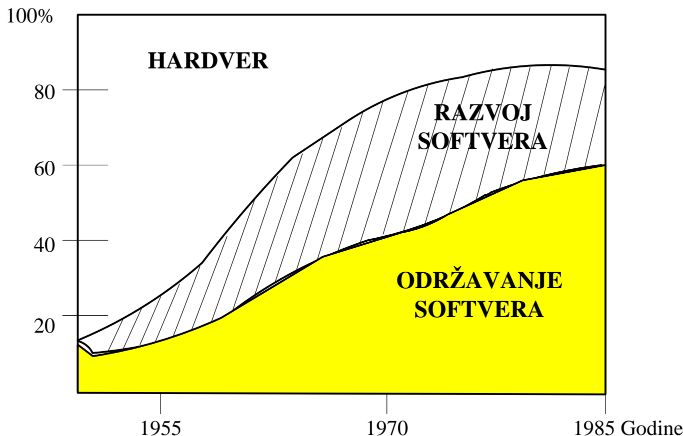
1. Istorija razvoja metoda i alata za projektovanje informacionih sistema

Softversko inženjerstvo i softverska kriza

Termin "Softversko inženjerstvo" se prvi put pojavio na jednoj NATO konferenciji još 1968. godine. Pod njim se podrazumevao skup metoda, tehnika i alata za projektovanje softvera, po principima projektovanja proizvoda, uređaja i objekata u drugim inženjerskim disciplinama. Softversko inženjerstvo se javilo kao odgovor na *"softversku krizu"*
Pod "softverskom krizom" se podrazumevaju svi, ne mali, problemi u razvoju softvera, prvenstveno niska produktivnost i visoki troškovi razvoja. Softverska kriza se obično ilustruje sledećim Boehm-ovim dijagramom:



- Adaptivno održavanje
- Perfektivno održavanje
- Korektivno održavanje

Pod terminom "kriza" podrazumeva se nešto prolazno. Da li softverska kriza još uvek traje?

Istorijat Softverskog inženjerstva će se diskutovati sa tri međusobno čvrsto povezana aspekta:
1. programski jezici i programiranje
2. modeli i metode razvoja sofvera
3. case alati za razvoj softvera

Istorijat, odnosno "revolucionarne promene" se dešavaju uvek ma isti način: Prvo u programskim jezicima, zatim u metodologiji razvoja softvera i na kraju u CASE alatima.

Revolucionarne promene - faze razvoja:
- "herojsko doba - rešavanje problema isključivo programiranjem
- strukturne metode
- modeli podataka, baze podataka i jezici iv generacije
- "doba zrelosti" - objektne metode

1. programski jezici i programiranje
Arhitektura programskih jezika:

1) Tipovi podataka (programsko-jezički, virtuelni tipovi). Pod tipom podatka se podrazumeva skup vrednost i skup operacija nad njima:
atomski tipovi

- agregirani tipovi - strukture podataka

2) kontrolne strukture - definisanje redosleda obavljanja operacija na tipovima:

- sekvencija
- selekcija
- iteracija

Jezici su se razlikovali po tome koje tipove podataka podržavaju i na koji način ostvaruju kontrolne strukture (koliko se “struktuirani”)

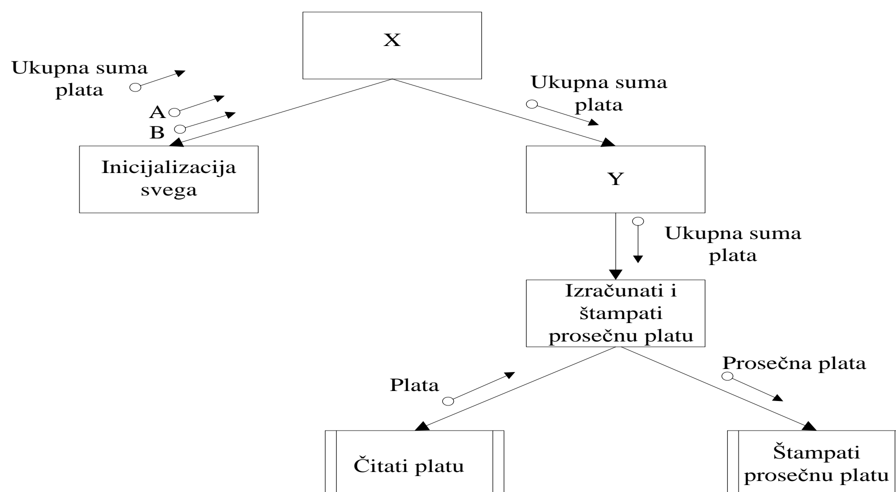
Strukturna faza 1965 – 1980

Najveća (gotovo jedina) pažnja u početnom periodu razvoja softverskog inženjerstva posvećivala se *kontrolnim strukturama* (algoritamske apstrakcije):
1) struktuirno programiranje:

- rešavanje sindroma “špageti koda”
- dobro definisane strukrure programa
- programiranje bez ili sa kontrolisanom primenom “go to” naredbe. “Go to considers harmful”

2) Strukturna analiza i projektovanje

- Structured System Analysis. Structured Design - Yourdon i Constantine
- SADT (Structured Analysis and Design Technique) - D.T.Ross, Sof. Tech. Inc., Današnji IDEF0 standard dela američke administracije (vojske i drugih)
- ISAC (Information System Work and Analysis of Changes), Langefors, Švedska



Pravila strukturnog projektovanja:

- nalaženje “centralne transformacije” ili “analiziatora transakcija” kao čvora stabla
- definisanje hijerarhijske strukture modula
- analiza povezanosti (cupling-a) i kohezije modula

Ova pravila su i danas aktuelna za definisanje “odgovornosti objekata” za pojedine operacije u objektnom projektovanju)

Uočavanje potrebe da se jasno odvoje specifikacija, projektovanje i implementacija softvera. Definisanje konvencionalnog (“vodopad”) “životnog ciklusa is”

1. ANALIZA ZAHTEVA I SPECIFIKACIJA IS.
Specifikacija treba da jasno, neprotivrečno i formalno odgovori na pitanje “šta” softver treba da radi;
2. PROJEKATOVANJE. Daje odgovor na pitanje “kako”
3. IMPLEMENTACIJA. Kodiranje i testiranje
4. UVOĐENJE
5. ODRŽAVANJE

CASE ALATI

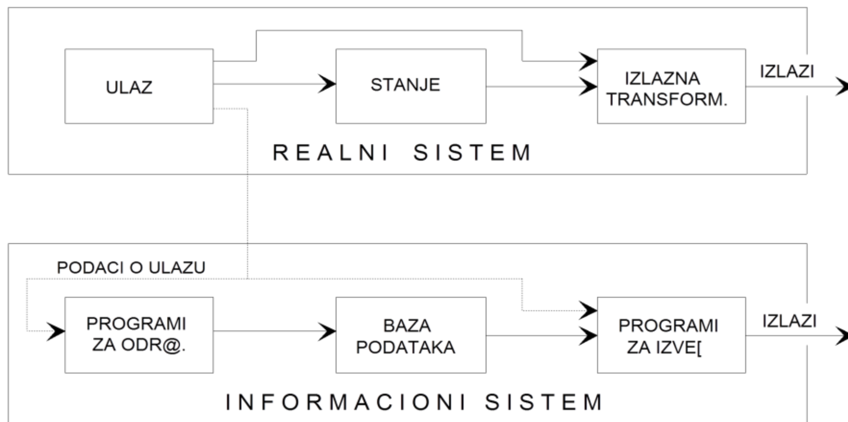
- PSL/PSA (Problem Statement Language - Problem Statement Language) ISDOS projekat University of Michigan, Prof D. Teichroew, učestvovao i FON. Verovatno prvi CASE alat.
- Information Engineering Workbanch - IBM
- BPwin
- ARTIST - FON
- ORACLE CASE
- Mnogi drugi

Pretežno “lower cases” - prve faze razvoja, analiza i specifikacija sistema.
Mnogi su još “živi”, jer se stalno proširuju i nadgrađuju.

Modeli podataka, baze podataka i jezici iv generacije 1980 -1995

BAZE PODATAKA, APSTRAKCIJE PODATAKA, MODELI PODATAKA I NEPROCEDURALNI JEZICI
Isključivo algoritamske transakcije nisu rešile mnoge probleme:

- redundasa podataka u masovnoj obradi podataka
- nezavisnost programa od logičke i fizičke strukture podataka
- uključivanje korisnika u razvoj softvera - pokušaj da korisnik samstalno zadovoljava svoje neposredne zahteve - razvoj neproceduralnih jezika
- prenošenje većeg dela semantike problema na podatke - razvoja modela podataka



Generacije modela podataka

I GENERACIJA: Jezici treće generacije sa relativno jednostavnim tipovima podataka.

II GENERACIJA: Modeli konvencionalnih Sistema za upravljanje bazom podataka - hijerarhijski mrežni i relacioni.

III GENERACIJA: Semantički bogati modeli Model objekti veze, SDM i drugi.

RELACIONI MODEL

Definicija preko tipova podataka:

- Skup vrednosti: skup relacija (tabela)
- Operacije: relaciona algebra

Znatno moćniji tipovi podataka nego u drugim konvencionalnim sistemima

Mogućnost definisanja neproceduralnog jezika S Q L

RELACIONI MODEL <-----> KOMERCIJALNI RELACIONI SUBP

E.F. Codd: "The Relational model for Database management", Version 2, Addison -Wesley, 1990.

Ipak nedovoljno semantički bogati:

- Ne postoje mehanizmi za implementaciju apstrakcija agregacije i generalizacije
- Ne postoji mogućnost definisanja apstraktnog tipa kao domena

MODEL OBJEKTI-VEZE

PROMENE U ŽIVOTNOM CIKLUSU

- UVOĐENJE TRANSFORMACIONIH PRINCIPA: Formalna specifikacija i automatska transformacija specifikacije u implementaciju
- PROTOTIPSKI RAZVOJ: Odbacivi i nadgradivi prototipovi.

CASE ALATI

Uglavnom isti kao i ranije, samo prošireni sa modelima podataka

- BPwin i ERwin
- ARTIST - FON
- ORACLE CASE
- Mnogi drugi

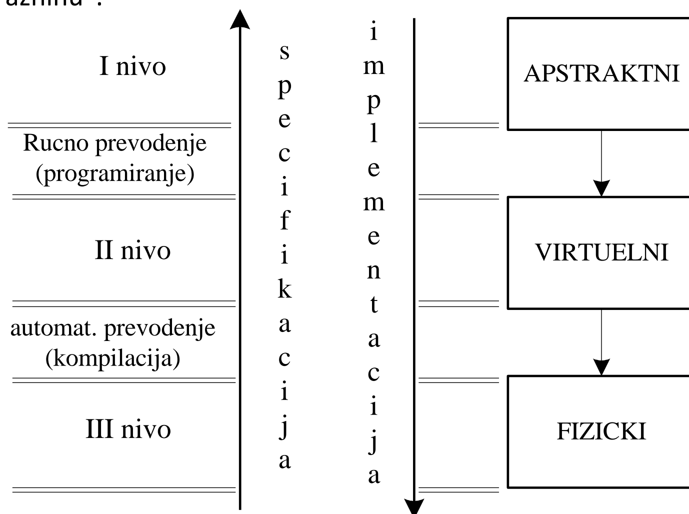
Ne samo "lower cases" - već delimični i "uper cases", generisanje baza podataka.

OBJEKTNE METODE 1980 -
apstraktni tipovi podataka - preteča
objektni jezici
izolovane objektne metode

STANDARDI - UML - ZRELOST – 1998

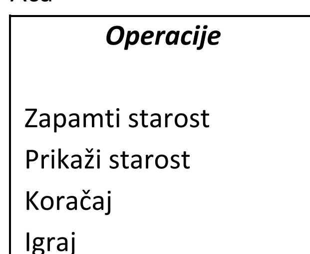
Apstraktni tipovi podataka
Realni sistem
Skup objekata, njihovih veza, njihovih atributa i operacija nad njima
Softverski sistem
Skup vrednosti (prostih struktura) i operacija nad njima

" Semantička praznina"
Apstraktni tipovi su korisnički definisani tipovi, pogodni za opis realnih sistema, koji treba da smanje ovu "semantičku prazninu".



Osnove objektnog pristupa: sistem je skup povezanih objekata

Pod objektom se podrazumeva entitet koji je sposoban da čuva svoja stanja i koji stavlja na raspolaganje okolini skup operacija preko kojih se ta stanja prikazuju ili menjaju.
Bitna karakteristika objekata u OO pristupima je učaurenje (encapsulation), sakrivanje informacija (information hiding).
U strogo objektnim pristupima jednini vidljivi deo objekta su operacje i to ne način na koji su implementirane, već samo njihovi efekti (specifikacija).



Skači

U složenom sistemu postojaće mnoštvo objekata, pa bi i njihov direktan opis bio veoma složen. Zbog toga je neophodno uvesti apstrakcije klasifikacije i generalizacije.
U apstrakciji klasifikacije definišu se pojam tipa, odnosno klase objekata.
U najnovijim pristupima se insistira na razlici koncepata tipa i klase objekata:
Pod tipom se podrazumeva kategorija objekata koji imaju isti skup stanja. Neki konkretan objekat se tretira kao pojavljivanje tipa. Tip objekta se definiše sa dva aspekta:
(1) kao jedan interfejs preko koga se definišu spoljne karakteristike objeketa toga tipa i
(2) kao jedna ili više klasa koje predstavljaju različite implementacije datog tipa.

STANDARDNI OPIS KLASA U UML-u

Ime-klase

vidljivost naziv-atributa-1: tip-podatka-1 = početna vrednost-1 iskaz osobina
vidljivost naziv-atributa-2: tip-podatka-1 = početna vrednost-1 iskaz osobina
....
vidljivost naziv-operacije-1 (lista-argumenata-1): tip-rezultata-1 iskaz osobina
vidljivost naziv-operacije-2 (lista-argumenata-1): tip-rezultata-2 iskaz osobina
....

Svaka operacija ima “pozvani” objekat kao implicitni argument. Ponašanje operacije zavisi od klase kojoj “pozvani” objekat pripada. Objekat “zna” svoju klasu, pa samim tim i način implementacije posmatrane operacije. Osobina da objekat poziva neku operaciju drugog objekta ne znajući kojoj klasi ovaj objekat pripada naziva se polimorfizam.

U objektnim pristupima jasno se razdvajaju modeli i metodologija

Za modele se definišu standardi: najpoznatiji standard je: UML

MODELI
MODELI ZA OPIS FUNKCIJA SISTEMA:
- Dijagrami slučajeva korišćenja: "Use Case Diagrams"
- (Strukturna sistemska analiza)

MODELI ZA OPIS STRIKTURE SISTEMA:
- Dijagrami klasa
- (Model objekti veze)

MODELI ZA OPIS DINAMIKE:
- Dijagrami sekvenci (Sequence Diagrams)
- Dijagrami kolaboracije (Collaboration Diagrams)
- Dijagrami promene stanja (State Transition Diagrams)
- Dijagrami aktivnosti (Activity Diagrams)

DIJAGRAMI ZA OPIS IMPLEMENTACIJE
- Deployment Diagrams

M E T O D O L O G I J A
Pod metodologijom se podrazumeva definisani proces razvoja softvera, gde se u različitim fazama primenjuju različiti standardni modeli
Podrazumeva se "sloboda" u definisanju metodologija, odnosno mogućnost definisanja sopstvene metodologije

SISTEMSKO-TEORIJSKA OO METODOLOGIJA
LABORATORIJA ZA IS FON-a
MODEL "ŽIVOTNOG CIKLUSA"
IDENTIFIKACIJA SISTEMA

- Definisanje funkcionalnog modela preko SSA
- Definisanje Slučajeva korišćenja iz primitivnih funkcija SSA

REALIZACIJA SISTEMA

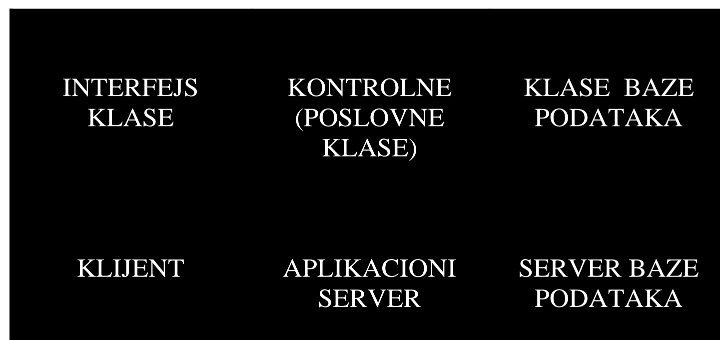
- Definisanje strukture objektnog modela (ili Modela objekti veze)
- Opis dinamike svakog Slučaja korišćenja
- Definisanje potpunog dijagrama fundamentalnih klasa

PROJEKTOVANJE

- Definisanje klasa baze podataka
- Definisanje interfejs klasa

IMPLEMENTACIJA

Potpuno odvajanje specifikacije i implementacije - troslojna arhitektura softera



CASE ALATI
U znatno većoj meri su transformacioni

- Select Enterprase Modeler - SELECT Software Tools, Inc.
- Rational Rose - Rational Software Corp.
- Paradigm Plus Enterprise Edition 3.51 - Platinum Technology, Inc.

2. Razvoj informacionih sistema

Složenost razvoja IS i modeli “Životnog ciklusa IS”

- Složenost razvoja IS savladava se:
    - Razbijanjem celokupnog razvoja na faze – način razbijanja na faze se naziva ***Model životnog ciklusa IS***

- Dekompozicijom samog sistema, odnosno definisanjem arhitekture sistema
    - Funkcionalna (strukturna) ili objektana dekompozicija
    - Dvoslojna, troslojna ili višeslojna arhitektura

Modeli "životnog ciklusa IS"

Tri osnovne grupe (principa):

- Konvencionalni razvoj: striktno praćenje svih pravila inženjerskog pristupa razvoju IS.
- Brzi razvoj: što pre doći do kakvog takvog rešenja, pa ga onda usavršavati
- Formalni (transformacioni) razoj definisanje formalnih modela i postupaka razvoja – formalna transformacija formalne specifikacije IS u implementaciju.

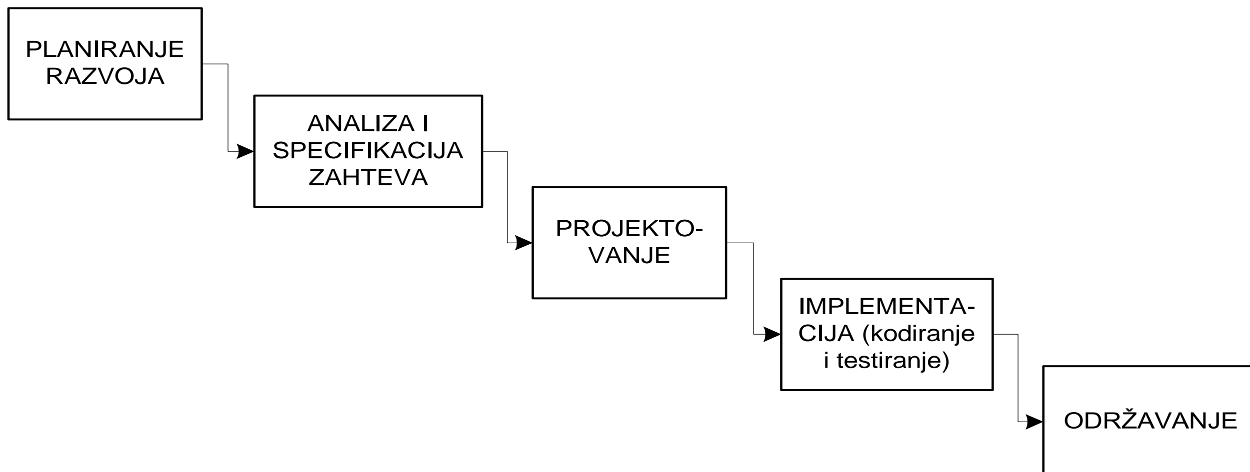
Konkretne motode i pristupi ne spadaju striktno ni u jednu grupu – kombinuju dva ili sva tri principa.

Konvencionalni razvoj

- Konvencionalni **"vodopad životni ciklus"** i sve njegove modifikacije:

    - "Fontana" model
    - Spralni model
    - (Iterativno) Inkrementalni pristup

KONVENCIONALNI "VODOPAD" ŽIVOTNI CIKLUS



KONVENCIONALNI ŽIVOTNI CIKLUS – STRUKURNI PRISTUP

1. **Planiranje – bsp metoda**
2. **Analiza i specifikacija zahteva**
    - Strukturna sistemska analiza nalaženje skupa "atomskih" fundamentalnih funkcija sistema, njihovih ulaza i izlaza
    - Opis ulaza, izlaza i skladi[ta preko re^nika ssa.
    - Opis pojedina^nih atomskih funkcija preko pseudokoda

3. Projektovanje

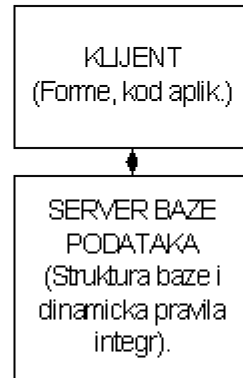
Logičko projektovanje

- Izgradnja odgovarajućeg modela podataka (model objekti-veze)

- Transformacija modela objekti veze u normalizovan relacioni model.
- Projektovanje strukturnih programa

Fizičko projektovanje

- Fizičko projektovanje baza podataka
- Projektovanje korisničkog interfejsa
- Dodavanje "fižičkih elemenata" strukturnim programima.

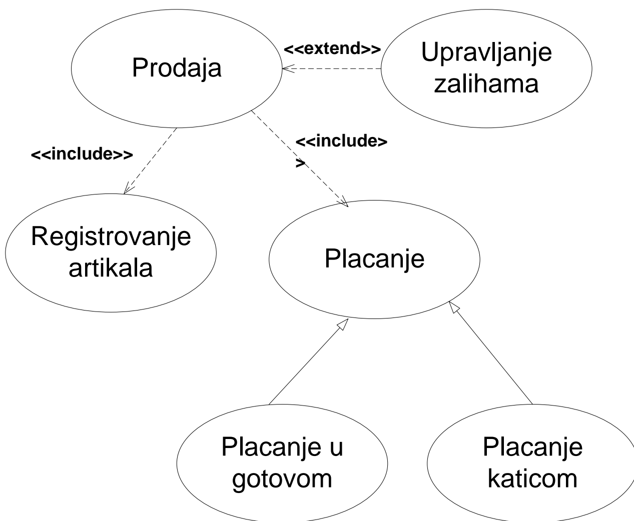


4. Implementacija

- Kodiranje u nekom strukturnom jeziku i testiranje ili
- Primena generatora aplikacija (jezika četvrte generacije)
- Relacione baze podataka i dvoslojna klijent-server arhitektura.

Konvencionalni životni ciklus – objektni pristup

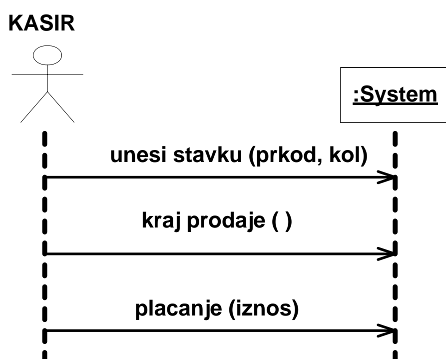
1. PLANIRANJE – Ništa specifično
2. ANALIZA I SPECIFIKACIJA ZAHTEVA
   - Izrada Slučajeva korišćenja: Dijagram slučajeva korišćenja i verbalni opis svakog SK
   - Izrada sistemskih dijsgrama sekvenci



Opis SK:

1. Učesnici
2. Namena
3. Kratak opis
4. Pre i Post uslovi
5. Opis osnovnog toka događaja
6. Opis alternativnog toka događaja

Sistemski dijagram sekvenci za slučaj korišćenja "prodaja"



3. PROJEKTOVANJE

- Logičko projektovanje
    - Izrada konceptualnog modela (dijagrama klasa bez operacija)
    - Izgradnja dijagrama sekvenci, dijagrama kolaboracije ili dijagrama aktivnosti za opis dinamike sistema (logike aplikacije)
    - Izgradnja kompletnog logičkog modela sistema (dijagram klasa sa operacijama definisanim preko dijagrama sekvenci kolaboracije ili aktivnosti)

- Fizičko projektovanje (zavisno od konkretnog okruženja)
    - Projektovanje baze podataka
    - Projektovanje korisničkog interfejsa
    - Dodavanje novih klasa na osnovu odgovarajućih uzora (pattern-a): MVC patern, Perzistentni brokeri i mnogi drugi
    - Izgradnja kompletnog fizičkog modela sistema

4. IMLEMENTACIJA

- Raspoređivanje delova modela na pojedine elemente višeslojne arhitekture,
- Transformacija fizičkog modela u konkretno implenemtaciono okruženje
- Dodatno kodiranje
- Testiranje

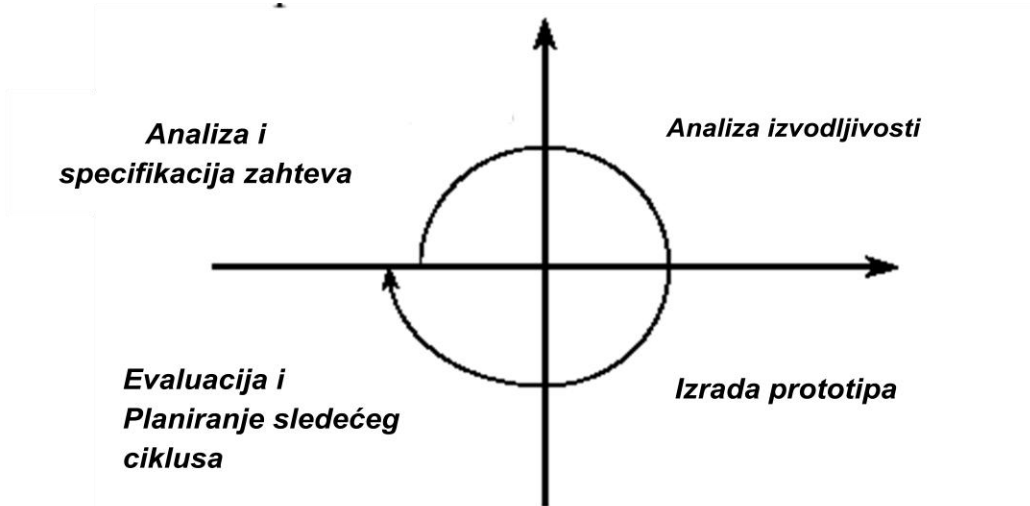
KONVENCIONALNI ŽIVOTNI CIKLUS – NEDOSTACI

- Problemi razvoja:
    - Teško je, gotovo nemoguće utvrditi sve zahteve na početku projekta i potpuno tačno;
    - Projekti su obično dugotrajni t teško je skrupulozno sprovesti metodologiju do kraja
    - Do prve verzije sistema (koja bi trebalo da bude konačna) dolazi se veoma sporo. Zaksneo povratni uticaj korisnika
    - Veoma obimna dokumentacija za velike projekte.

- Problemi održavanja:
    - Veoma skupo održavanje
    - Dvostruko održavanje, održavanje koda i održavanje projektne dokumentacije
    - U slučaju izmene zahteva treba proći ponovo kroz sve faze projektaovanja i izvršiti izmene. To se obično ne radi, pa na kraju, i pored ogromnog truda imamo nesaglasnost koda i projektne dokumentacije.

Svi ovi nedostaci i znatno manjoj meri dolaze do izražaja u objektnim pristupima!

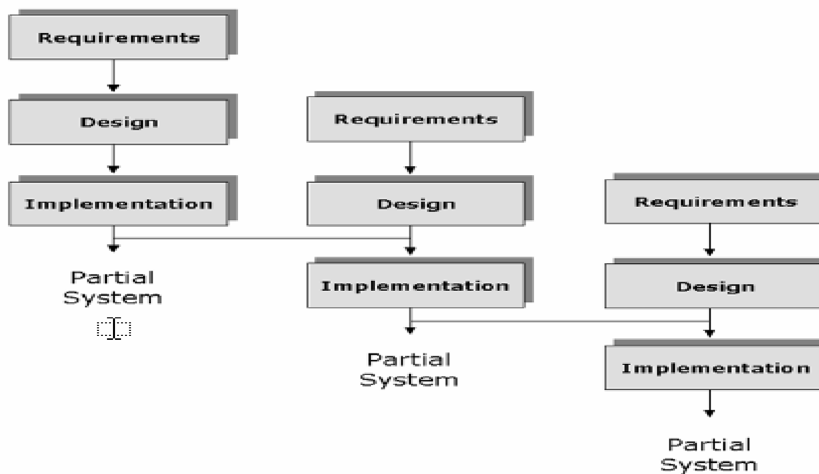
Modifikacije konvencionalnog životnog ciklusa: **Spiralni model**



Prototipovi služe za:

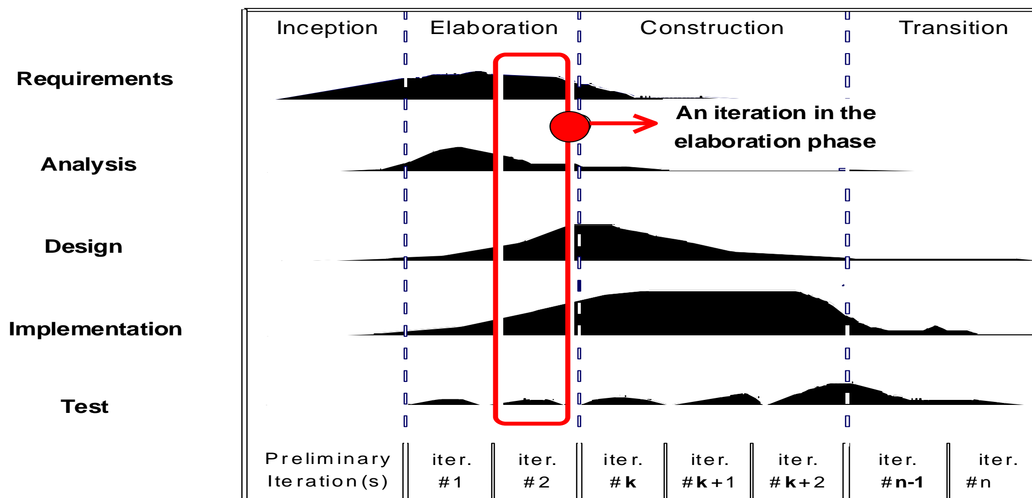
- Validaciju kor. Zahteva
- Experimenat sa implementacionim okruženjem

Modifikacije konvencionalnog životnog ciklusa: **Inkrementalni model**



Prethodi podela na podsisteme, odnosno definisanje polazne arhitekture sistema

Modifikacije konvencionalnog životnog ciklusa: **Iterativno-inkrementalni razvoj**
(Unified Software Development process)



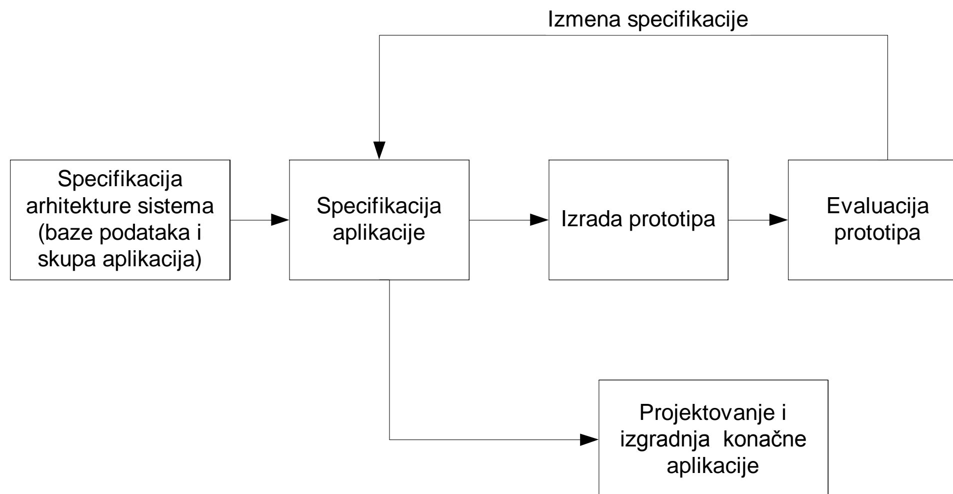
Brzi (rapid, agile) razvoj

U ovom pristupu, umesto da se trudimo da detaljnom analizom pokušamo da utvrdimo potpune zahteve korisnika, pokušavamo da razvijemo sistem sa nekompletno definisanim zahtevima.

- Prototipski razvoj- rapid prototyping
- Agile Software Development

Prototipski razvoj

- Prototipovi aplikacija a ne celog sistema
- Prethodno definisanje arhitekture sistema: skupa aplikacija i modela (baze) podataka
- Postojanje alata za izradu prototipova, prototip mora brzo i jeftino da bude urađen (Jezici četvrte generacije)
- Loša dokumentacija, teško održavanje ovako izgrađenog sistema



Agile Software Development

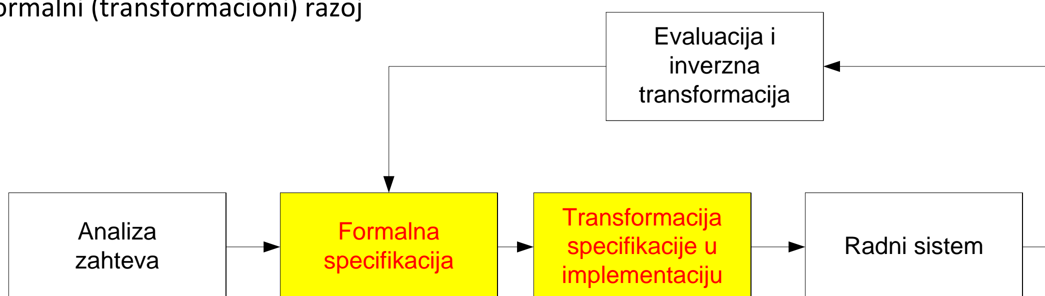
- ***Cilj je kod koji radi, a ne perfektna dokumentacija.*** Više se vodi računa o organizaciji ljudi koji rade na projektu, a manje o metodologijama i alatima za razvoj.
- Najpopularniji Agile Development pristup je ***XP (Extreme Programming). XP*** je projektovan za male timove koji treba da brzo razviju neki softver u okruženju sa stalno promenljivim zahtevima. U takvom okruženju konvencionalni životni ciklus bi bio neupotrebljiv.

XP (Extreme Programming)

XP se zasniva na sledećim principima:

1. ***Planiranje***. Korisnik daje procenu dobiti za realizaciju pojedinih zahteva, a programeri odgovarajuću procenu troškova i na osnovu toga se određuju zahtevi koji će biti realizovani odmah i zatevi koji će biti odloženi.
2. ***Jednostavna realizacija***. XP tim proizvodi jednostavne sisteme i a zatim ih učestalo menja i poboljšava
3. ***Metafora***. XP timovi koriste zajednički "sistem imena" i zajedniučki opis sistema.
4. ***Jednostavno projektovanje***. XP podrazumeva da se svaki program pravi na najednostavniji način koji zadovoljava zahteve.
5. ***Testiranje.*** Stalno se vrši validacija sistema. Prvo se definiše test, a onda se razvija softver koji treba da ga zadovolji. Korisnik definiše test za prihvatanje sistema.
6. ***Refactoring***. Softver se stalno poboljšava, s tim da se očuva njegova jednostavnost. Ne prave se duplikati.
7. ***Programiranje u parovima***. Dva programera razvijaju jedan kod kontrolišući jedan drugog. Rade na istoj mašini. Pokazuje se da ovakav razvoja daje znatno bolje rezultate od individualnog razvoja.
8. ***Kolektivno vlasništvo***. Ceo kod pripada svim programerima. Kad god je potrebna neka izmena bilo ko može da je izvrši.
9. ***Stalna integracija***. Integracija razvijenih delova radi se više puta dnevno. To zahteva učešće svih programera i ubrzava proces razvoja.
10. ***40-časovna radna nedelja.*** Prekovremeni rad je izuzetak. Programeri treba da budu odmorni, zdravi i produktivni.
11. ***Stalno prisustvo korisnika***. Značajno se poboljšava komunikacija, smanjuje potreba za prepiskom i dokumentacijom.
12. ***Standardno kodiranje.*** Svi programeri treba da pišu kod na isti, standardni, način da bi se principi XP-a mogli da sprovedu.

Formalni (transformacioni) razoj



- Izvršni kod se dobija isključivo transformacijom formalne specifikacije u implementaciju
- Ovakav pristup omogućuje jednostavnu promenu implementacionog okruženja (platforme) - jednostavno "adaptivno održavanje"
- Perfektivno održavanje sistema se značajno pojednostavljuje – nema "dvostrukog održavanja" (dokumentacije i koda)

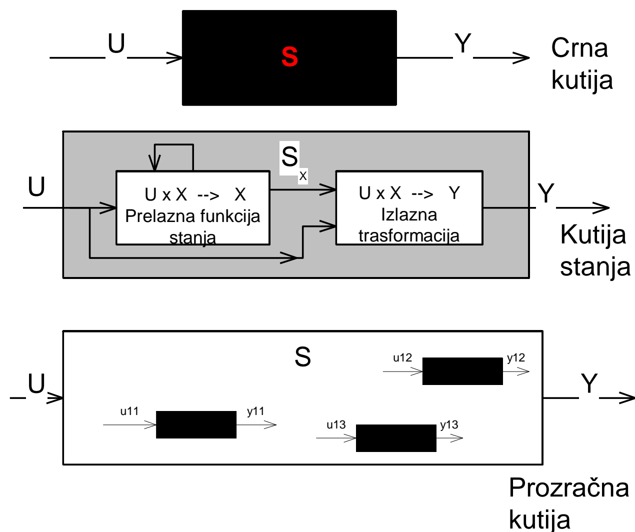
- Koji se jezici (modeli) koriste za formalnu specifikaciju?
- Kako se vrši transformacija formalne specifikacije u radni sistem (kod)?
- Kako se vrši formalna inverzna transformacija?

U vreme kada je definisan transformacioni pristup je bio moguć samo za veoma specifične i jednostavne sisteme za koje ke bilo moguće definisti formalni specifikacioni jezik i odgovarajuću transformaciju

CASE alati delimično podržavaju ovakav pristup (Model objekti-veze kao jezik za specifikaciju i SQL ko radni sistem, na primer)
Osnovna ideja, značajne prednosti ovoga pristupa, sve bolji jezici (modeli) za specifikaciju i CASE alati stalno unapređuju transformacioni pristup razvoju IS.

