prikazana na slici 9.29, ilustruje kako se pojedinačni mikroservisi (dizajnirani prema slojevima) integrišu u koherentnu celinu putem API Gateway-a i Event Bus-a, zadržavajući pritom potpunu autonomiju svojih podataka



Slika 9.29. Prikaz mikroservisne arhitekture zasnovane na DDD principima sa API Gateway-om i asinhronom komunikacijom

Dok prethodna slika 9.29 prikazuje čistu DDD strukturu sa graničnim kontekstima, slika 9.30 prikazuje **tehnološku realnost** ekosistema:

- Različite tipove klijenata (Web, IoT, Mobile).
- Različite tipove baza podataka (SQL i NoSQL), što dodatno objašnjava tvoj "Database-per-service" koncept.
- Mehanizam **Message Queue** (Publish/Subscribe), što vizuelno potvrđuje tvoju priču o asinhronoj komunikaciji.



Slika 9.30. Tehnološki ekosistem mikroservisne arhitekture sa heterogenim klijentima i mešovitim modelima skladištenja podataka (SQL/NoSQL) (de la Torre et al, 2022)

Tabela 9.6. Komparativna analiza: DDD mikroservisi nasuprot distribuiranih monolita

Karakteristika	**Dobri mikroservisi** (DDD pristup)	**Loši mikroservisi** (Distribuirani monolit)
Povezanost	**Labava** (Loosely coupled): Promena u jednom servisu ne zahteva izmene u drugima.	**Tesna** (Tightly coupled): Izmena jednog servisa izaziva "domino efekat" i zahteva ažuriranje celog sistema.
Baza podataka	**Database-per-service**: Svaki servis poseduje svoje podatke (enkapsulacija).	**Zajednička baza**: Više servisa direktno pristupa istim tabelama, stvarajući skrivene zavisnosti.
Kohezija	**Visoka**: Servis je fokusiran na jedan jasan poslovni kontekst (Bounded Context).	**Niska**: Servis obavlja previše nepovezanih zadataka ili je logika jednog procesa rasuta svuda.
Značenje podataka	Jasno definisano: Termini imaju specifično značenje unutar granica konteksta.	**Konfuzno**: Isti objekti se koriste za sve, što dovodi do semantičkih konflikata.
Skalabilnost	**Selektivna**: Skaliraju se samo servisi koji su pod opterećenjem.	**Otežana**: Zbog zavisnosti, često morate skalirati više servisa istovremeno.
Instalacija (Deployment)	**Nezavisna**: Servisi se puštaju u rad bez koordinacije sa drugim timovima.	**Orkestrirana**: Servisi se moraju instalirati tačno određenim redosledom da sistem ne bi pao.

Glavna opasnost u razvoju mikroservisa nije tehnološka, već arhitekturalna. Bez stroge primene granica (strateški DDD) i pravilnog modelovanja agregata (taktički DDD), timovi rizikuju stvaranje sistema koji je teži za održavanje od originalnog monolita.

9.4.6. Studija slučaja: Mikroservisna arhitektura sistema za dostavu dronom

Kompanija pokreće uslugu dostave dronovima, gde scenario uključuje čitav niz kompleksnih operacija, kao što su:

- Kupac zakazuje preuzimanje, nakon čega sistem dodeljuje dron i procenjuje vreme isporuke.
- Korisnik prati lokaciju drona uz realno ažuriranje vremena dolaska (ETA - Estimated Time of Arrival).
- Sistem upravlja korisničkim nalozima, paketima i vrši analizu istorijskih podataka radi optimizacije.

Ovi zahtevi, zajedno sa planovima za buduće proširenje na nova tržišta i uvođenje novih funkcionalnosti (poput deljenja dronova ili video nadzora), čine ovaj sistem idealnim za upotrebu mikroservisne arhitekture.

Analiza domena

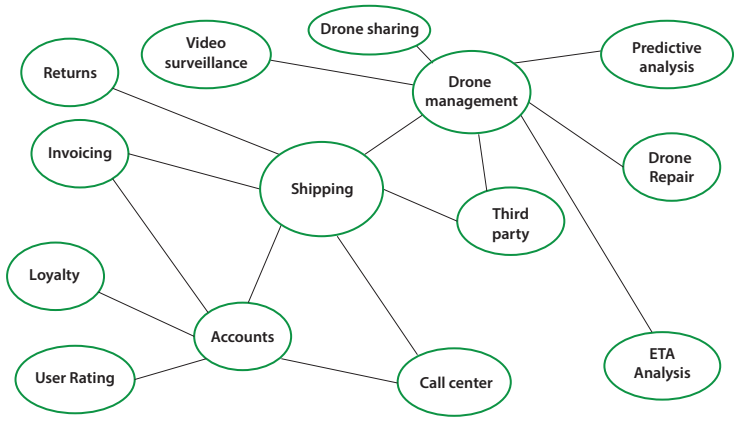
Proces počinje modelovanjem poslovnog domena. Cilj je kreiranje modela domena, koji predstavlja apstraktni prikaz poslovnih funkcija i njihovih interakcija, pre bilo kakvih odluka o tehnologiji. Tokom crtanja mape domena, fokus je na sledećim pitanjima:

- Koje funkcije predstavljaju srž poslovanja (Core Domain)?
- Koje funkcije su pomoćne prirode (Supporting Subdomains)?
- Gde je neophodna integracija sa eksternim sistemima trećih strana (npr. procesori plaćanja)?

Analizom domena, u ovoj studiji slučaja, identifikovane su ključne veze (Slika 9.33):

- Dostava (Shipping) je u centru jer predstavlja primarnu vrednost sistema.
- Upravljanje dronovima (Drone management) obuhvata održavanje, popravke i prediktivnu analizu kvarova.
- Korisnički nalozi (Accounts) podržavaju sisteme lojalnosti i fakturisanje (Invoicing).

U ovoj fazi se ne donose odluke o bazama podataka, već se definiše kako se biznis procesi preklapaju. Čak i funkcije koje neće biti u prvoj verziji, poput video nadzora, uključuju se u model kako bi arhitektura bila spremna za rast.



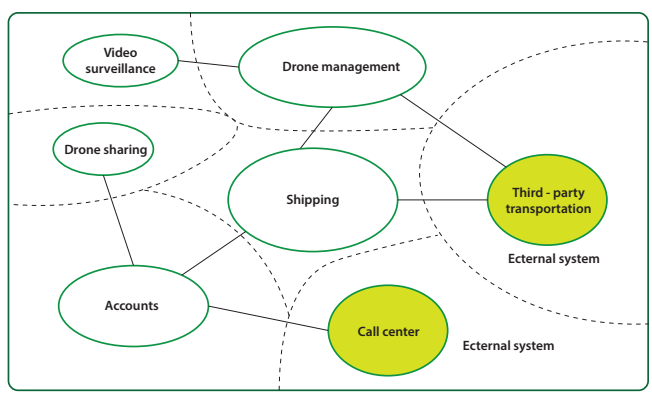
Slika 9.31. Dijagram modela domena za sistem dostave dronom

Definisanje graničnih konteksta

Model domena uključuje entitete iz stvarnog sveta (dron, paket, korisnik), ali oni ne moraju imati identičnu reprezentaciju u svakom delu sistema. Na primer, dron se posmatra na dva načina:

1. U kontekstu popravke i prediktivne analize, dron je definisan serijskim brojem, kilometražom, istorijom servisa i karakteristikama baterije.
2. U kontekstu zakazivanja dostave, dron je definisan isključivo statusom dostupnosti i lokacijom.

Ukoliko bismo koristili jedan "univerzalni" model drona, on bi bio previše složen za održavanje, a timovi bi stalno morali da usklađuju izmene polja koja im nisu potrebna. Rešenje je granični kontekst (Bounded Context), koji postavlja jasne granice unutar kojih jedan model ima specifično značenje (Slika 9.32).



Slika 9.32. Dijagram graničnih konteksta u domenu dostave dronom

Interakcije između konteksta se odvijaju putem dobro definisanih API-ja, koristeći dva ključna DDD obrasca:

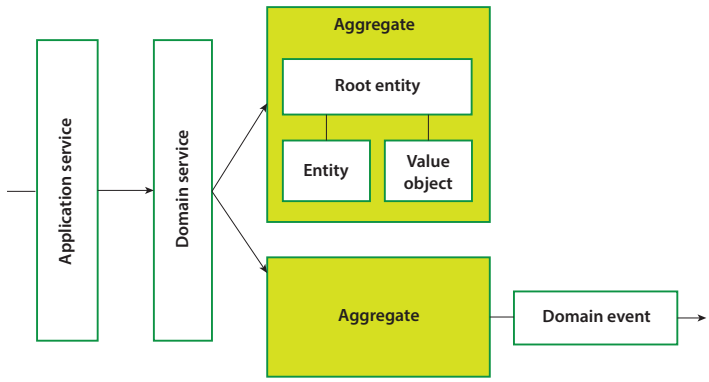
- *Open Host Service* (OHS): Servis definiše stabilan protokol preko kojeg mu drugi pristupaju.
- *Published Language* (PL): Objavljivanje zajedničkog formata podataka koji omogućava laku integraciju između različitih timova.

