

**UNIVERZITET U BANJOJ LUCI  
ELEKTROTEHNIČKI FAKULTET**

**Prof. dr Dražen Brđanin**

**PROJEKTOVANJE SOFTVERA  
/uvod/**

**Banja Luka  
2024.**

# Dizajn (projektovanje) sistema



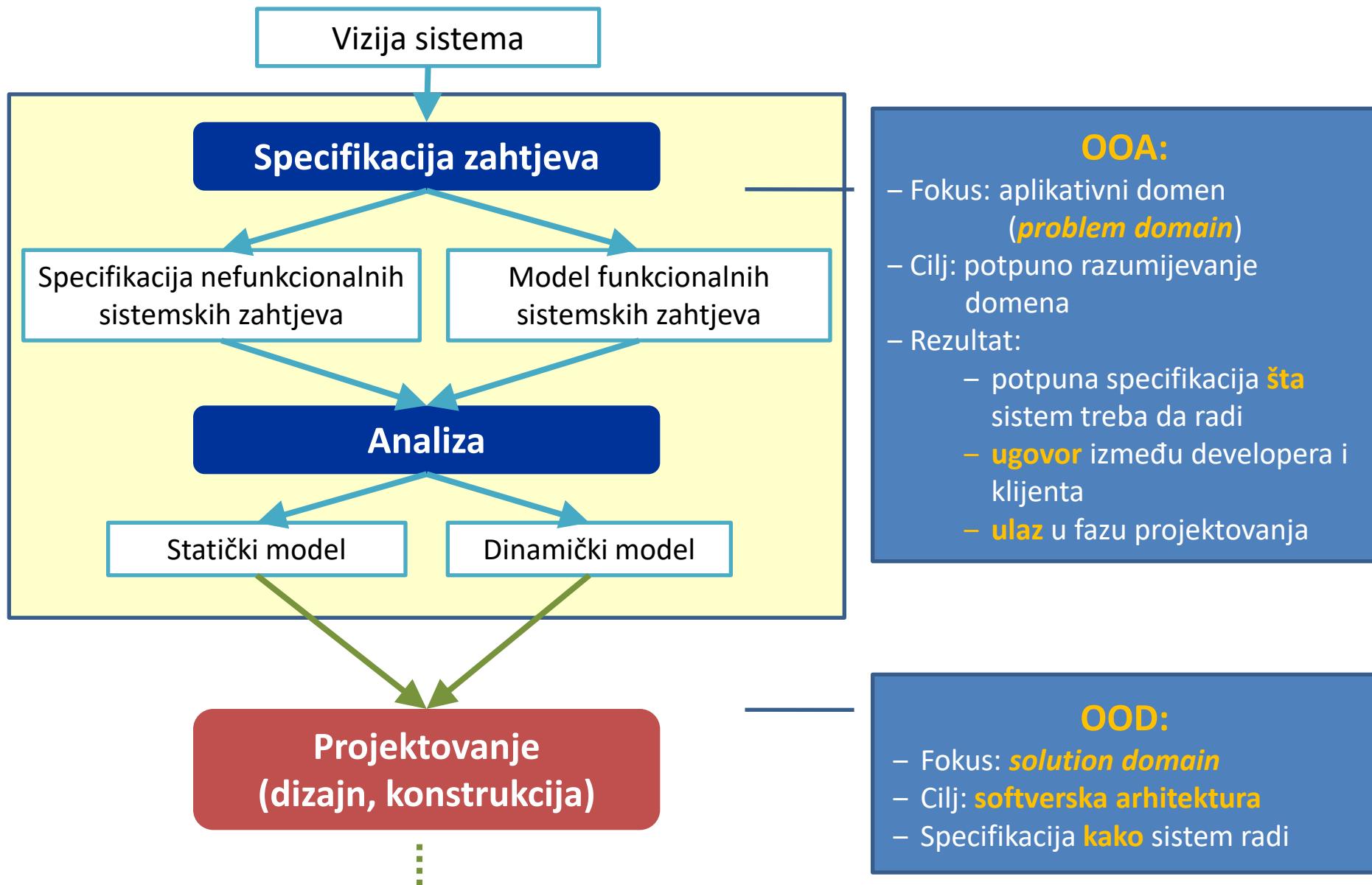
## Definicija:

*Projektovanje (dizajn/konstrukcija) je proces rješavanja problema čiji je cilj pronalaženje i specifikacija načina za realizaciju funkcionalnih sistemskih zahtjeva uz uvažavanje ograničenja i obezbjeđenje kvaliteta.*

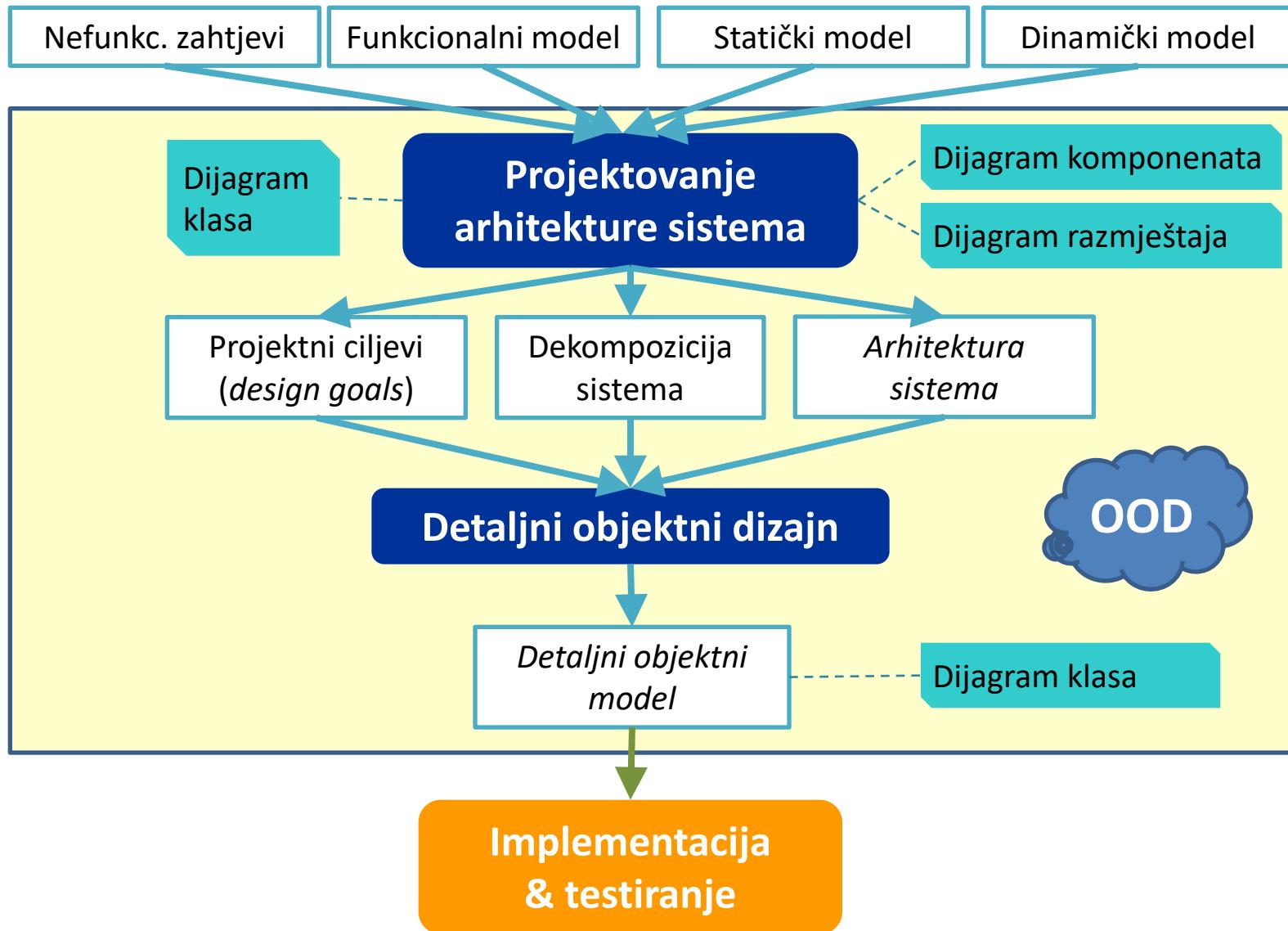
**Projektovanje nije jednostavno!**

- U poređenju sa analizom, dizajn je zahtjevniji i teži.
- Analiza zavisi samo od aplikativnog domena, a dizajn i od analize i od implementacije.
- Projektant mora da vodi računa o mapiranju sistemskih modela (koji reprezentuju aplikativni domen) na odgovarajući hardver – to nije egzaktna i potpuno poznata nauka – postoje heuristike, ali je njihovo vrijeme (polu)zastarijevanja veoma kratko (2-5 godina).
- Razvoj tehnologija je intenzivan (mainframe – prije 30 godina, klijent-server, web, mobilno računarstvo – danas)