Transformacioni razvoj:***Cleanroom engineering***

- ***Cleanroom engineering*** koji je predložio IBM (druga polovina 80-tih) je rigorozni metod razvoja u kome "greške nisu dozvoljene" (za razliku od drugih pristupa koji se zasnivaju na "trial-and-error" pristupima).
- ***Cleanroom engineering*** polazi od činjenice da su ***programi način realizacije matematičkih funkcija***. Za specifikaciju programa neophodno je definisati funkciju koja u potpunosti opisuje ponašanje koje se od programa zahteva. Nalaženje procedure koja realizuje tu funkciju je osnova razvoja softvera.



Transformacioni razvoj:"Sistemsko-teorijski životni ciklus" – FON kraj prošlog veka

1. **Identifikacija sistema**: nalaženje funkcionalnog modela sistema, na osnovu nekog posmatranja (analize):

S:šFa : T x U ---> Y, a **e** A}

2. Realizacija sistema: nalaženje modela sistema u izabranom prostoru stanja.

φ: U x X $\rightarrow$ X (Funkcija prelaza stanja)

η = U x X $\rightarrow$ Y (Izlazna transformacija)

3. **Implementacija:** kodiranje i testiranje

Transformacioni razvoj:"Sistemsko-teorijski životni ciklus" – FON kraj prošlog veka

- **Identifikacija sistema –** skup funkcija (bilo koji alat za funkcionalnu specifikaciju sistema, Dijagrami tokova podataka, najbolje, a mogu i Slučajevi korišćenja)
- Realizacija sistema:
  - Izbor najpovoljnijeg modela za realizaciju funkcija
  - Nalaženje minimalne realizacije u okviru izabranog modela.

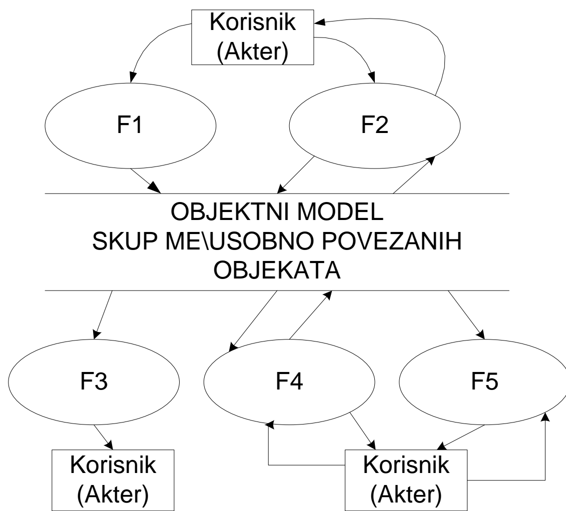
SISTEMSKO-TEORIJSKI "ŽIVOTNI CIKLUS"

Ponekad se u "Sistemsko-teorijski životni ciklus" daodaje i četvrta faza:

4. Sinteza upravljanja

čime se realizuje upravljačka funkcija IS, odnosno MIS (Management Information System)

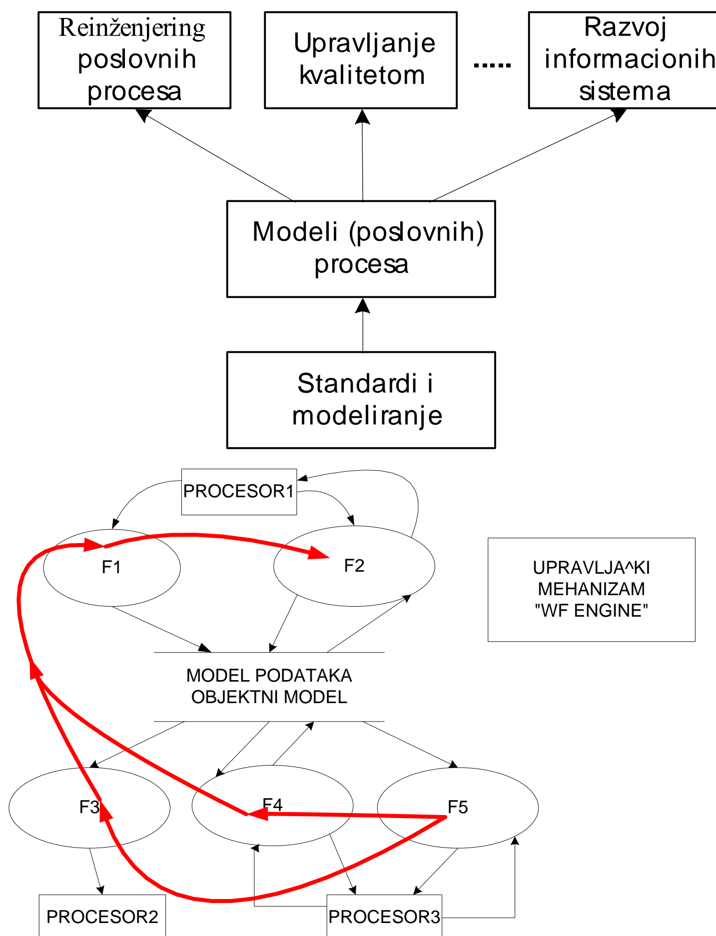
Upravljački informacioni sistemi



- Samosinhronzovani skup Aplikacija
- Sinhronizacija preko podataka u bazi
- Omogućeno upravljanje pojedinačnim funkcijama
- Savremeni menadžment je orjentisan prema upravljanju procesima koji predstavljaju ***orkestraciju funkcija*** u cilju obavljanja nekog zadatka.

Upravljanje procesima- Modeli procesa

Danas postoji gotovo opšta saglasnost da je upravljanje međusobno povezanim i međusobno zavisnim poslovnim procesima ("process centered management approach") osnova uspešnog funkcionisanja bilo koje organizacije



Odnos poslovnih procesa i poslovnih funkcija

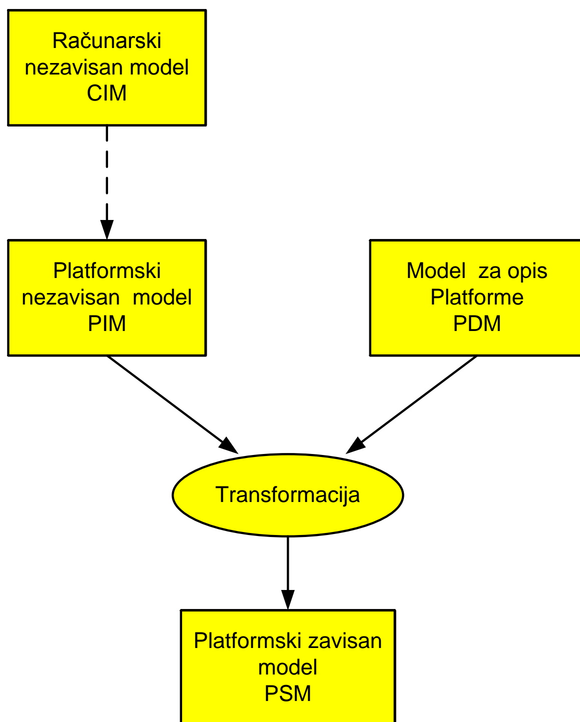
- Praćenje realizacije nekog ugovora,
- Praćenje realizacije narudžbenice kupca,
- Sprovođenje nekog upravnog postupka i slično.

Upravljački mehanizam sadrži:

- Model poslovnih procesa
- Model organizaciono-tehnološkog okruženja
- Operativni plan obavljanja poslovnih procesa
- Operacije za upravljanje: lansiranje u suspedndovanje posla, preraspodela izvršioca i slično

Transformacioni razvoj:***Modelom vođeni razvoj***

- Model Driven Architecture (MDA) koju je 2001 predložila Object Management Group je "*an approach to using models in software development*". Zato je bolje koristiti prevod *Modelom vođeni razvoj* mego *Modelom vođene arhitekture.*
- Pojam *arhitektura* se vezuje za činjenicu da se preko asptraktnih modela (PIM) može da ostvari inteoperabilnost heterogenih sistema.



Computation Independent Model –CIM – model odgovarajućeg domena, zajednički rečnik za korisnika i projektanta

Platform Independent Model – PIM.
Model IS nezavisan od implementacione platforme. Specifikacija sistema

Platform Description Model –PDM
Model implementacione platforme

Platform Specific Model- PSM
Model IS implementiran u datom okruženju.

- Modelom vođeni razvoj omogućava:
  - Specifikaciju sistema nezavisnu od bilo kakve implementacije;
  - Specifikaciju platforme;
  - Izbor platforme za implementaciju specifikaovanog sistema;
  - Transformaciju specifikacije sistema u izabranu platformu.

3. Analiza zahteva i specifikacija is

- Model je "subjektivni odraz objektivne stvarnosti". Zbog toga može da postoji više različitih modela istog sistema, sa istog ili različitih aspekata.
- Model je neka simplifikacija realnosti
- Modeli se izgrađuju da bi se bolje razumeo realni sistem
- Modelovanje je način da se savlada složenost nekog sistema
- Modelovanje je opšti pristup u svim unženjerskim disciplinama
- U svakoj oblasti postoji, obično više, intelektualnih alata (jezika) za modelovanje sistema- **UML za oblast razvoja softvera**

CILJEVI UML-a

- Opšti vizuelni jezik za modelovanje softverskih sistema i razmenu dobijenih modela
- Mogućnost proširenja i specijalizacije osnovnih koncepata u skladu sa potrebama vrste sistema koja se modelira
- Podrška specifikaciji nezavisnoj od razvojne metodologije i implementacionog okruženja
- Uvođenje formalizma koji će omogućiti razumevanje jezika
- Podrška konceptima viših nivo apstrakcije: komponenta, kolaboracija, aplikacioni kostur

ASPEKTI MODELA U UML-U
Za svaki aspekt daje se statički i dinamički opis sistema

Aspekt slučajeva korišćenja

- Opisuje se ponašanje sistema sa tačke gledišta korisnika prvenstveno, a koristi se u analizi i testiranju takođe. *Pretstavljaju funkcionalnu specifikaciju sistema.*
- Statički opis ovoga aspekta daje se preko ***Dijagrama slučajeva korišćenja***, a dinamički, prvenstveno takstualno, a zatim i preko ***dijagrama interakcija, dijagrama promene stanja ili dijagrama aktivnosti.***

Aspekt projektovanja

- Aspekt projektovanja pretstavlja realizaciju sistema u "objektnom prostoru stanja".
- Statički opis ovoga aspekta daje se preko ***Dijagram klasa i Dijagrama objekata.***
- Dinamički opis se daje preko ***dijagrama interakcija, dijagrama promene stanja i dijagrama aktivnosti.***

Aspekt implementacije

- Aspekt implementacije pretstavlja komponente i fajlove preko kojih se sistem fizički ostvaruje.
- Statički opis ovoga aspekta daje se preko ***Dijagrama komponenti.***
- Dinamički opis se daje preko ***dijagrama interakcija, dijagrama promene stanja i dijagrama aktivnosti.***

Aspekt procesa

- Aspekt procesa opisuje način odvijanja procesa u sistemu, "niti" procesa, konkurentnost i sinhronizaciju.
- Koriste se isti dijagrami kao i u aspektu projektovanja i za statički i za dinamički opis, a njviše dijagrami aktivnosti

Aspekt razmeštanja

- Aspekt razmeštanja pretstavlja topologiju sistema, računarsko-komunikacionu mrežu. Pokazuje se kako su u ovoj mreži razmeštene komponente koje pretstavljaju fizičku realizaciju sistema
- ***Dijagrami razmeštaja*** se koriste za statički opis.
- Dinamički opis se daje preko ***dijagrama interakcija, dijagrama promene stanja i dijagrama aktivnosti.***

Funkcionalni model sistema.

Strukturna sistemska analiza slučajevi korišćenja

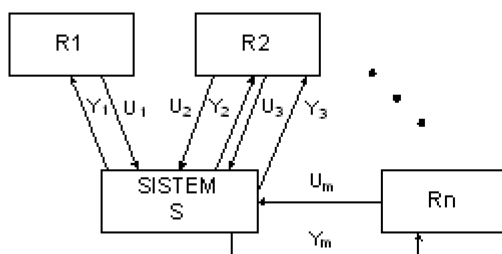
Informacioni sistemi mogu biti veoma složeni. OO model složenog IS može sadržati i nekoliko hiljada različitih objekata sa mnoštvom njihovih atributa i veza. Zbog toga prvi modeli u razvoju nekog sistema ne mogu da budu objektni, **moraju biti funkcionalni**

Kao alati za modelovanje funkcija sistema (transformacije ulaza u izlaz) koristiće se strukturna sistemska analiza(konvencionalni model) i model slučajeva korišćenja (uml)

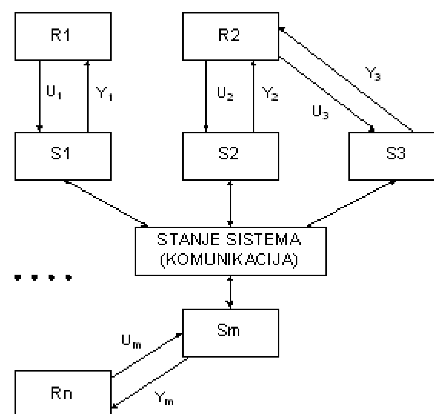
- Pretstavlja sistem kao "crnu kutiju"
- Pretstavlja se funkcionalnost sistema na način kako je vide spoljni objekti
- Pretstavljaju se ulazi i izlazi iz sistema i funkcije koje transformišu ulaze (pobudu, stimulaciju) u izlaze
- Pretstavlja model zahteva jer treba da pokaže potpuno, precizno i nedvosmisleno kako će objekti van sistema (korisnici, akteri) koristiti posmatrani sistem)

Zadatak funkcionalnog modelovanja je:

- *dekomponovanje složenog sistema na skup podsistema* (“logičkih jedinica posla”, “atomskih transakcija”, “slučajeva korišćenja”) **- SSA**
- *opis pojedinačnih podmodela* tako da, s jedne strane budu razumljivi korisniku, a sa druge da posluže da se iz njih na organizovan način mogu da formiraju ostali (objektni) modeli, odnosno nastavi i kontrološe dalji proces razvoja softverskog sistema - **SK**



(a) Sistem kao celina



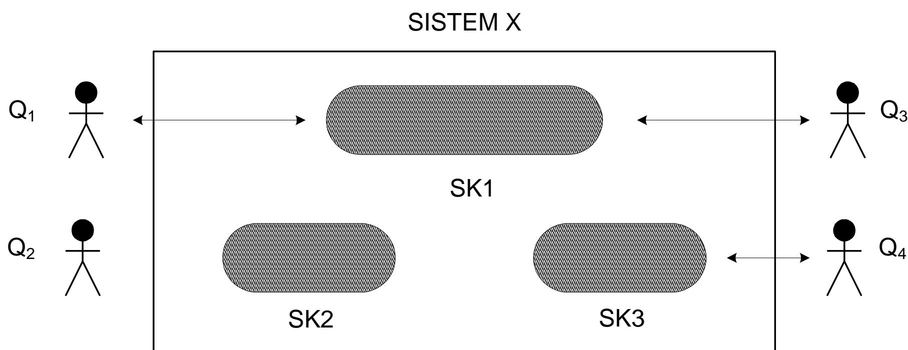
(b) Dekomponovani sistem sa podsistemima koji komuniciraju samo preko stanja sistema

Model slučajeva korišćenja

- **Pod terminom "slučaj korišćenja" podrazumeva se jedan specifičan način korišćenja IS,** jedna "atomska" funkcija IS. Preko "slučaja korišćenja" opisuje se interkcija nekog objekta van sistema sa samim IS. Skup "slučajeva korišćenja" predstavlja sve pretpostavljene načine korišćenja sistema.

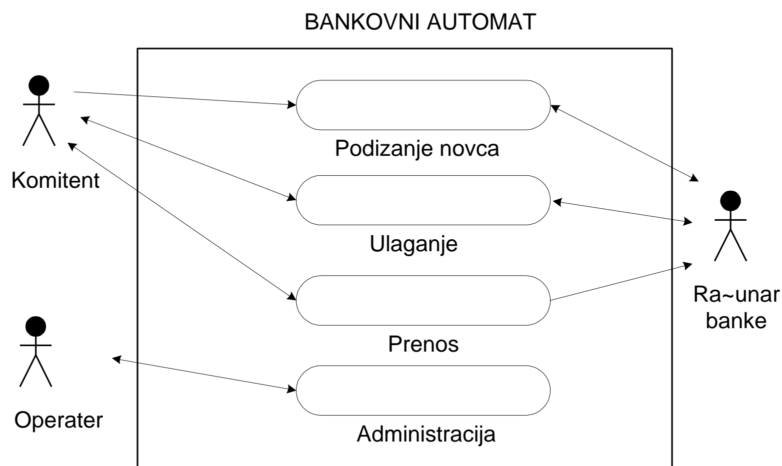
- **Model slučajeva korišćenja je graf sa dve vrste čvorova: čvorovima koje predstavljaju aktere i čvorovima koji predstavljaju slučajeve korišćenja.** Akter je objekat van sistema koji predstavlja tip (vrstu) korisnika. Akter je bilo šta što stupa u interakciju sa IS, ništa drugo van posmatranog IS nema nikakav uticaj na sistem. Akter može biti korisnik (čovek) ili neki drugi sistem. Treba praviti razliku između korisnika i aktera. Korisnik je čovek koji koristi sistem, dok je akter specifična uloga koju korisnik ima u komunikaciji sa sistemom.

Opšti model slučajeva korišćenja



Direktna komunikacija između dva aktera i dva konkretna (oni sa kojima komuniciraju akteri) slučaja korišćenja se ne može predstaviti na modelu (grafu). Međutim, kako će kasnije biti prikazano, moguće je definisati asocijaciju između klasa slučajeva korišćenja i klasa aktera (apstraktni akteri i apstrakti slučajevi korišćenja), da bi se jednostavnije prikazao neki složeni model.

Primer slučaja korišćenja



Opis slučaja korišćenja – scenario

- Svaki slučaj korišćenja treba da bude detaljno opisan. Mada je moguće davti i formalan opis slučaja korišćenja (dijagrami kolaboracije, dijagram promene stanja) preporučuje se da se u prvoj fazi koristi struktuirani verbalni opis, jer je on neophodan čak i ako se da neki formalni opis.

- Uobičajeno je, takođe, da se posebno daje opis normalnog toka događaja u slučaja korišćenja, a posebno mogući izuzeci.
- Jedan slučaj korišćenja pretstavlja **skup sekvenci događaja**. Jedna sekvenca događaja se naziva **scenario.** Postoji osnovni scenario i skup mogućih izuzetaka i alternativnih funkcionisanja

PODIZANJE NOVCA: **osnovni scenario**

Provera kartice: Komitent ubacuje karticu u automat. Automat čita karticu i proverava da li je prihvatljiva. Ako je prihvatljiva, zahteva se od komitenta da unese "tajnu šifru".

Proveravanje šifre: Komitent unosi tajnu šifru. Ako je šifra korektna zahteva se da korisnik izabere transakciju.

Unos tipa transakcije: Komitent bira "podizanje novca" i automat šalje računaru banke tajnu šifru da bi se dobili brojevi komitentovih računa. Dobijaju se komitentovi brojevi računa i prikazuju na ekranu automata.

Podizanja novca: Komitent bira račun i unosi iznos koji podiže.Automat šalje računaru banke zahtev za podizanje datog iznosa sa datog računa. Priprema se štampanje izveštaja za komitenta.

Kraj: Automat vraća karticu karticu komitentu. Izdaje se izveštaj komitentu

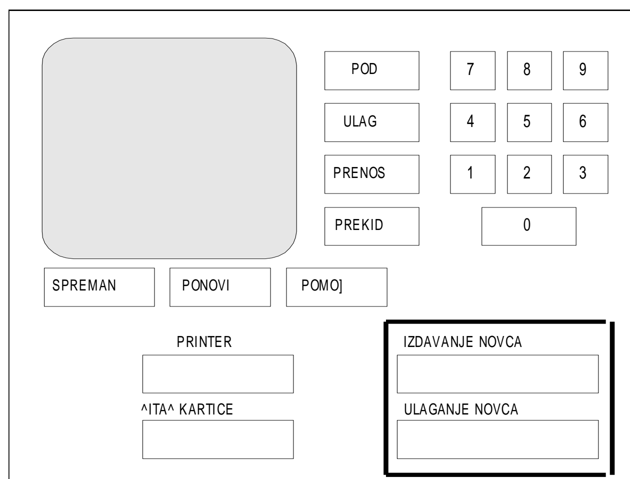
PODIZANJE NOVCA:- **alternativna scenaria**

Kartica nije prihvatljiva: Kartica se vraća korisniku sa zvučnim signalom.

Nekorektna tajna šifra: Odgovarajuća poruka se prikazuje na ekranu i daje se šansa korisniku da je ponovo unese. Dozvoljava se tri pokušaja, a zatim se vraća kartica korisniku.

Prekid: Korisnik može u svakom trenutku da prekine transakciju. Poništiće se svi dotadašnji efekti i vratiti kartica korisniku.

Mada SK treba, prvenstveno, da bude logički opis korišćenja sistema, treba imati u vidu i buduću arhitekturu sistema, a ponekad se opis daje preciznije ako je prethodno definisan korisnički interfejs. To ne sme da implicira zavisnost buduće aplikacije od interfejsa

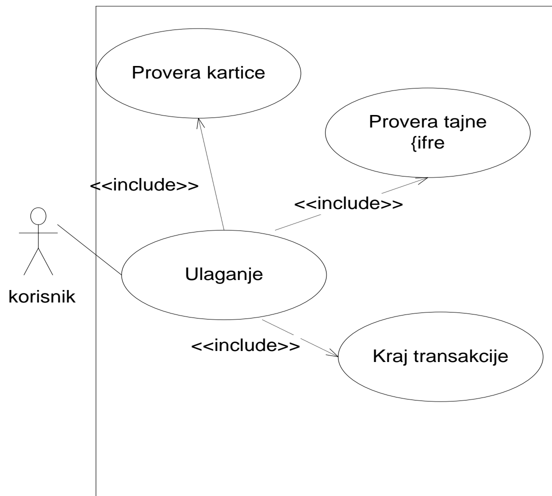


Veze u dijagramima slučajeva korišćenja

- **ASOCIJACIJA-** prikazana veza između aktera I slučaja korišćenja
- **GENERALIZACIJA-** veza opštijeg i specifičnijeg slučaja korišćenja koji nasleđuje opis opštijeg
- **<<extend>> -** stereotip veze zavisnosti koja referencira(ubacuje) moguće dodatno"ponašanje" opisano u posebnom apstraktnom SK, u osnovni SK
- **<<include>> -** stereotip veze zavisnosti koja eksplicitno ubacuje dodatno"ponašanje" opisano u posebnom apstraktnom SK, u osnovni SK.

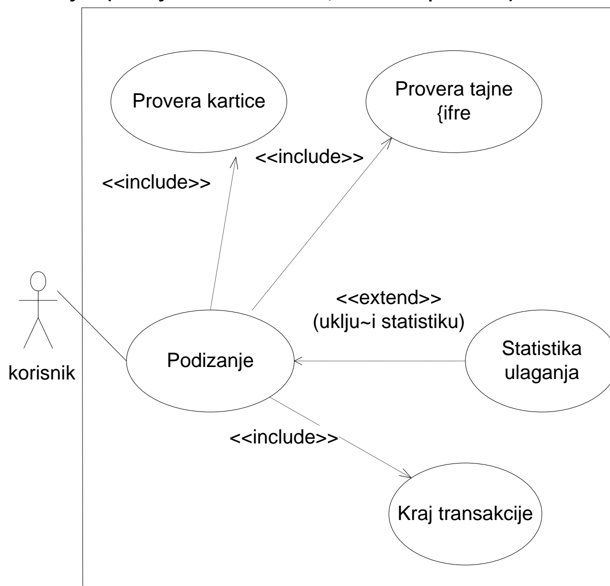
ILUSTRACIJE VEZE <<include>>:

Osnovni SK eksplicitno uključuje ponašanje opisano sa apstraktnim SK. Služi da se izbegne višestruko opisivanje istog ponašanja



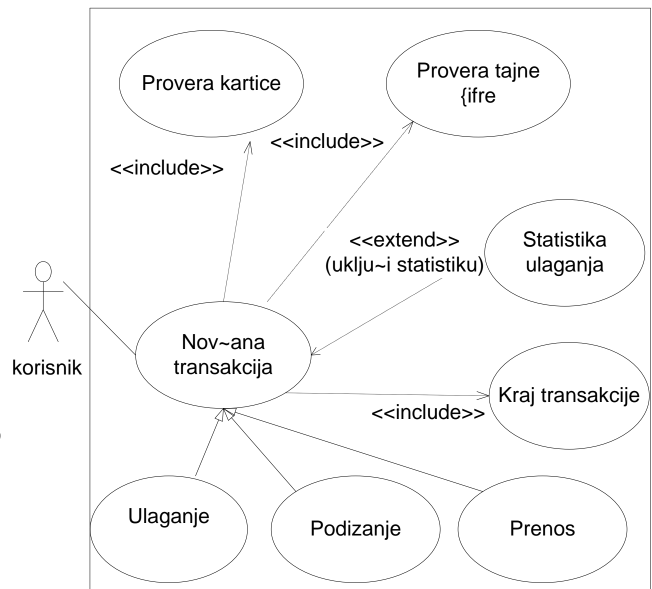
PRIMER VEZE <<extend>>

Osnovni SK implicitno proširuje ponašanje opisano u apstraktnom SK. Proširenje se vrši u tzv"tačkama proširenja"("uključi statistiku", za dati primer)



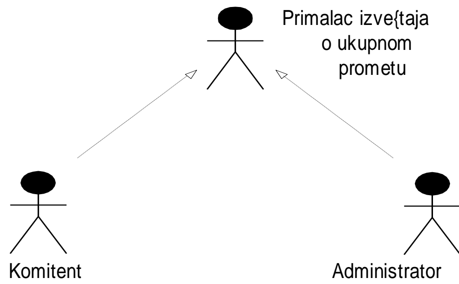
Primer generalizacije sk

Gde god se koristi SK nadtip, može se koristiti i SK podtip



Apstraktni akter

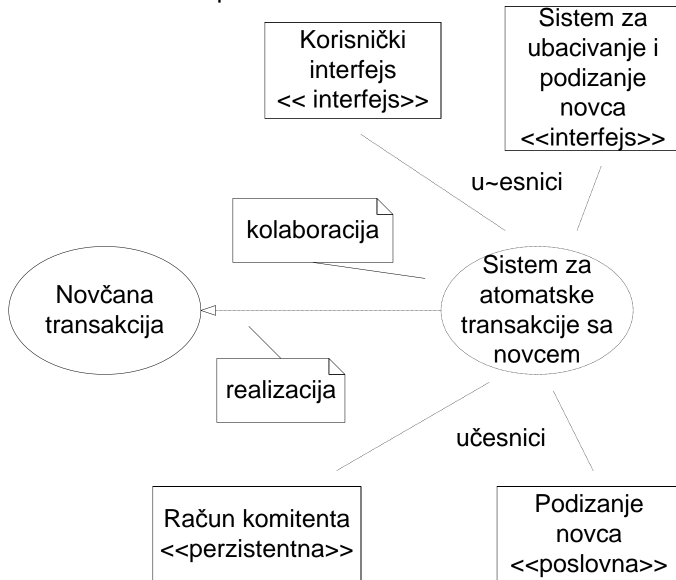
Kada dva aktera imaju slične uloge u odnosu na sistem oni mogu naslediti zajedničkog apstraktnog aktera.   Ako se isti slučaj korićenja može da poveže sa različitim akterima, pogodno je definisati apstraktnog aktera i opisati samo jedan slučaj korišćenja. Koncept apstraktnog aktera je takođe je koristan za opisivanje privilegija u korišćenju nekog sistema.



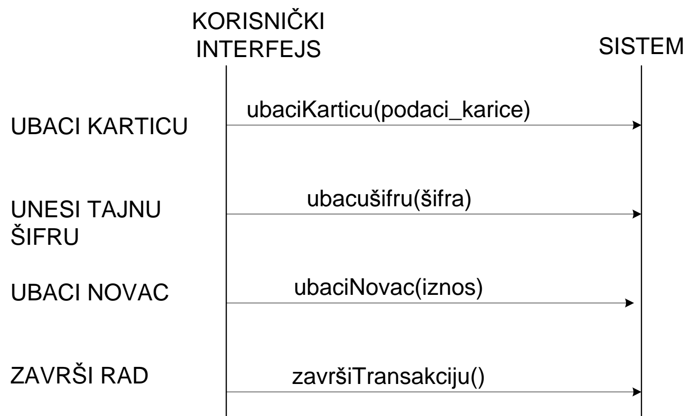
Kolaboracija i slučaj korišćenja

Kolaboracija "Sistem za automatske transakcije sa novcem" implemntira (realizuje) SK "Podizanje".
Kolaboracija je asocijacija elemenata koji u međusobnoj saradnji realizuju neki zahtev. (Navedene klase "učesnici")

"Use Case Driven Development Process"



Opis scenarija preko sistemskog dijagrama sekvenci



Problemi

- U nekom složenom sistemu broj slučajeva korišćenja može da bude veoma veliki. Kako definisati taj skup slučajeva korišćenja?
- Kako pokupiti "znanje" o strukturama podataka u postojećem sistemu, korisno da se izgradi konceptualni model sistema (dijagram klasa)?

- **Metoda SSA, kao metoda funkcionalne dekompozicije, može da bude odgovor, na ovo a i na neka druga pitanja modelovanja (modelovanje poslovnih procesa)**

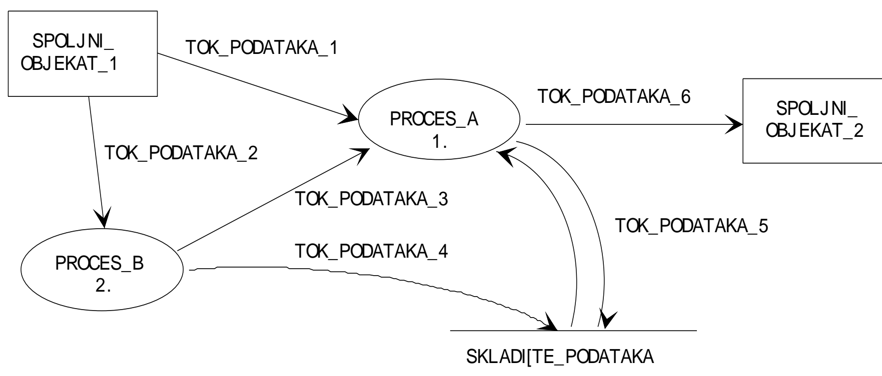
Strukturna sistemska analiza

- Strukturna sistemska analiza (SSA) je jedna potpuna konvencionalna metoda za specifikaciju informacionog sistema, odnosno softvera. Ona se na različite načine može povezati sa drugim metodama i modelima za projektovanje softvera, pa i sa objektnim metodama.
- SSA posmatra informacioni sistem kao funkciju (proces obrade) koja, na bazi ulaznih, generiše izlazne podatke. Ulazni podaci se dovode u proces obrade, a izlazni iz njega odvode preko tokova podataka. . Imajući u vidu zahtev da specifikacija treba da se oslobodi svih implementacionih detalja od interesa su samo sadržaj i struktura ulaznog toka, a ne i medijum nosilac toka.

Osnovni koncepti za specifikaciju IS u SSA su:

- **funkcije**, odnosno procesi obrade podataka,
- **interfejsi**, objekti van sistema sa kojima sistem komunicira preko tokova podataka,
- **tokovi podataka**,preko kojih se podaci prenose između interfejsa, funkcije i skladišta,
- **skladišta podataka**, u kojima se permanentno čuvaju stanja sistema.

Njihov međusobni odnos se prikazuje preko dijagrama toka podataka koji prikazuje vezu interfejsa, odnosno skladišta kao izvora odnosno ponora podataka, sa odgovarajućim procesima, kao i međusobnu vezu procesa.



Imajući u vidu sve rečeno, jednu potpunu specifikaciju IS čine:

1.Hijerarhijski organizovan **skup dijagrama toka podataka**;
2.**Rečnik podataka** koji opisuje sadržaj i strukturu svih tokova skladišta podataka;
3.**Specifikacija logike primitivnih procesa**;

Pored dijagrama tokova podataka uobičajeno je za jedan sistem da se prikaže i Dijagram dekompozicije koji prikazuje celokupnu dekompoziciju sistema, od Dijagrama konteksta do primitivnih funkcija. Ovde se predlaže da se za izradu Dijagrama dekompozicije koriste Jackson-ovi dijagrami, jer se sa njima može opisati i logika primitivnuh procesa i time izvršiti bolja priprema za konstukciju Dijagram slučajeva korišćenja

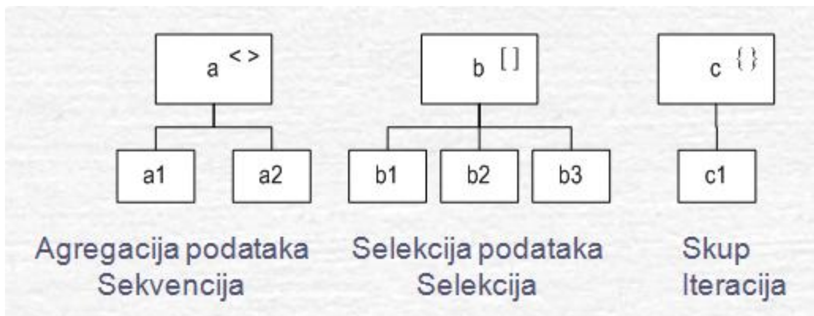
SSA- sintaksna i metodološka pravila

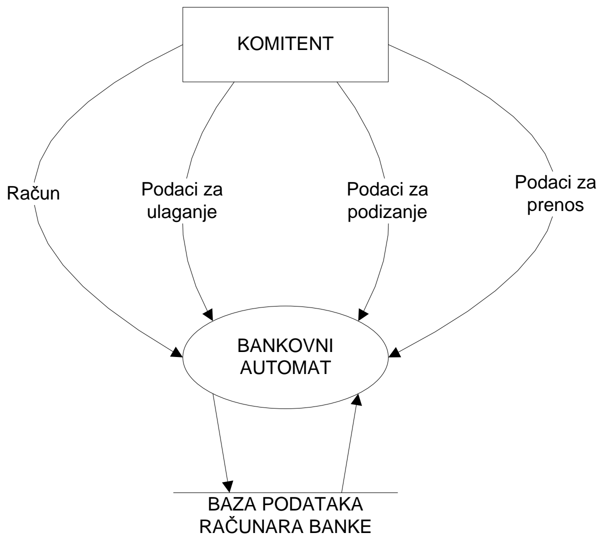
- Za formiranje DTP - ova postoji čitav skup formalnih (sintaksnih) pravila. Najznačajnije je pravilo koje se mora poštovati pri dekompoziciji procesa, **pravilo balansa tokova**: Ulazni i izlazni tokovi na celokunom DTP-u koji je dobijen dekompozicijom nekog procesa P moraju biti ekvivalentni sa ulaznim i izlaznim tokovima toga procesa P na dijagramu višeg nivoa. Pri tome se uzima u obzir dekompozicija tokova predstavljena u rečniku podataka.

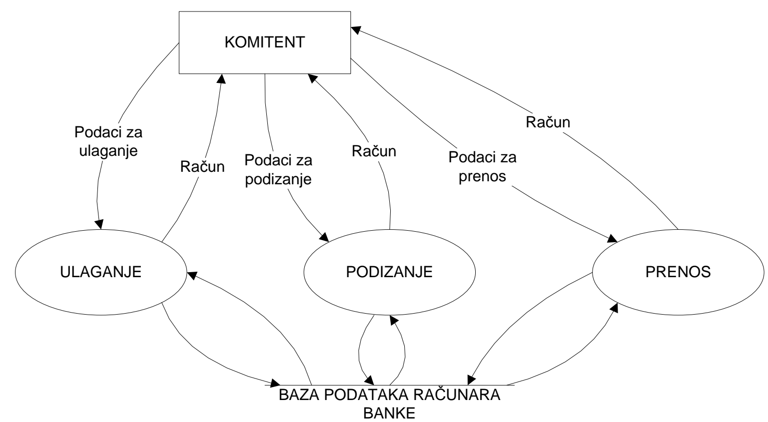
- Kao najvažnije metodološko pravilo koristi se pravilo da **funkcije na DTP-u između sebe treba da komuniciraju isklljučivo preko skladišta.** Korišćenje ovoga pravila vodi uvek ka identifikaciji nekih fundametalnih funkcija u sistemu, fundamentalnih sa sledeće dve tačke gledišta:
  (I) funkcije su autonomne, jedna od druge zavise isključivo preko skladišta;
  (II) bilo koja složenija funkcija dobija odgovarajućim kobinovanje fundamentalnih

JACKSON-ovi dijagrami za opis strukture podataka i programa

- Potiču iz Jackson-ove metode razvoja programa koja polazi od stava da se struktura program može dedukovati iz strukture podataka na njegovom ulazu I izlazu.
- Koriste se sledeće oznake:

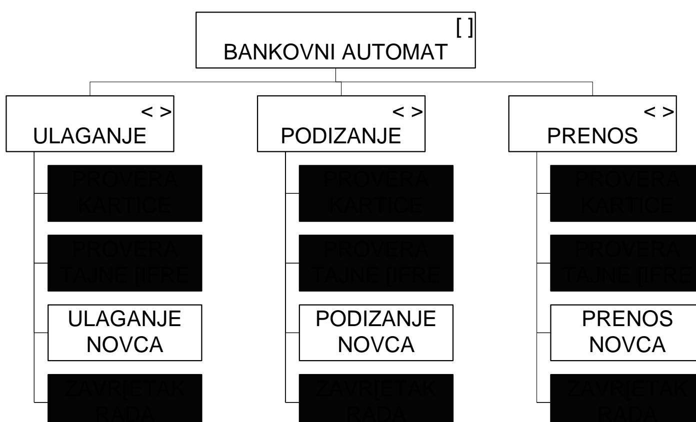






Jackson-ov dijagram dekompozicije

Označeni su zajednički delovi u različitim primitivnim procesima – kadidati za apstraktne slučajeve korišćenja



4. Konceptualno modelovanje

- Konceptualni model predstavlja suštinske karakteristike sistema za koji se projektuje baza podataka.
- Opisuje se na visokom nivou apstrakcije, preko modela podataka.
- Predstavlja celokupan mogući informacioni sadržaj baze podataka, nezavisno od toga koji je deo tog sadržaja i kako implementiran.