Ovim je završena strateška faza, a sistem je spreman za taktički DDD, gde se unutar svakog konteksta definišu entiteti i agregati.

Primena taktičkog DDD-a: Unutrašnji dizajn mikroservisa

Cilj taktičkog dizajna je identifikacija prirodnih granica unutar programskog koda. Osnovno pravilo je da mikroservis ne bi trebalo da bude manji od jednog agregata niti veći od jednog graničnog konteksta. Pre konkretne primene na sistem dronova, neophodno je razumeti četiri osnovna gradivna bloka:

- **Entiteti** (Objekti sa jedinstvenim identitetom) - To su objekti koje sistem prati putem njihovog jedinstvenog broja (ID-a), bez obzira na promenu njihovih osobina. Na primer: Isporuka je entitet. Čak i ako dva drona nose identične pakete na istu lokaciju, to su dve različite dostave jer svaka ima svoj jedinstveni *DeliveryID*. Isporuka ostaje isti entitet čak i kada promeni status iz "u toku" u "završena".
- **Vrednosni objekti** (Zamenljivi opisi) - Ovi objekti nemaju sopstveni identitet. Oni služe samo da opišu karakteristike nečega. Ako se promeni jedna vrednost u objektu, to se smatra potpuno novim objektom. Na primer: Lokacija ili Dimenzije Paketa. Lokacija (geografska širina i dužina) nema identitet; ona samo opisuje tačku na mapi. Ako dron skrene na nove koordinate, stara lokacija se jednostavno zamenjuje novom.
- **Agregati** (Sistemi "glavnog i podređenih" objekata) - Agregat je grupa povezanih entiteta i vrednosnih objekata koji čine granicu transakcione konzistentnosti. Svaki agregat ima korenski entitet (Aggregate Root) koji kontroliše sve izmene. Na primer: Porudžbina je agregat. Koren agregata je sama *Porudžbina*, dok su *Stavke porudžbine* podređeni entiteti. Ne možete dodati novu stavku direktno u bazu podataka; morate proći kroz "korenski entitet" (Porudžbinu) koji proverava pravila (npr. da li je porudžbina već zaključana) i ažurira ukupnu cenu. Ako obrišete porudžbinu, brišu se i sve njene stavke.
- **Domenski događaji** (Vesti unutar sistema) - To su zapisi o značajnim promenama stanja objekta koji služe za obaveštavanje drugih delova sistema. Na primer: Događaj "Dron je sleteo" (DroneLanded). Kada se ovaj događaj desi, servis za dostavu ga objavljuje na magistrali (sabirnici) poruka (Message Bus). Na tu "vest" reaguju drugi servisi: servis za obaveštenja šalje SMS kupcu, a servis za održavanje dronova beleži da je dron spreman za punjenje baterije.



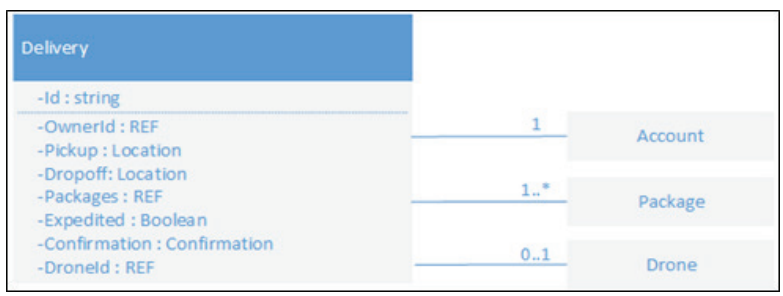
Slika 9.33. Hijerarhija i interakcija taktičkih DDD obrazaca

Modelovanje agregata i koordinacija

Na osnovu ovih principa, razvojni tim je identifikovao ključne agregate sistema:

1. Isporuka (Delivery): Glavni agregat koji sadrži reference na vlasnika, dron i lokacije preuzimanja i isporuke.
2. Paket (Package): Agregat koji definiše fizičke karakteristike robe i oznake (npr. barkod).
3. Dron (Drone): Agregat koji prati status drona u realnom vremenu.
4. Nalog (Account): Agregat koji upravlja podacima o korisniku.

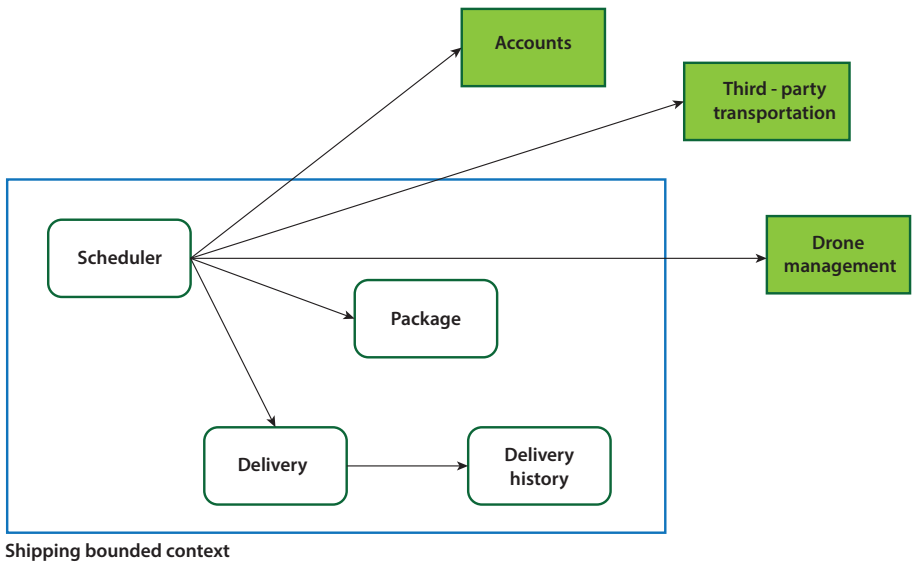
Radi detaljnije ilustracije taktičkog dizajna, na slici 9.34 prikazan je UML dijagram agregata **Isporuka** (Delivery). Ovaj agregat, kao korenski entitet, čuva direktne reference na druge agregate unutar sistema, kao što su **Nalog** (Account), **Paket** (Package) i **Dron** (Drone). Ovakva struktura omogućava strogu kontrolu nad procesom dostave: agregat *Isporuka* ne mora poznavati unutrašnje detalje drona ili paketa, već se oslanja na njihove jedinstvene identifikatore (REF), čime se održava labava povezanost (loose coupling) između različitih mikroservisa.



Slika 9.34. UML dijagram agregata "Isporuka" sa zavisnostima ka ostalim agregatima

Radi preciznije koordinacije, u dizajn su uvedene i dve **domenske usluge**:

1. **Planer** (Scheduler): Koordinira korake zakazivanja isporuke između različitih agregata.
2. **Supervizor** (Supervisor): Prati status svakog koraka kako bi detektovao zastoje ili neuspehe u realnom vremenu.



Slika 9.35. Logička dekompozicija i međuzavisnosti unutar graničnog konteksta isporuke

Kao što je prikazano na slici 9.35 unutar graničnog konteksta isporuke, domenska usluga *Scheduler* upravlja procesima između agregata *Package* i *Delivery*, istovremeno održavajući neophodne veze sa eksternim sistemima. Ovaj nivo detalja osigurava da svaki mikroservis bude kohezivan i otporan na greške.

Identifikovanje granica mikroservisa

Finalni korak u procesu projektovanja je transformacija logičkih agregata u fizičke granice mikroservisa. Glavni izazov je postići balans: servis mora biti dovoljno autonoman da bi se nezavisno razvijao, ali ne i previše usitnjen, jer preveliki broj mikro-komponenti nepotrebno usložnjava komunikaciju unutar sistema.