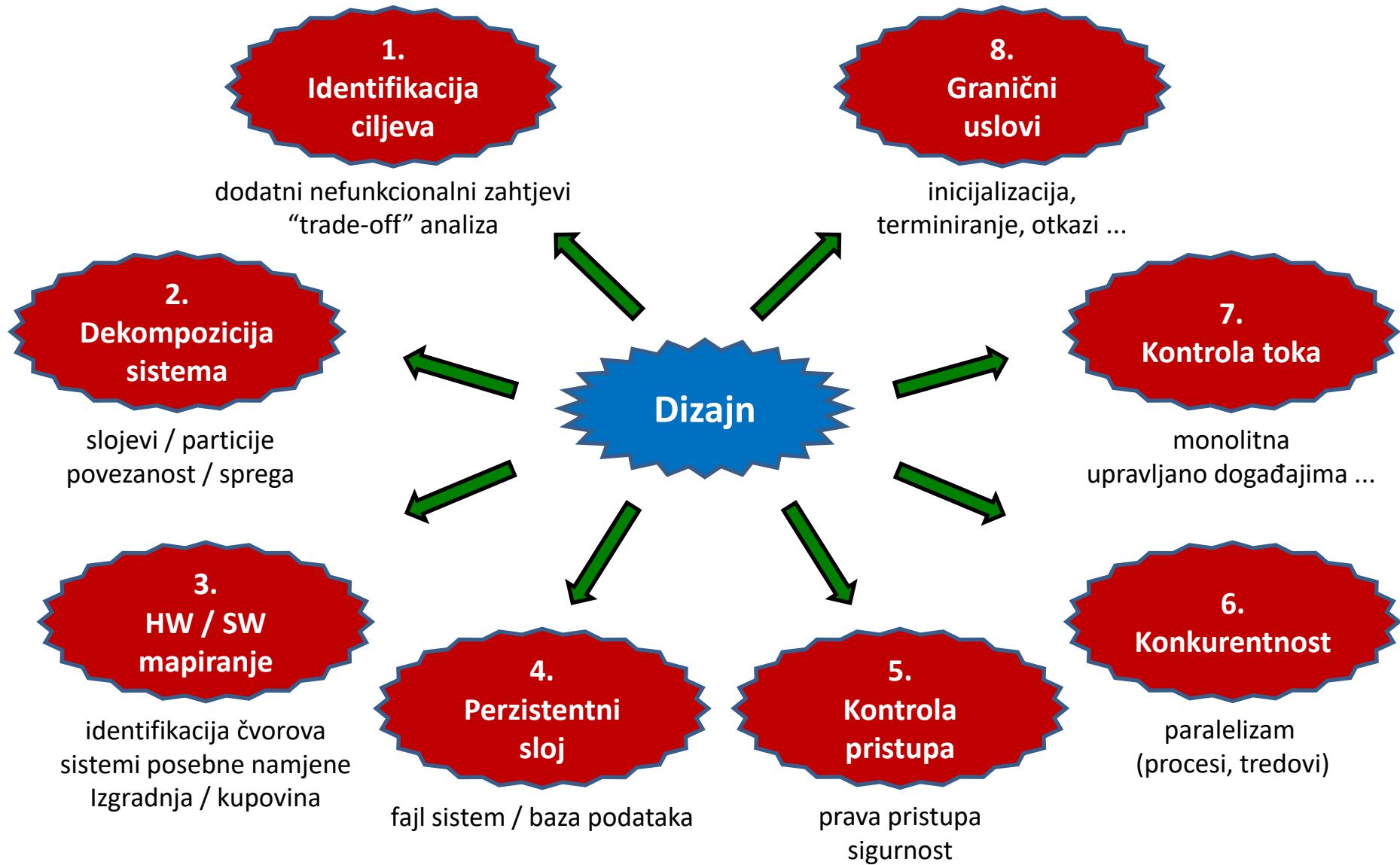
# Objektno-orientisana analiza



# Objektno-orientisano projektovanje



# 8 bitnih aktivnosti u projektovanju

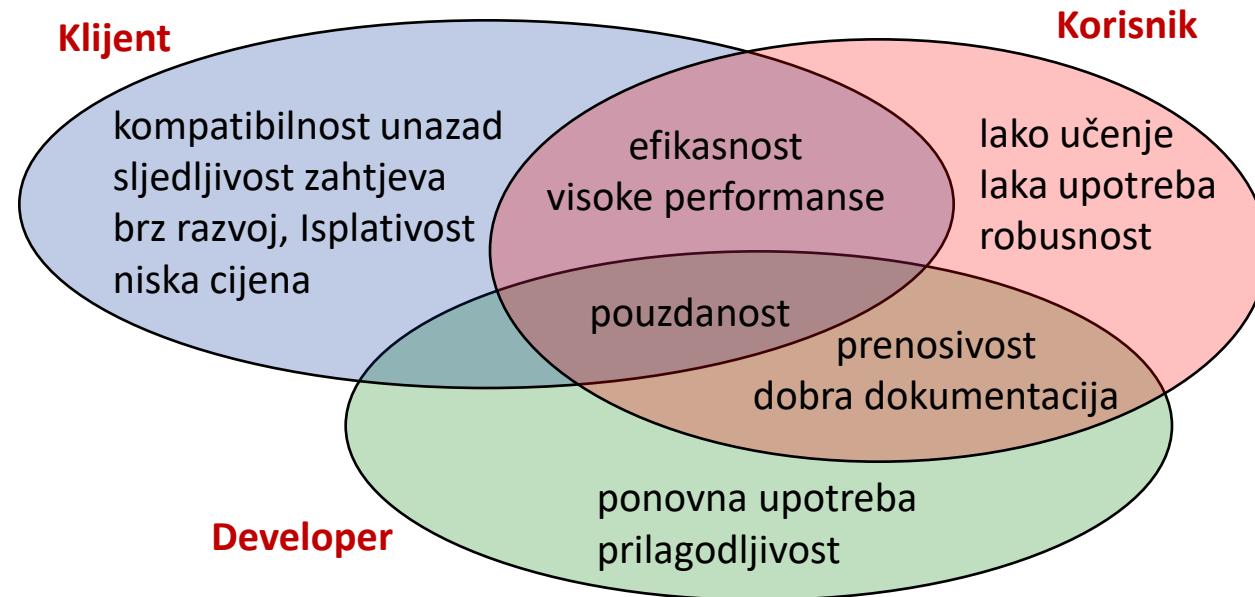


# 1. Projektni ciljevi

## Primjeri projektnih ciljeva:

- pouzdanost
- niska cijena
- visoke performanse
- ponovna upotreba
- efikasnost
- prenosivost
- kompatibilnost unazad
- isplativost
- robusnost
- sljedljivost zahtjeva
- prilagodljivost
- dobra dokumentacija
- dobro definisani interfejsi
- brz razvoj
- minimalan broj grešaka
- laka upotreba
- lako održavanje
- ...

## Zainteresovani (*stakeholderi*) imaju različite ciljeve:



## Trade-off analiza:

cijena ↔ robusnost

cijena ↔ reuse

:

**UNIVERZITET U BANJOJ LUCI  
ELEKTROTEHNIČKI FAKULTET**

**Prof. dr Dražen Brđanin**

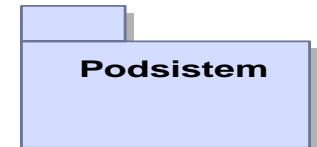
**PROJEKTOVANJE SOFTVERA  
/dekompozicija sistema/**