Konstruisanje konceptulanog modela

- Koriste se "intelektualni alati" kao što su model podataka ' PMOV, IDEF1X, Dijagram klasa,
- Postupak modelovanja je uvek "veština", zavisi od sposobnosti, znanja i iskustva analitičara.
- Ne mogu se dati neka stroga pravila modelovanja koja bi, bez obzira na to ko modelovanje vrši, vodila do jedinstvenog modela složenog realnog sistema.
- Mogu se dati samo opšte metodološke preporuke, opšti metodološki pristupi, kao pomoć u ovom složenom poslu..

Pristupi konceptualnom modelovanju

Postoji više metodoloških pristupa konceptualnom modelovanju, a u razvoju modela nekog konkretnog sistema oni se gotovo uvek kombinuju.

1. Integracija podmodela;
2. Direktno modelovanje na bazi verbalnog opisa sistema;
3. Konkretizacija opštih (generičkih) modela, odnosno korišćenje uzora ("paterna");
4. Normalizacija relacija; - (specifican za relacioni model)
5. Transformacija jednog modela u drugi ("direktno" i "inverzno" inženjerstvo).

PROJEKTOVANJE RELACIONIH BAZA PODATAKA

1. Primena normalnih formi na analizu relacija
2. Primena normalnih formi na sintezu relacija
3. Transformacija drugih modela u relacioni

NORMALNE FORME - projektovanje relacija normalizacijom

Normalizacija je postupak projektovanja logičke strukture baze podataka. Uobičajeno je da se koristi za projektovanje logičke strukture relacionog modela. Međutim, postupak normalizacije ima opštiji značaj i treba ga primenjivati i na druge modele baze podataka (mrežni i hijerarhijski, relaciono-objektni), a i za projektovanje strukture zapisa u obradi podataka zasnovanoj na kolekciji izolovanih datoteka.

Anomalije u ažuriranju (dodavanju zapisa, izbacivanju zapisa, izmeni vrednosti atributa)

Anomalije (nesimetričnost) u izveštavanju

Relacija R je u Prvoj normalnoj formi (1NF) ako su sve vrednosti njenih atributa atomske

FUNKCIONALNE ZAVISNOSTI ATRIBUTA RELACIJE

1. Data je relacija R sa atributima X i Y, moguće složenim. Atribut Y je funkcionalno zavisan od atributa X (ili X funkcionalno određuje Y),

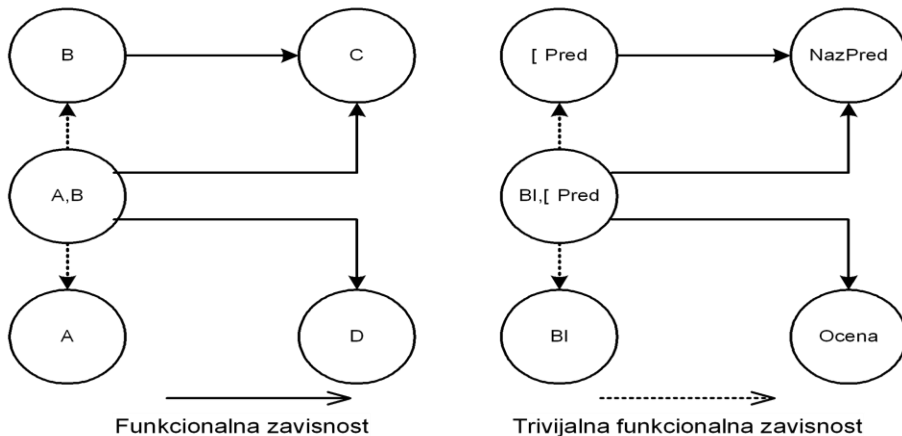
R.X --> R.Y,

ako i samo ako svakoj vrednosti X odgovara jedna i samo jedna vrednost Y.

Definicija funkcionalne zavisnosti se može dati i na sledeći način:
Atribut Y relacije R je funkcionalno zavisan od atributa X relacije R ako i samo ako, kad god dve n-torke relacije R imaju istu x-vrednost one moraju imati i istu y-vrednost.

Očigledno je da iz definicije funkcionalne zavisnosti sledi i nova definicija ključa i nadključa relacije:
Atribut X, moguće složeni, je nadključ neke relacije R ako i samo ako funkcionalno određuje sve ostale atribute relacije R.
Atribut X, moguće složeni, je ključ relacije R ako je nadključ relacije R, a nijedan njegov pravi podskup nema tu osobinu.

2. Atribut Y relacije R je potpuno funkcionalno zavisan od atributa X relacije R ako je funkcionalno zavisan od atributa X, a nije funkcionalno zavisan ni od jednog pravog podskupa atributa X.



(a) Graf definicije potpune funkcionalne zavisnosti (b) Primer potpune funkcionalne zavisnosti.

Tranzitivna funkcionalna zavisnost se definiše na sledeći način:
Data je relacija R sa atributima A, B i C, moguće složenim.
Ako u relaciji R važi:

A	--->	B
B	--->	C
A	--->	C
B	-/->	A
C	-/->	A,

atribut C je tranzitivno funkcionalno zavisan od atributa A.

Jednostavnije rečeno, *atribut C je tranzitivno funkcionalno zavisan od atributa A ako je funkcionalno zavisan od A i ako je funkcionalno zavisan od nekog atributa B koji je i sam funkcionalno zavisan od A*

- Relacija R je u Drugoj normalnoj formi (2NF) ako i samo ako je u 1NF i svi njeni neključni atributi potpuno funkcionalno zavise od primarnog ključa. (Neključni atributi su atributi koji nisu kandidati za ključ, niti deo kandidata za ključ)
- Relacija R je u Trećoj normalnoj formi (3NF) ako i samo ako je u 2NF i ako svi njeni neključni atributi netranzitivno zavise od primarnog ključa.

Relacija R je u 2NF i u 3NF ako svi njeni atributi daju jednoznačne činjenice o celom ključu i samo o celom ključu.

- Boyce-Codd-ova (BCNF) normalna forma uklanja neke nepreciznosti u definisanju 2NF i 3NF. Za definiciju BCNF uvodi se i pojam **determinante relacije.**
- Determinanta relacije R je bilo koji atribut, prost ili složen, od koga neki drugi atribut u relaciji potpuno funkcionalno zavisi.
- Relacija R je u Boyce-Codd-ovoj normalnoj formi (BCNF) ako i samo ako su sve determinante u relaciji i kandidati za ključ.

- Relacija R je u 4NF ako u njoj nisu date dve (ili više) nezavisne višeznačne činjenice.
- Relacija je u 5NF onda kad se njen informacioni sadržaj ne može rekonstruisati iz relacija nižeg stepena, s tim što se slučaj relacija nižeg stepena sa istim ključem isključuje.
- Relacija je u Normalnoj formi ključeva i domena (DK/NF) ako je svako ograničenje na vrednosti njenih atributa posledica definicije ključeva i domena.

Četvrta, peta i normalna forma kjučeva i domena

U relaciji R(A, B, C) postoji višeznačna zavisnost A ->-> B ako za datu vrednost A, postoji skup od nula, jedne ili više vrednosti B, a taj skup vrednosti ni na koji način ne zavisi od vrednosti atributa C. Atributi A, B i C mogu biti složeni.

Formalna definicija višeznačnih zavisnosti može se dati i na sledeći način:
U relaciji R(A,B,C) postoji višeznačna zavisnost A ->-> B ako i samo ako kad god u njoj postoje n-torke <a,b,c> i <a,b',c'>, postoje takođe i n-torke <a,b,c'> i <a,b',c>. Atributi A, B i C mogu biti složeni.

U relaciji R(X, Y, ..., Z) postoji zavisnost spajanja ako i samo ako relacija R rezultuje iz prirodnog spajanja njenih projekcija po X,Y, ..., Z, gde su X,Y, ..., Z podskupovi atributa relacije R.
Relacija R je u Petoj normalnoj formi ako i samo ako se svaka zavisnost spajanja može pripisati kandidatu za ključ.

Primena normalnih formi na projektovanje baze podataka analizom relacija

- **Dat je skup nenormalizovanih relacija**. (na primer, svaka “stavka” rečnika podataka strukturne sistemske analize može se tretirati kao nenormalizovana relacija)
- **Primenom definicija normalnih formi, relacije se normalizuju**. Pri tome se može odmah primeniti dk/nf, ali je mnogo praktičnije postepeno primenjivati definicije 2nf, 3nf, bcnf 4nf, 5nf.
- **Dobijene relacije sa istim ključem mogu da se konsoliduju** (spoje) ako se time ne narušava semantika sistema.

Transformacija modela objekti-veze u relacioni model

P1. Pravilo za objekte. Svaki objekat PMOV postaje relacija sa odgovarajućim imenom. Atributi ove relacije su:

- svi atributi posmatranog objekta,
- svi identifikatori objekata prema kojima posmatrani objekat ima preslikavanje sa donjom i gornjom granicom kardinalnosti jednakom 1. Pri tome treba imati u vidu i"trivijalna" preslikavanja koja se na DOV ne prikazuju (preslikavanje od slabog prema nadredjenom, od agregacije prema komponenti, od podtipa prema nadtipu), a čije kardinalnosti su uvek (1,1).

Ključ relacije je:

- identifikator objekta, za objekte "jezgra" (objekte koji nisu ni agregacije, ni slabi, ni podtipovi),
- za slab objekat, identifikator nadređenog objekta, proširen sa jednim ili više atributa slabog, koji jedinstveno identifikuju slabi objekat u okviru nadredjenog,
- za agregaciju, složeni kljuc koga čine identifikatori objekata koji čine agregaciju,
- za podtip, identifikator nadtipa.

P2. Dodatno pravilo za veze.

Preko pravila (P1) u relacioni model su prevedeni svi objekti i sve veze koje imaju barem jedno preslikavanje kardinalnošću (1,1) iz MOV.

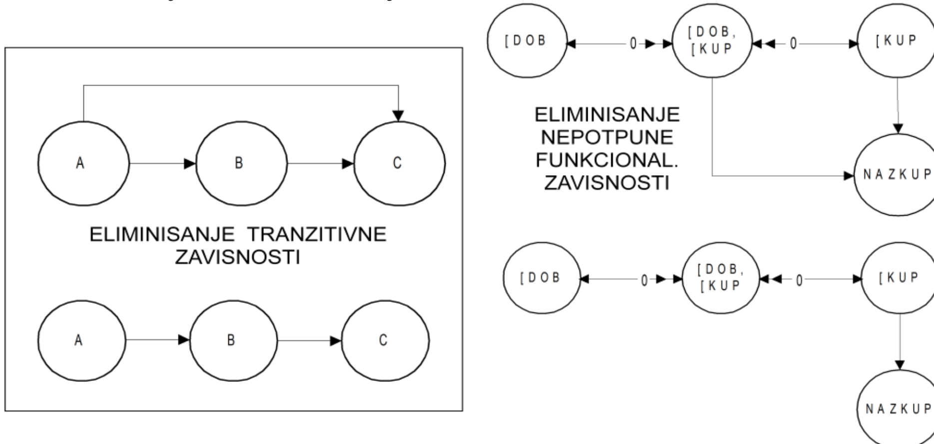
Sve ostale veze postaju posebne relacije. Ime relacije je ime veze.

Atributi ovih relacija su identifikatori objekata koji čine vezu.

Ključ ovako dobijene relacije je:

- složeni ključ koga čine oba identifikatora, ako je gornja granica kardinalnosti oba preslikavanja M,
- identifikator objekta sa gornjom granicom kardinalnosti preslikavanja 1, ako jedno preslikavanje ima gornju granicu 1, a drugo M
- identifikator bilo kog objekta, ako oba preslikavanja imaju gornju granicu kardinalnosti preslikavanja jednaku 1.

Sinteza relacija na osnovu teorije funkcionalnih zavisnosti



- Prethodni grafovi ukazuju na mogućnost definisanja postupka za sintezu relacija na osnovu zadatog skupa atributa i njihovih međusobnih funkcionalnih zavisnosti Da bi se ovaj problem formalno definisao uvedimo i sledeće definicije:
- **Zatvaranje skupa funkcionalnih zavisnosti** F je skup F^+svih funkcionalnih zavisnosti koje se mogu dedukovati iz F, preko negog skupa pravila zaključivanja.
- Posmatrajmo dva skupa funkcionalnih zavisnosti F i G. Kaže se da je skup **G prekrivanje skupa F** ako je $G^+ = F^+$, tj ako su im zatvaranja ista. U tom slučaju se, kaže da su skupovi G i F ekvivalentni.

Formalno se problem projektovanja baze podataka može definisati na sledeći način:
Za skup funkcionalnih zavisnosti definisan modelom sistema naći minimalno prekrivanje i implementirati ga preko odgovarajućeg skupa relacija.

Pravila zaključivanja za izvođenje zatvaranja

- Pravila zaključivanja su osobine funkcionalnih zavisnosti na osnovu kojih, iz jedne ili više funkcionalnih zavisnosti nekog skupa, možemo logički dedukovati neke druge.

- Armstrong-ove aksiome predstavljaju, jedan neprotivrečan i potpun skup pravila zaključivanja. Neprotivrečan, u smislu da se iz njega mogu dedukovati samo važeće funkcionalne zavisnosti, a potpun, u smislu da se na osnovu njega mogu dedukovati sve funkcionalne zavisnosti.

ARMSTRONG-ove AKSIOME

Oznaka U se nadalje koristi kao oznaka za skup atributa neke relacije.

A1. REFLEKSIVNOST. Ako je $Y \subseteq X \subseteq U$, tada važi X -> Y. Ovo pravilo generiše trivijalne funkcionalne zavisnosti. Trivijalne funkcionalne zavisnosti definisane su skupom atributa U neke relacije, a ne samim funkcionalnim zavisnostima koje u relaciji važe.

A2. PROŠIRENJE. Ako važi X -> Y i ako je $Z \subseteq U$, tada važi i XZ -> YZ. (Oznaka XZ je skraćena oznaka za X U Z).

A3. TRANZITIVNOST. Ako postoje funkcionalne zavisnosti X -> Y i Y -> Z, tada postoji i X -> Z.

Iz Armstrongovih aksioma mogu se izvesti i sledeća korisna dodatna pravila zaključivanja:

D1.ADITIVNOST (pravilo unije). Ako postoje funkcionalne zavisnosti X -> Y i X -> Z, tada postoji i Y -> YZ

D2. PSEUDOTRANZITIVNOST. Ako postoje funkcionalne zavisnosti X -> Y i YW -> Z, tada postoji i funkcionalna zavisnost XW -> Z.

D3.DISTRIBUTIVNOST(pravilo dekompozicije). Ako postoji funkcionalna zavisnost X -> YZ, tada važi i X -> Y i X -> Z.

Uslovi da je h minimalni prekrivač f

1.Desne strane funkcionalnih zavisnosti u H su pojedinačni atributi.

2. Za svaku funkciju f: X ---> A iz F, ako je ispunjeno
   $[F - \{X \text{ ---> } A\}]^+ = F^+$, tada $f \notin H$. (Eliminiše redundantne funkcionalne zavisnosti.)
3. Ni za jedno X ---> A iz H, za bilo koji pravi podskup Z od X ($Z \subset X$), $[H - \{X \text{ ---> } A\} \text{ U } \{Z \text{ ---> } A\}]$ nije ekvivalentno sa H. (Ne postoji nepotreban atribut na levoj strani)
4. $F^+ = H^+$, odnosno zatvarači skupa funkcionalnih zavisnosti i njegovog minimalnog prekrivača su jednaki. (Garantuje da su sve fukcionalne zavisnosti sačuvane u minimalnom prekrivaču.)

BERNSTEIN-ov algoritam za nalaženje minimalnog prekrivanja

Na osnovu pravila dekompozicije, svesti skup funkcionalnih zavisnosti F na skup funkcionalnih zavisnosti F1 čije su desne strane pojedinačni atributi.

1. Eliminisanje nepotrebnih atributa. Eliminiši nepotrebne atribute iz F1 i formiraj skup funkcionalnih zavisnosti G koji je ekvivalentan sa F1 ($G^+ = F1^+$).
2. Nalaženje neredundantnog prekrivača. Nađi neredundantni prekrivač H skupa funkcionalnih zavisnosti G.
3. Podela i grupisanje. Podeli skup funkcionalnih zavisnosti u H u grupe H_k tako da u svakoj grupi leve strane budu identične.
4. Spajanje ekvivalentnih ključeva. Za svaki par grupa H_i i H_j, sa levim stranama X i Y spoji H_i i H_j zajedno ako postoji bijekcija X <-----> Y u H. Za svako $A \in Y$ izbaci X ---> A iz H. Za svako $B \in X$ izbaci Y ----> B iz H. Formiraj skup J u koji se stavljaju zavisnosti X ---> Y i Y -- -> X za svaku bijekciju grupa.
5. Eliminisanje novodobijenih tranzitivnih zavisnosti posle koraka 4. Nađi prekrivač $H' \subseteq H$ tako da je $(H', J)^+ = (H, J)^+$ i da nijedan pravi podskup od H' nema tu osobinu.
6. Konstruisanje relacija. Za svaku grupu H_k konstruiši jednu relaciju sa svim atributima koji se pojavljuju u grupi. Leva strana funkcionalnih zavisnosti u grupi je ključ relacije.

- Nalaženje minimalnog prekrivača nije jednoznačan zadatak, odnosno da jedan skup funkcionalnih zavisnosti ima više minimalnih prekrivača.

- Osnovni problem u gornjem algoritmu je nalaženje minimalnog prekrivača i njegovog zatvarača H^+. Algoritam nalaženja minimalnog prekrivača skupa funkcionalnih zavisnosti F može se realizovati tako što će se, postepeno, eliminisati iz skupa F one funkcionalne zavisnosti f_i za koje važi $(F - f_i)^+ = F^+$. To znači da se algoritam nalaženja minimalnog prekrivača svodi na algoritam nalaženja zatvarača.

Algoritam za nalaženje zatvarača

- Nalaženje zatvarača F^+ skupa funkcionalnih zavisnosti F direktnom primenom Armstrongovih aksioma je eksponencijalne složenosti. Zbog toga se primenjuje postupak nalaženja tzv. "Zatvarača skupa atributa": Neka je F skup funkcionalnih zavisnosti nad skupom atributa U i neka je $X \subseteq U$. Tada je X^+, zatvarač podskupa X u odnosu na F, skup atributa A takav da se X ---> A može dedukovati iz F pomoću Armstrongovih aksioma.
- Može se dokazati sledeća lema: Neka funkcionalna zavisnost X ---> Y može se dedukovati iz F pomoću Armstrongovih aksioma, ako i samo ako je $Y \subseteq X^+$, a X^+ je zatvarač skupa X a odnosu na F.

Algoritam za nalaženje zatvarača skupa atributa

Ulaz: Skup atributa U, skup funkcionalnih zavisnosti F nad U i podskup $X \subseteq U$.

Izlaz: X^+, zatvarač X u odnosu na F.

Postupak: Sračunava se niz skupova atributa $X^{(0)}$, $X^{(1)}$, ..., preko pravila:

1. $X^{(0)} = X$
2. $X^{(i,1)} = X^{(i)}$ U V, gde je V takav skup atributa A da postoji neka funkcija Y ---> Z, gde je $A \in Z$, a $Y \subseteq X^{(i)}$. Kako se $X^{(j,1)}$ sračunava samo na osnovu $X^{(j)}$ i kako je skup U konačan, postupak se završava kada postane $X^{(i,1)} = X^{(i)}$.

PRIMER BERNSTEIN-ovog ALGORITMA

Dat je skup atributa U = {A, B, C, D, X1, X2}

Dat je skup funkcionalnih zavisnosti:

G = {X1X2 -> AD, CD -> X1X2, AX1 -> B, BX2-> C, C -> A}

Korak 0:

f1: X1X2 ---> A f5: AX1 ---> B

f2: X1X2 ---> D f6: BX2 ---> C

f3: CD ---> X1 f7: C ---> A

f4: CD ---> X2

Korak 1: Nema nepotrebnih atributa.

Korak 2: Proverava se redom da li se svaka fi može dedukovati iz tranzitivnog zatvarača preostalih, odnosno da li je

$fi \in [F - fi]^+$.

Za funkciju f1, na primer, to je ekvivalentno proveri da li je $A \in X1X2$ u odnosu na F - f1:

$X^{(0)}$ = X1X2 $X^{(1)}$ = X1X2 (zbog f2)

Prema tome X1X2 ---> A ne može se dedukovati iz $(F - f1)^+$ pa zbog toga pripada neredundantnom prekrivaču.

Isto se može zaključiti i za preostale fi iz F.

Korak 3: Podela i grupisanje. Vrši se grupisanje funkcionalnih zavisnosti u sledeće grupe:

H1={f1, f2}, H2={f3, f4}, H3={f5}, H4={f6}, H5={f7}

Korak 4:Spajanje ekvivalentnih ključeva. Proverava se da li između levih strana funkcionalnih zavisnosti nekih grupa postoji bijekcija. Bijekcija postoji ako je $N \in M^+$, a $M \in N^+$.

Sračunajmo X1X2′ u odnosu na skup H koji je u ovom slučaju jednak skupu F:

$X^{(0)}$ = X1X2 $X^{(1)}$ = X1X2AD (zbog f1 i f2)
$X^{(2)}$ = X1X2ADB (zbog f5)
$X^{(3)}$ = X1X2ADBC(zbog f3, f4, f5, f6)
X1X2′ = X1X2ADBC

Sračunajmo CD′ u odnosu na skup H:
$X^{(0)}$ = CD
$X^{(1)}$ = CDX1X2A (zbog f3, f4 i f7)
$X^{(2)}$ = CDX1X2AB (zbog f5)
$X^{(3)}$ = CDX1X2AB (zbog f6)
CD′ = X1X2ADBC

Prema tome postoji bijekcija X1X2 <---> CD. Zbog toga se zavisnosti X1X2---> A, X1X2 ---> B, CD ---> X1, CD --->X2 spajaju u grupu (X1, X2, C, D, A). U skup J treba dodati gornju bijekciju, odnosno četiri zavisnosti: X1X2 ---> C, X1X2 ---> D, CD ---> X1 i CD ---> X2. Kako zadnje tri već postoje u F, dodaje se samo f8: X1X2 ---> C.

Korak 5. Eliminisanje novodobijenih tranzitivnih zavisnosti. Novi skup funkcionalnih zavisnosti H , J obuhvata zavisnosti f1 do f8. Eliminisanje tranzitivnih zavisnosti se radi ponovo preko Koraka 2, uključujući i f8. Prvo se ispituje da li je sada f1 redundantna. Sračunava se X1X2′ u odnosu na H , J - f1.
$X^{(0)}$ = X1X2
$X^{(1)}$ = X1X2DC (na osnovu f2 i f8)
$X^{(2)}$ = X1X2DCA (na osnovu f7)
X1X2′ = X1X2DCA
Kako je A $\in$ X1X2′ funkcionalna zavisnost f1 je redundantna. Ostale funkcionalne zavisnosti (koje bi proveravali na isti način) nisu.

Korak 6. Konstruišu se sledeće relacije:
R1(<u>X1, X2</u>, C, D)
R2(<u>A, X1,</u> B)
R3(<u>B, X2</u>, C)
R4(<u>C, A</u>)

Fizičko projektovanje relacionih baza podataka

- Fizičko projektovanje baza podataka se vrši nakon potpunog logičkog projektovanja, na osnovu jasne logičke strukture baze podataka.
- Fizičko projektovanje baza podataka veoma je zavisno od konkretnog subp.
- Nije uobičajeno da se vrše neki detaljni “fizički proračuni”, radije se primenjuju ekspertska znanja i kasnije podešavanje fizičke strukture.

Fizičko projektovanje relacionih baza podataka - koraci:

1. Prilagođavanje logičke strukture konkretnom skupu aplikacija - denormalizacija
2. Distribucija baze podataka - različite “klijent- server” arhitekture
3. “klasterovanje” - podaci koji se zajedno koriste treba da budu fizički bliski
4. Određivanje metoda pristupa (indeksiranje i eventualno “hešing”)

5. Konceptualno modelovanje

Metodologija modelovanja

1. Integrisanje podmodela
2. Direktno modeliranje
3. Kombinovani metod

Integracija podmodela

Postupak integracija podmodela podrazumeva:

1. Izgradnja poslovnog modela sistema i specifikacija identifikovanih aplikacija
    - korišćenjem Strukturne sistemske analize
    - Ili opisa slučajeva korišćenja preko Sistemskog dijagrama sekvenci sa, na primer XML opisom argumenata poruka
    - ili Dijagramom aktivnosti sa strukturom objektnih tokova opisanih preko nekog modela podataka.
2. Izgradnja podmodela podataka za svaku specifikovanu aplikaciju, odnosno primitivni poslovni proces (funkciju).
3. Integracija podmodela u jedinstveni model celog sistema.

Struktura tokova i skladišta

Izgled toka otprenica dobaljača (otprdob)

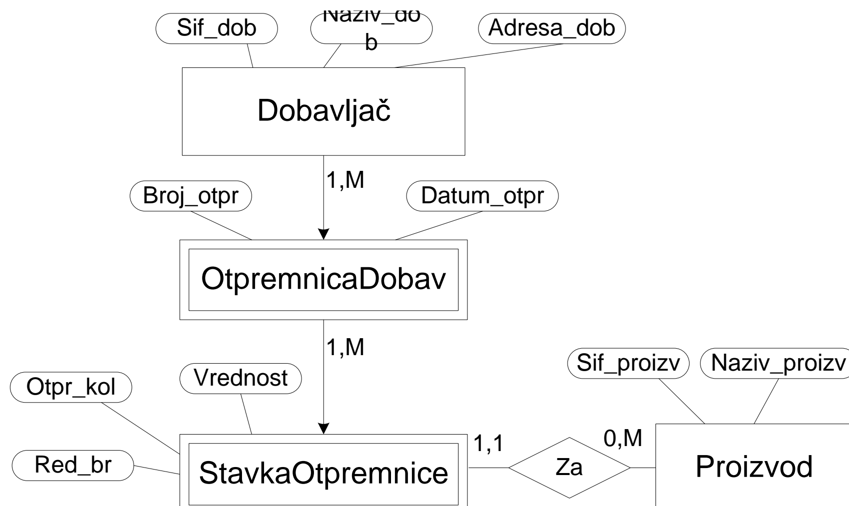
[if_dob .....   Naziv_dob ....   Adresa_dob ....
Br_otpr .....   Datum_otpr.......

Red_br	[if_proizv	Naziv_proizv	Otpr_kol	Vrednost
...	....	.....	.....	.....
....	.....	......	.,.....	.....

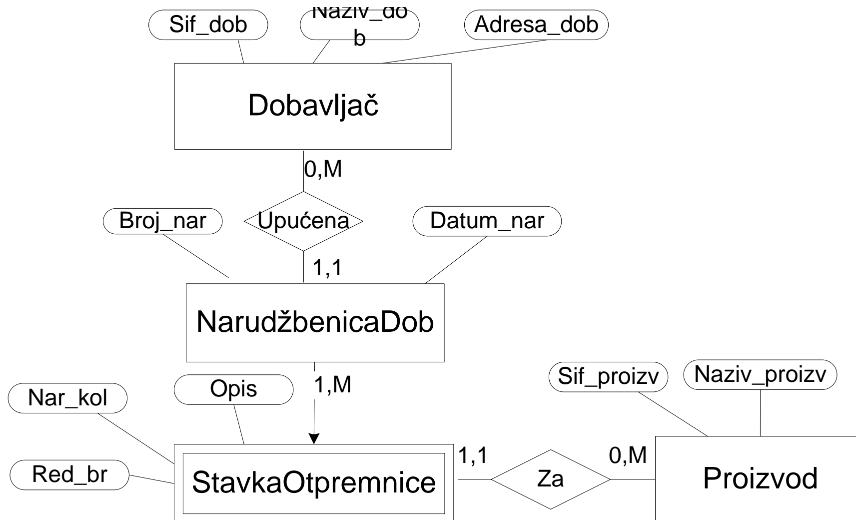
Definicija u rečniku ssa

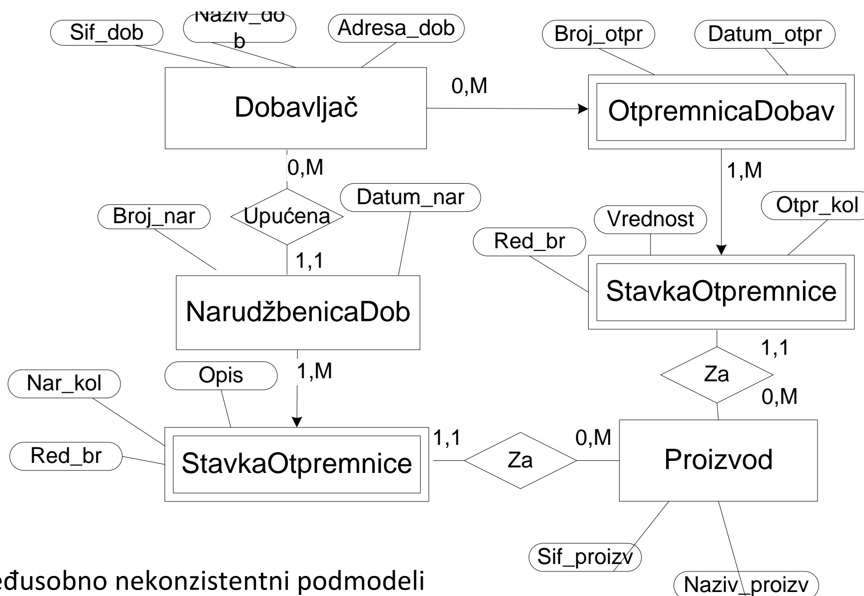
OTPDOB: < Sif_dob, Naziv_dob, Adresa_dob, Br_otpr, Datum_otpr,{<Red_br , Sif_proizv, Naziv_proizv, Otpr_kol, Vrednost>}>

Podmodel za otpremnicu dobavljača

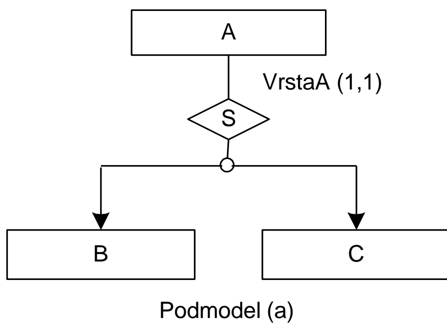


NARDOB: < [if_dob, Naziv_dob, Adresa_dob, Br_nar, Datum_nar,{<Red_br , [if_proizv, Naziv_proizv, Nar_kol, Opis>}>

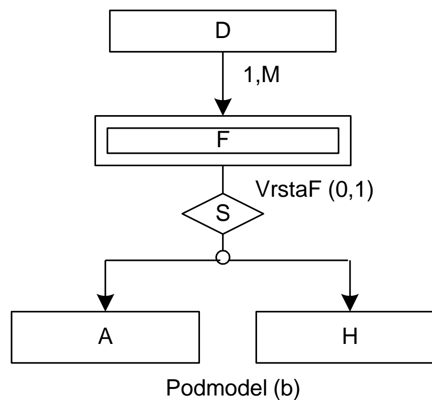




Međusobno nekonzistentni podmodeli



Podmodel (a)



Podmodel (b)

Uzor za konstruisanje model podataka u poslovnim sistemima

Strukture tokova i skladišta podataka su osnova za nalaženja modela. Delovi navedenog primera mogu se uopštiti, odnosno poslužiti kao uzor:

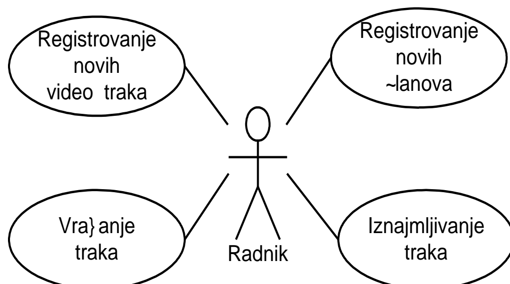
- Neki tok ili skladište podataka se sastoji od jednog ili više poslovnih dokumenata. Jedan poslovni dokument prikazuje neki skup povezanih objekata. Neophodno je identifikovati te objekte, utvrditi koja polja dokumenta predstavljaju njihove atribute i razlučiti veze objekata koje su na njemu prikazane.
- Može se konstatovati da, najčešće, poslovni dokumenti imaju jednu od sledeća dva tipa struktura: (1) "ravni dokumenat" u kome se sva polja pojavljuju samo jednom (Faktura, UplataDob) i (2) "dokumenat sa stavkom" u kome postoje stavke (tabele), odnosno skup polja koje se višestruko ponavljaju (NarudžbenicaDob, Otremnica, Katalog). Pogodno je stavke tretirati kao posebne "slabe" objekte u modelu. Isto tako je pogodno pretpostaviti da stavke imaju redne brojeve.
- Dokumenta koja se primaju iz "okoline" treba tretirati kao "slabe objekte", sa njihovim pošiljaocima kao nadređenim objektom (Otpremnica, Faktura, Katalog). Očigledno je da BrojDokumenta ne može biti njihov jedinstveni identifikator, već se identifikacija mora proširiti i sa identifikatorom pošiljaoca.

Direktno modelovanje

Pod direktnim modelovanjem podrazumeva se razvoj modela podataka na osnovu:

- Poznavanja realnog sistema,
- Iskustva,
- Tekstualnog opisa,
- Poznavanja generičkih modela podataka (paterna)

Dijagram slučajeva korišćenja za "Video klub"



Struktuirani tekstualni opis slučajeva korišćen

Registrovanje novih video traka

1. Za svaku novu traku formiraju se odgovarajući podaci (*vrsta trake*, *naziv*, *broj kopija*, *datum nabavke*, *naziv dobavljača*, *adresa dobavljača*, *dnevna renta)*.
2. Pri ubacivanju podataka o traci u bazu, proverava se, preko *imena* i *vrste*, da li takav zapis već postoji. Ako postoji, povećava se u postojećem zapisu *broj kopija*. Ako ne postoji automatski se dodeljuje ***šifra trake*** datoj traci i registruje odgovarajući zapis.

Registrovanje članova

1. Formiraju se podaci o članu (*broj lične karte*, *ime*, *prezime*, *adresa*, *telefon*).
2. Podaci o članu se unose u bazu.

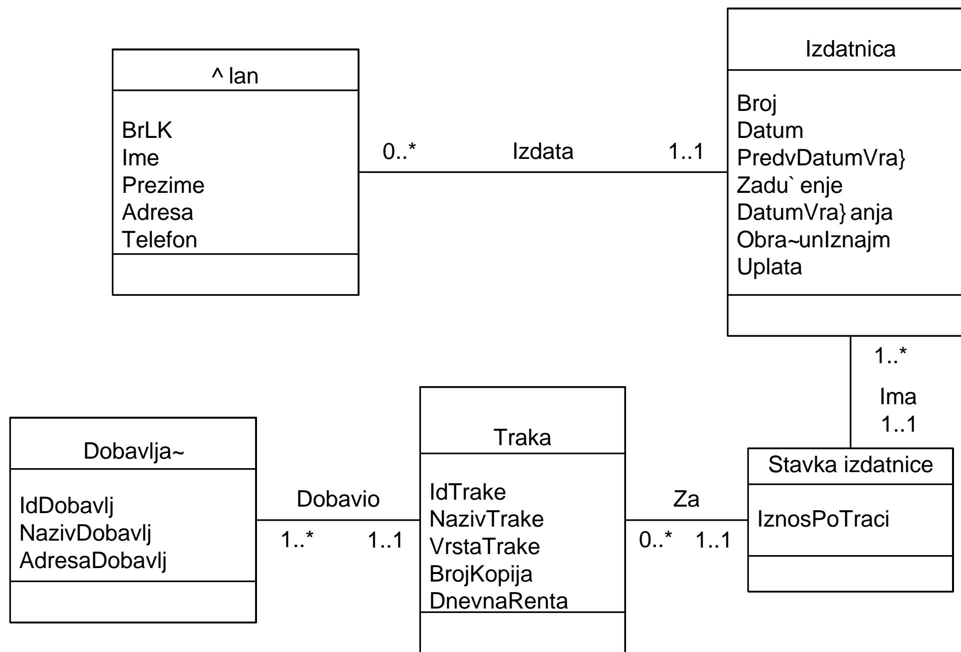
Iznajmljivanje traka

1. Član bira jednu ili više traka koje želi da iznajmi.
2. Formira se dokumenat Izdatnica u koga se unose podaci o članu, a kao posebne stavke izdatnice unose se podaci o iznajmljenim trakama. Unosi se i *datum izdavanja* i *predviđen datum vraćanja* traka.
3. Na osnovu podataka o *dnevnoj renti* traka, sračunava se i *ukupan iznos* koji član treba da plati, ako trake vrati na vreme.
4. Član potpisuje izdatnicu, dobija njenu kopiju i odnosi trake.

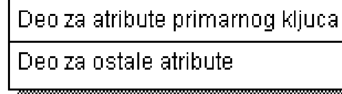
Vraćanje traka

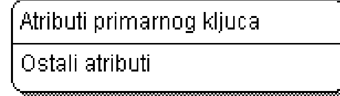
1. Član vraća trake.
2. U izdatnicu se unosi *datum kada su trake vraćene* i ukupna *cena iznajmljivanja*.
3. Član plaća iznajmljivanje.