Kriterijumi za grupisanje agregata

U sistemu dostave dronom, granice servisa su određene primenom sledećih pravila:

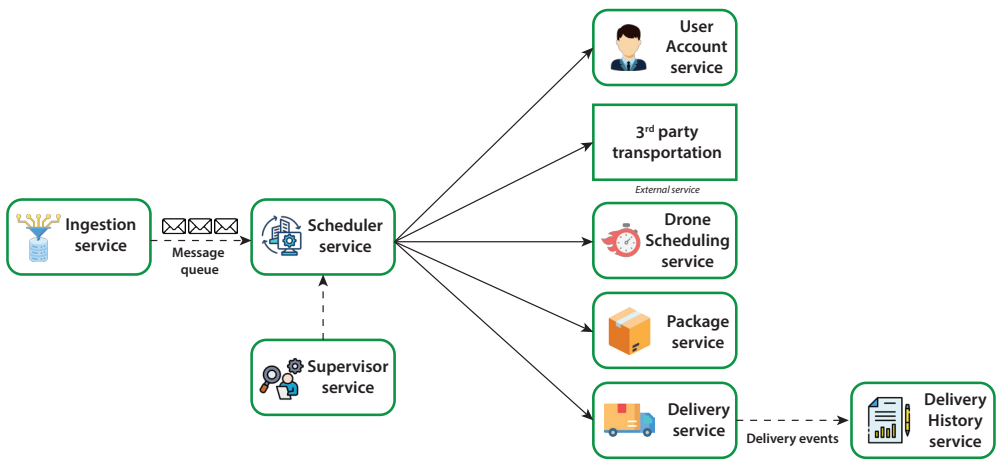
- **Granični kontekst kao primarna granica**: Osnovno pravilo nalaže da jedan mikroservis pokriva jedan granični kontekst, što omogućava jasnu podelu odgovornosti među razvojnim timovima.

- **Transakciona konzistentnost**: Ukoliko grupa entiteta (npr. Isporuka i Paket) mora biti ažurirana istovremeno u okviru jedne baze podataka da bi sistem ostao ispravan, oni se grupišu u isti mikroservis.
- **Specifični zahtevi za skalabilnost**: Delovi sistema sa različitim opterećenjem se razdvajaju. Na primer, funkcija praćenja drona u realnom vremenu generiše mnogo više saobraćaja nego servis za fakturisanje, pa se oni izoluju kako bi se nezavisno skalirali.

Arhitektura mikroservisa sistema za dostavu dronom

Na osnovu analize, definisana je sledeća struktura servisa:

- **Delivery Service** (Servis za dostavu): Centralni servis koji upravlja agregatima *Delivery* i *Package*, prateći sve faze isporuke od zakazivanja do završetka.
- **Drone Administration Service** (Servis za dronove): Specijalizovan za tehničko upravljanje flotom dronova, uključujući dostupnost, održavanje i popravke.
- **Package Service** (Servis za pakete): Opciono izdvojen servis koji upravlja fizičkim dimenzijama i specifikacijama robe radi lakše integracije sa automatizovanim skladištima.
- **Account Service** (Servis za korisnike): Upravlja profilima korisnika, istorijom njihovih porudžbina i podacima za naplatu.



Slika 9.36. Mapiranje agregata i graničnih konteksta u konkretne mikroservise (fizička arhitektura)

Asinhrona komunikacija i fleksibilnost

Nakon uspostavljanja fizičkih granica, mikroservisi komuniciraju asinhrono putem događaja. Na primer, kada *Delivery Service* emituje događaj "**Isporuka završena**", on ne poziva direktno ostale servise. Umesto toga, *Invoicing Service* (koji "sluša" taj događaj) automatski generiše račun, dok servis za lojalnost ažurira bodove korisnika. Ovakva struktura omogućava da se svaki servis razvija, testira i skalira nezavisno, čime se postiže maksimalna fleksibilnost i otpornost celokupnog sistema na greške.

Vežba 9.2. Dekompozicija bankarskog sistema primenom DDD principa

Scenario: Digitalna transformacija banke. Banka želi da prebaci svoj nasleđeni monolitni "Core Banking" sistem na mikroservisnu arhitekturu. Banka nudi tri ključne grupe usluga:

1. **Tekući računi i transakcije**: Otvaranje računa, uplate, isplate i transferi novca.
2. **Kreditni proizvodi**: Obrada zahteva za keš i stambene kredite, provera kreditne sposobnosti i otplata rata.
3. **Upravljanje platnim karticama**: Izdavanje debitnih i kreditnih kartica, postavljanje limita i blokada u slučaju krađe.

Problem: U trenutnom sistemu, entitet "**Klijent**" je postao preobiman (tzv. God Object). On sadrži sve — od kućne adrese, preko stanja duga po kreditu, do PIN koda za karticu. Svaka promena u modulu za kartice zahteva testiranje celog sistema za kredite, što usporava razvoj.

Zadatak 1: Identifikacija graničnih konteksta (Strateški DDD). Analizirajte scenario i definišite najmanje tri granična konteksta (Bounded Contexts). Za svaki kontekst navedite naziv i primarnu poslovnu odgovornost.

Zadatak 2: Sveprisutni jezik (Ubiquitous Language) **i semantičke razlike**. Objasnite kako se pojam "Račun" razlikuje u kontekstu Transakcija u odnosu na kontekst Kredita. Koje informacije su bitne za jedan, a koje za drugi model?

Zadatak 3: Predlog mikroservisa i agregata. Na osnovu identifikovanih konteksta, opišite strukturu mikroservisa. Navedite nazive servisa i identifikujte koji bi bio korenski entitet (Aggregate Root) u svakom od njih.

Rešenje vežbe

Rešenje zadatka 1. Identifikovane su sledeće logičke granice:

- **Core Banking Context**: Odgovoran za knjigovodstveno stanje, pravila o protoku novca i validaciju transfera.
- **Lending Context** (Kreditni kontekst): Odgovoran za procenu rizika, istoriju otplate i plan amortizacije.
- **Card Management Context**: Odgovoran za bezbednost, upravljanje limitima i komunikaciju sa kartičarskim organizacijama (Visa/Mastercard).

Rešenje zadatka 2 (Analiza semantičkih razlika): Iako se u oba slučaja koristi reč "Račun", u okviru **Sveprisutnog jezika** (Ubiquitous Language) specifičnog za svaki kontekst, modeli su potpuno različiti:

- **U Transakcijama**: "Račun" je izvor ili odredište novca. Bitni podaci su IBAN i raspoloživa (likvidna) sredstva.
- **U Kreditima**: "Račun" predstavlja kreditnu partiju. Bitni podaci su preostala glavnica, kamatna stopa i datum dospeća sledeće rate.

Ovo nisu isti objekti. Njihovo razdvajanje sprečava da promena u logici kredita slučajno pokvari sistem za isplate plata.

Rešenje zadatka 3 (Arhitektura servisa): Svaki od servisa prati princip **Database-per-service**, čime se osigurava da nijedan drugi servis ne može direktno menjati podatke u tuđoj bazi.

1. **Microservice A** (Transactions):
    1. **Koren agregata**: *Account* (Račun).
    2. **Specifičnost**: Koristi SQL bazu za garantovanu konzistentnost svakog centa.
2. **Microservice B** (Lending):
    1. **Koren agregata**: *Loan* (Kredit).
    2. **Specifičnost**: Može koristiti poseban sistem za bodovanje (scoring engine) i NoSQL bazu za čuvanje obimne dokumentacije klijenata.
3. **Microservice C** (Cards):
    1. **Koren agregata**: *Card* (Kartica).
    2. **Specifičnost**: Visoko bezbedan servis integrisan sa spoljnim API-jima kartičarskih mreža.

Literatura

[1] Avison, D. E., & Fitzgerald, G. (2006). Information systems development: Methodologies, techniques and tools (4th ed.). McGraw-Hill Education.

[2] Bass, L., Weber, I., & Zhu, L. (2015). *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional.

[3] Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM, 13*(6), 377–387.

[4] Coronel, C., & Morris, S. (2015). *Database systems: Design, implementation, & management* (11th ed.). Cengage Learning.

[5] Date, C. J. (2004). An introduction to database systems (8th ed.). Addison-Wesley.

[6] Date, C. J. (2012). Database Design and Relational Theory: Normal Forms and All That Jazz. O'Reilly Media.

[7] de la Torre, C., Wagner, B., & Rousos, M. (2022). .NET Microservices: Architecture for Containerized .NET Applications. Microsoft Press. Preuzeto sa https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

[8] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In M. Mazzara & B. Meyer (Eds.), Present and Ulterior Software Engineering (pp. 195–216). Springer.

[9] Dumas, M., La Rosa, M., Mendling, J., & Reijers, H. A. (2018). Fundamentals of Business Process Management (2nd ed.). Springer.

[10] Elmasri, R., & Navathe, S. B. (2016). Fundamentals of Database Systems (7th ed.). Pearson.

[11] Erl, T. (2016). Service-Oriented Architecture: Analysis and Design for Services and Microservices (2nd ed.). Prentice Hall.

[12] Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional.

[13] Fagin, R. (1981). A Normal Form for Relational Databases that Is Based on Domains and Keys. ACM Transactions on Database Systems (TODS), 6(3), 387-415.

[14] Fowler, M., & Lewis, J. (2014). Microservices: a definition of this new architectural term. martinfowler.com.

[15] Humble, J., & Farley, D. (2010). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Pearson Education.

[16] IEEE. (1998). IEEE Std 830-1998: IEEE Recommended Practice for Software Requirements Specifications. IEEE Computer Society.