**Banja Luka  
2024.**

# 2. Dekompozicija sistema

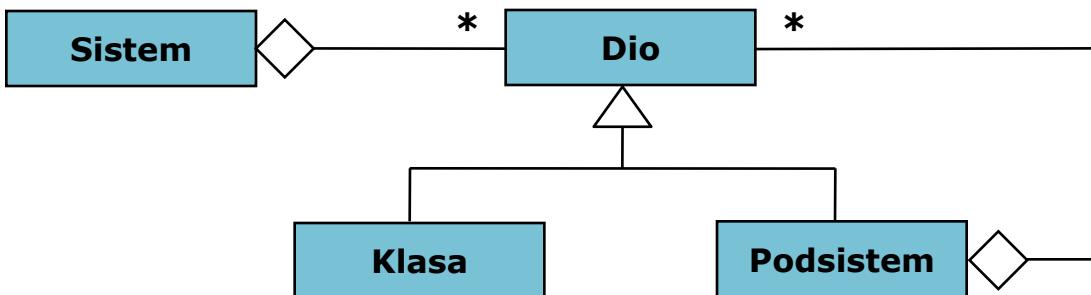
Divide et impera!

Dekompozicija sistema je postupak kojim se sistem reprezentuje kao kolekcija podsistema.



## Podsistemi

- **Podsistemi** je kolekcija blisko povezanih klasa, asocijacija, operacija, događaja i ograničenja.
- Podsistemi se u UML modeluju kao paketi.
- Podistem je **zamjenljivi dio sistema sa dobro definisanim interfejsima**, a koji inkapsulira stanje i ponašanje svih sadržanih klasa.
- Podistem tipično korespondira obimu posla kojim može biti dodijeljen jednom programeru ili razvojnom timu. Dekompozicijom sistema na (relativno nezavisne) podsisteme, više timova može istovremeno da razvija pojedine podsisteme uz minimalnu međusobnu komunikaciju.
- U slučaju kompleksnih sistema, dekompozicija može rekursivno da se primjenjuje – podsistemi mogu da se dekomponuju na jednostavnije podsisteme.



Podsistemi se različito realizuju u programskim jezicima:

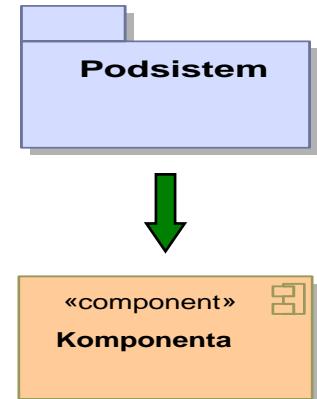
Java (paket)

C++ (ne podržava podsisteme)

# 2. Dekompozicija sistema

**Podsistem = komponenta ciljnog softverskog sistema**

- U kasnijim fazama, podsistemi se mapiraju u komponente sistema:
  - **logička komponenta**  
podsistem koji nema eksplicitni run-time ekvivalent
  - **fizička komponenta**  
podsistem koji ima eksplicitni run-time ekvivalent, npr. DB server



**Proces dekompozicije**

- **Iterativno-inkrementalni proces:**
  - iterativne revizije postojeće dekompozicije i inkrementalno adresiranje novog
  - **dijeljenje/spajanje i dodavanje/izbacivanje**
- **Inicijalna dekompozicija:**
  - zasnovana na funkcionalnom modelu
  - **slučaj upotrebe → podsistem** (dijelovi struktturnog modela koji pripadaju istom slučaju upotrebe pripadaju jednom podsistemu)
- **Naknadne transformacije:**
  - tehnike: **raslojavanje i particonisanje** (*layering/partitioning*)
  - arhitekturni stilovi (šabloni za dekompoziciju)
    - klijent-server, MVC, repozitorijum, 3-slojna, 4-slojna, SOA, ...
    - ciljna softverska arhitektura je konkretna instanca nekog arhitekturnog stila