Konceptualni model "Video kluba"

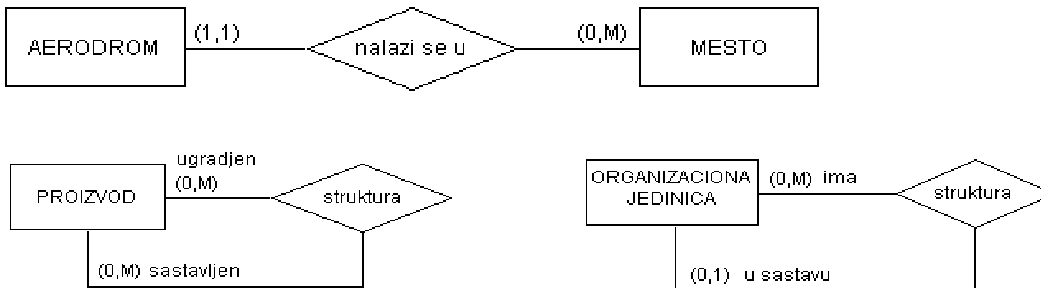


VERZIJE MOV-a: IDEF1x standard

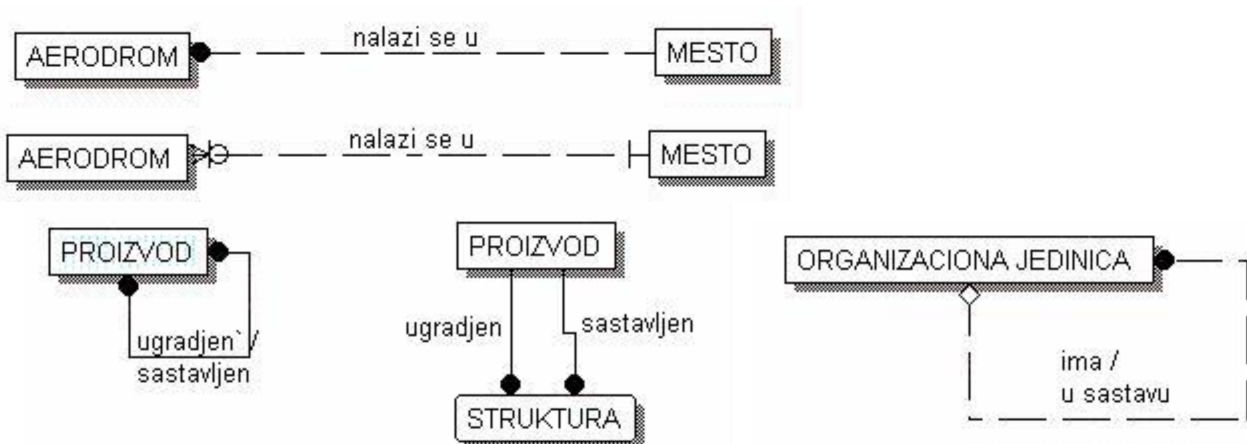




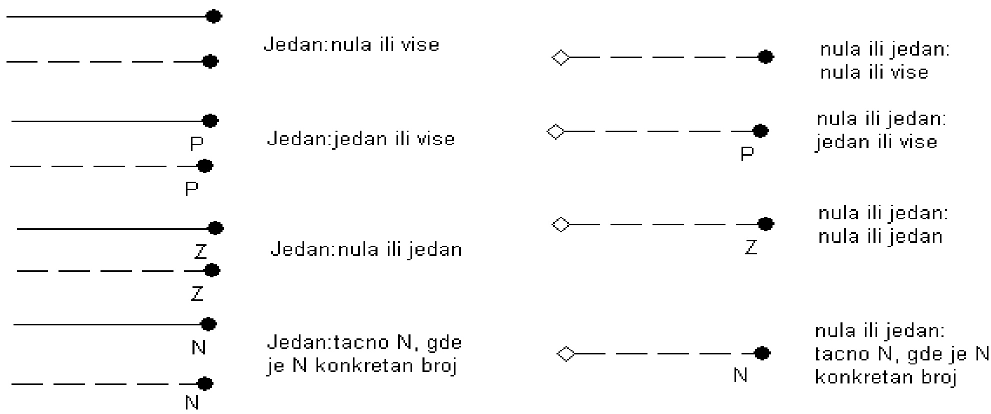
Veze po PMOV sintaksi



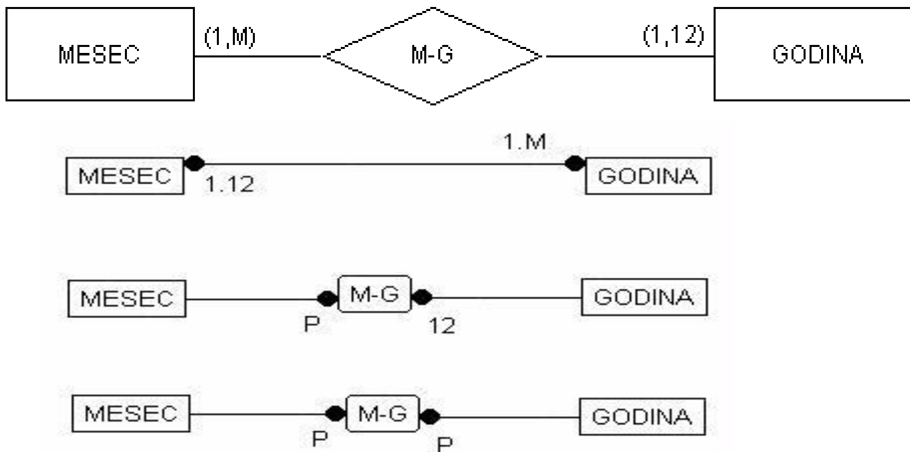
Veze po IDEF1x i IE standardu



VERZIJE MOV-a: IDEF1x standard



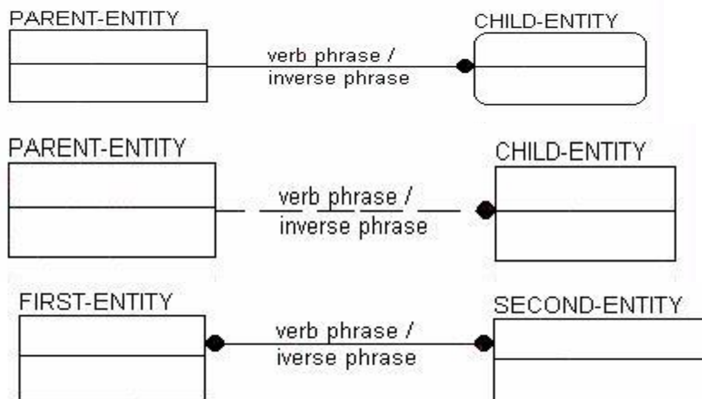
VERZIJE MOV-a: IDEF1x standard



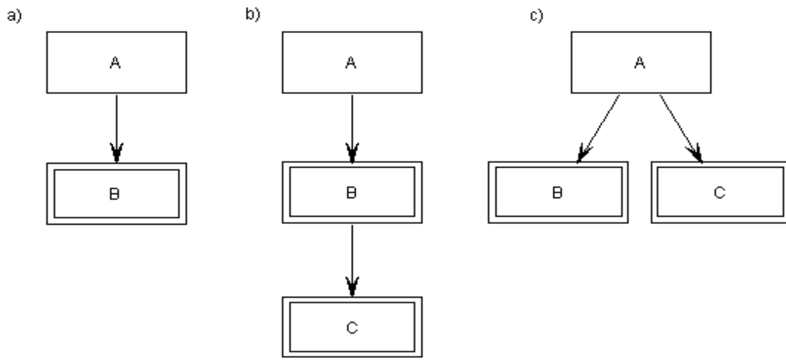
a) jedan:više identifikujuća veza

b) jedan:više neidentifikujuća veza

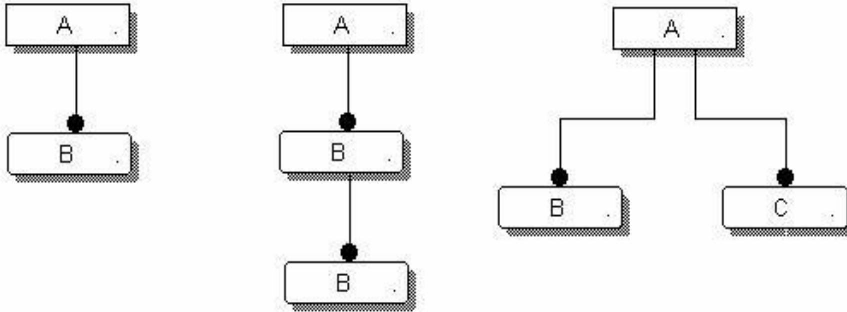
c) nespecificirana veza



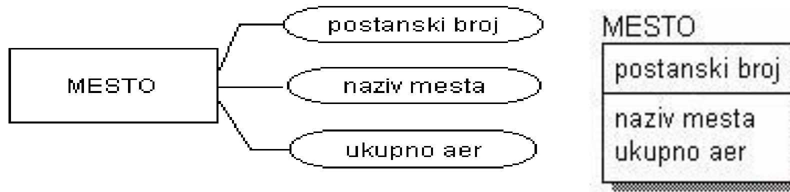
Objekti po PMOV sintaksi



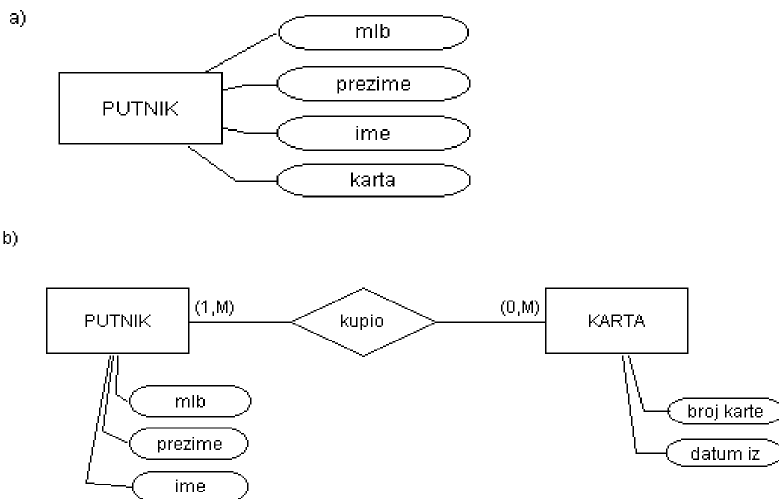
Objekti po IDEF1x standard

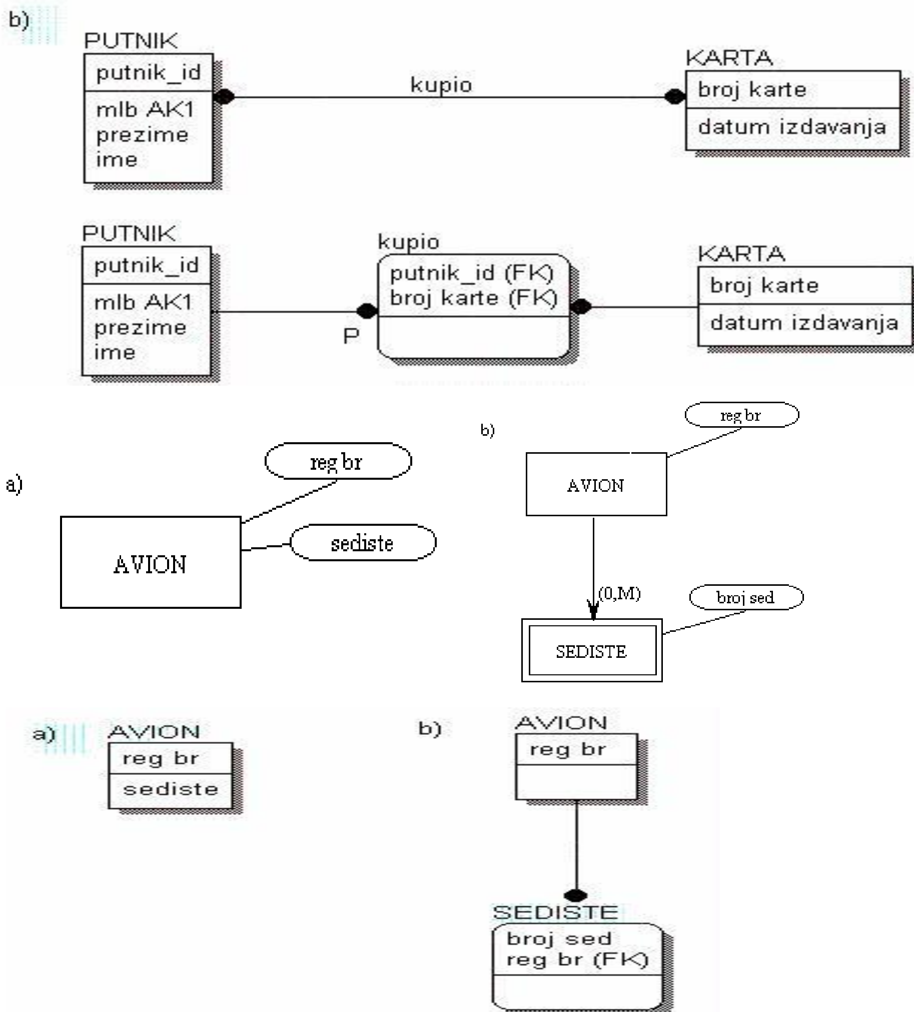


Atributi i domeni

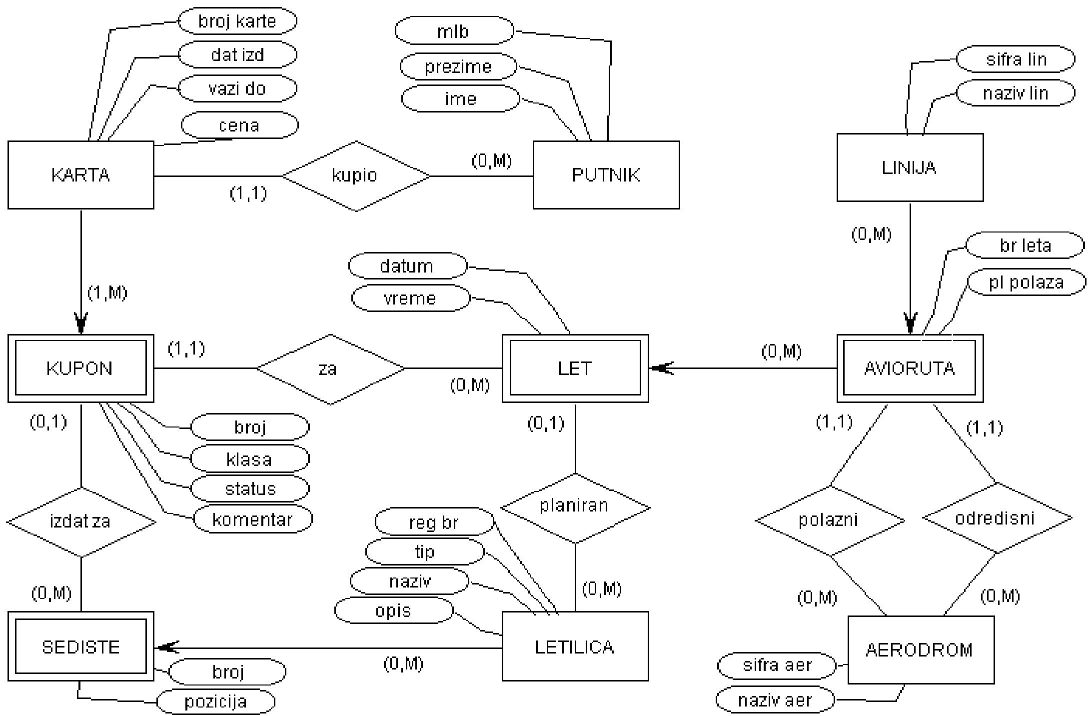


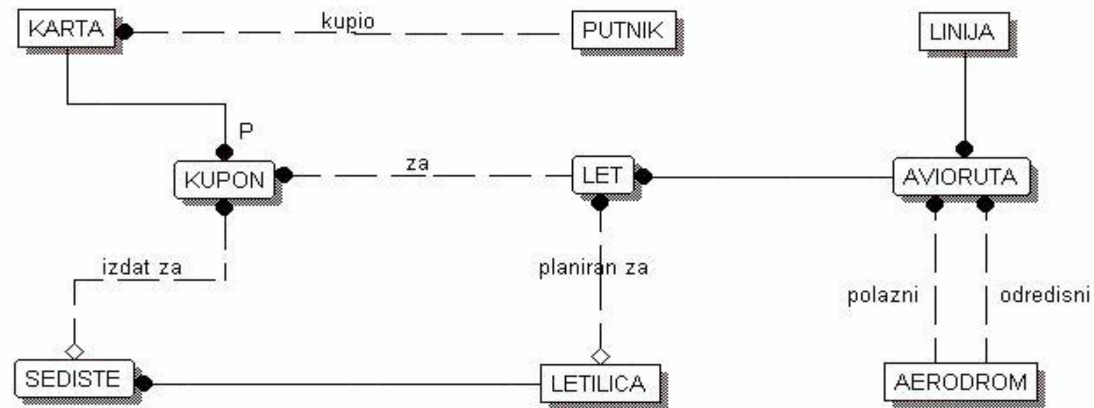
Višeznačni atributi

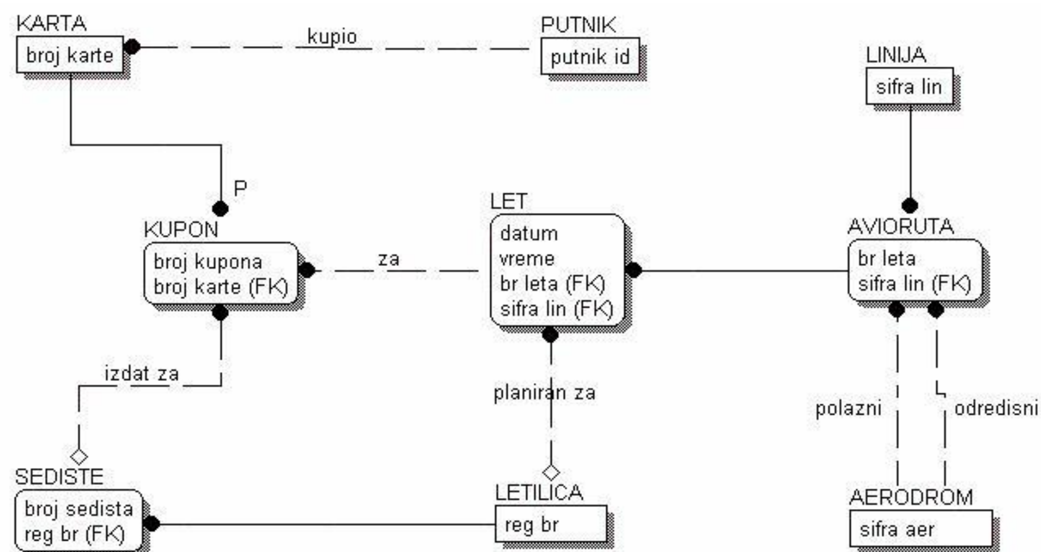


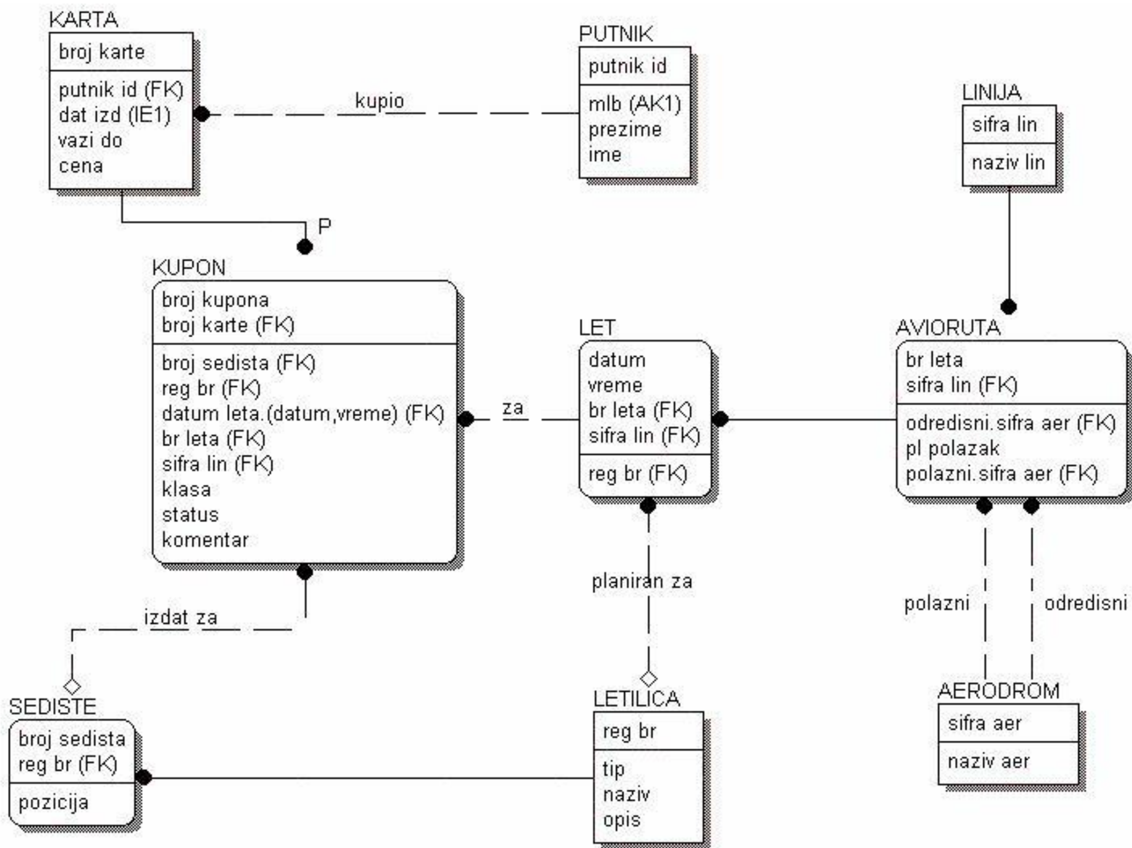


Primer 1: Avionska karta za jednu standardnu avio-liniju može biti sastavljena od više kupona. Jedna linija može da uključi više letova na relaciji izmedju mesta polaska i mesta krajnjeg odredišta. Svaki avion obično ima nekoliko letova u toku dana (let je identifikovan preko datuma i vremena poletanja aviona). Karta sadrži podatke o avionskoj liniji, prezimenu i imenu putnika, mestu polazišta, mestu krajnjeg odredišta, datumu izdavanja, roku važenja i ceni. Kuponi karte sadrže identične podatke i podatke o pojedinačnim letovima izmedu polazišta i krajnjeg odredišta: mesto poletanja, mesto sletanja, osnovni podaci o avionu. broj leta, klasa sedišta, datum i vreme poletanja.

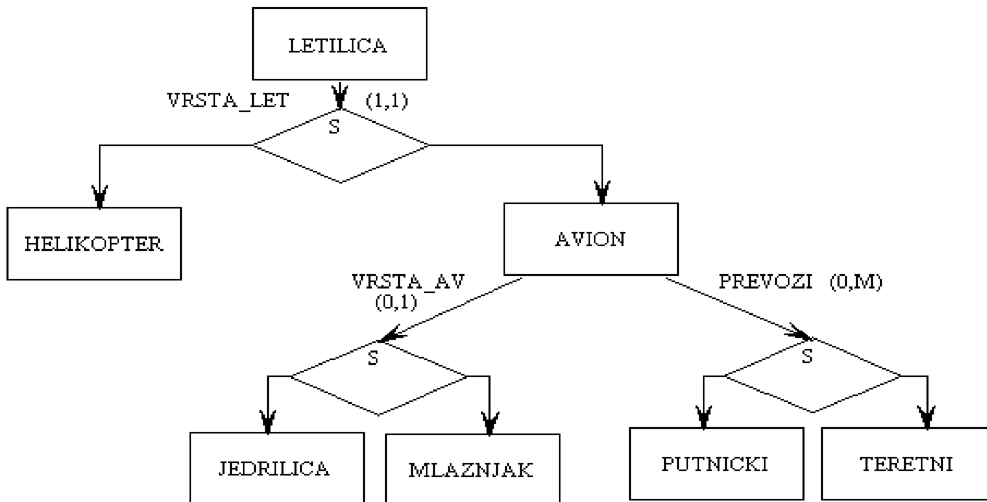

broj karte
dat izd
vazi do
cena
KARTA
kupio
(1,1)
(0,M)
PUTNIK
mlb
prezime
ime
sifra lin
naziv lin
LINIJA
(0,M)
br leta
pl polaza
(1,M)
datum
vreme
KUPON
(1,1)
za
(0,M)
LET
(0,M)
AVIORUTA
(0,1)
broj
klasa
status
komentar
izdat za
(0,1)
planiran
reg br
tip
naziv
opis
(1,1)
(1,1)
polazni
odredisni
(0,M)
(0,M)
(0,M)
(0,M)
SEDISTE
broj
pozicija
(0,M)
LETILICA
AERODROM
sifra aer
naziv aer


KARTA
kupio
PUTNIK
LINIJA
P
KUPON
za
LET
AVIORUTA
izdat za
planiran za
polazni
odredisni
SEDISTE
LETILICA
AERODROM

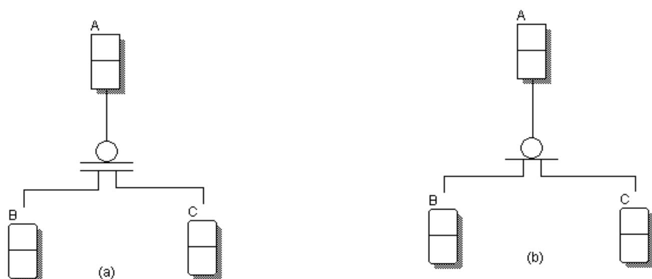

KARTA
broj karte
kupio
PUTNIK
putnik id
LINIJA
sifra lin
P
LET
datum
vreme
br leta (FK)
sifra lin (FK)
KUPON
broj kupona
broj karte (FK)
za
AVIORUTA
br leta
sifra lin (FK)
izdat za
planiran za
polazni
odredisni
SEDISTE
broj sedista
reg br (FK)
LETILICA
reg br
AERODROM
sifra aer

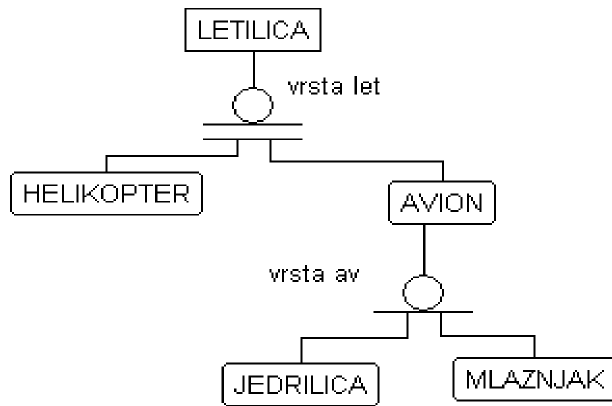


Generalizacija i specijalizacija

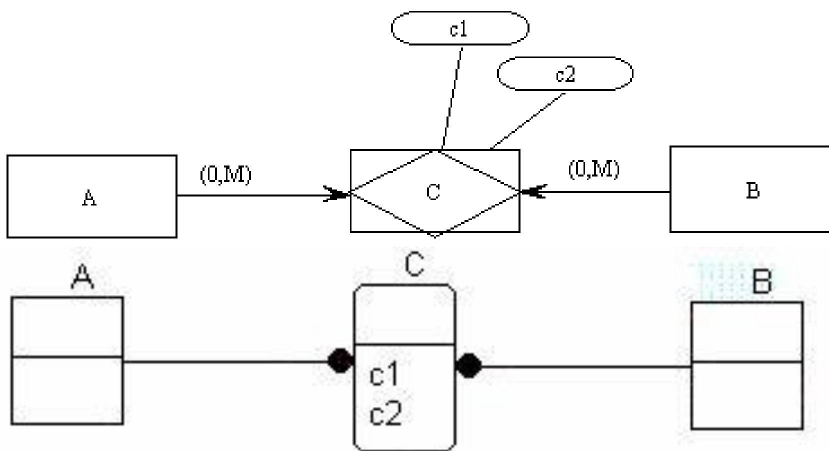


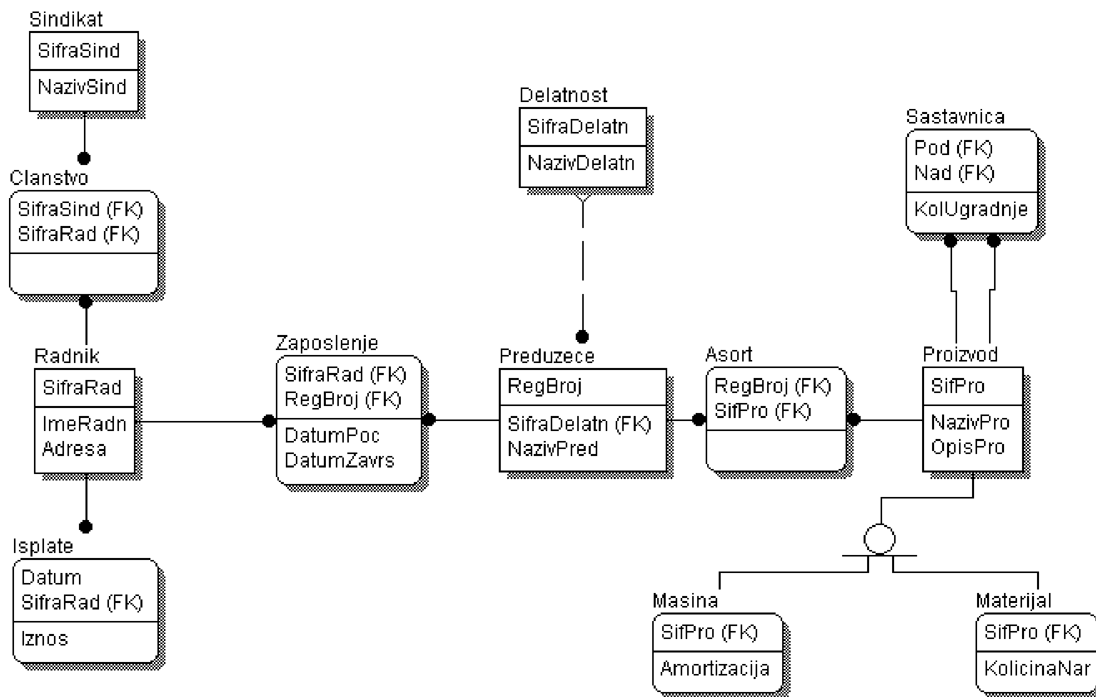
VERZIJE MOV-a: IDEF1x standard





Agregacija



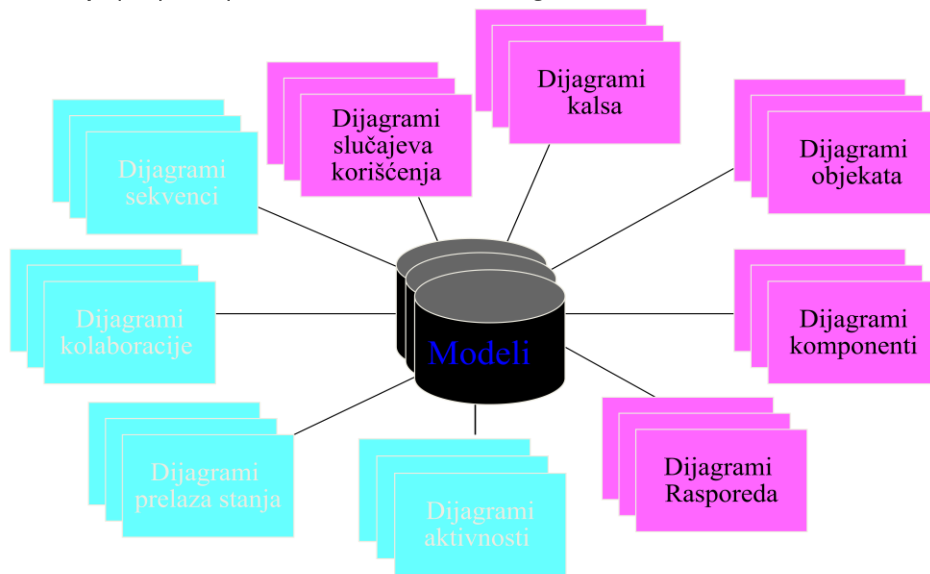


8. UML Modeli

- Aspekt projektovanja -

Za svaki aspekt daje se statički i dinamički opis sistema

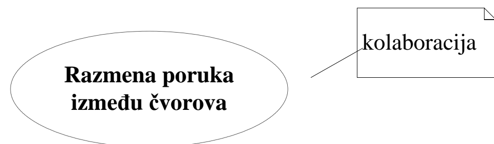
Model je potpun opis sistema sa neke tačke gledišta



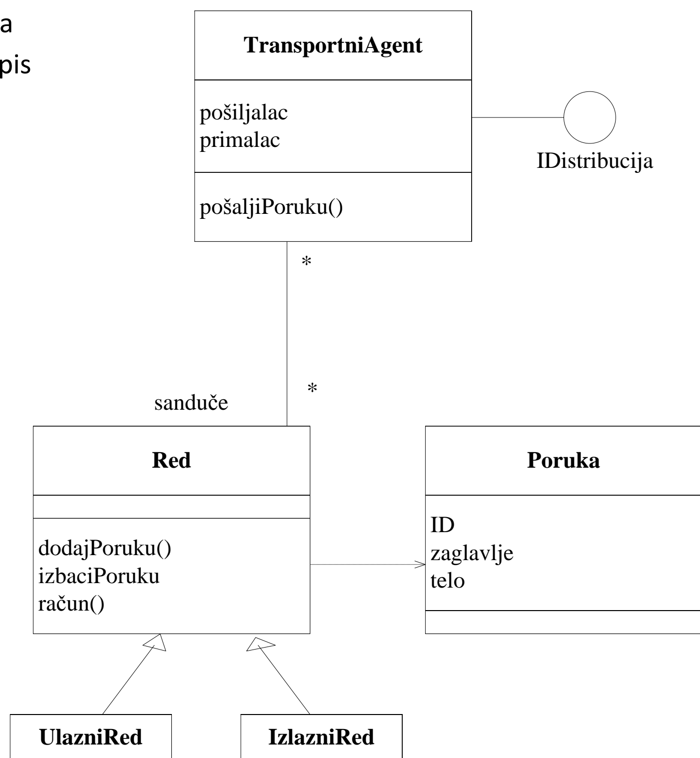
- Aspekt projektovanja pretstavlja realizaciju sistema u "objektnom prostoru stanja".
- Statički opis ovoga aspekta daje se preko **Dijagram klasa i Dijagrama objekata.**
- Dinamički opis se daje preko **dijagrama interakcija, dijagrama promene stanja i dijagrama aktivnosti.**

Dijagrami klasa

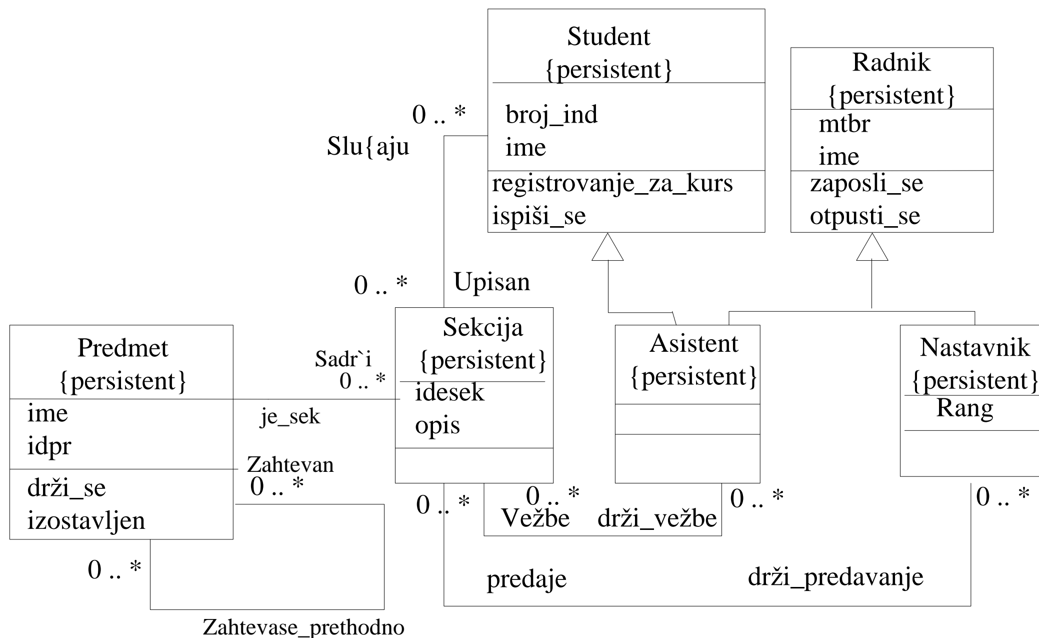
- Dijagram klasa najčešće korišćeni dijagam u OO pristupima razvoju softvera
- On pretstavlja skupove klasa, interfejsa, kolaboracija i njihove međusobne veze
- Koristi se da pretstavi:
  1. *Osnovni "rečnik sistema" – definiše pojmove koji se u tom sistemu koriste,*
  2. *Opiše strukturu neke kolaboracije,*
  3. *Pretstavi logičku šemu baze podataka*



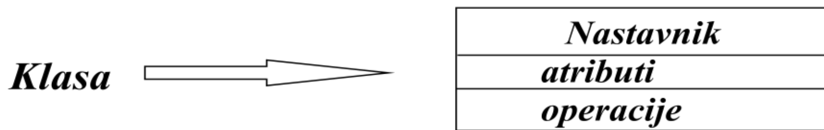
Dijagram klasa
Kao statički opis
Kolaboracije

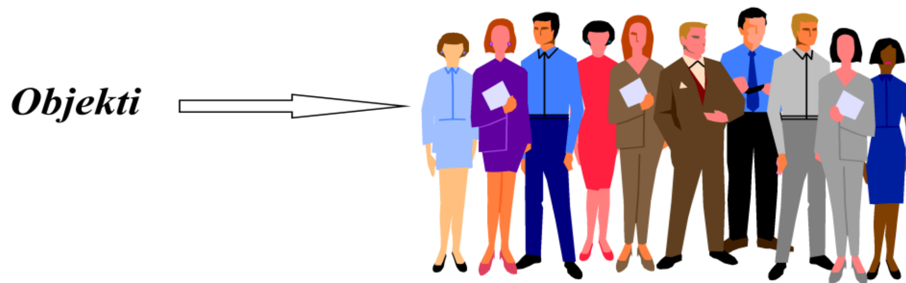


Primer dijagrama klasa kao logičke šeme baze podataka



Klasa je okvir (template) tj.specifikacija za kolekciju objekata koji deli isti skup atributa i operacija.





Veza je ono sto klasa ili neki objekat zna o durgoj klasi ili objektu.

[Klase] Generalizacija (Generalization)

Nasladjivanje

- Primer: Osoba - Nastavnik, Student, Sluybenik...
- Primer: TransportnoSredstvo - Avion, Voz, Auto, Brod...