[17] ISO/IEC/IEEE. (2018). ISO/IEC/IEEE 29148:2018 Systems and software engineering — Life cycle processes — Requirements engineering.

[18] Kendall, K. E., & Kendall, J. E. (2013). Systems analysis and design (9th ed.). Pearson Education.

[19] Kim, G., Humble, J., Debois, P., & Willis, J. (2016). The DevOps Handbook: How to Create World-Class Agility, *Reliability, and Security in Technology Organizations*. IT Revolution Press.

[20] Lau, F. (2000). Toward a framework for action research in information systems studies. Information Technology & People, 13(3), 148–175. https://doi.org/10.1108/09593840010339843

[21] Marca, D. A., & McGowan, C. L. (1988). Structured analysis and design technique. McGraw-Hill.

[22] Newman, S. (2019). Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly Media.

[23] OASIS. (2024). Web Services Business Process Execution Language (WS-BPEL). Retrieved from https://www.oasis-open.org

[24] OMG (Object Management Group). (2024). Business Process Model and Notation (BPMN). Retrieved from https://www.omg.org/spec/BPMN

[25] Oracle Academy. (2018). *HealthOne Medical Database Design Project*. Oracle Corporation.

[26] Pahl, C. (2015). Containerization and the PaaS cloud. IEEE Cloud Computing, 2(3), 24–31. https://doi.org/10.1109/MCC.2015.51

[27] Pressman, R. S., & Maxim, B. R. (2020). Software Engineering: A Practitioner's Approach (9. izd.). McGraw-Hill Education.

[28] Ramakrishnan, R., & Gehrke, J. (2003). Database management systems (3rd ed.). McGraw-Hill.

[29] Richards, M., & Ford, N. (2020). Fundamentals of software architecture: An engineering approach. O'Reilly Media.

[30] Richardson, C. (2018). Microservices Patterns: With examples in Java. Manning Publications.

[31] Robertson, S., & Robertson, J. (2012). Mastering the requirements process: Getting requirements right (3rd ed.). Addison-Wesley.

[32] Rosson, M. B., & Carroll, J. M. (2002). Usability engineering: Scenario-based development of human-computer interaction. Morgan Kaufmann.

[33] Sommerville, I. (2016). Software engineering (10th ed.). Pearson Education.

[34] UN/CEFACT. (2024). Core Component Technical Specification. Retrieved from https://unece.org/uncefact

[35] van der Aalst, W. M. P. (2004). Business process management: A personal view. Business Process Management Journal, 10(2), 98–102. https://doi.org/10.1108/14637150410530285

[36] W3C. (2024). Standards Overview. Retrieved from https://www.w3.org/standards/

[37] Weske, M. (2012). Business Process Management: Concepts, Languages, Architectures (2nd ed.). Springer.

[38] Wiegers, K., & Beatty, J. (2013). Software Requirements (3. izd.). Microsoft Press.

[39] Yourdon, E., & DeMarco, T. (1979). Structured Analysis: A Modern Tool for Systems Development. Prentice Hall.

[40] zur Muehlen, M., & Rosemann, M. (2005). Multiperspective enterprise modeling: An ontological foundation. BPM Center Reports.



PRILOG

REŠENJA ZADATAKA

Zadatak 3.1. Kreirajte *Product Backlog* i *Sprint Backlog* za Instagram aplikaciju

Korisničke priče koje bi mogle biti deo Product Backlog-a za aplikaciju Instagram su:

1. Korisnička autentifikacija i profil

- Kao korisnik, želim da se registrujem ili prijavim na aplikaciju putem email-a, Facebooka ili Google naloga.
- Kao korisnik, želim da kreiram svoj profil sa osnovnim informacijama (ime, korisničko ime, profilna slika).

2. Deljenje fotografija i video snimaka

- Kao korisnik, želim da objavljujem fotografije i video snimke kako bih ih podelio/la sa svojim pratiocima.
- Kao korisnik, želim da dodam opis i *hashtagove* svojim objavama kako bih ih lakše pronašao/la i pretražio/la.

3. Pregled i interakcija sa objavama

- Kao korisnik, želim da pregledam objave korisnika koje pratim na svom *feed*-u.
- Kao korisnik, želim da lajkujem i komentarišem objave drugih korisnika.

4. Upotreba storija (Stories)

- Kao korisnik, želim da kreiram privremene storije (priče) koje će biti dostupne samo 24 sata.
- Kao korisnik, želim da dodam efekte, filtere i tekst na svoje storije.

5. Direktna poruka i grupni čatovi

- Kao korisnik, želim da šaljem direktno poruke drugim korisnicima.
- Kao korisnik, želim da kreiram ili da se pridružim grupnim čatovima sa više korisnika.

6. Podešavanja profila i privatnost

- Kao korisnik, želim da menjam podešavanja privatnosti svog profila (javni, privatni, samo za pratioce).
- Kao korisnik, želim da imam kontrolu nad tim ko može da komentariše moje objave i šalje mi direktno poruke.

7. Pretraga korisnika i hashtagova

- Kao korisnik, želim da pretražujem druge korisnike po njihovom imenu ili korisničkom imenu.
- Kao korisnik, želim da pretražujem objave po *hashtag*-ovima kako bih pronašao/la relevantne sadržaje.

8. Obaveštenja i aktivnosti

- Kao korisnik, želim da dobijam obaveštenja o novim lajkovima, komentarima ili pratiocima.
- Kao korisnik, želim da pregledam istoriju aktivnosti (koga sam zapratio/la, ko me prati itd.).

9. Administracija i moderacija

- Kao administrator, želim alate za moderaciju sadržaja (npr. brisanje objava, blokiranje korisnika).
- Kao administrator, želim izveštaje o nepravilnostima i mogućnost da reagujem na njih.

10. Integracija sa drugim platformama

- Kao korisnik, želim da povežem svoj Instagram nalog sa drugim društvenim mrežama (npr. Facebook, Twitter).

Sprint Backlog ciljevi i zadaci tokom sprintova bi bili:

Sprint 1

1. **Korisnička autentifikacija i profil**
   - Zadatak 1: Implementirati backend funkcionalnost za registraciju korisnika putem email-a, Facebooka ili Google naloga.
   - Zadatak 2: Razviti frontend formu za kreiranje profila sa poljima za ime, korisničko ime i profilnu sliku.
   - Zadatak 3: Testirati integraciju različitih metoda autentifikacije i registracije.
2. **Deljenje fotografija i video snimaka**
   - Zadatak 1: Razviti backend API za objavljivanje fotografija i video snimaka.
   - Zadatak 2: Implementirati frontend komponente za unos opisa i hashtagova prilikom objavljivanja.
   - Zadatak 3: Validirati i testirati proces objavljivanja sadržaja.

Sprint 2

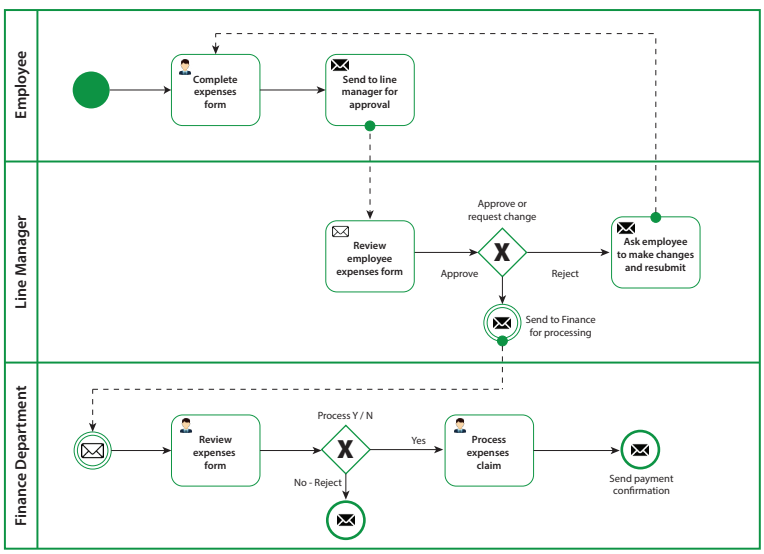
3. **Pregled i interakcija sa objavama**
   - Zadatak 1: Razviti backend API za prikaz objava na feed-u korisnika.
   - Zadatak 2: Implementirati frontend za prikaz objava sa opcijama za lajkovanje i komentarisanje.
   - Zadatak 3: Testirati interakciju korisnika sa objavama (lajkovanje, komentarisanje).
4. **Upotreba Storija** (Stories)
   - Zadatak 1: Implementirati backend funkcionalnost za kreiranje i upravljanje storijama.
   - Zadatak 2: Razviti frontend interfejs za dodavanje efekata, filtera i teksta na storije.
   - Zadatak 3: Testirati stabilnost i performanse storija tokom 24-časovnog ciklusa.