# 2. Dekompozicija sistema (nastavak)

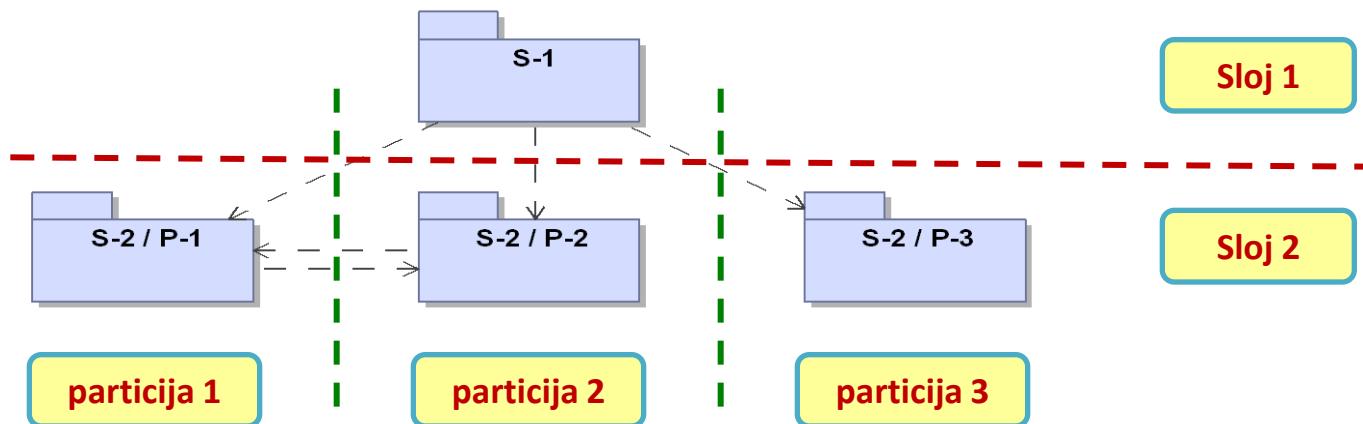
Tehnike za dekompoziciju (tehnike za smanjivanje stepena spregnutosti)

## Vertikalna dekompozicija (raslojavanje)

- **Sloj** je podsistem koji obezbjeđuje servis drugom podsistemu, uz sljedeća ograničenja:
  - sloj zavisi samo od servisa nižih slojeva,
  - sloj nema znanje o višim slojevima.

## Horizontalna dekompozicija (particionisanje)

- Sloj može horizontalno da se izdijeli na više manjih (nezavisnih) podsistema – **particije**
  - particije obezbjeđuju servise drugim particijama istog sloja
  - particije se često nazivaju **slabo spregnuti podsistemi** (*weakly coupled*)



# 2. Dekompozicija sistema (nastavak)

## Veze između podsistema

### Vertikalne veze (veze između slojeva)

#### – “compile time” zavisnosti

- sloj A **zavisi od** sloja B
- npr. *import, build*

#### – “run time” zavisnosti

- sloj A **poziva** sloj B  
(npr. web klijent poziva web server)
- slojevi nisu procesorski čvorovi  
(SW/HW mapiranje je predmet naknadne analize)

### Horizontalne veze (veze između particija)

#### – “Peer-to-Peer” veze

- particije su ravnopravne i imaju uzajamno znanje jedna od drugoj
- particija A poziva particiju B i particija B poziva particiju A

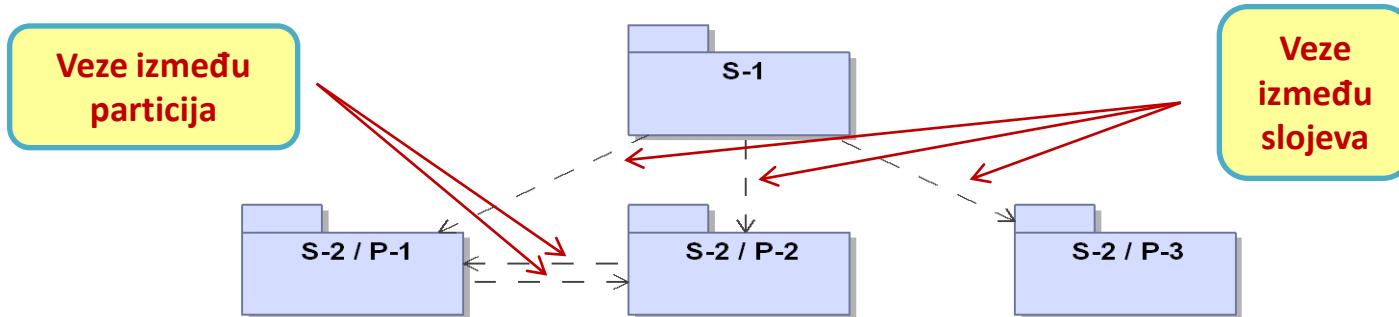
## Neke heuristike za dekompoziciju

#### – Ukupan broj podsistema: **≤10**

- više podsistema povećava stepen sprege u sistemu  
(više servisa)

#### – Ukupan broj slojeva: **≤5**

- **dobar dizajn: 3 sloja**



# 2. Dekompozicija sistema (nastavak)

## Osnovni cilj dekompozicije: REDUKCIJA SLOŽENOSTI

### Mjere složenosti

#### Povezanost (*coherence*)

- mjera zavisnosti između klasa u podsistemu

→ **visoka povezanost (high coherence)**

- klase u podsistemu imaju slične uloge, izvršavaju slične zadatke, i uzajamno su povezane većim brojem asocijacija
- postiže se ako je većina interakcija unutar podistema, a ne preko granica podistema
- **Uvijek pitanje:**

Da li dati podistem stalno poziva neki servis drugog podistema?