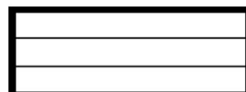
[Objekat] Asocijacija (Associations)

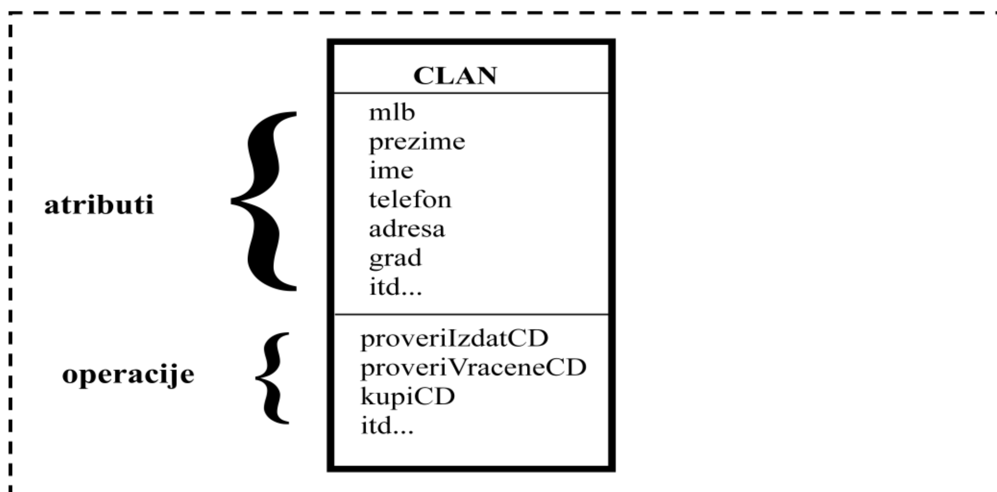
- Nastavnik – Predmet
- Student - Predmet

[Objekat] Aggregations & Composition (Whole-Part)

- Assembly – Parts, Group – Members, Container - Contents

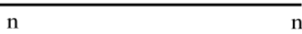
UML Class Diagram Notation



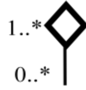


Class
Generalization
Relationship

Object Association

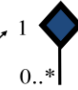


Object
Aggregation
Association



1..*
0..*

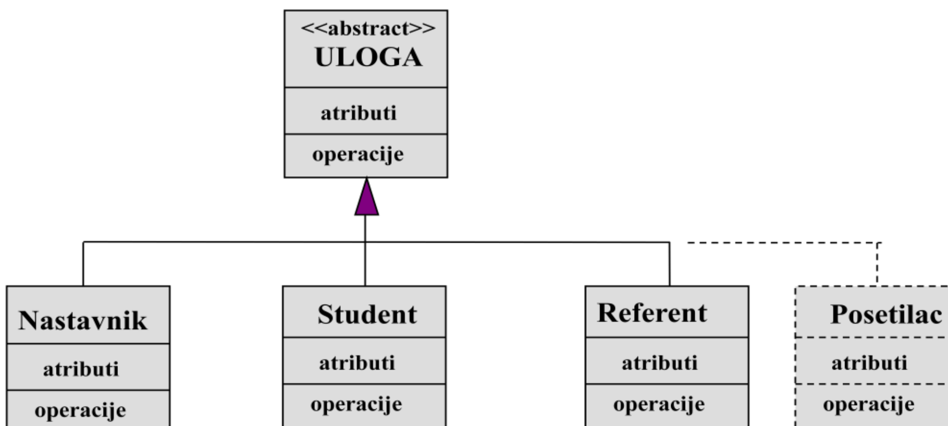
Object Composition
Association



1
0..*

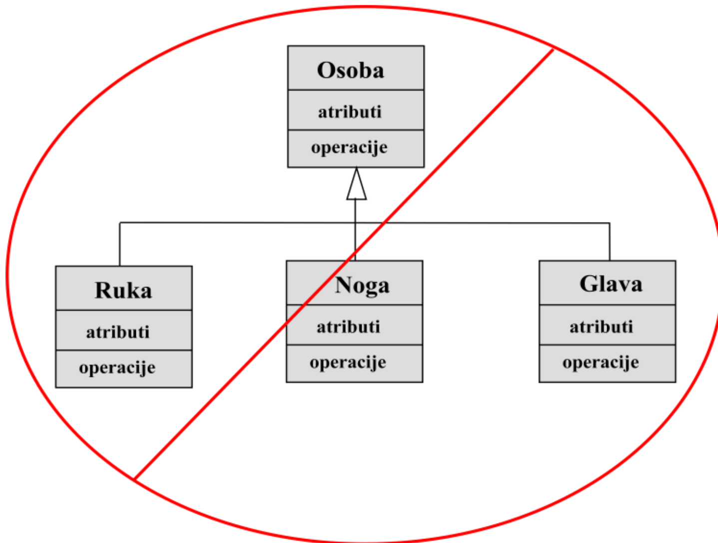
Will always be “1”

Primer Generalizacije

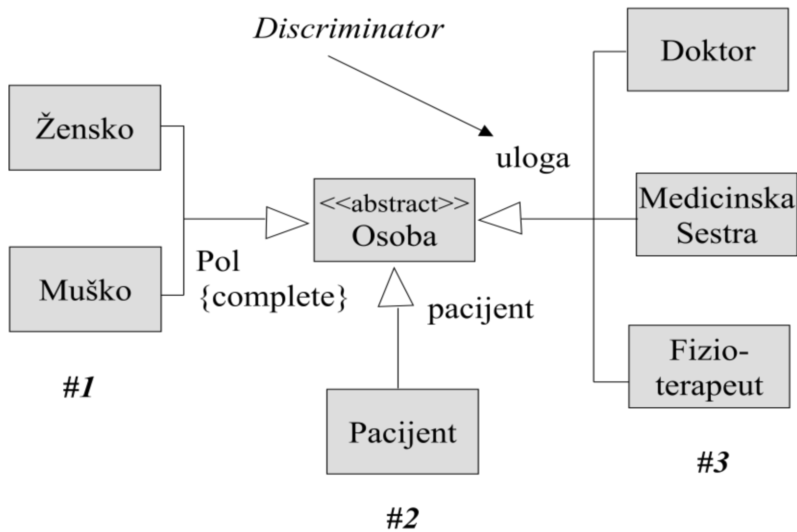


Note: <<abstract>> = no objects

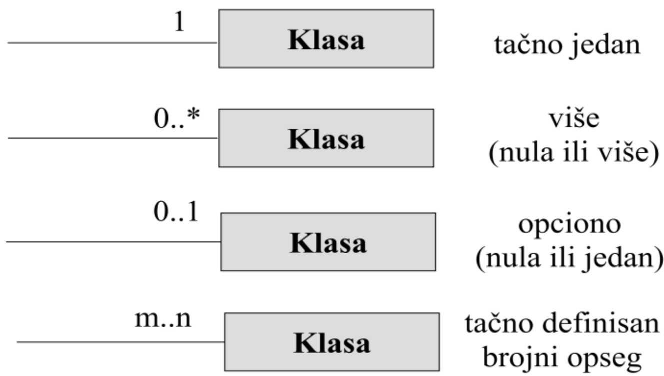
Primer loše generalizacije (narušava “is a” ili “is a kind of”)



Generalizacija- Višestruka klasifikacija

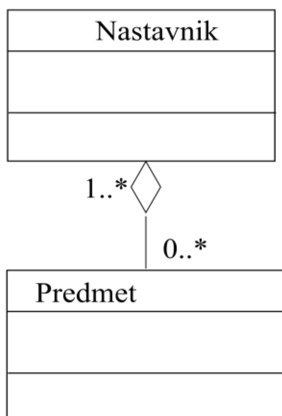


Kardinalnosti

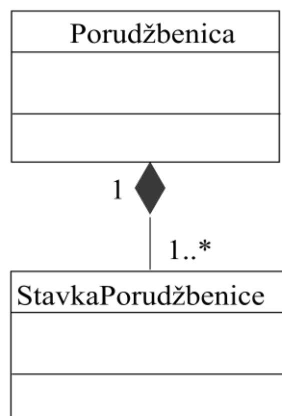


Primer:

Aggregation



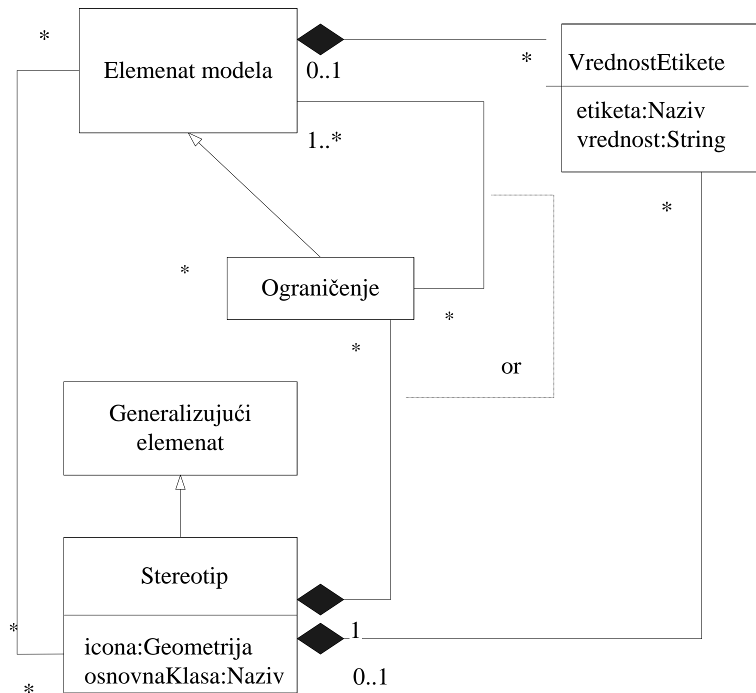
Composition



(drugi primer: ruka --> prst)

Definisanje semantike uml-a preko dijagrama klasa

Metamodel za mehanizme proširenja



Dijagram klasa (Stereotipi)

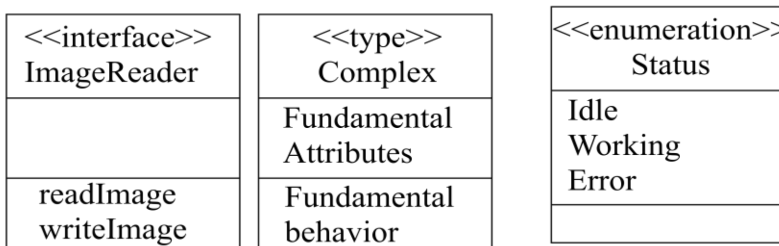
Važni stereotipi:

<<interface>> specificira kolekciju operacija za specifikaciju servisa

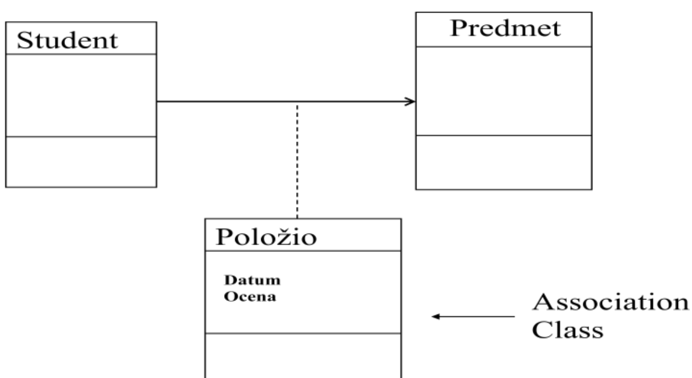
<<type>> specificira strukturu i ponašanje (ne implementaciju)

<<enumeration>> specificira predefinisane vrednosti

<<implementationClass>> pomoćna klasa kreirani u detaljnom dizajnu



Asocijacija kao klasa

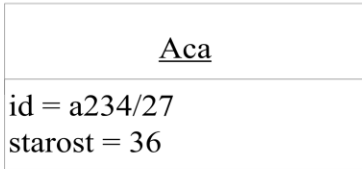


Dijagram objekata (pojavljivanja)

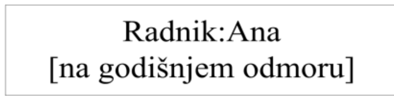
- Dijagram objekata pokazuje skup objekata i njihovih veza u jednom trenutku vremena.
- Dijagrami objekata modeliraju stanje sistema u jednom trenutku vremena, odnosno daju zamrznutu sliku sistema u jednom trenutku vremena.
- Dijagrami objekata daju statičku strukturu za opis dinamičkog ponašanja sistema preko dijagrama interakcije. Oni su pojavljivanje odgovarajućeg dijagrama klasa i statički deo dijagrama interakcije
- Pretstavljaju skup objekata i pojavljivanja veza, mogu da sadr`e i komentare i ograničenja, da budu podeljeni u pakete i podsisteme.

Način označavanja objekata (pojavljivanja)

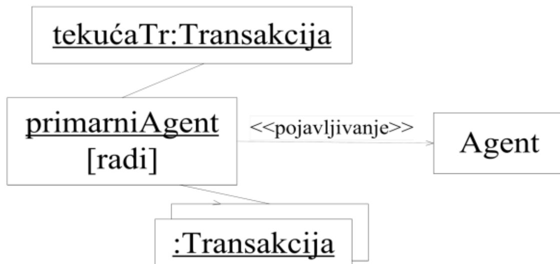
Oznaka	Opis
Ana : Lice	Naziv objekta i naziv odgovarajuće Klase
: Lice	"Anonimni" objekat date klase
Aca	Objekat čija se klasa podrazumeva (očigledna je)
agent:	"Siroče", pojavljivanje nedefinisane klase
:Odeljenje::OrgStruktura	Anonimno pojavljivanje sa definisanom "putanjom" za naziv klase
:Odelenja	"Multiobjekat", pretstavlja kolekciju anonimnih objekata.



Prikazivanje vrednosti atributa objekta

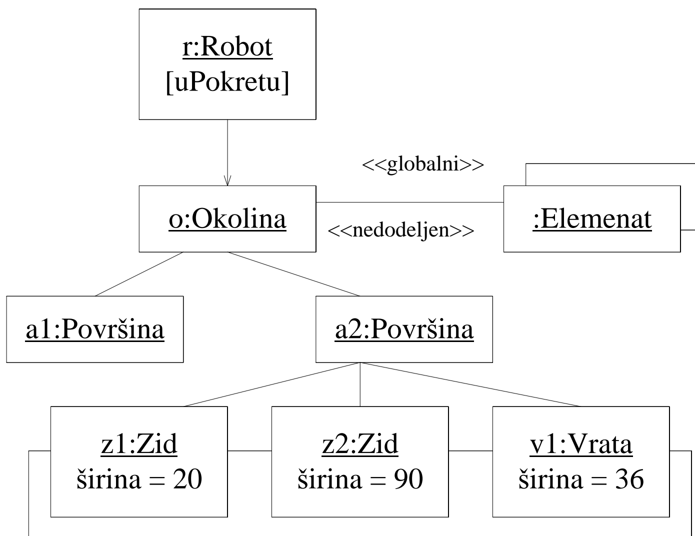


Eksplicitno prikazivanje stanja objekta

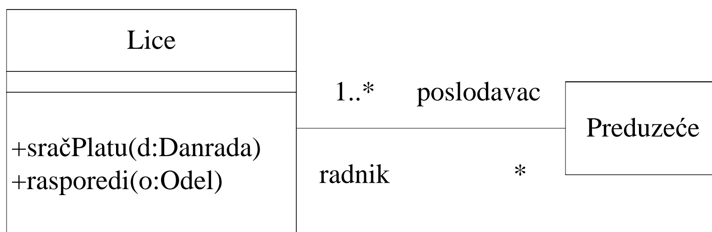


Primer jednostavnog Dijagrama objekata sa stereotipnom vezom koja pokazuje pripadnost klasi

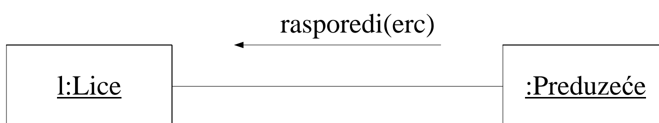
Primer dijagrama objekata



Klase, asocijacije pojavljivanja, veze i poruke



Klase, asocijacija, uloge, kardinalnosti



pojavljivanja , linkovi i poruke

9. **UML Modeli**

Aspekt projektovanja - dinamički opis sistema

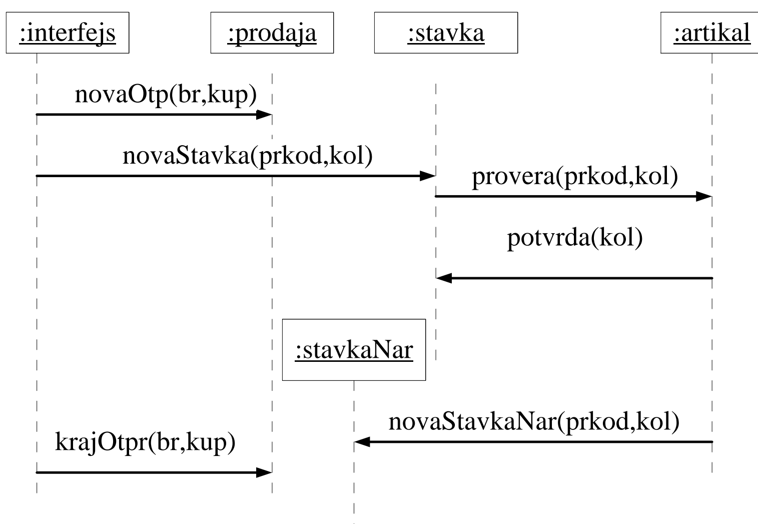
Dijagrami sekvenci i dijagrami kolaboracije

Interakcije se mogu modelovati na dva načina:

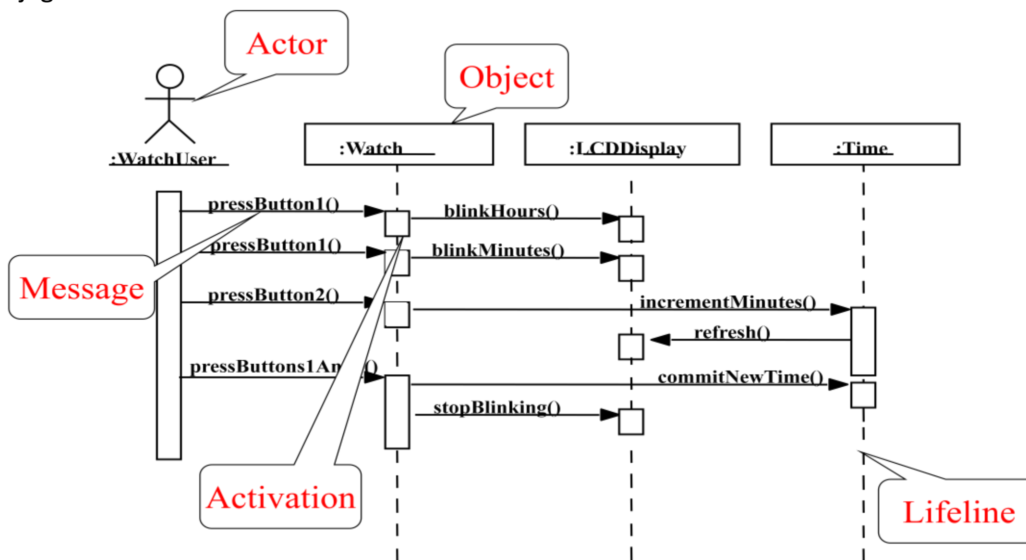
- Prikazujući vremenski redosled poruka: DIJAGRAMI SEKVENCI
- Prikazujući interakciju u kontekstu neke organizacije (strukture) objekata: DIJAGRAMI KOMUNIKACIJE

Moguće je automatski prevesti jedan oblik u drugi.

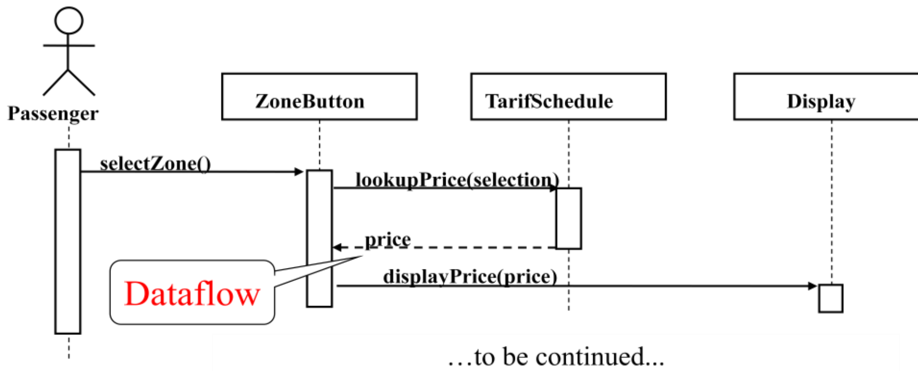
Primer dijagrama sekvenci



Dijagram sekvenci

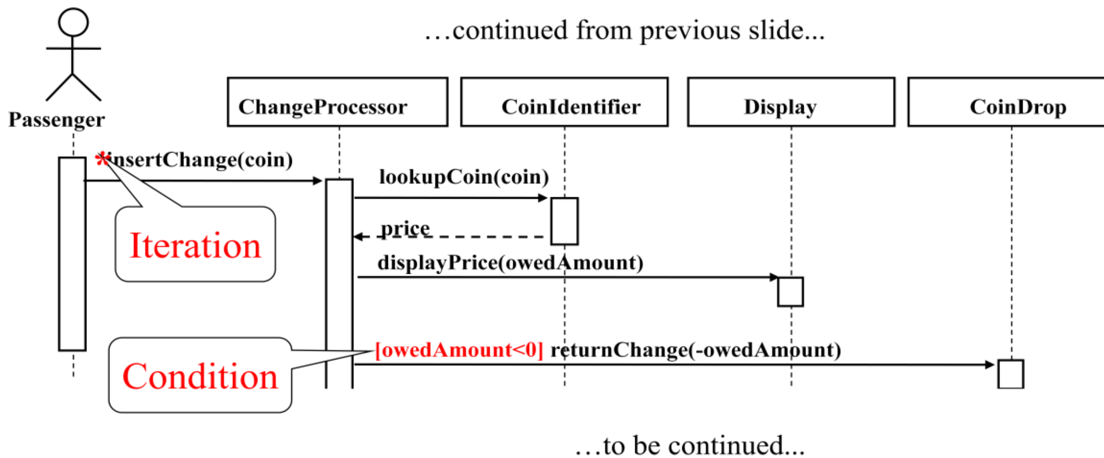


Nested messages



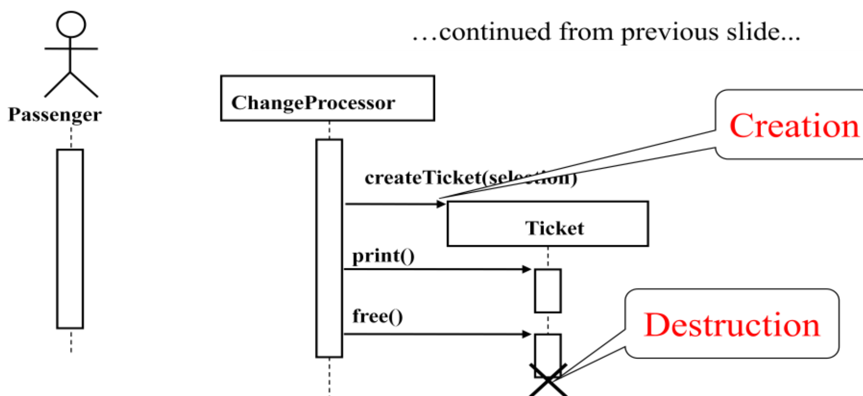
- The source of an arrow indicates the activation which sent the message
- An activation is as long as all nested activations
- Horizontal dashed arrows indicate data flow
- Vertical dashed lines indicate lifelines

Iteration & condition



- Iteration is denoted by a * preceding the message name
- Condition is denoted by boolean expression in [ ] before the message name

Creation and destruction

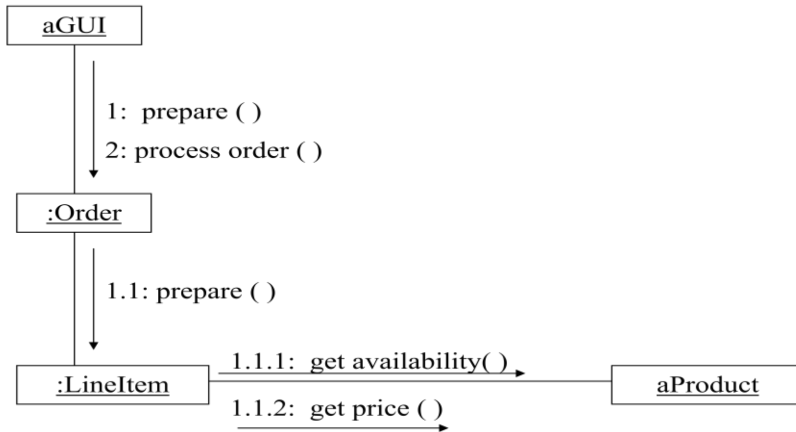


- Creation is denoted by a message arrow pointing to the object.
- Destruction is denoted by an X mark at the end of the destruction activation.
- In garbage collection environments, destruction can be used to denote the end of the useful life of an object.

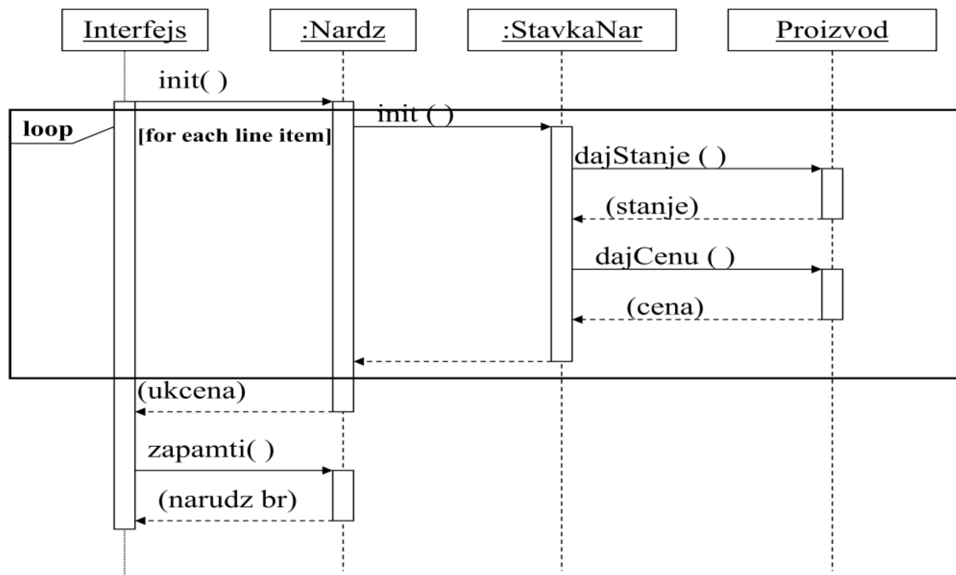
Sequence Diagram Summary

- UML sequence diagram represent behavior in terms of interactions.
- Useful to find missing objects.
- Time consuming to build but worth the investment.
- Complement the class diagrams (which represent structure).

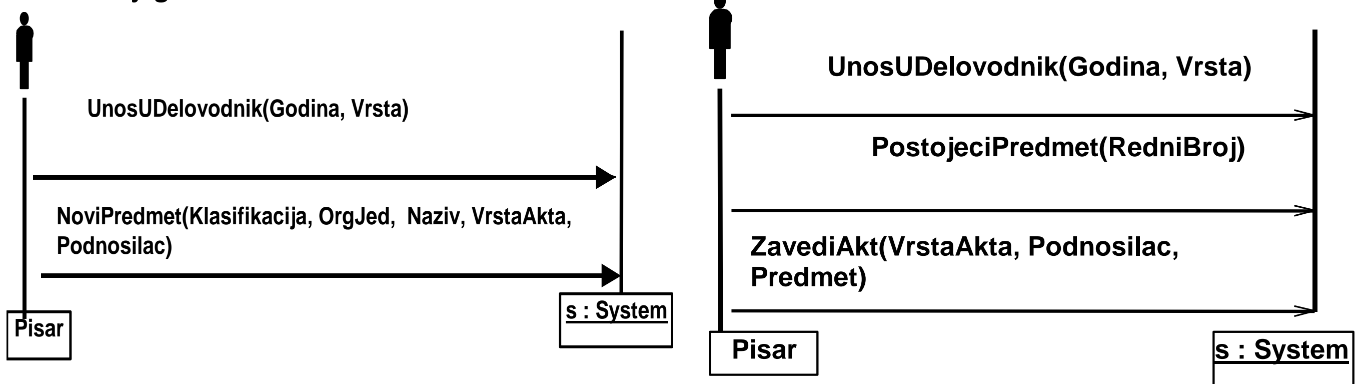
Communication diagram for: "Make a Purchase"

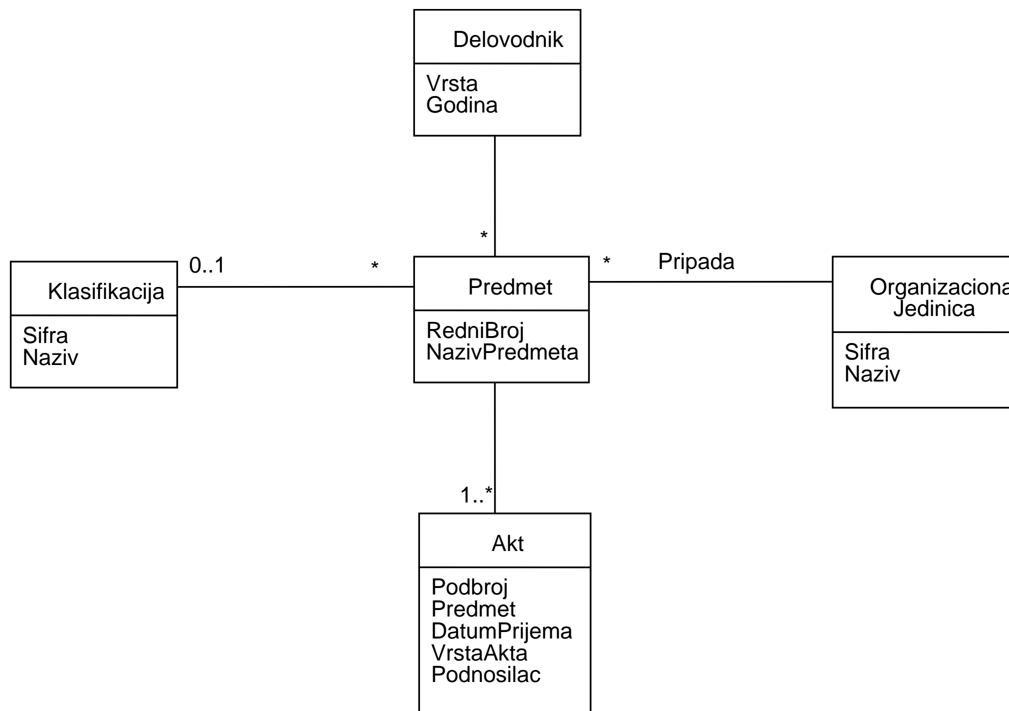


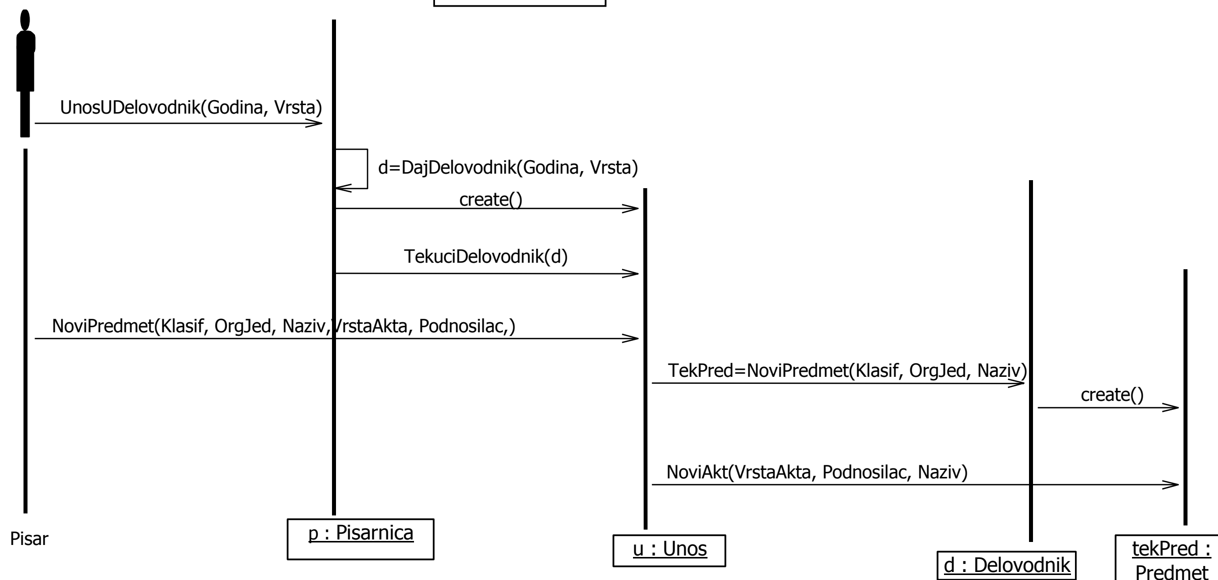
Dijagram sekvenci za: "Izrada Narudzbenice"



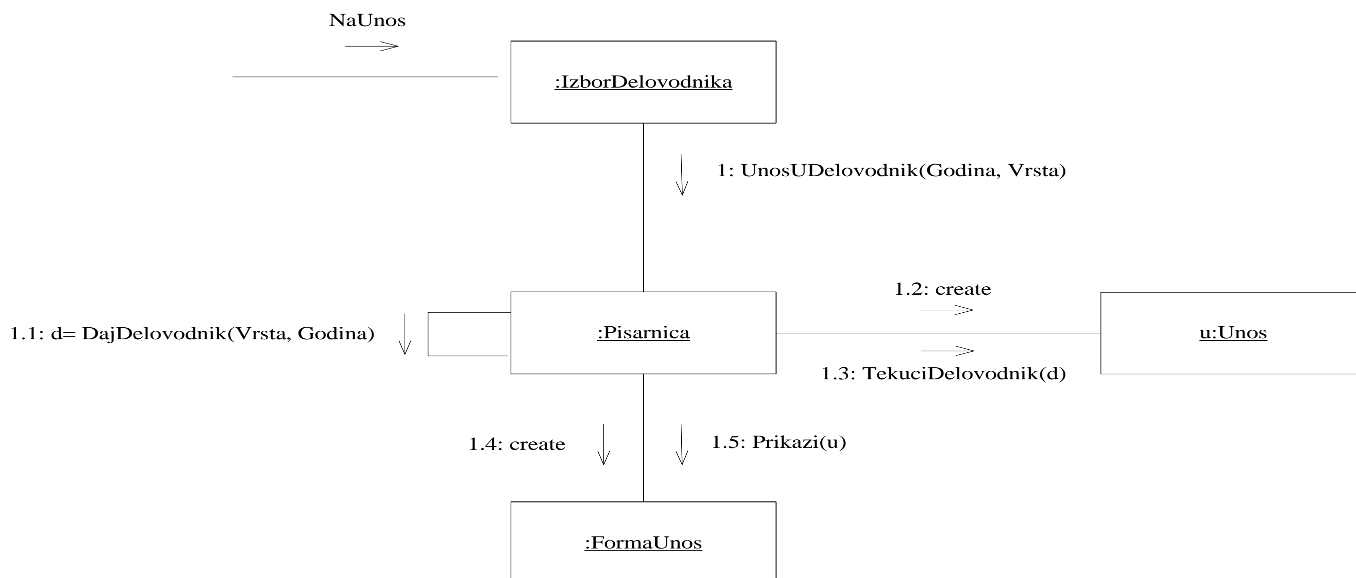
Sistemski dijagram sekvenci

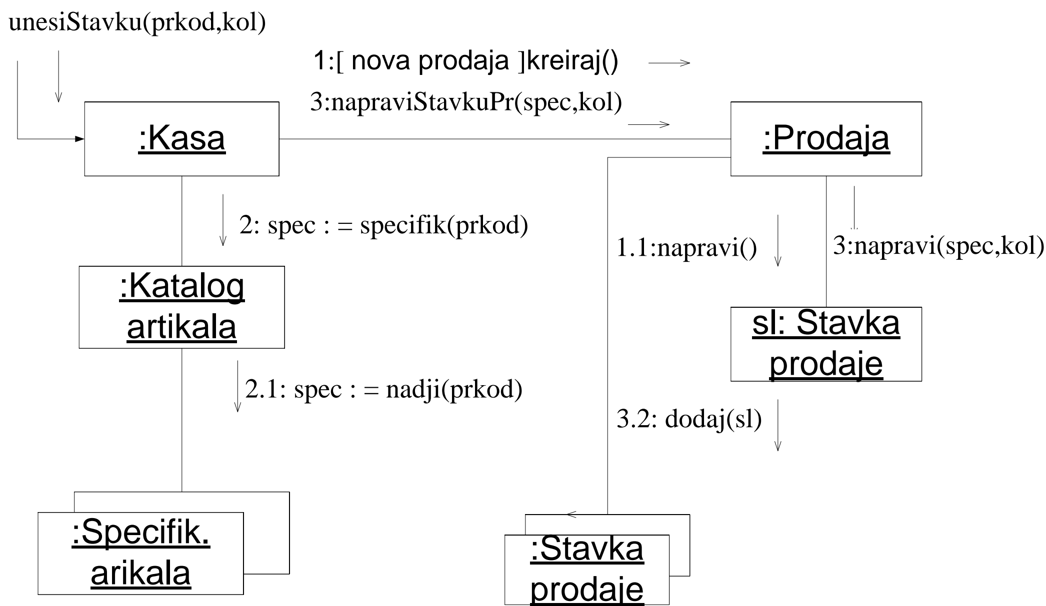






Primer dijagrama kolaboracije





Dijagrami promene(prelaza) stanja

Pri opisivanju dinamike sistema preko dijagrama prelaza stanja koriste se sledeći pojmovi:

(1) **Sistem (objekat) je skup objekata, njihovih atributa i njihovih veza**. Struktura sistema - odnos njegovih objekata, veza i atributa opisuje se preko modela opisanih u prethodnom delu.

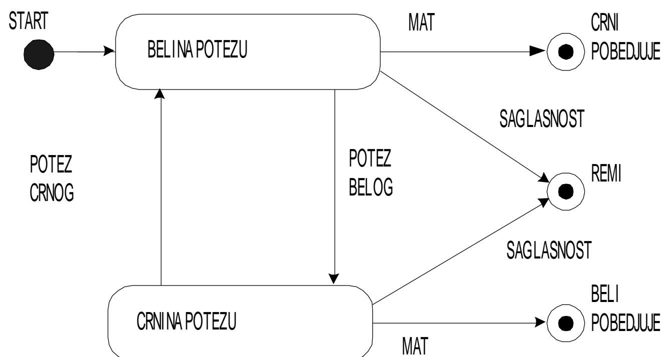
(2) **Stanje sistema (objekta)** u jednom trenutku vremena predstavlja skup vrednosti atributa svih objekata i "vrednosti" svih veza u tom trenutku. Termin "vrednost veze" opisuje par (za binarne veze ili n-torku uopšte) identifikatora pojavljivanja objekata koji su u vezi.