Sprint 3

5. **Direktna poruka i grupni četovi**
   - Zadatak 1: Razviti backend API za slanje direktnih poruka i kreiranje grupnih čatova.
   - Zadatak 2: Implementirati frontend interfejs za pregled i upravljanje direktnim porukama.
   - Zadatak 3: Testirati komunikaciju između korisnika putem direktnih poruka i grupnih čatova.
6. **Podešavanja profila i privatnost**
   - Zadatak 1: Implementirati backend logiku za podešavanje privatnosti profila (javni, privatni, samo za pratioce).
   - Zadatak 2: Razviti frontend za promenu podešavanja privatnosti i kontrole nad objavama.
   - Zadatak 3: Testirati sigurnost i pristupnost podešavanja privatnosti.

Svaki Sprint je fokusiran na specifičan set funkcionalnosti koje su identifikovane kao prioritetne na osnovu korisničkih priča. Svaki zadatak je definisan na način koji omogućava precizno planiranje, implementaciju, testiranje i evaluaciju tokom sprinta. Sprint Backlog je dinamičan i može se prilagođavati na osnovu dostupnih resursa, prioritetnih zahteva i novih saznanja tokom razvoja aplikacije.

Zadatak 4.1. Kreirati BPMN procesa refundacije troškova zaposlenih



Izvor: https://www.codelessplatforms.com/approval-process-flow

Zadatak 5.5. Primeniti 4NF na sledeću relaciju:

Postupak primene Četvrte normalne forme (4NF) na relaciju **KURS_UČESNIKA** (KursID, UčesnikID, Instruktor) zahteva pažljivo analiziranje poslovnih pravila i uočenih funkcionalnih zavisnosti.

Pretpostavimo da za ovu relaciju važe sledeća pravila:

- **Jedan kurs** (KursID) **ima više učesnika** (UčesnikID).
- **Jedan kurs** (KursID) **ima više instruktora** (Instruktor).
- **Lista učesnika za jedan kurs je potpuno nezavisna od liste instruktora za taj isti kurs.**

Ovakva situacija ukazuje na prisustvo dve nezavisne višeznačne zavisnosti, koje su uzrok redundantnosti:

1. **KursID** →→ **UčesnikID**: Jedan kurs ima više učesnika.
2. **KursID** →→ **Instruktor**: Jedan kurs ima više instruktora.

Iako ova relacija može biti u BCNF, ona i dalje sadrži značajna ponavljanja podataka. Na primer, ako se na kursu K01 nalaze učesnici U01 i U02, a vode ga instruktori I01 i I02, tabela bi izgledala ovako:

KursID	UčesnikID	Instruktor
K01	U01	I01
K01	U01	I02
K01	U02	I01
K01	U02	I02

Ovo dupliranje otežava održavanje baze. Ako se na kurs prijavi novi učesnik U03, morali bismo uneti dva nova reda, ponavljajući sve instruktore.

Postupak dekompozicije u 4NF

Da bi se eliminisala redundantnost i relacija dovela u 4NF, moramo je razdvojiti (dekomponovati) na dve manje relacije. Svaka nova relacija će sadržati po jednu od nezavisnih višeznačnih zavisnosti.

1. **Izdvojite prvu zavisnost u novu relaciju**. Kreirajte novu tabelu koja će beležiti samo vezu između kurseva i učesnika. Nazovimo je **KURS_UČESNICI**.

   KURS_UČESNICI (KursID, UčesnikID)

2. **Izdvojite drugu zavisnost u drugu relaciju**. Napravite drugu tabelu koja će pratiti samo vezu između kurseva i instruktora. Nazovimo je **KURS_INSTRUKTORI**.

   KURS_INSTRUKTORI (KursID, Instruktor)

Ovaj postupak je eliminisao redundanciju i osigurao da su podaci konzistentni. Dodavanje novog učesnika sada zahteva samo jedan unos u tabelu KURS_UČESNICI, bez uticaja na tabelu KURS_INSTRUKTORI. Primarni ključevi u obe nove relacije su složeni: (KursID, UčesnikID) i (KursID, Instruktor).

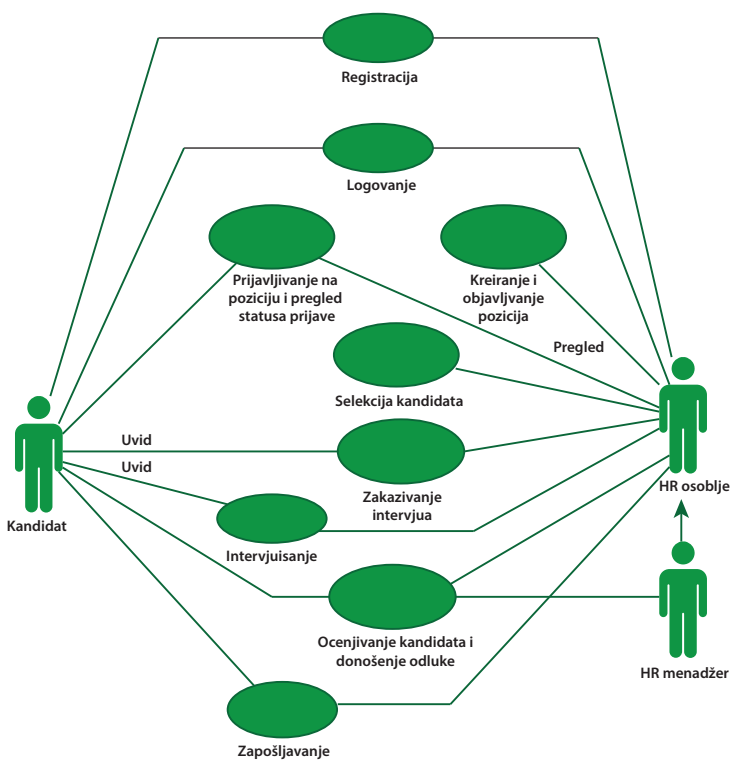
Zadatak 6.1. Kreirati *use case* dijagram za HR aplikaciju koja podržava procese selekcije, regrutovanja i zapošljavanja kandidata, na osnovu prikupljenih zahteva od korisnika

Use case-ove rasporediti hronološki i to: proces regrutacije, selekcije, intervjuisanja i zapošljavanja.

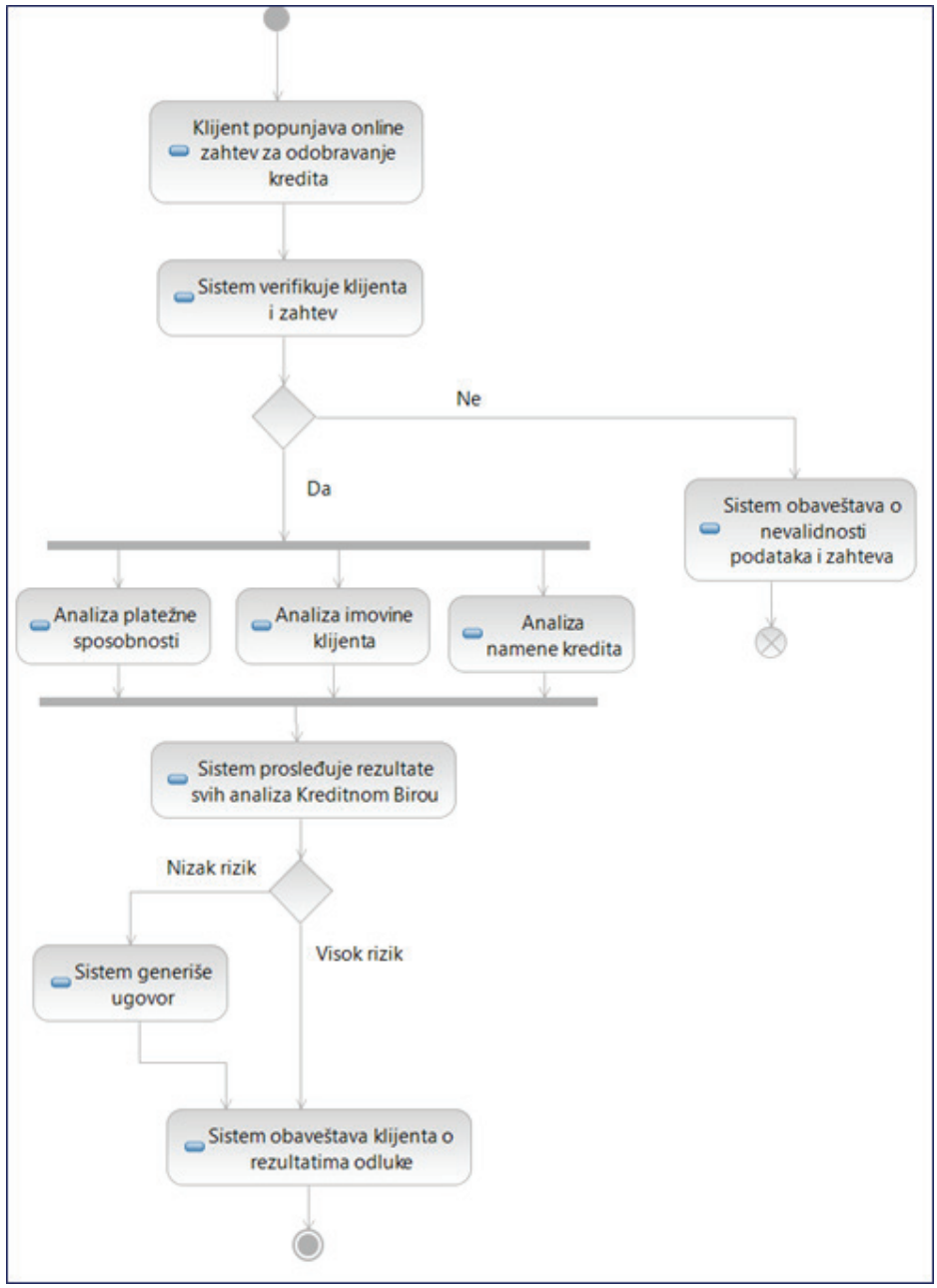
S obzirom da su Registracija i logovanje potpuno odvojene funkcije, tj. korisnik može da izabere Registraciju direktno iz menija bez prethodnog logovanja, u tom slučaju ne treba koristiti relaciju „extend" kao u prethodnim primerima, već oba *use case*-a stoje odvojeno i akteri imaju asocijaciju sa oba.