(3) **Događaji** iniciraju promene stanja sistema. Odziv sistema na neki događaj zavisi od stanja u kome se on nalazi. Događaj mo`e da prouzrokuje promenu stanja sistema i/ili da indukuje novi događaj. Događaj se zbiva u jednom trenutku vremena, događaj nema trajanje (u vremenskoj skali u kojoj se posmatra dati sistem). Ponekad se događaj i poruka tretiraju kao sisnonimi. Međutim, precizno, poruka je pojavljivanje događaja, što će kasnije biti pokazano.

(4) Dijagram prelaza (promene) stanja je apstrakcija koja pokazuje stanja, događaje i prelaze (tranzicije) iz stanja u stanje kao mogući odziv na događaje.

(5) **Dijagram promene stanja** povezuje stanja (konkretna, imenovana) sa događajima u sistemu. Promena stanja izazvana događajem naziva se **tranzicija (prelaz)**. Dijagram promene stanja je usmereni graf u kome su čvorovi stanja, a grane tranzicije, sa usmerenjem od polaznog do prouzrokovanog stanja. Granama grafa daju se nazivi događaja koji prouzrokuju tranziciju. (Jedan događaj mo`e da prouzrokuje više tranzicija, pa više grana mo`e da ima isto ime !!!).

(6) **Početna i krajnja stanja**. Mo`e se uvesti i koncept početnog i krajnjeg stanja, za objekte (sisteme) koji imaju "ograničen `ivot". U tom slučaju, početno stanje je rezultat kreiranja odgovarajućeg objekta, a krajnje podrazumeva njegovo "uništenje" (nestanak). Početna i krajna stanja na grafu imaju specijalne oznake, a mogu imati i imena.



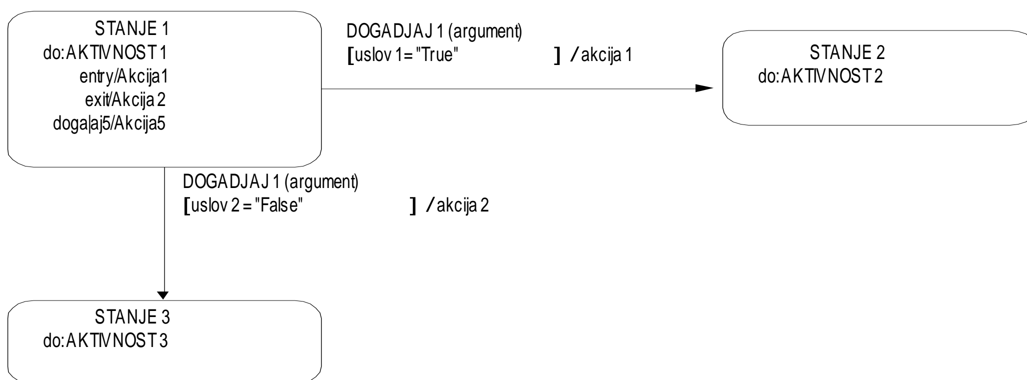
(7) **Uslovi**. Uslov je Bulova funkcija nad vrednostima atributa i veza. Stanja sistema se mogu opisati preko uslova. Iskaz da je objekat u nekom stanju je uslov. Pored toga, uslovi se mogu koristiti da ograniče tranzicije prouzrokovane događajima. Ponekad, za prelaz sistema iz jednog stanja u drugo potrebno je, pored događaja, da bude ispunjen i neki uslov. Na dijagramu prelaza stanja uslov se iskazuje uz naziv događaja, unutar uglaste zagrade.

(8) **Akcija** pretstavlja jedno "atomsko sračunavanje"koje prouzrokuje promenu stanja sistema ili vraća neku vrednost. Neka akcija okida događaj koji će sistem prevesti iz jednog u drugo stanje.Akcija mo`e da pozove operaciju nekog objekta, da kreira ili uništi neki objekat ili da pošalje signal nekom objektu. Akcije se mogu pridružiti stanjima i tranzicijama. Ako se akcije pridružuju stanjima one mogu biti:

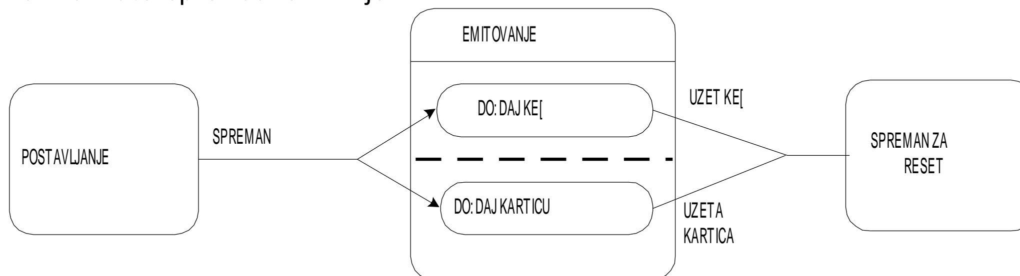
- **"entry"** – akcija koja se obavlja uvek pri ulazu u stanje, bez obzira koja je tranzicija to prouzrokovala;
- **"exit"** - akcija koja se obavlja uvek pri napuštanju stanja, bez obzira koja je tranzicija to prouzrokovala;
- **"inerna tranzicija"** – akcija koja ne menja stanje sistema.

Akcija se mo`e obaviti i pri prelazu iz jednog u drugo stanje Na tranziciji akcije se iskazuju uz naziv događaja, iza uslova i oznake "/".

(9) **Aktivnosti**. Kada je objekat (ili deo sistema) u nekom stanju, on je bilo neaktivan ili obavlja neku **aktivnost** dok ga događaj ne prevede u neko drugo stanje.Drugim rečima, dok obavlja određenu aktivnost, sistem je u datom stanju. Na dijagramu prelaza stanja aktivnosti se definišu nazivom iza ključne reči DO, u okviru stanja (čvora).



(9) **Sinhronizacija konkurentnih aktivnosti**. U jednom stanju se može obavljati više konkurentnih aktivnosti.. Ove aktivnosti ne moraju biti sinhronizovane, mogu se obavljati bilo kojim redom, ali sve one moraju biti obavljene pre nego što se izvrši tranzicija u drugo stanje. Konkurentne aktivnosti u jednom stanju se prikazuju podelom stanja (čvora) na delove razmaknute isprekidanom linijom.

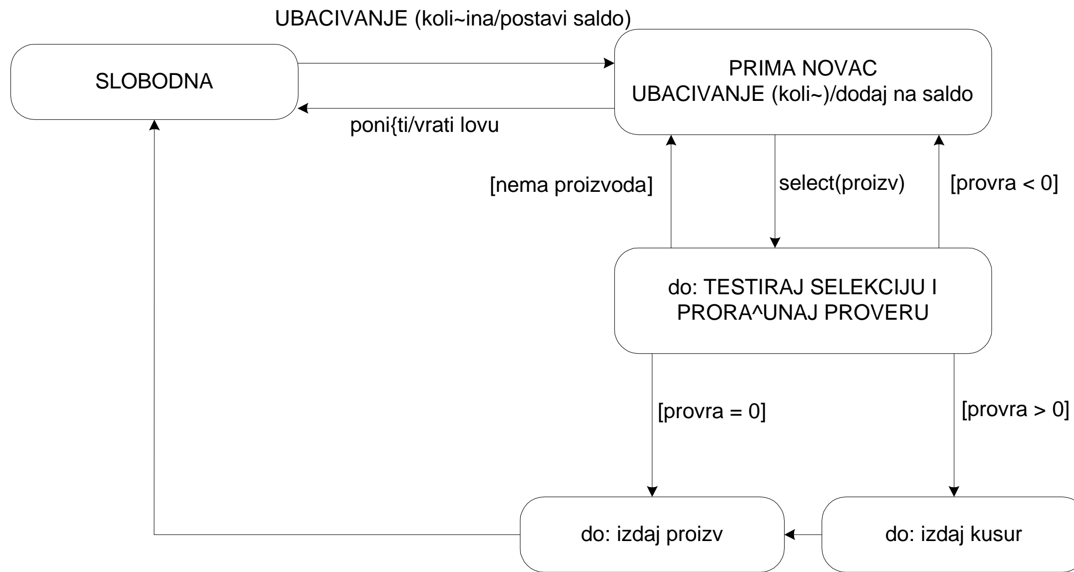


(10) **Neoznačena ili automatska tranzicija**. Koristi se da bi se prikazala automatska tranzicija iz jednog stanja u drugo koja se obavlja čim se aktivnost u nekom stanju obavi. Kraj aktivnosti u nekom stanju mo`e se tretirati kao neimenovani događaj. Taj neimenovani događaj "okida" neimenovanu tranziciju u drugo stanje.

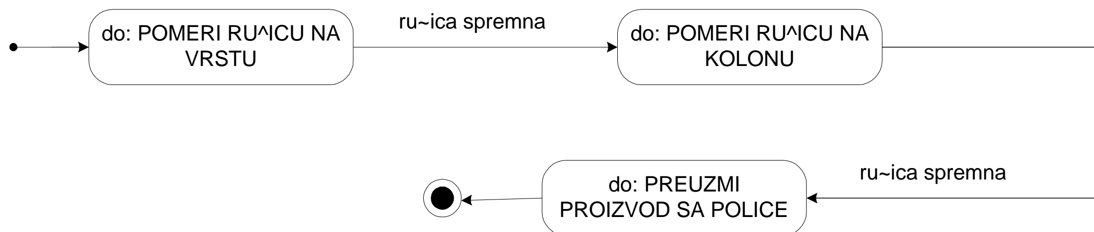
(11) **Dekompozicija dijagrama prelaza stanja** Dijagrami prelaza stanja se mogu dekomponovati na sledeće načine:

(i) Kompozitno stanje, odnosno sekvencijalna podstanja.
(ii) Generalizacija stanja.
(iii) Agregacija stanja - agregaciona konkurentnost.

Kompozitno stanje. Svaka aktivnost u okviru stanja se može predstaviti posebnim Dijagramom dekompozicije. Dobijeni dijagram ima ulaz i izlaz, odnosno početno i završno stanje. Ova stanja i odgovarajući događaji su nevidljivi sa dijagrama višeg nivoa i predstavljaju samo detaljan opis aktivnosti. Ugnježđeni dijagram stanja "zamenjuje" jedno stanje na dijagramu višeg nivoa. Sve ulazne tranzicije sa dijagrama višeg nivoa prenose se na početno stanje, a sve izlazne transakcije na završno. Izlazno stanje može da generiše poruku - događaj svome nadređenom stanju.

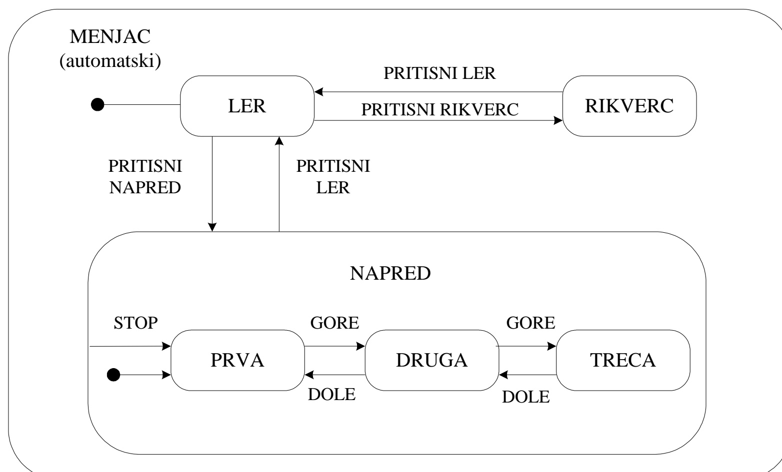


(a)



(b) Dekompozicija stanja "izdaj proizvod" kao ugnje`dena stanja

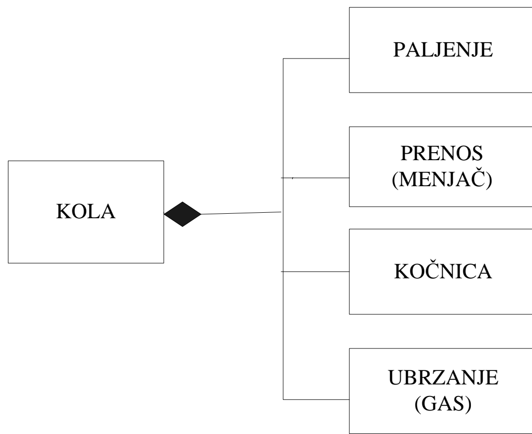
Generalizacija stanja. Pod generalizacijom stanja će se ovde podrazumevati odnos između stanja i podstanja u kome podstanje nasleđuje osobine stanja, promenljive stanja i tranzicije stanja (precizno, izlazne tranzicije). Ako je neki događaj primljen kada je objekat u datom podstanju, sve tranzicije nadstanja su potencijalno primenljive, ako nisu "prekrivene" istoimenom tranzicijom na podstanju. Nadstanje se predstavlja kao kontura koja zaokružuje podstanja. Moguće su tranzicije od nadstanja u podstanje i obrnuto, kao i tranzicije iz podstanja u neko drugo stanje van konture.

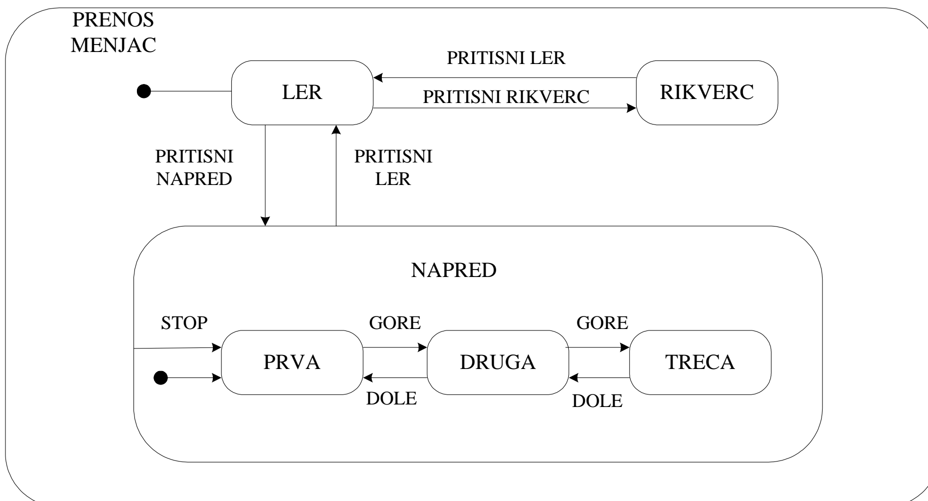


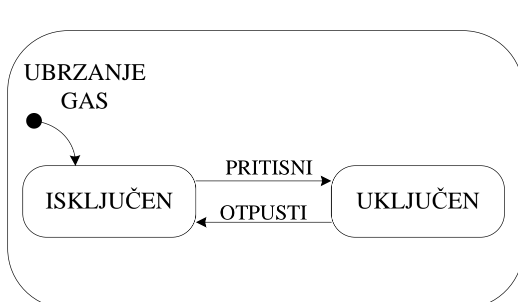
Agregacija stanja - agregaciona konkurentnost. Konkuretna stanja su posledica postojanja složenih objekata koji su agregacija svojih komponenti ili postojanja višestrukih paralelnih aktivnosti.

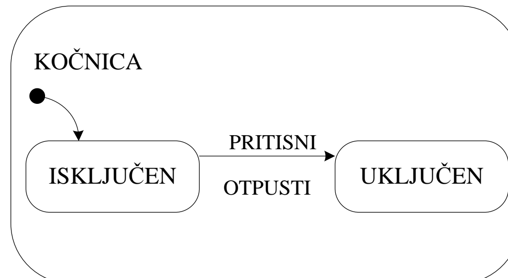
U slučaju složenih objekata, svaka komponenta može da ima svoja stanja i svoj dijagram prelaza stanja. Stanje agregiranog objekta je Dekartov proizvod stanja njegovih komponenti.

Slučaju višestrukih paralelnih aktivnosti je prikazan ranije

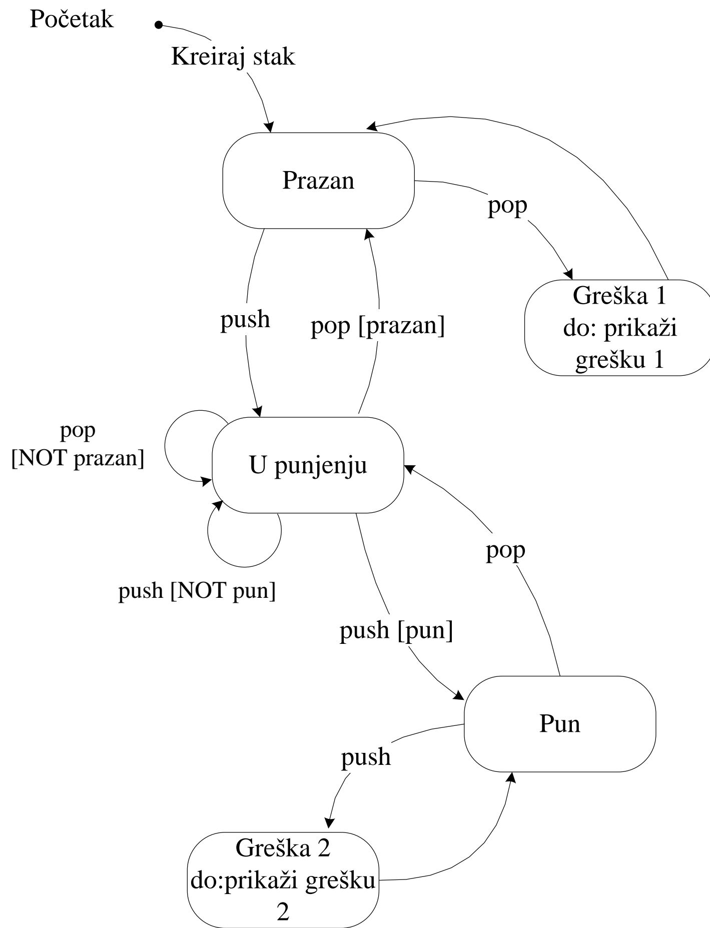




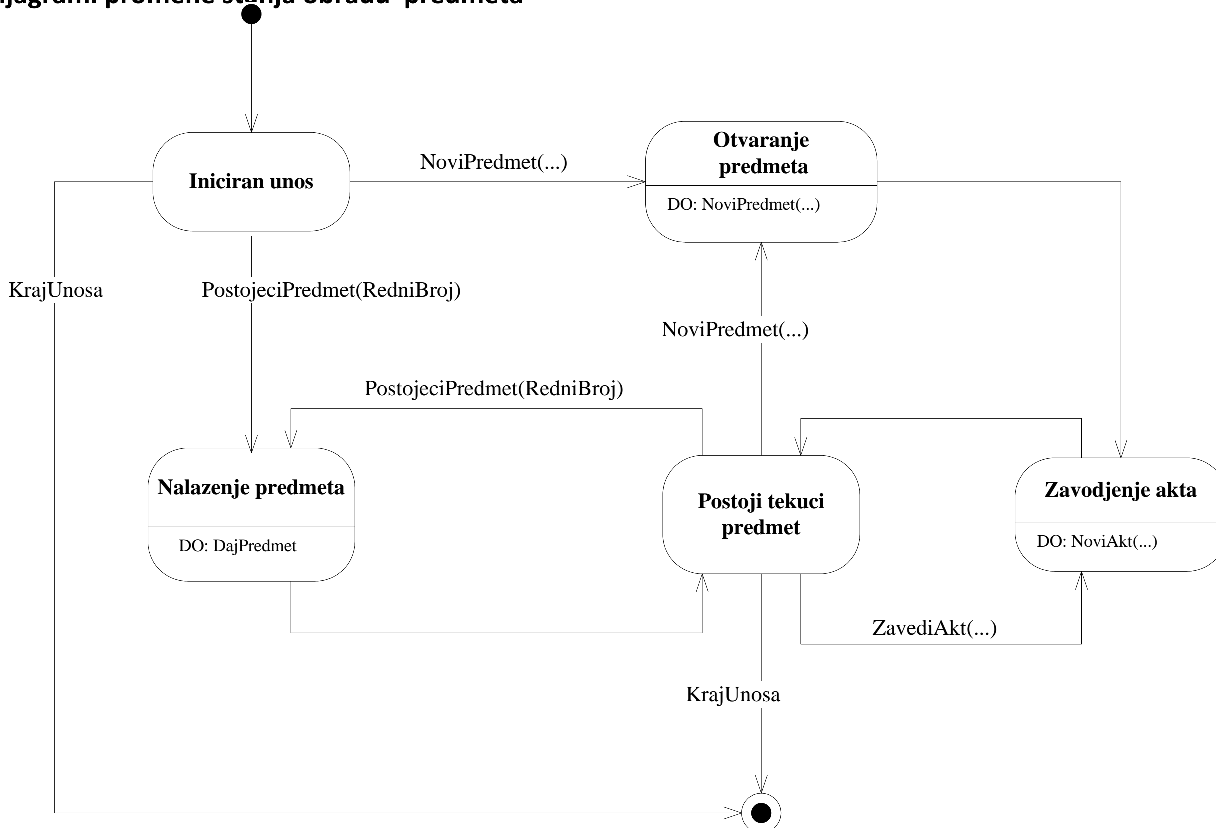


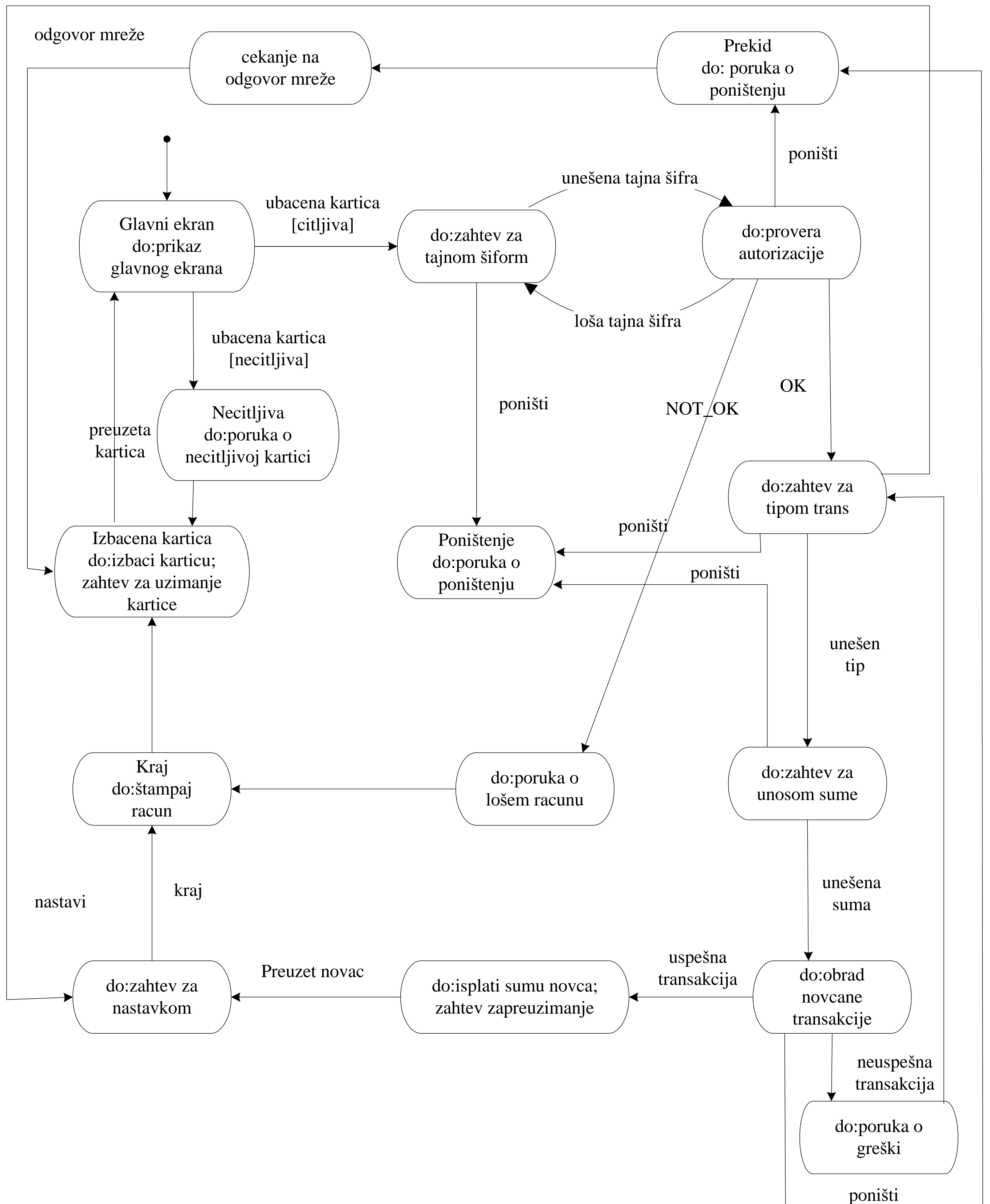


Dijagram promene stanja za stak

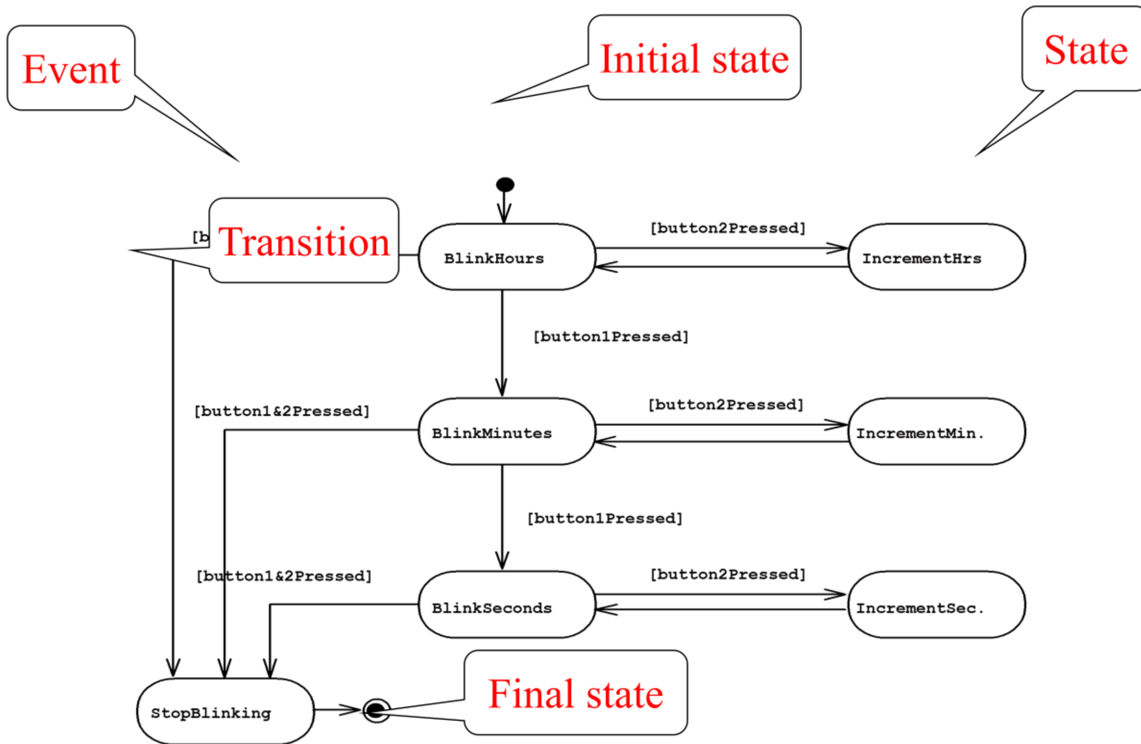


Dijagrami promene stanja obradu predmeta




odgovor mreže
cekanje na odgovor mreže
Prekid do: poruka o poništenju
poništi
unešena tajna šifra
Glavni ekran do:prikaz glavnog ekrana
ubacena kartica [citljiva]
do:zahtev za tajnom šiform
do:provera autorizacije
loša tajna šifra
ubacena kartica [necitljiva]
Necitljiva do:poruka o necitljivoj kartici
preuzeta kartica
poništi
NOT_OK
OK
do:zahtev za tipom trans
Izbacena kartica do:izbaci karticu; zahtev za uzimanje kartice
Poništenje do:poruka o poništenju
poništi
poništi
unešen tip
Kraj do:štampaj racun
do:poruka o lošem racunu
do:zahtev za unosom sume
unešena suma
nastavi
kraj
uspešna transakcija
do:zahtev za nastavkom
Preuzet novac
do:isplati sumu novca; zahtev zapreuzimanje
do:obrad novcane transakcije
neuspešna transakcija
do:poruka o greški
poništi

State Chart Diagrams

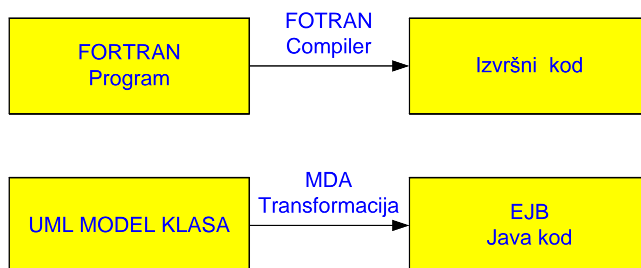


Represent behavior as states and transitions

10. Modelom vođeni razvoj (arhitektura)

– 16 str.

Modelom vođeni razvoj, kao i svi drugi pristupi u transformacionom ražvoju softvera, nije ništa revolucionarno novo. To je samo jedan viši nivo kompilacije jednog u drugi jezik.

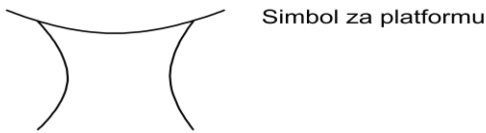


MDA i interoperabilnost

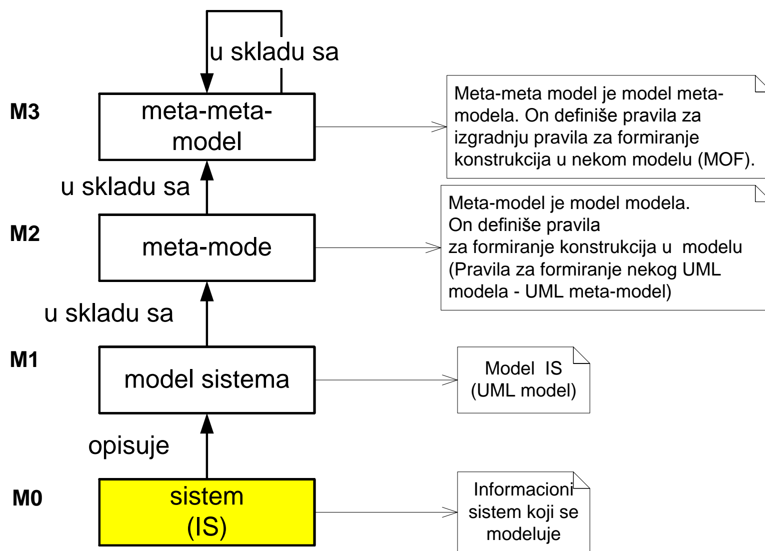
- Sa tačke gledišta korisnika heterogeni distribuirani računarski sistem treba da se posmatra kao "jedinstveni svet", bez ikakvih "platformski specifičnih barijera".
- Broj različitih platformi stalno raste i ne može se očekivati standardizacija (ptpuna unificiranost) na tom nivou.
- Preostaje da se standardizacija ostvari na nivo specifikacije, odnosno modela i da se preko tih opštih modela ostvari interoperabilnost delova sistema realizovanih na različitim platformama.

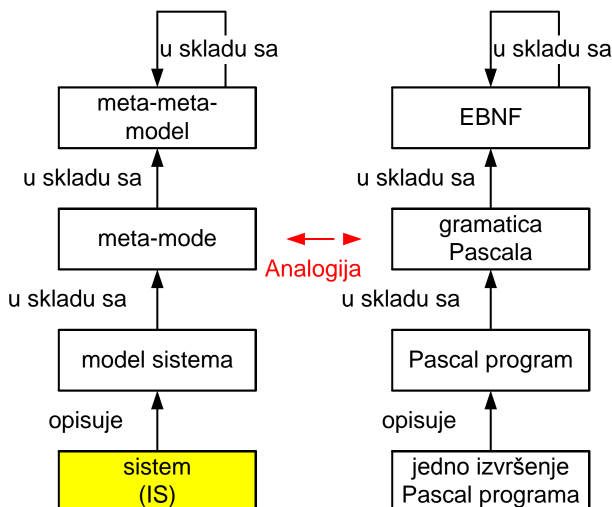
- “***Model driven***” zato što se u svim fazama razvoja (spefifikacija,projrktovanje, implementacija, održavanje) koriste modeli.
- “***Arhitektura***”. Arhitektura nekog sistema, uopšte, je specifikacija njegovih delova i pravila koja definišu način komunikacije između delova. U MDA pristupu definišu se modeli koji se koriste u razvoju softvera i pravila njihovog povezivanja.
- ***Platforma*** je sistem sa definisanim skupom funkcija (bez detaljne specifikacije njihove implementacije) preko koji je moguće implementirati neki softver.

- Vrste platformi:
  - ***Generičke****:Objektna, Batch, Tok podataka*
  - ***Specifične tehnologije****: CORBA, J2EE*
  - ***Tehnologije pojedinih proizvođača***: *CORBA: Borland VisiBroker;J2EEs: IBM WebSphere, Oracle;Microsoft .NET*.

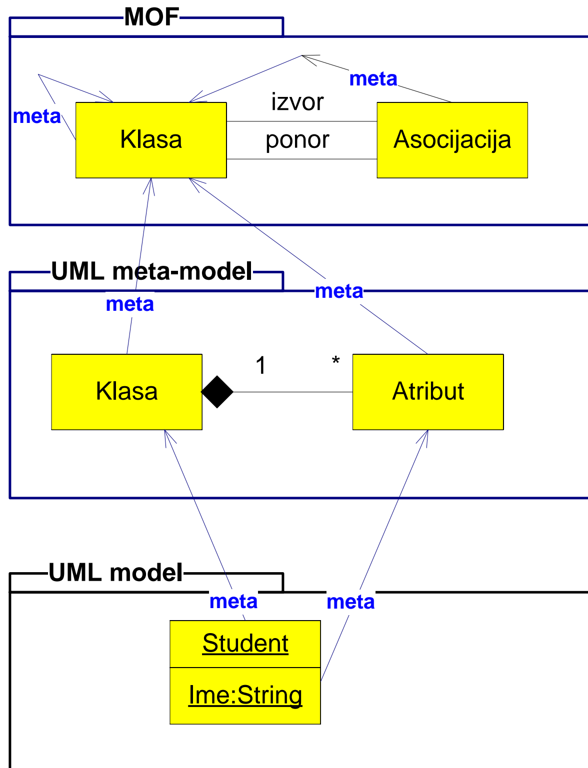


MODELI- četvoronovoska hijerarhija meta-modela



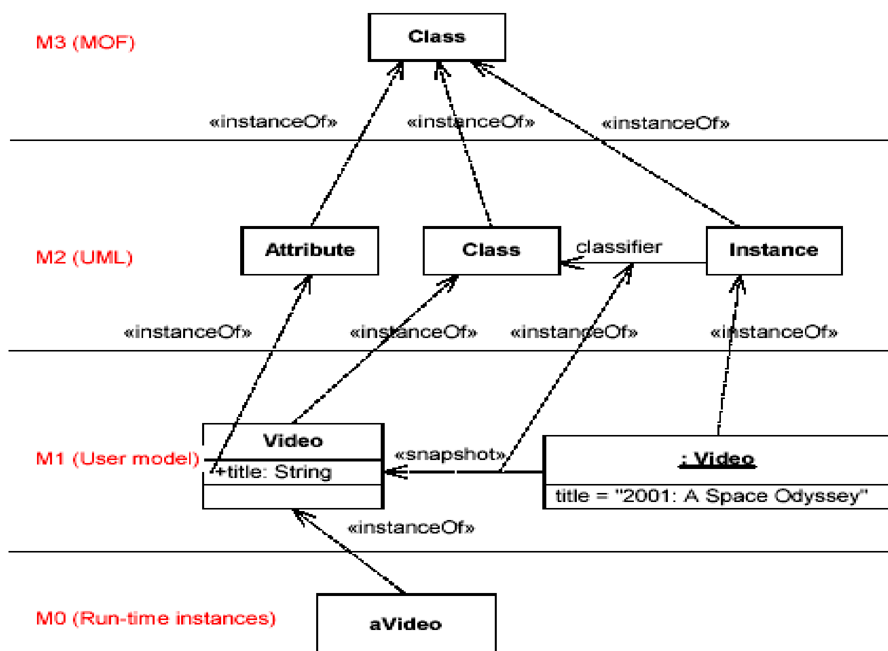


Modeli i metamodeli



MODELI- četvoronovoska hijerarhija meta-modela

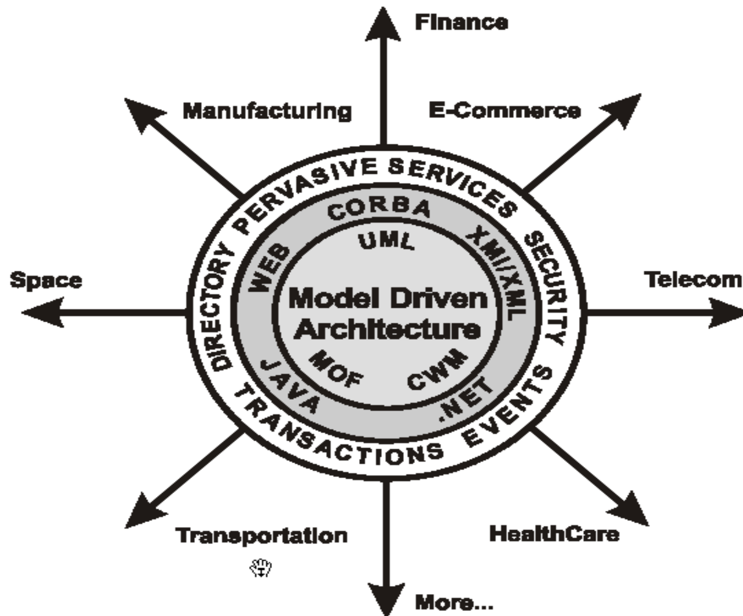
Kada se kaže da je meta-model "pojavljivanje" meta-meta modela podrazumeva se da je svaki koncepta meta-modela jedno pojavljivanje met-meta modela.



MDA OMG standardi

Osnovni MDA OMG astandardi su:

- UML
- MOF
- CWM
- XMI



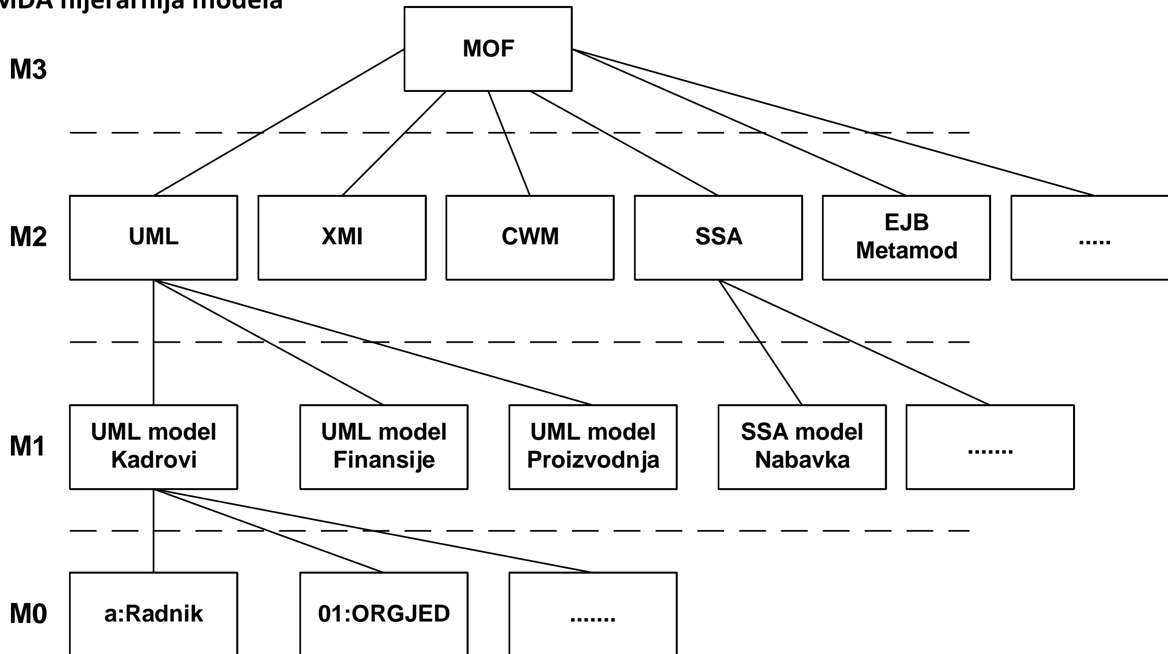
- **UML** – OMG standardni jezik za modelovanje diskretnih sistema:
  - UML 2.0 standard dat je u dva osnovna dokumenta: (1) *UML 2.0: Infrastructure - osnovni koncept jezika i (2) UML 2.0: Superstructure*- nadgradnja sa konstrukcijama korisničkog nivoa.
- **MOF** (Meta object facility) – standardni jezik za specifikaciju meta modela. Mof sdarži dva osnovna dela:
  - Abstrakni model generičkih objekata i njihovoh asocijacija (vrlo sličan sa UML jezgom)
  - Skip pravila za preslikavanje MOF modela u jezički nezavisne interfejse. Implementacija ovih interfejsa omogućava pristup i ažuriranje bilo kog modela zasnovanog na MOF-u.

- **XMI** (XML Metadata Interchange). Definiše XML tagove preko kojih se može opisati serijalizovan model baziran na MOF-u. Služi prvenstveno za razmenu modela. Transformacija iz jednog u drugi često se vrši i na sledeći način:

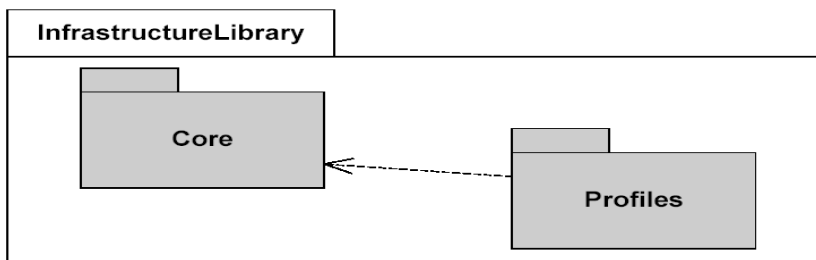
Model1 → XMI1→ XSLT →XMI2 → Model2

- **CWM *(****The Common Warehouse Metamodel)* definiše metamodele koji predstavljaju stovarišta podataka (data warehousing) i odgovarajuće OLAP analize.
- **JMI (**Java Metadata Interface) – omogućava preslikavanje MOF modela u Javu da bi se aostavrila mogućnost održavavanja MOF metabaza iz Jave.

MDA hijerarhija modela



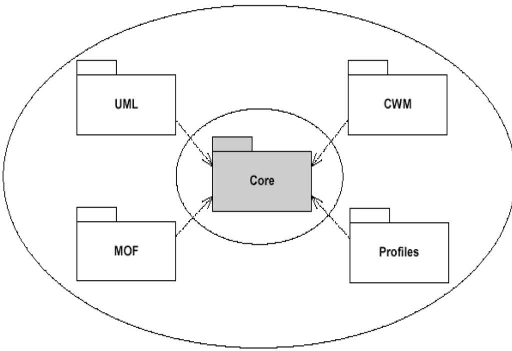
UML Infrastructure specifikacija

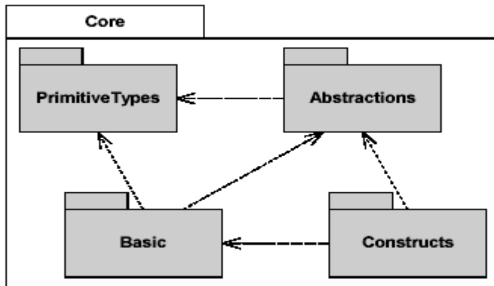


- UML specifikacija se daje preko UML metamodela.
- Jedan od osnovnih principa UML-a je proširljivost. Zbog toga
  UML 2.0 Infrastructure librarary sadrži dva osnovna dela:
  - ❑ Jezgro (Core)
  - ❑ Profiles - proširenja. Profil – mehanizam prilagođavanja modela konkretnim platformama (J2EE/EJB, .NET/COM+ i drugim)

UML- Odnos “Jezgra” i ostalih modela
Jezgro se koristi za definisanje drugih metamodela i kreiranje drugih jezika preko koncepta profila.
Jezgro je praktično zajednički deo za sve ove modele.

Paketi u UML jezgru





PrimitiveTypes sadrže nekoliko predefinisanih tipova potrebnih za metamodelovanje, posebno za UML i MOF.
Abstractions sadrži nekoliko abstraktnih metaklasa koje se višestruko koriste(specijalizuju) i različitim metamodelima.
Basic sadrži podskup koncepata iz paketa *Constructs* koji se prvenstveno koriste za objektne jezike i XMI.
Constracts sadrži metaklase koje su konkretneklase okrenute prema konceptima objektno-orjentisanog modelovanja (koriste se za UML i MOF i njihovo usaglašavanje)

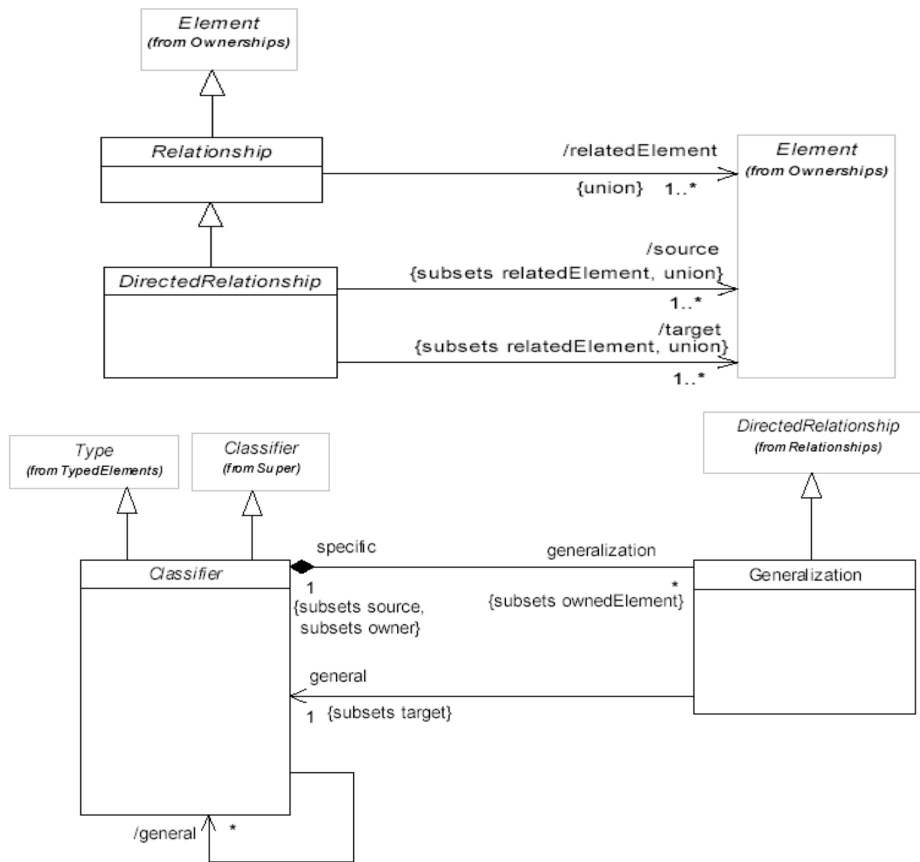
UML specifikacija

- Svaki koncept iz odgovarajućeg paketa se opisuje sa:
    - Dijagramom klasa
    - Opštim opisom
    - Opisom svih atributa
    - Opisom svih asocijacija
    - Specifikacijom semantike

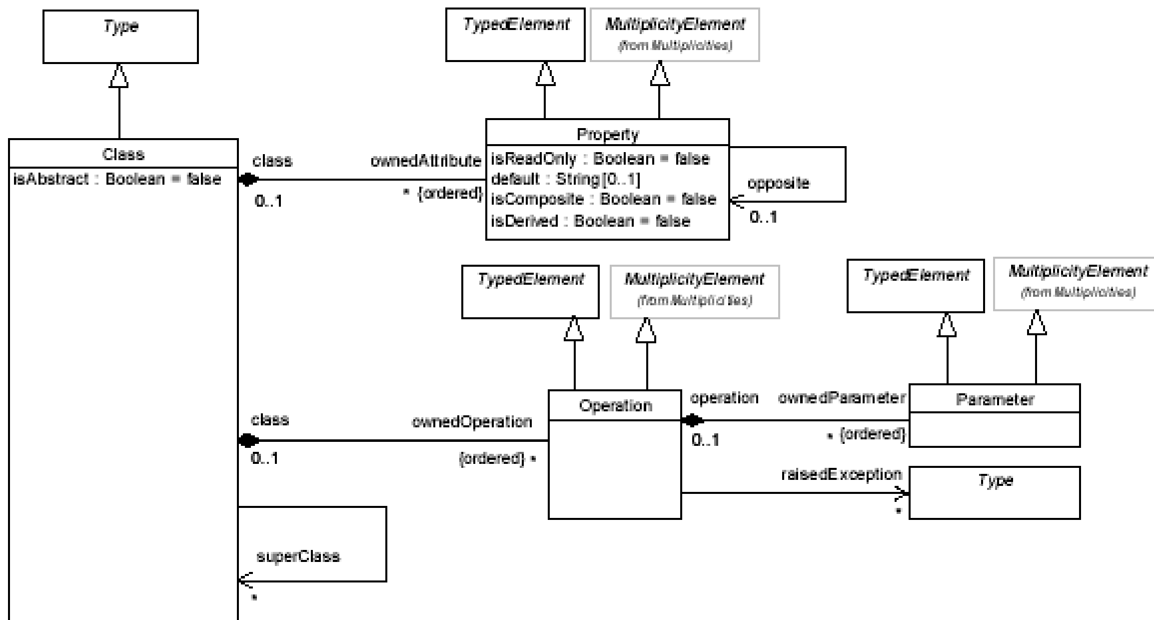
Generalization je podtip DirectRelationship

To znači da nasleđuje sve odgovarajuće osobine pa i asocijacije. Preslikavanje *generalization* je podskup preslikavanja *target*, a *specific* podskup preslikavanja *source*
To znači da ako se za neko pojavljivanje klase ***generalization*** zatheva ***target*** dobiće se instanca asocijacije ***general.***

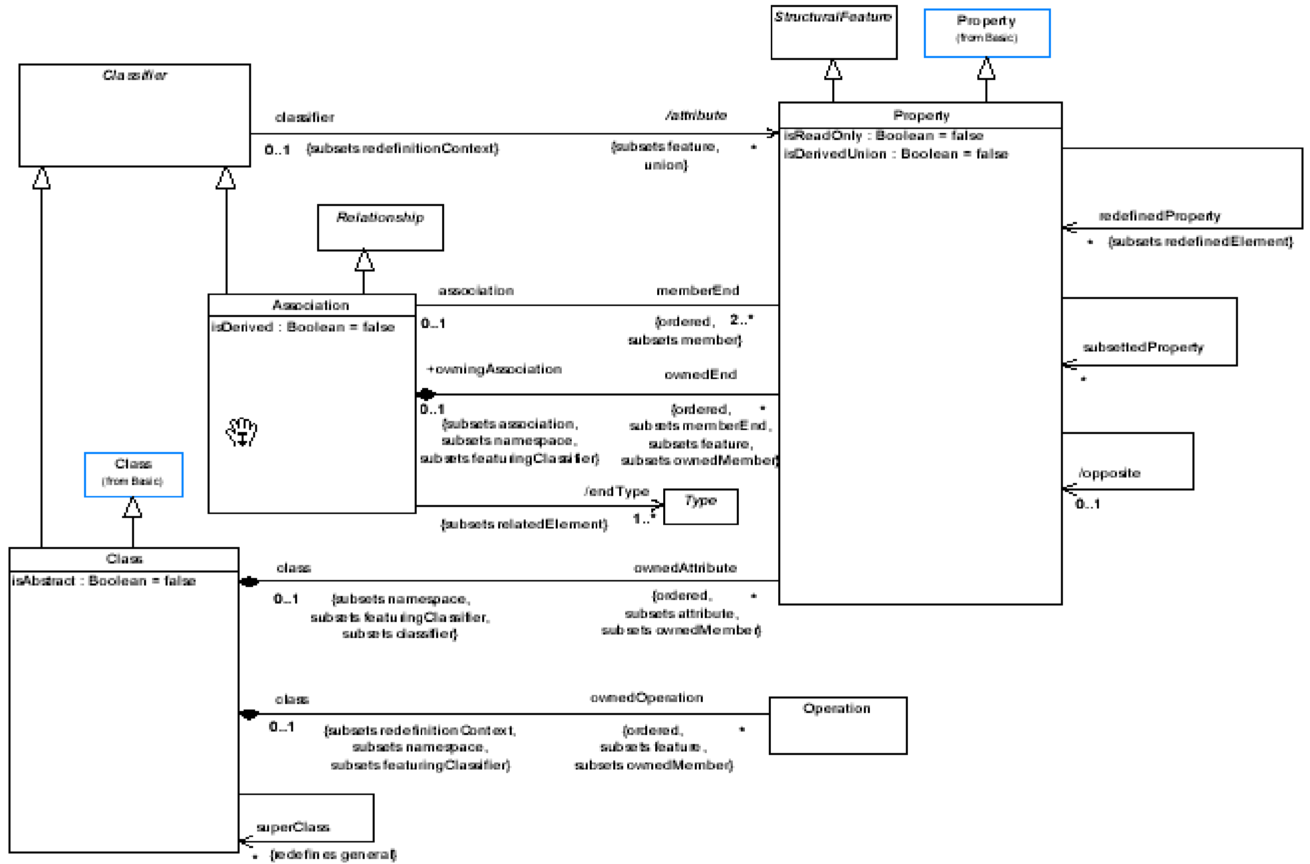
Primeri koncepata UML met-modela



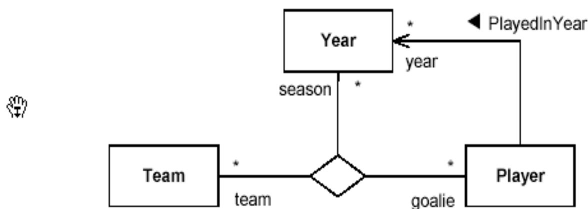
Basic – specifikacija klase



Constructions – specifikacija Dijagrama klasa



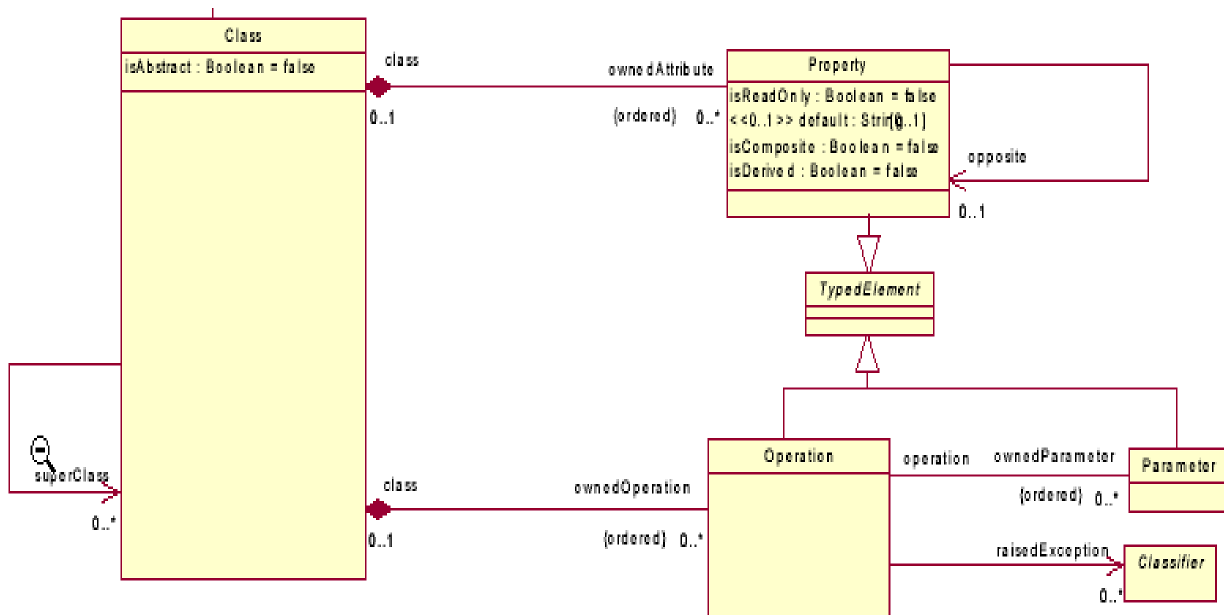
Višestruke asocijacije



Binarna i ternarna asocijacija

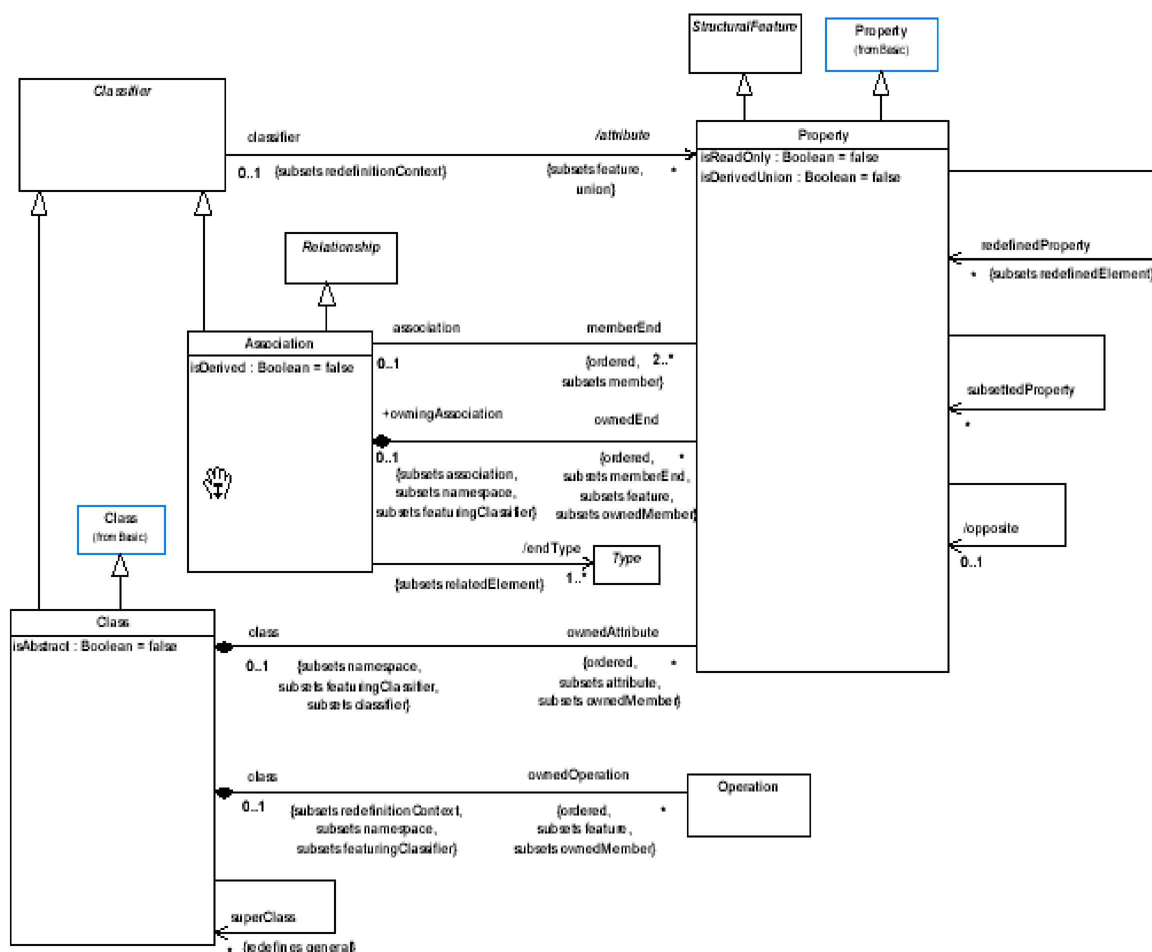
Essential MOF (EMOF)

Koristi osnovne elemente iz paketa Basic i Abstractions iz UML2 i ne dodaje svoje sopstvene klase. Na primer sam pojam klase se specifikuje na isti način. EMOF može da posluži za specifikaciju metamodela objektnih jezika i XMI



The Complete MOF (CMOF) se koristi specifikaciju drugih modela na primer UML-a

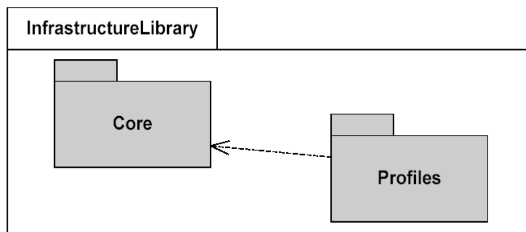
Specifikacija Dijagrama klasa



MOF

- Za konstrukciju meta-modela koristi se MOF
- Pošto je meta-model model modela, na njega se može primeniti isti jezik koji se koristi za izgradnju modela nekih sistema, pod uslovom da je takav jezik reflesivan (što praktično znači dovoljno opšt)
- UML dijagram klasa ima tu osobinu i zato je on osnova MOF-a.

UML profil



Profil je mehanizam prilagođavanja meta- modela konkretnim platformama (J2EE/EJB, .NET/COM+, i drugim) Nikakva postojeće ograničenja u UML meta-modelu se ne ukidaju, profilom se samo dodaju nova.

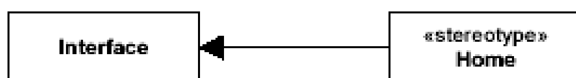
Razlozi za definisanje profila su:

- Uvođenje terminologije specifične za neku platformu
- Uvođenje specifičnih oznaka
- Proširenje semantike nekog koncepta meta-modela
- Ograničavanje upotrebe koncepata metamodela
- Dodavanje informacija za transformisanje PIM→PSM

Na pitanje kada koristiti profil, a kada formirati novi Metamodel (jezik), nema unapred definisanog odgovora.

Mehnaizmi za proširenje uML metamodela su **“tagged values” ograničenja i stereotipovi**. Osnovni mehanizam je ***stereotip.*** Stereotip definiše kako se proširuje neka metaklasa (ili prethodno definisani stereotip). Ne može da se koristi sam nego zajedno sa metaklasom koju proširuje. Jedna metaklasa može da ima više različituh stereotipova. Streotip se označavi bilo sa

- ***<< naziv stereotipa>>*** ili
- Novim grafičkim simbolom



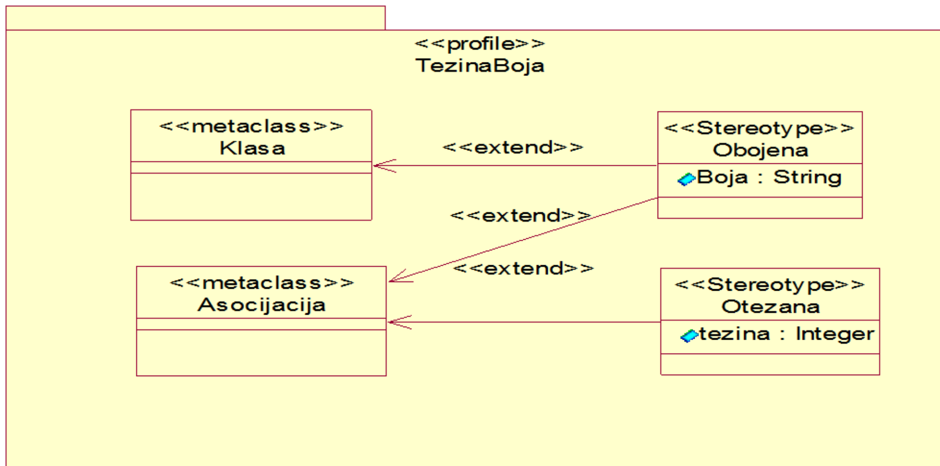
Extension specijalna vrsta asocijacije koja definiše proširenje

I sam UML sadrži neke predefinisane profile. Time su uvedeni novi koncepti koji proširuju semantiku nekih osnovnih UML koncepata sa ili bez uvođenja novih oznaka.

- ✓ <<include>> i <<extend>> u Dijagramima slučajeva korišćenja kao proširenje veže zavisnosti.
- ✓ <<interface>> stereotip klase
- ✓ <<executable>> stereotip komponente
- ✓ <<framework>> stereotip paketa
- ✓ **<<profile>> stereotip paketa**

Primer definisanja profila

Profil TežinaBoja

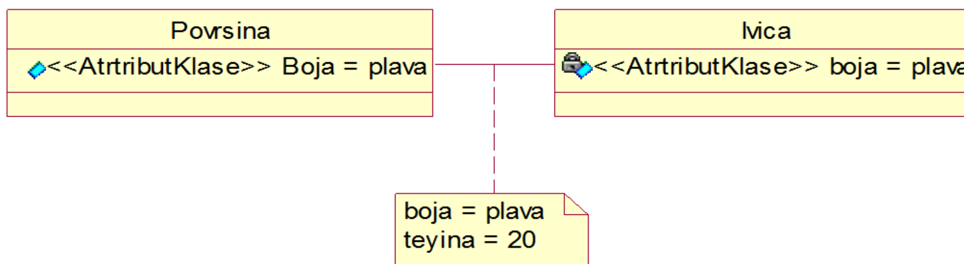


I za klasu i za asocijaciju u UML-u se može definisati boja, a za asocijaciju samo “težina”

- Za uvedene koncepte u profilu se mogu definisati ograničenja (OCL ili prirodni jezik)
    - Sa obojenom asocijacijom se mogu povezati klase koje imeju tu istu boju.

context UML::InfrastructureLibrary::Core::Constructs::Association
inv: self.isStereotyped(“Obojena”) **implies**
self.connection -> **forAll** (isStereotyped(“Obojena”)
implies boja=self.boja)

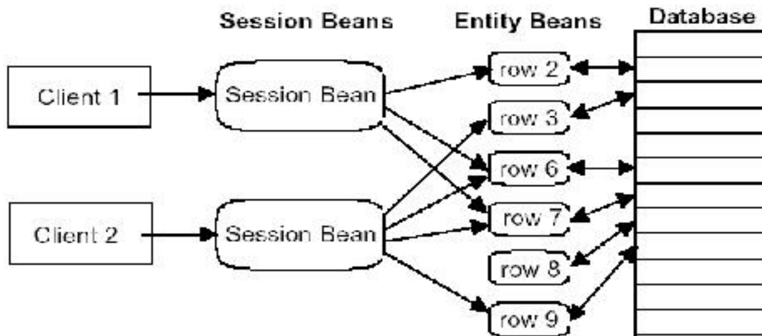
Tagged value su meta-atributi prudruženi metaklasi (boja, težina).
U modelu se predstavljaju kao atributi klasa preko para (naziv taga, vrednost)



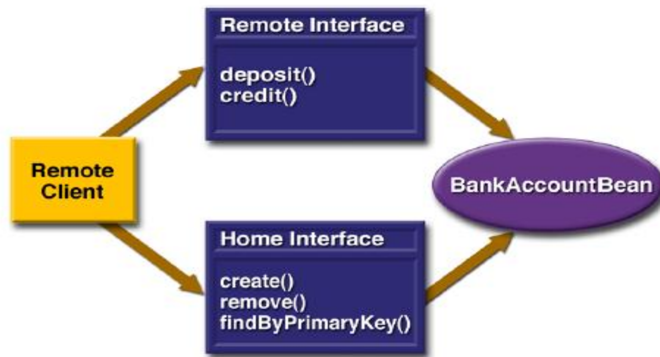
Enterprise Java-Beans (EJB) tehnologija
EJB su komponente J2EE komponente koje se izvršavaju u okviru EJB kontejnera koji obezbeđuje upravljanje transakcijam i sigurnost.
Enterprise beans su komponente u kojima je učaurena poslovna Logika sistema (aplikacije).

Session bean	Jedan ***Session bean*** predstavlja jednog klijenta na serveru. Da bi pristupio aplikaciji na serveru kljijent koristi Seesion Bean. On sakriva kompleksnost aplikacije od kljijenta. Nije perzistentan (Statless, Stateful)

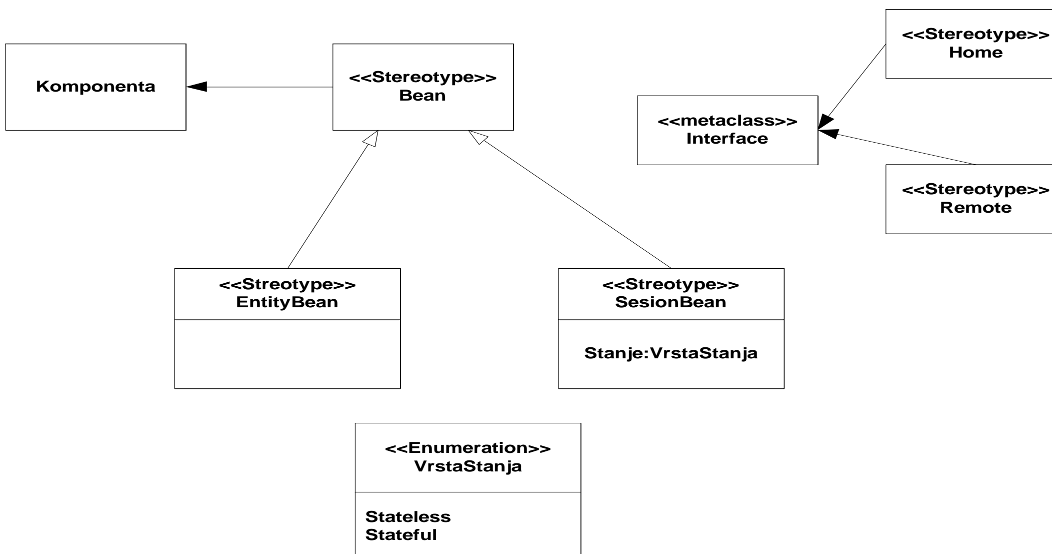
Entity bean	Predstavlja poslovni entitet koji postoji u bazi. Povezan je sa jednom tabelom ili pogledom u relacionoj bazi. Jedno njegovo pojavljivanje odgovara redu u tabeli. Jedan **entity bean** može da koristi više klijenata. Svaki **entity bean** ima jedinstveni identifikator (primary key)
Message-driven been	Predstavlja "listener for the Java Message Service API", za procesiranje asinhronih poruka.



Klijent može da ptristupi Session ili Entity beans samo preko predefinisanih interfejsa



- *Remote interface* definiše za dati bean specifične poslovne metode.
- *Home interface* definiše "životni ciklus" i metode za nalaženje bean-a.



Preslikavanje PIM →PSM

- MDA preslikavanje definiše transformacije iz PIM u specifičan PSM u skladu sa odgovarajućim modelom platforme.
- **Preslikavanje tipova** - definišu se pravila preskivanja u koja mogu da budu uključene i vrednosti iz pojavljivanja. *Preslikavanje na nivou metamodela* je preskikavanje tipova.
- **Preslikavanje pojavljivanja modela.** *Uvođenje oznaka*

Oznake (Marks) mogu da budu:

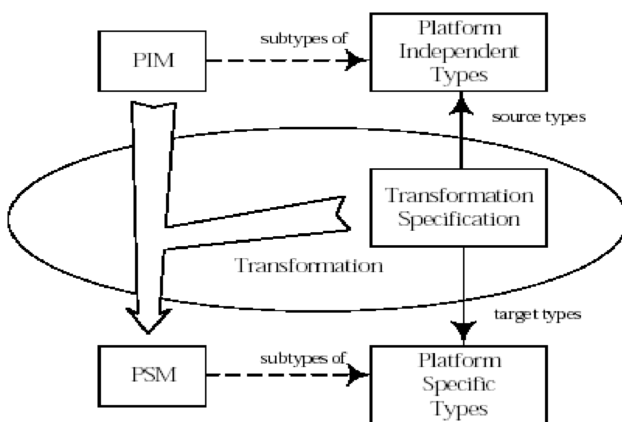
- Tipovi modela (klase, asocijacije, uloge u asocijacijama ili drugi tipovi)
- Stereotipovi iz UML profila
- Elementi metamodela (MOF-a, na primer)
- Neki nefunkcionalni zahtevi (specifikacija kvaliteta ili zahtevanih performamsi).
  - Primer: **Entity** oznaka se može primeniti na klasu ili objekat u PIM-u čime se naznačava da će se na te klase ili objekte primenti **temlejt** sa nazivom Entity.

Tempejt je model koji definiše transformaciju (neka vrsta "design pattern").

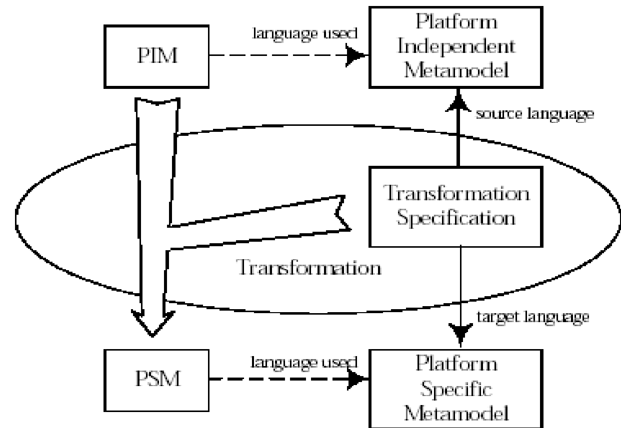
- Primer**:** *CORBA Entity* template specifikuje da se objekta iz PIM-a označen sa *Entity* preslikava u CORBA PSM model u objekat tipa *HomeInterface* i objekat tipa *EntityComponent* i njihovu međusobnu vezu.

Jezik za preslikavanje*:* prirodni jezik, neki algoritamski jezik ili jezik za preslikavanje modela (XSLT za XMI1→XMI2). The current. MOF Query/View/Transformation RFP (Request for Proposal) zahteva da se u okviru MOF-a definiše standardni jezik za preslikavanje modela.

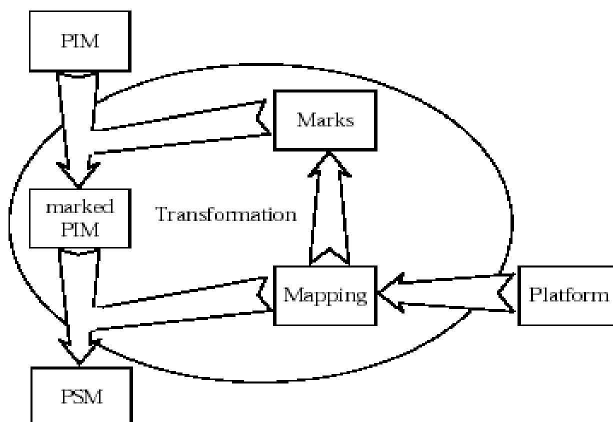
PIM →PSM preslikavanje tipova



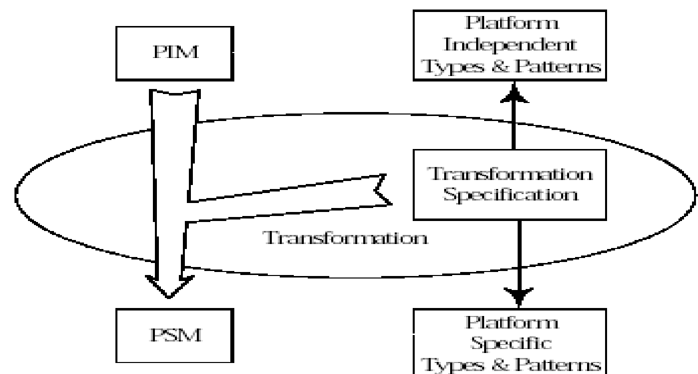
Preko metamodela



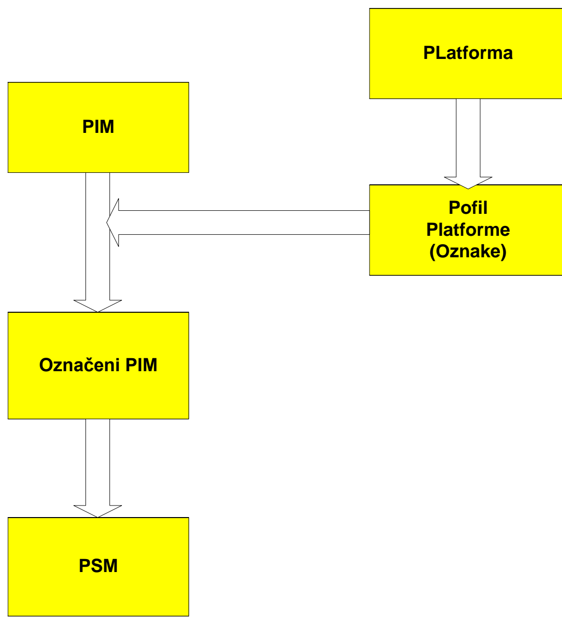
Preslikavanje PIM →PSM preko oznaka



Preko tipova i pefinisanih paterna meta-modela



Preslikavanje PIM →PSM preko profila



- Izgrađuje se profil platforme u koji se PIM preslikava – koncepti UML meta-modela ili MOF-a se proširuju da bi se obuhvatile specifičnosti platforme
- Koncepti PIM modela označavaju se sa oznakama koje odgovaraju konceptima profila platforme
- Vrši se preslikavanje preko pravila koja se definišu na ovako označenim PIM-u

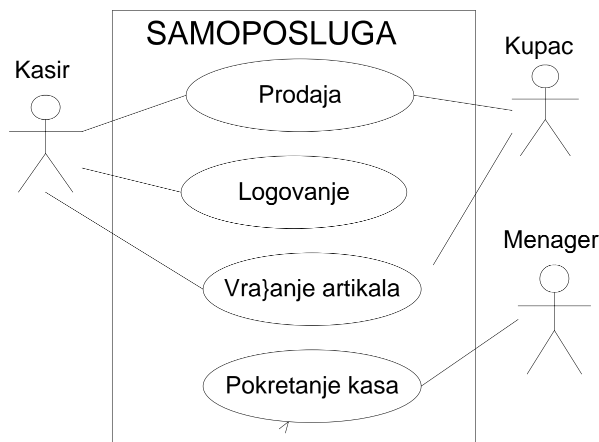
Izgradnja PIM-a

PIM se izgradjuje nekim postupkom specifikacije IS.