HR Menadžer ima sve funkcije HR Osoblja, plus dodatne, stoga se koristi relacija generalizacije između ova dva aktera. Generalizacija se koristi kada jedan akter nasleđuje sve funkcije drugog aktera, ima dodatne funkcionalnosti i predstavlja specijalizovanu verziju opšteg aktera. S obzirom da HR Menadžer jeste specijalizovan HR radnik zato generalizacija ovde ima smisla.

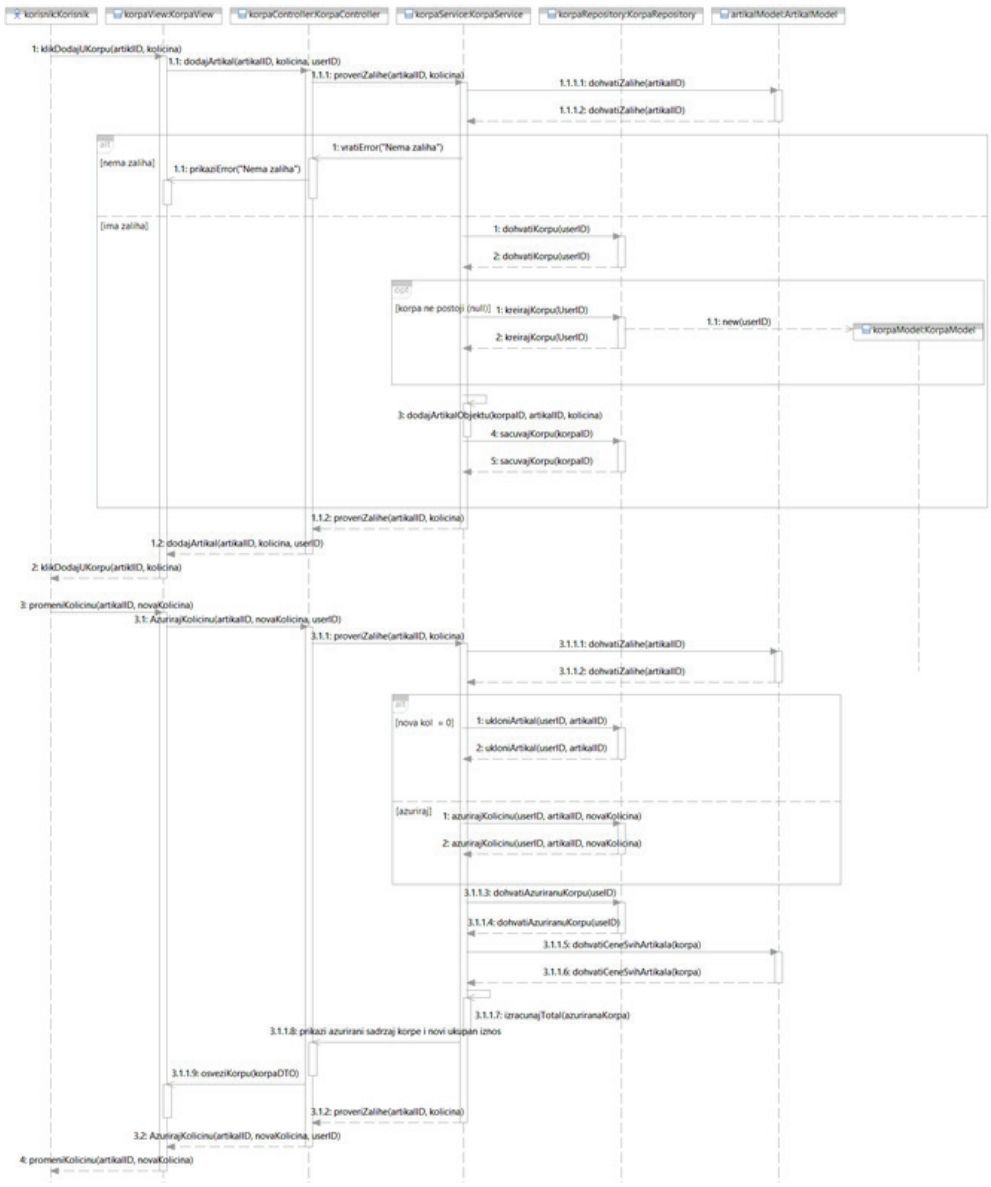
Iako su u postavci zadatke opisani koraci, kao npr. pregled pristigle prijave kandidata, treba se zapitati da li je to stvarno *use case* ili ćemo samo imentovati relaciju kao pregled ili uvid u rezultat obrade nekog *use case*-a. Da bi nešto bilo *use case*, mora da ima min dve aktivnosti koje obrađuje. Pregled nema nikakvih aktivnosti, stoga nije *use case*.



Zadatak 6.2. Kreirati dijagram aktivnosti za *use case* "Online zahtev za odobravanje kredita", koji prati sledeća scenarija:



Zadatak 7.1. Kreirati dijagram sekvenci za *use case* "Kreiranje i editovanje šoping korpe"



Zadatak 7.2. Kreirati UML dijagram klasa koji modeluje osnovne elemente onlajn prodavnice

Korpa i Porudzbenica se često koriste naizmenično, ali u kontekstu razvoja softvera i UML modelovanja, one predstavljaju dve različite faze u životnom ciklusu kupovine.

Koncept	**Šoping Korpa** (Cart)	**Porudžbenica / Kupovina** (Order)
Uloga	**Privremeni kontejner** za selekciju proizvoda.	**Trajni, pravni zapis** o transakciji.
Stanje	**Nestabilno** (Klijent stalno dodaje/ oduzima proizvode).	**Stabilno** (Sadržaj i cena su zaključani u trenutku izdavanja).
Implikacije na zalihe	**Ne blokira** zalihe**.**	**Blokira/rezerviše** zalihe čim nastane.
Atributi	status: AKTIVNA, datumKreiranja.	statusPlaćanja, adresaIsporuke, ukupnaCena.
Povezanost	Vezana samo za **Korisnika** (može se obrisati ako istekne vreme).	Vezana za **Korisnika, Plaćanje, Isporuku.**
Plaćanje	**Ne vezuje** se za Plaćanje.	**Vezuje** se za Plaćanje (šta se plaća).

U većini *e-commerce* sistema, korisnik može da puni samo jednu korpu istovremeno. Bilo bi nelogično da ima dve različite, aktivne korpe koje istovremeno plaća. Kada se ta jedna Korpa konvertuje u Porudžbenicu, ona prestaje biti aktivna, a korisnik može da započne novu, praznu Korpu. Razdvajanje ova dva koncepta je ključno za funkcionalnost sistema:

- Korpa brine o tome šta korisnik želi.
- Porudžbina brine o tome kako će se želja izvršiti (platiti, poslati, pratiti).

Jedna Porudžbina može imati **više pokušaja Plaćanja** (npr. jedan neuspeli i jedan uspešan).

Sada kada razumemo funkcionalnu razliku, pogledajmo kako se to odražava u strukturi sistema:

- **Tranzicija Korpa - Porudžbina** (Nema Asocijacije!)

  Budući da je **Korpa privremena i sklona čestim modifikacijama**, a **Porudžbina trajna** i **zaključana**, ne smemo imati direktnu, trajnu asocijaciju između ovih dveju klasa. Jedna klasa se **transformiše** u drugu. To se ne modeluje vezom (asocijacijom), već **metodom**. Na klasi Korpa postoji metoda: **+ kreirajPorudžbinu**(korisnik). Kada se ova metoda pozove, ona uzima sadržaj Korpe, kreira novu instancu klase Porudžbina, fiksira cene i arhivira/briše staru Korpu.

- **Veza Porudžbina - Plaćanje**

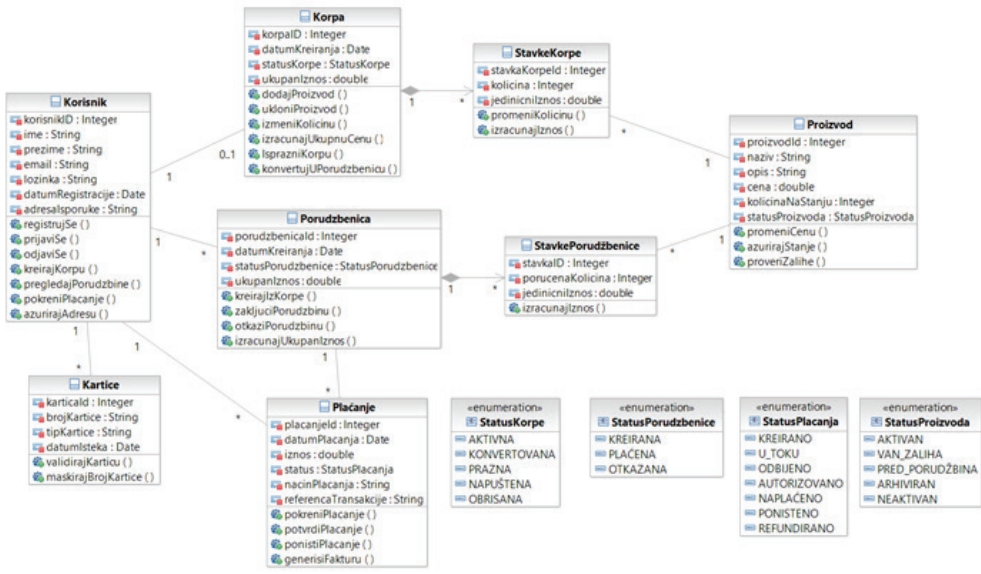
  Porudžbina je zaključani ugovor. Svaka platna transakcija se odnosi na **tačno jednu** Porudžbinu. Da bismo pokrili realnost e-trgovine, Porudžbina mora dozvoliti neograničen broj Plaćanja. Ako korisnik dva puta neuspešno pokuša da plati, sistem mora da zabeleži ta **tri zasebna zapisa Plaćanja** (dva neuspešna i jedan uspešan). **Zvezdica** (*) omogućava beleženje te **istorije pokušaja** plaćanja povezanih sa tim ugovorom.

- **Veza Plaćanje - Korisnik**

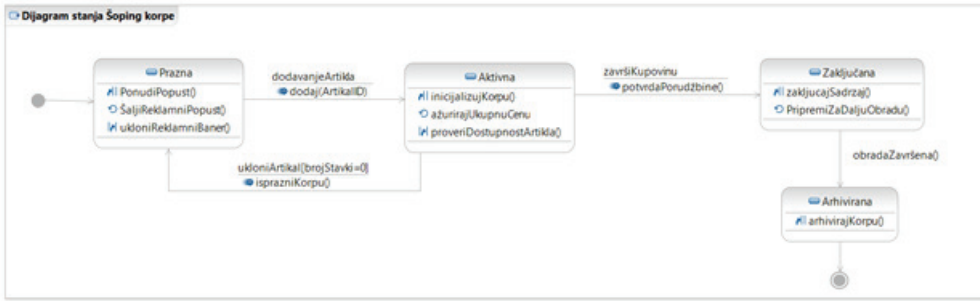
  Možda se čini suvišnim, jer je Plaćanje već indirektno vezano za Korisnika preko Porudžbine. Međutim, direktna veza je ključna iz razloga revizije i praćenja finansija:

  - **Revizija i pretraga**: Finansijski i revizijski timovi često moraju da filtriraju sve finansijske transakcije (uspešne, neuspešne, refundacije) direktno po ID-u Korisnika, bez potrebe da prolaze kroz tabelu Porudžbina.
  - **Praćenje toka novca**: Direktan odnos **Korisnik - Plaćanje** omogućava sistemu da izgradi potpunu finansijsku istoriju za svakog korisnika. To je posebno važno u kompleksnim sistemima gde plaćanje ne rezultira uvek porudžbinom (npr. plaćanje pretplate, depozita, itd.).

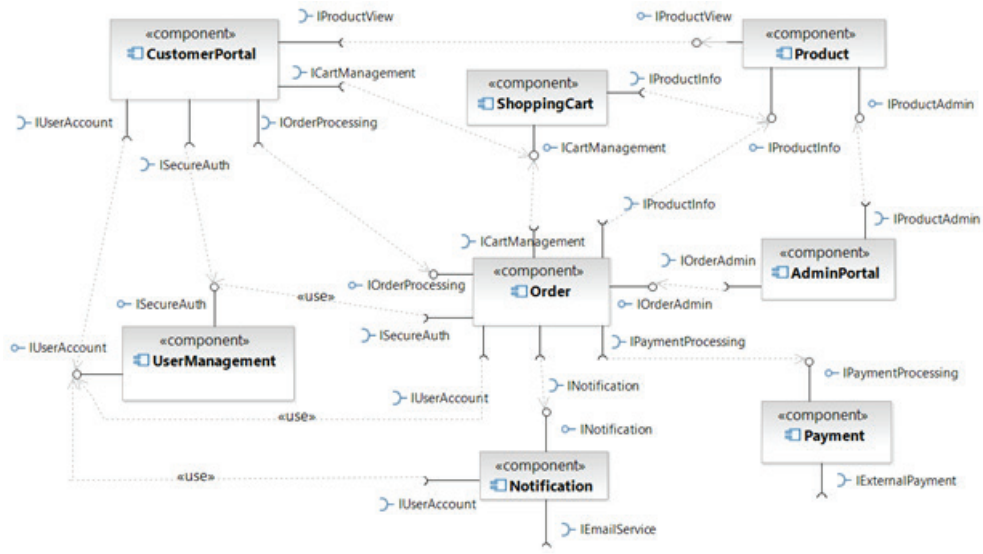
Iako je Plaćanje strukturalno zavisno od Porudžbine za iznos, ono je **operativno** direktno zavisno od Korisnika za autorizaciju i finansijsko praćenje.



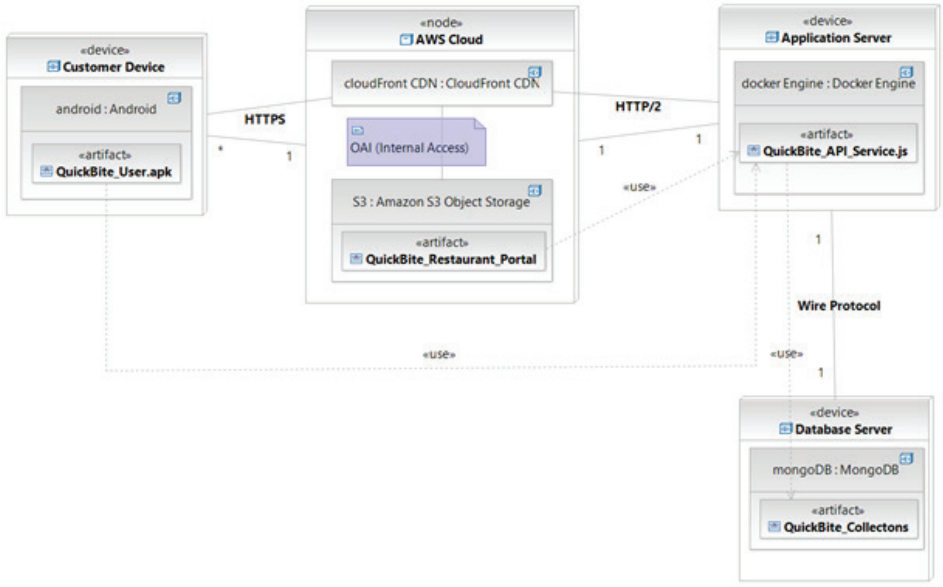
Zadatak 8.1. Kreirati dijagram stanja klase „ŠopingKorpa“



Zadatak 8.2. Kreirati dijagram komponenti za sistem onlajn prodavnice



Zadatak 8.3. Kreirati dijagram raspoređivanja za Cloud-native arhitekturu sistema „QuickBite“