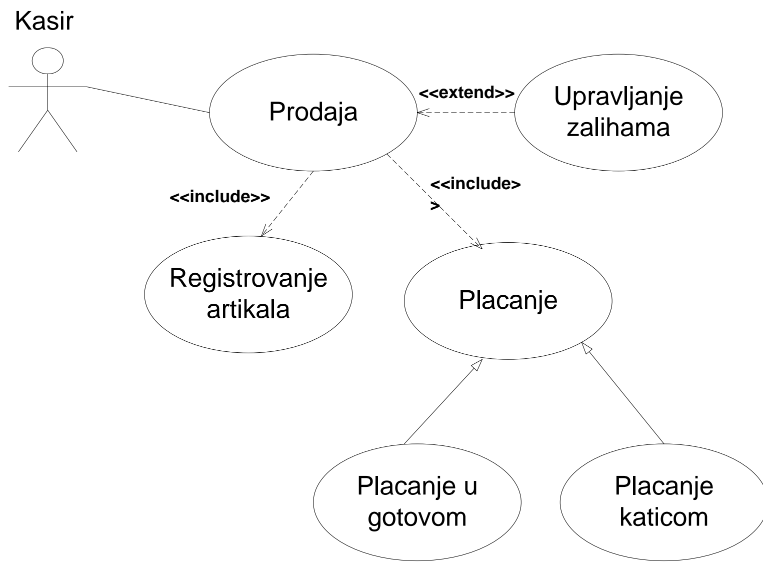
Specifikacija IS preko "Sitemsko-teorijskog životnog ciklusa:

1. **Identifikacija** – nalaženje funkcionalnog modela
    - Definisanje slučajeva korišćenja
    - Definisanje sistemskih dijagrama sekvenci
2. **Realizacija**
    - Specifikacija strukturnog (konceptualnog) modela
    - Specifikacija dinamike sistema (Dijagmi sekvenci, kolaboracije, aktivnosti)
    - Specifikacija konačnog objedinjenog strukturnog i dinamičkog modela

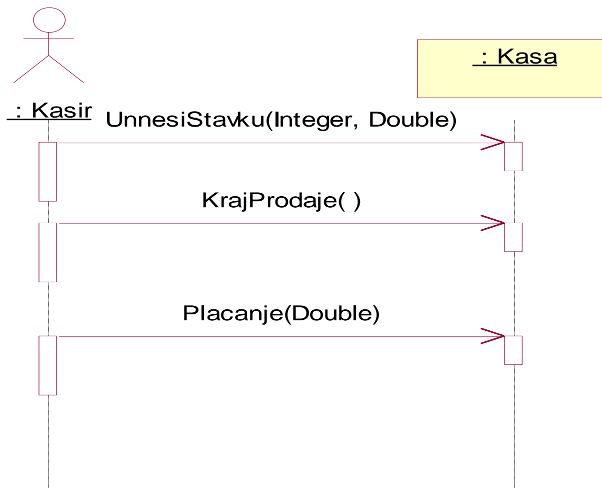
SLUČAJEVI KORIŠĆENJA- DIJAGRAM



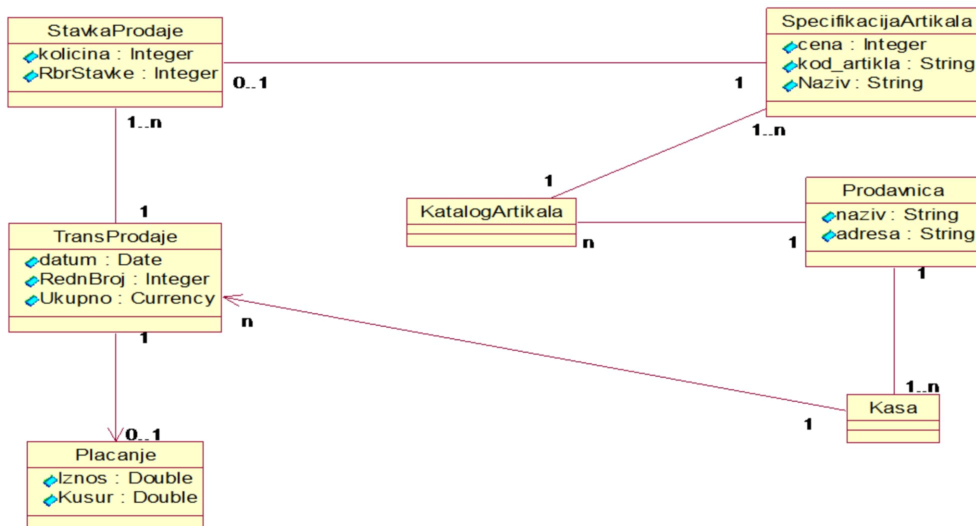
SLUČAJ KORIŠĆENJA PRODAJA



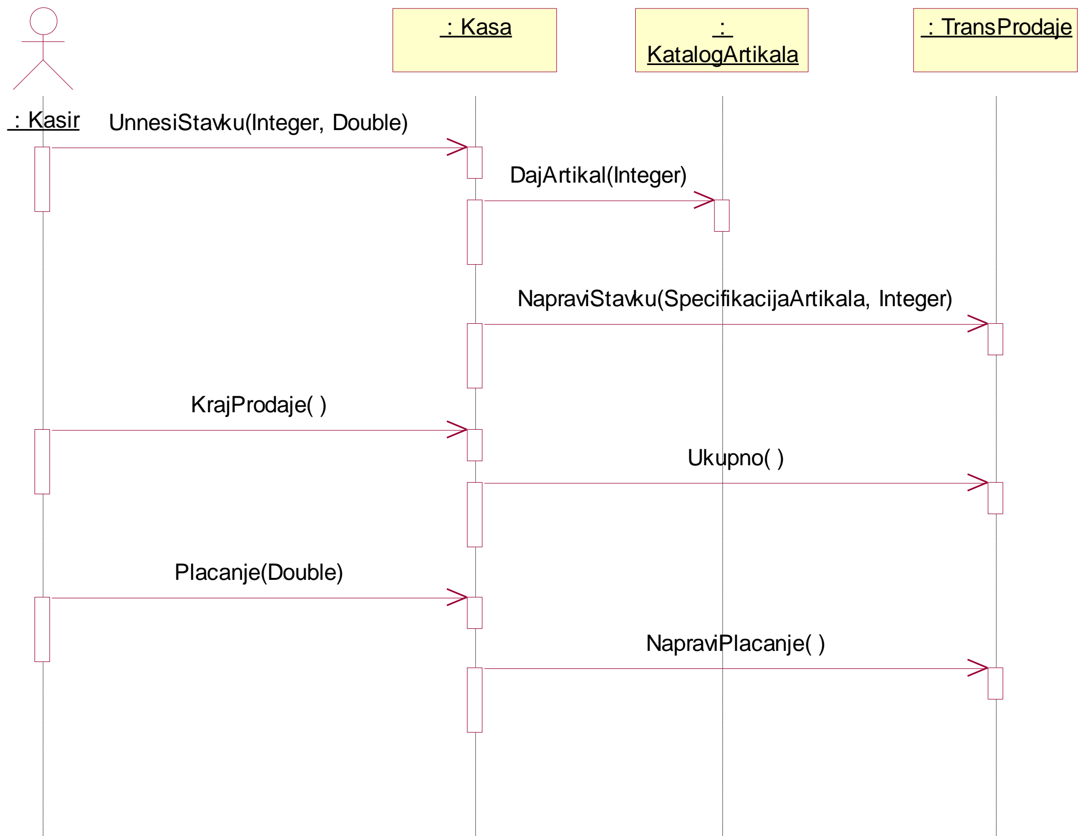
Sistemski dijagram sekvenci



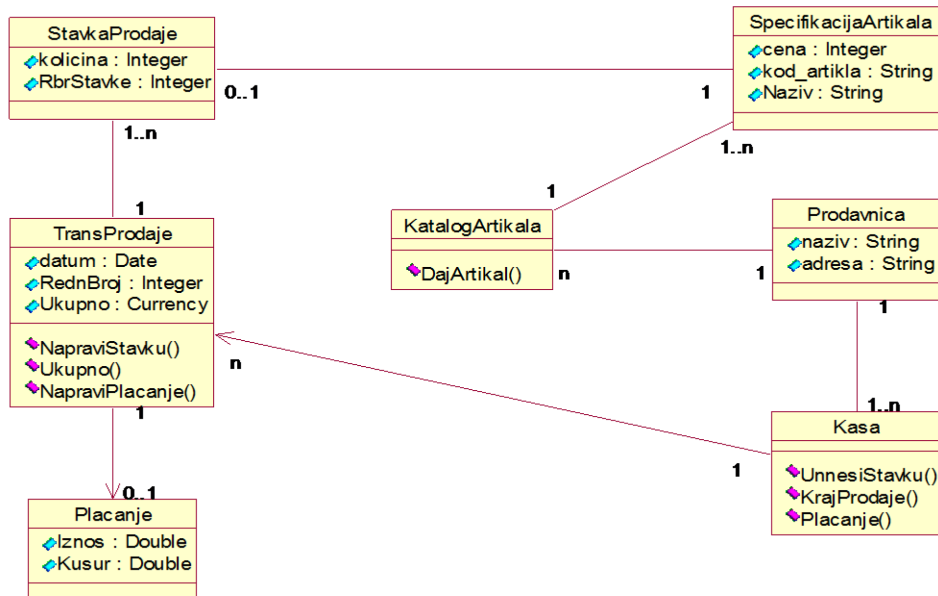
Konceptualni model



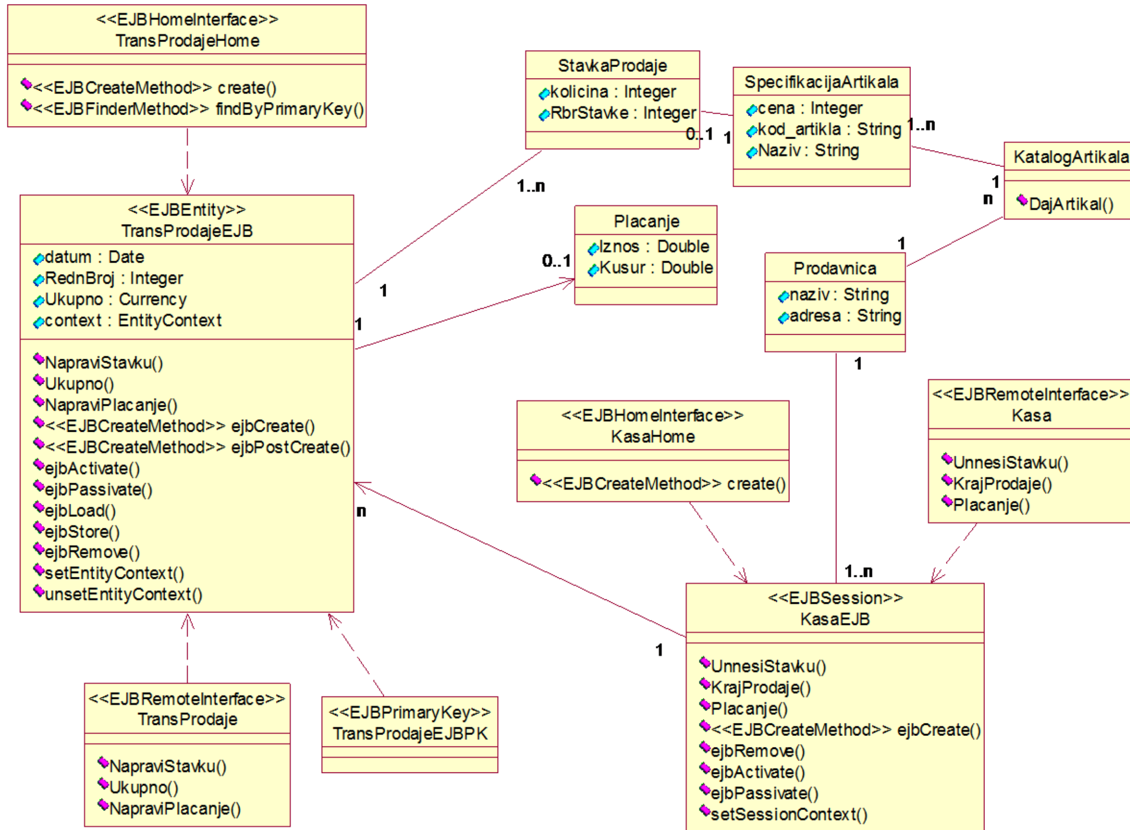
Dinamika sistema – Dijagram sekvenci



Konačni PIM



EJB PSM



Deo generisanog koda

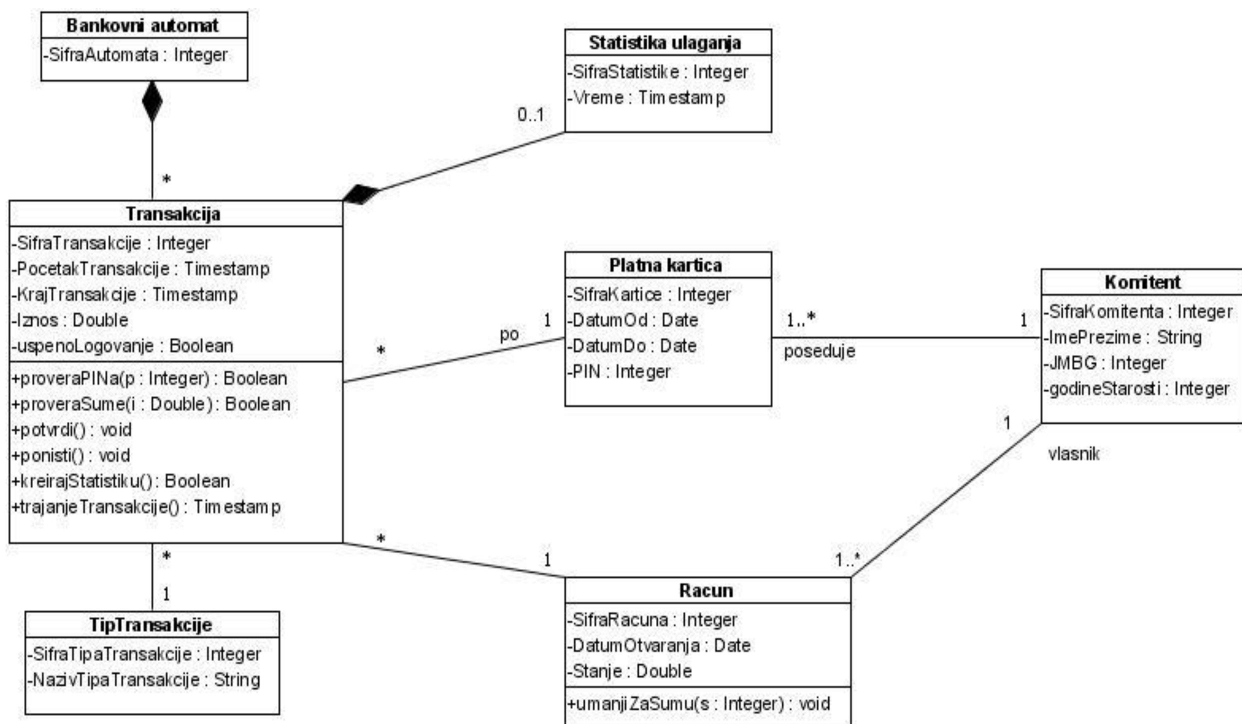
Java Code Generation Process --

```
// Import Statements
import java.rmi.RemoteException;
import javax.ejb.*;
public class KasaEJB implements javax.ejb.SessionBean {
   public Prodavnica theProdavnica;
   public TransProdajeEJB theTransProdajeEJB;
   /*
     Method: Default Constructor
   */
   public KasaEJB()  {}
   /*
     Method:  UnnesiStavku
   */
   public Double UnnesiStavku(Integer PrKod, Double Kol)  {}
   /*
     Method:  KrajProdaje
   */
   public Prodaja KrajProdaje()  {}
   /*
     Method:  Placanje    */    public Double Placanje(Double Iznos)  {}
```

OCL (Object constraint language)

- **Object Constraint Language (OCL)** je formalni jezik za definisanje izraza u UML modelima.
  Izrazi najčešće predstavljaju *ograničenja* koja moraju biti zadovoljena u okviru sistema koji se modeluje ili *upite* nad objektima u modelu.
- Deklarativan jezik
  Specificira se ono što mora biti zadovoljeno, ne ono što treba da se uradi
- Nema spoljašnjih efekata (side - effect)
  Evaluacija OCL izraza nikada ne dovodi do promene stanja sistema
- Nije programski jezik
  Nije moguće definisati programsku logiku niti kontrolu toka upotrebom OCL - a

- Prva verzija razvijena 1995. godine, od strane IBM – a
- Trenutna verzija je OCL 2.2
- OCL je deo UML standarda
- Može se koristiti za definisanje ograničenja ili upita nad svim vrstama UML dijagrama
  najčešće je to UML Class diagram



Context

Kontekst (Context) definiše kontekst upotrebe OCL izraza u okviru UML modela.

npr. za UML Class Diagram, kontekst OCL izraza se koristi za identifikovanje elementa (klasa, interfejs...) na koji se odnosi OCL izraz

Sintaksa:

```
context <classifier>
```

Primer: OCL izraz se odnosi na klasu Komitent

```
context Komitent
```

Invariant

- Konstanta (Invariant) koristi se za definisanje OCL ograničenja koje treba da bude zadovoljeno tokom celokupnog životnog ciklusa objekta.
- Drugačije rečeno, Invariant ograničenje uvek mora biti zadovoljeno.
- Označava se klauzulom inv.
- Sintaksa:
  context <classifier>
  inv[<naziv ograničenja>]: <Boolean OCL izraz>

  Primer: datum isteka važenja kartice mora uslediti nakon datuma početka važenja kartice (atributi DatumOd i DatumDo klase PlatnaKartica)

  context PlatnaKartica
  inv : DatumDo > DatumOd
  context PlatnaKartica
  inv : self.DatumDo > self.DatumOd
  context PlatnaKartica
  inv datumOgraničenje: self.DatumDo > self.DatumOd

Precondition

- Ograničenje koje se odnosi na operaciju klase u UML class diagramu.
- Ograničenje koje mora biti zadovoljeno ***u trenutku*** izvršavanja operacije.
- Označava se klauzulom pre.
- Sintaksa:
  context <classifier> :: <operacija>(<parametri>)
  pre[<naziv ograničenja>]: <Boolean OCL izraz>

  Primer: može se proveriti raspoloživost sredstava za uneti iznos, ako je prethodno uneti PIN odgovarajući
  context Transakcija :: proveraSume(i : Double)
  pre : self.uspesnoLogovanje=true
  Primer: uneti iznos mora biti veći od 0
  context Transakcija :: proveraSume(i : Double)
  pre : i>0

Postcondition

- Ograničenje koje se odnosi na operaciju klase u UML class diagramu.
- Ograničenje koje mora biti zadovoljeno ***po završetku*** izvršavanja operacije.
- Označava se klauzulom post.
- Sintaksa:
  context <classifier> :: <operacija>(<parametri>)
  post[<naziv ograničenja>]: <Boolean OCL izraz>

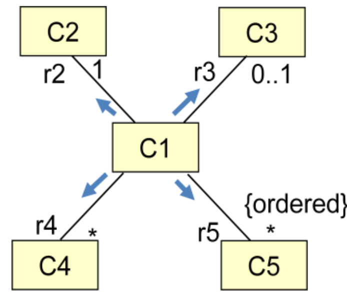
- Primer: trajanje transakcije mora biti jednako razlici početka i kraja transakcije
  context Transakcija::trajanjeTransakcije():Timestamp
  post: result = self.krajTransakcije – self.pocetakTransakcije

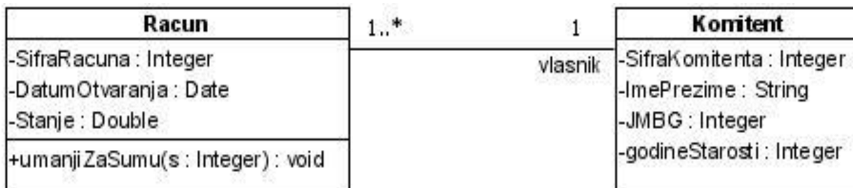
- result klauzula
    - odnosi se na rezultat (povratnu vrednost) operacije nad kojom je definisano Postcondition ograničenje.

- Primer: stanje na računu posle podizanja novca biće jednako početnom stanju umanjenom za podignutu sumu
  context Racun:: umanjiZaSumu(s:Integer):void
  post: stanje = stanjet@pre - s
- @pre klauzula
    - odnosi se na vrednost atributa pre izvršenja operacije (npr. stanje@pre).
    - npr. stanje se odnosi na vrednost posle izvršenja operacije
    - @pre klauzula može se koristi isključivo u postcondition ograničenjima.

OCL navigacija

- Nazivi uloga (preslikavanja) koriste se za pristup odgovarajućim objektima preko međusobne veze.
  ukoliko naziv uloge nije definisan, koristi se naziv klase sa malim početnim slovom.
- U kontekstu instance x klase C1:
    - x.r2 je tipa C2
    - x.r3 je tipa C3
    - x.r4 je tipa Set(C4)
    - x.r5 je tipa OrderedSet(C5)





Primer: Vlasnik računa mora imati više od 17 godina
context Racun
inv : self.vlasnik.godineStarosti > 17

OCL tipovi podataka

- Predefinisani tipovi podataka
    - Primitivni tipovi: String, Integer, Real, Boolean
    - Tipovi kolekcije: Collection, Set, OrderedSet, Bag, Sequence
    - Tip n-torka: Tuple
    - Specijalni tipovi: OclAny, OclType...
- Korisnički definisani tipovi podataka
    - Korisničke klase, npr. Komitent, Racun, Transakcija...

Operacije i operatoti nad primitivnim tipovima

Tip	Vrednost	Operatori i operacije
Boolean	true, false	=, <>, and, or, xor, not, implies, if-then-else-endif
Integer	-1, 0, 1, ...	=, <>, >, <, >=, <=, *, +, - ,/ , abs(), max(b), min(b), mod(b), div(b)
Real	1.5, ...	=, <>, >, <, >=, <=, *, +, - , /, abs(), max(b), min(b), round(), floor()
String	'a', 'John'	=, <>, size(), concat(s2), substring(lower, upper) (1<=lower<=upper<=size), toReal(), toInteger()

Tipovi kolekcije i tip n-torka

Sintaksa	Opis	Primeri
Collection(T)	Kolekcija elemenata tipa T	
Set(T)	Neuređena kolekcija, nisu dozvoljeni duplikati	Set{1 , 2}
OrderedSet(T)	Uređena kolekcija, nisu dozvoljeni duplikati	OrderedSet {2, 1}
Bag(T)	Neuređena kolekcija, dozvoljeni duplikati	Bag {1, 1, 2}
Sequence(T)	Uređena kolekcija, dozvoljeni duplikati	Sequence {1, 2, 1} Sequence {1..4} (= {1,2,3,4})
Tuple(field1: T1, ... , fieldn : Tn)	N-torka(sa imenovanim delovima)	Tuple {age: Integer = 5, name: String = 'Joe' } Tuple {name = 'Joe', age = 5}

Collection(T) operacije i iteratori

<table>
<tr><th>Operacija</th><th>Opis</th></tr>
<tr><td>size(): Integer</td><td>Vraća broj elemenata u kolekciji.</td></tr>
<tr><td>isEmpty(): Boolean</td><td>Vraća true ukoliko je size = 0.</td></tr>
<tr><td>notEmpty(): Boolean</td><td>Vraća true ukoliko je size > 0.</td></tr>
<tr><td>count(object: T): Integer</td><td>Vraća broj pojavljivanja elemenata tipa T u kolekciji.</td></tr>
<tr><td>sum(): T</td><td>Vraća sumu svih elemenata u kolekciji (T mora podržavati “+”)</td></tr>
<tr><th>Iterator</th><th>Opis</th></tr>
<tr><td>exists(iterators | body) : Boolean</td><td>Vraća true ukoliko je body uslov zadovoljen za BAR JEDAN element u kolekciji.</td></tr>
<tr><td>forAll(iterators | body): Boolean</td><td>Vraća true ukoliko je body uslov zadovoljen za SVE elemente u kolekciji.</td></tr>
<tr><td>one(iterator| body): Boolean</td><td>Vraća true ukoliko je body uslov zadovoljen za TAČNO jedan element u kolekciji.</td></tr>
</table>

Set(T) operacije

Operacija	Opis
union(s: Set(T)): Set(T)	Unija seta nad kojim se poziva operacija i seta *s*.
intersection(b: Bag(T)): Set(T)	Presek seta nad kojim se poziva operacija i bag - a *b*.
including(object: T): Set(T)	Vraća originalni set uključujući element *object*.
excluding(object: T): Set(T)	Vraća originalni set isključujući element *object*.
asOrderedSet(): OrderedSet(T)	Vraća originalni set kao OrderedSet.
asSequence(): Sequence(T)	Vraća originalni set kao Sequence.
asBag(): Bag(T)	Vraća originalni set kao Bag.

Bag(T) operacije

Operacija	Opis
union(b: Bag(T)): Bag(T)	Unija bag-a nad kojim se poziva operacija i bag-a *b*.
intersection(s: Set(T)): Set(T)	Presek bag-a nad kojim se poziva operacija i set-a *s*.
including(object: T): Bag(T)	Vraća originalni bag uključujući element *object*.
excluding(object: T): Bag(T)	Vraća originalni bag isključujući element *object*.
asSequence(): Sequence(T)	Vraća originalni bag kao Sequence.
asSet(): Set(T)	Vraća originalni bag kao Set.
asOrderedSet(): OrderedSet(T)	Vraća originalni bag kao OrderedSet.

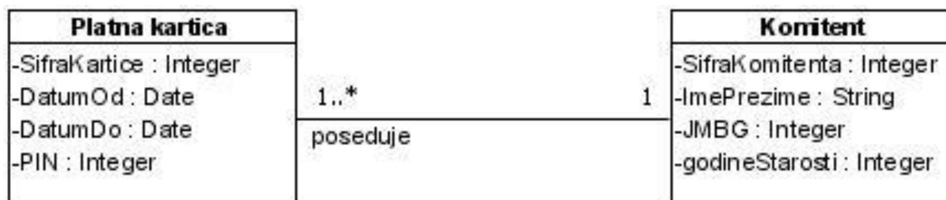
Sequence(T) operacije

Operacija	Opis
at(i : Integer) : T	Vraća i-ti element sekvence (1<=i<=size).
indexOf(object : T) : Integer	Vraća indeks elementa *object* u sekvenci.
first() : T	Vraća prvi element sekvence.
last() : T	Vraća poslednji element sekvence.
asBag(): Bag(T)	Vraća originalnu sekvencu kao Bag.
asSet(): Set(T)	Vraća originalnu sekvencu kao Set.
asOrderedSet(): OrderedSet(T)	Vraća originalnu sekvencu kao OrderedSet.
append(object: T): Sequence(T)	"Lepi" element *object* na kraj originalne sekvence.
prepend(obj: T): Sequence(T)	"Lepi" element *object* na početak originalne sekvence.
insertAt(i : Integer, object : T) : Sequence(T)	Dodaje element *object* na i-to mesto u skevenci(1<=index<=size+1).

OrderedSet(T) operacije

Operacija	Opis
at(i : Integer) : T	Vraća i-ti element uređenog seta(1<=i<=size).
indexOf(object : T) : Integer	Vraća indeks elementa *object* u uređenom setu.
first() : T	Vraća prvi element uređenog seta.
last() : T	Vraća poslednji element uređenog seta.
append(object: T): OrderedSet(T)	"Lepi" element *object* na kraj originalnog uređenog seta.
prepend(obj: T): OrderedSet (T)	"Lepi" element *object* na početak originalnog uređenog seta.
insertAt(i : Integer, object : T) : OrderedSet (T)	Dodaje element *object* na i-to mesto u uređenom setu(1<=index<=size+1).

Pozivanje OCL operacija



Za pozivanje operacija koristi se " ->", ne "."

Primer: u svakom trenutku, komitent mora posedovati barem jednu platnu karticu

context Komitent

inv : self.poseduje -> size() >= 1

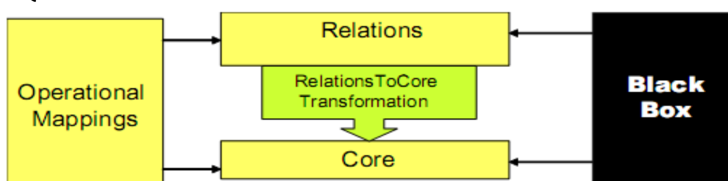
OCL alati

- Dresden OCL Toolkit
- Eclipse MDT/OCL
- Incremental OCL
- Kent OCL Library
- OCLE
- OSLO – Open Source Library for OCL

QVT (Query/View/transformation)

- QVT predstavlja standard za transformacije modela definisan od strane OMG – a (Object Management Group).
- Trenutna verzija standarda je QVT 2.0 .
- QVT specifikacija je hibridne deklarativno/imperativne prirode.
- Deklarativni deo je podeljen na dva nivoa.

QVT arhitektura



Relations jezik

- Deklarativna specifikacija veza između MOF modela.
- Podržava
  - Kompleksne paterne mapiranja objekata.
  - Implicitno kreiranje trace klasa i njihovih instanci, radi praćenja izvršenja transformacije.
- Poseduje i grafičku sintaksu.

Core jezik

Deklarativan jezik.
Mali model/jezik koji podržava mapiranja među promenljivim kroz evaluaciju uslova definisanih nad ovim promenljivim u odnosu na skup modela.
Jednake ekspresivnosti kao i Relations jezik, sa jednostavnijom semantikom.
Trace modeli se moraju definisati eksplicitno, nisu izvedeni iz same transformacije, kao što je slučaj sa Relations jezikom.

Operational Mappings jezik

Operational Mappings jezik je imperativan jezik koji proširuje Relations i Core jezike.
Oslanja se na OCL.
Sintaksa Operational Mappings jezika sadrži standardne imperativne konstrukcije (npr. različite vrste petlji).
Operational mapiranje podrazumeva mapiranje u potpunosti definisano upotrebom Operational Mappings – a.

Black Box implementacija

Dozvoljava implementaciju složenih algoritama u bilo kom programskom jeziku.
Dozvoljava upotrebu domain – specific biblioteka za izračunjavanje vrednosti propertija modela.

Relations jezik – transformacija

Transformacija modela predstavlja skup relacija koje moraju biti zadovoljene kako bi transformacija bila uspešna.
Transformacija se može koristiti za:

- Proveru konzistentnosti modela
- Izmenu jednog modela kako bi se održala konzistentnost

Modeli koji se transformišu moraju biti imenovani.
Modeli koji se transformišu su određenog tipa modela.

Primer: definisanje transformacije između UML i relacionog modela

```
transformation umlRdbms (uml : SimpleUML,
                         rdbms : SimpleRDBMS) { ... }
```

umlRdbms – naziv transformacije
uml i rdbms – modeli koji se transformišu
SimpleUML i SimpleRDBMS – tipovi modela koji se transformišu

Relations jezik – relacija i domen

Relacije u transformaciji definišu ograničenja koja moraju biti zadovoljena od strane elemenata modela uključenih u transformaciju.
Domen je tipizirana promenljiva koja se može "upariti" sa elementom modela određenog tipa modela.

Domen može imati definisan i patern koji predstavlja skup promenljivih i ograničenja koja elementi modela, vezani za te promenjlive, moraju zadovoljiti kako bi ispunili patern.

Primer: mapirati svaki UML paket u šemu baze podataka. Nazivi paketa i šeme moraju biti isti.

```
relation PackageToSchema{
    domain uml p:Package {name=pn}
    domain rdbms s:Schema {name=pn}
   }
```

Relations jezik – Checkonly i Enforce domeni

Kada se relacija izvršava u smeru **checkonly** domena, vrši se provera da li u tom domenu postoji element koji zadovoljava relaciju.

Kada se relacija izvršava u smeru **enforce** domena, vrši se provera da li u tom domenu postoji element koji zadovoljava relaciju. Ukoliko ne postoji takav element, model se menja kako bi zadovoljio relaciju.

Primer:

```
relation PackageToSchema{
    checkonly domain uml p:Package {name=pn}
    enforce domain rdbms s:Schema {name=pn}
   }
```

Relations jezik – Patern

Vezuje se za domen.

Template izraz koji se koristi da bi povezao elemente modela uključene u transformaciju sa promenljivama definisanim u okviru domena.

Primer:

```
    domain uml c:Class {
      namespace = p:Package {},
      kind='Persistent',
      name=cn
   }
```

Relations jezik – When i Where klauzule

- **When** klauzula definiše uslove pod kojima relacija mora da bude zadovoljena.
- **Where** klauzula definiše uslov koji mora biti zadovoljen od strane svih elemenata modela koji učestvuju u relaciji.
- When i where klauzule mogu sadržati i OCL izraze.

Primer: relacija ClassToTable mora važiti ukoliko važi relacija PackageToSchema. Ukoliko važi relacija ClassToTable, moraju važiti i sve odgovarajuće relacije AttributeToColumn.

```
relation ClassToTable{
   domain uml c:Class  {...}
   domain rdbms t:Table {...}
   when {
```

```
PackageToSchema(p, s);
    }
  where{
AttributeToColumn(c, t);
  }
```

Relations jezik – Top-level relacije

U izvršavanju transformacije, sve **top-level** relacije moraju biti zadovoljene.
Ostale relacije moraju biti zadovoljene samo ako su pozvane iz where klauzule neke druge relacije.

Primer:

```
transformation umlRdbms (uml : SimpleUML, rdbms :
SimpleRDBMS) {
    top relation PackageToSchema {…}
    top relation ClassToTable {…}
    relation AttributeToColumn {…}
}
```

Operational Mapping jezik

Operational Mapping jezik omogućava:

- Definisanje transformacija upotrebom potpuno imperativnog pristupa (operational transformacije)
- Dopunu relacionih transformacija imperativnim operacijama koje implementiraju relacije (hibridni pristup)

Operational Mapping jezik – Operational transformacije

Operational transformacija predstavlja definiciju jednosmerne transformacije koja je izražena na imperativan način.
U svom potpisu definiše modele uključene u transformaciju i ulaznu tačku transformacije *(main).*
Kao i klasa, operational transformacija može se instancirati zajedno sa svojim propertijima i operacijama.

```
transformation Uml2Rdbms(in uml:UML,out rdbms:RDBMS) {
    main() {....}
    ....
    ........................................
    .... helperi...
    .... operacije mapiranja...
}
```

Potpis: definiše naziv transformacije, početne i ciljne metamodele.

Ulazna tačka: Početna tačka transformacije koja počinje izvršenjem operacije u okviru main()

Elementi transformacije: Helperi i operacije mapiranja definišu logiku transformacije.

Operacije mapiranja

Operacija mapiranja mapira jedan ili više elemenata početnih modela u jedan ili više elemenata ciljnih modela.

Uvek jednosmerna.

Izvršava operacije kako bi kreirala elemente ciljnih modela.

Može pozivati druge operacije mapiranja.

Sa drugim operacijama mapiranja može uspostavljati odnose nasleđivanja.

Primer:

```
mapping Package::packageToSchema() : Schema
   when { self.name.startingWith() <> "_"}
{
   name := self.name;
   table := self.ownedElement->map class2table();
}
```

Pozivanje operacija mapiranja

```
mapping Class::class2table() : Table   when {self.isPersistent()}
{
  name := 't_' + self.name;
  column := self.attribute->map attr2Column());
  key := self.map class2key(result.column);
}
mapping Attribute::attr2Column() : Column {
  name:=self.name; type:=getSqlType(self.type);
}
```

Helperi

Helper je operacija vezana za tip podatka koja vraća određenu vrednost.

Može se vezivati i za primitivne tipove i za tipove modela.

Mogu se koristiti za navigaciju kroz izvorne modele.

Dve vrste helpera:

- Query (bez spoljnih efekata)
- Helper (sa spoljnim efektima, vrši se izmena ulaznih paametara)

Primeri:

```
query Association::isPersistent() : Boolean =
    (self.source.kind='persistent' and self.destination.kind='persistent');
..........
helper Package::computeCandidates(inout list:List) : List {
   if (self.nothingToAdd()) return list;
   list.append(self.retrieveCandidates());
   return list;
}
```

Kontrola toka

Compute

compute (v:T := initexp) { … self.getSomething() …};

While

self.myprop := **while**(v:T = initexp; v<>null) { … self.getSomething() …}

forEach

self.ownedElement->**forEach**(i|i.isKindOf(Actor)) { … }

Break

Continue

If-then-else

Kritike QVT – a

Ne postoji potpuna implementacija standarda

Samo Relations i Operational Mappings jezici

Ograničena upotreba

Vezan za XMI format

Obimna specifikacija

Prerana standardizacija

QVT alati i implementacije

QVT Operational Mappings

- Borland Together
- SmartQVT
- Eclipse M2M Operational QVT

QVT Core

- OptimalJ (prekinut dalji razvoj)

QVT Relations

- MediniQVT
- ModelMorf
- Eclipse M2M Declarative QVT