SPISAK VEŽBI

VEŽBA 3.1. KREIRATI ***P****RODUCT BACKLOG* I ***S****PRINT BACKLOG* ZA AMAZON *E-COMMERCE* APLIKACIJU. 62
VEŽBA 4.1. NACRTATI BPMN ZA PROCES "ODOBRAVANJE KREDITA" 111
VEŽBA 4.2. KREIRATI BPMN DIJAGRAM ZA PROCES "PODNOŠENJE I OBRADA ZAHTEVA ZA GODIŠNJI ODMOR" 113
VEŽBA 5.1. NA OSNOVU SLEDEĆEG DOKUMENTA, ODGOVORITE NA PITANJA I KREIRAJTE MODEL PODATAKA. 128
VEŽBA 5.2. PRIMENITI BCNF NA SLEDEĆU RELACIJU POSAO (ZAPOSLENIID, PROJEKATID, LOKACIJA), GDE VAŽE SLEDEĆA POSLOVNA PRAVILA: 137
VEŽBA 5.3. PRIMENITI 4NF NA RELACIJU ZAPOSLENI_PROJEKTI (ZAPOSLENIID, PROJEKATID, VEŠTINA). POSLOVNA PRAVILA GLASE: 140
VEŽBA 5.4. SVESTI SLEDEĆU RELACIJU NA 5NF: RASPORED (STUDENTID, PREDMETID, PROFESORID). OVA RELACIJA BELEŽI KO OD STUDENATA POHAĐA KOJI PREDMET I KOD KOG PROFESORA. POSLOVNA PRAVILA SU: 142
VEŽBA 5.5. NA OSNOVU PRIKUPLJENOG DOKUMENTA »UGOVOR O RENTIRANJU VOZILA« KREIRAN JE POČETNI OBJEKAT KOJI JE POTREBNO ANALIZIRATI PRIMENOM PRVE TRI NORMALNE FORME: 144
VEŽBA 6.1. KREIRANJE *USE CASE* DIJAGRAMA ZA E-COMMERCE APLIKACIJU (PRIMER: AMAZON) 159
VEŽBA 6.2. KREIRATI *USE CASE* DIJAGRAM ZA APLIKACIJU „AIRBNB ZA REZERVACIJU SMEŠTAJA" 166
VEŽBA 6.3. KREIRATI DIJAGRAM AKTIVNOSTI ZA *USE CASE* „AUTENTIKACIJA" KOD CHATGPT SISTEMA 176
VEŽBA 6.4. KREIRATI DIJAGRAM AKTIVNOSTI ZA *USE CASE* „PLAĆANJE KREDITNOM KARTICOM" KORISTEĆI PARTICIJE (*SWIMLINES*). 180
VEŽBA 7.1. KREIRATI DIJAGRAM SEKVENCI ZA *USE CASE* „PLAĆANJE" 209
VEŽBA 7.2. KREIRATI PROŠIRENI DIJAGRAM KLASA ZA RENTACAR AGENCIJU KOJA JE UVELA NOVA PRAVILA POSLOVANJA. 238
VEŽBA 7.3. KREIRATI KONCEPTUALNI DIJAGRAM KLASA BANKARSKOG SISTEMA I TRANSAKCIJA 240
VEŽBA 8.1. KREIRATI JEDNOSTAVAN DIJAGRAM STANJA NEKOG UREĐAJA KOJI IMA SAMO DVA STANJA UKLJUČEN (*ON*) I ISKLJUČEN (*OFF*) KORIŠĆENJEM IBM RSA ALATA. 265
VEŽBA 8.2. U IBM RSA ALATU NACRTATI SLEDEĆI DIJAGRAM STANJA ATM MAŠINE 270
VEŽBA 8.3. KREIRATI DIJAGRAM STANJA ZA STANJA KLASE STUDENT 271
VEŽBA 8.4. DIJAGRAM KLASA „ONLAJN PRODAVNICE" SA PRIMENOM OBRAZACA 285
VEŽBA 8.5. NACRTATI DIJAGRAM KOMPONENTI ZA SISTEM ZA REZERVACIJU I PRODAJU KARATA U IBM RSA ALATU 298
VEŽBA 8.6. KREIRATI KOMPOZITNI STRUKTURNI DIJAGRAM SISTEMA *TICKETINGSYSTEM*. 303
VEŽBA 8.7. KREIRANJE DIJAGRAMA RASPOREĐIVANJA E-COMMERCE APLIKACIJE U IBM RSA ALATU 316
VEŽBA 8.8. KREIRATI DIJAGRAM RASPOREĐIVANJA ZA MOBILNU APLIKACIJU ZA UPOZNAVANJE „MEETU" 323
VEŽBA 9.1. MODELOVANJE SERVISNO ORIJENTISANE APLIKACIJE ZA OBRADU ZAHTEVA ZA BANKARSKI KREDIT KORIŠĆENJEM BPMN NOTACIJE 347
VEŽBA 9.2. DEKOMPOZICIJA BANKARSKOG SISTEMA PRIMENOM DDD PRINCIPA 382

SPISAK ZADATAKA

ZADATAK 4.1. KREIRATI BPMN PROCESA REFUNDACIJE TROŠKOVA ZAPOSLENIH 114
ZADATAK 5.1. PRIMENITI 1NF NA SLEDEĆU RELACIJU: 131
ZADATAK 5.2. PRIMENITI 2NF NA SLEDEĆU RELACIJU: 133
ZADATAK 5.3. PRIMENITI 3NF NA SLEDEĆU RELACIJU: 136
ZADATAK 5.4. PRIMENITI BCNF NA SLEDEĆU RELACIJU: 138
ZADATAK 5.5. PRIMENITI 4NF NA SLEDEĆU RELACIJU: 141
ZADATAK 6.1. KREIRATI *USE CASE* DIJAGRAM ZA HR APLIKACIJU KOJA PODRŽAVA PROCESE SELEKCIJE, REGRUTOVANJA I ZAPOŠLJAVANJA KANDIDATA. PRIKUPLJENI SU SLEDEĆI ZAHTEVI OD KORISNIKA: 168
ZADATAK 6.2. KREIRATI DIJAGRAM AKTIVNOSTI ZA *USE CASE* "ONLINE ZAHTEV ZA ODOBRAVANJE KREDITA", KOJI PRATI SLEDEĆA SCENARIJA: 182
ZADATAK 7.1. KREIRATI DIJAGRAM SEKVENCI ZA *USE CASE* "KREIRANJE I EDITOVANJE ŠOPING KORPE" 216
ZADATAK 7.2. KREIRATI UML DIJAGRAM KLASA KOJI MODELUJE OSNOVNE ELEMENTE ONLAJN PRODAVNICE. 243
ZADATAK 8.1. KREIRATI DIJAGRAM STANJA KLASE „ŠOPINGKORPA" 272
ZADATAK 8.2. KREIRATI DIJAGRAM KOMPONENTI ZA SISTEM ONLAJN PRODAVNICE 306
ZADATAK 8.3. KREIRATI DIJAGRAM RASPOREĐIVANJA ZA CLOUD-NATIVE ARHITEKTURU SISTEMA „QUICKBITE" 324

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.41(075.8)

ЊЕГУШ, Ангелина, 1971-
Specifikacija i modelovanje softvera / Angelina Njeguš. - 1. izd. - Beograd : Univerzitet Singidunum, 2026 (Beograd : Birograf). - VIII, 435 str. : ilustr. ; 24 cm

Tiraž 800. - Bibliografija: str. 418-419.

ISBN 978-86-7912-857-7

а) Софтверско инжењерство

COBISS.SR-ID 189678601

© 2026

Sva prava zadržana. Nijedan deo ove publikacije ne može biti reprodukovan u bilo kom vidu i putem bilo kog medija, u delovima ili celini bez prethodne pismene saglasnosti izdavača.

www.singidunum.ac.rs

9 788679 128577



Angelina Njeguš

Specifikacija i modelovanje softvera

Udžbenik „Specifikacija i modelovanje softvera“ je prevashodno namenjen studentima Tehničkog fakulteta i Fakulteta za informatiku i računarstvo Univerziteta Singidunum, ali i svima onima koji svoju karijeru žele da grade u pravcu razvoja softvera, a posebno u ulogama arhitekte softvera, poslovnog analitičara ili projektanta sistema. Cilj ovog udžbenika je da pomogne studentima u razvijanju sistemskog razmišljanja prilikom realizacije projekata razvoja softvera, sa posebnim fokusom na početne faze prikupljanja zahteva i modelovanja.

Specifikacija i modelovanje softvera zavise od izabrane metodologije razvoja, vrste aplikacije i same arhitekture. U ovom udžbeniku detaljno su objašnjene aktuelne metodologije i prikazani praktični pristupi modelovanju različitih vrsta softverskih rešenja.

Ovaj udžbenik predstavlja most između teorijskih postavki softverskog inženjerstva i praktičnih izazova sa kojima se inženjeri susreću u realnom poslovnom okruženju, pripremajući studente za kreiranje softvera koji je pouzdan, održiv i u potpunosti usklađen sa potrebama korisnika.

Univerzitet Singidunum