Da: Objediniti podisteme ako je moguće!

#### niska povezanost (low coherence)

- velik broj pomoćnih i uslužnih klasa u podsistemu, koje su povezane malim brojem asocijacija

#### Sprega (*coupling*)

- mjera zavisnosti između podistema
- **visoka sprega (high coupling)**

- promjena u jednom podistemima ima velik uticaj na drugi podistem

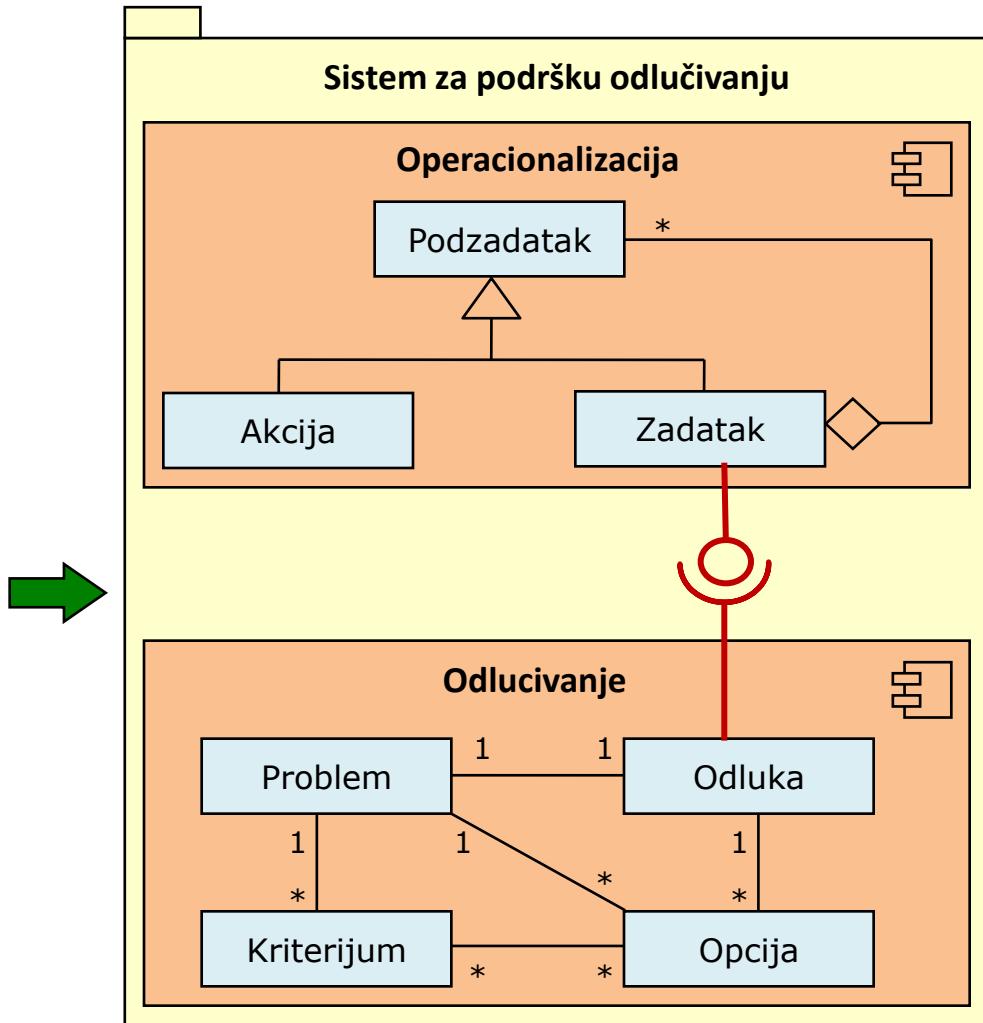
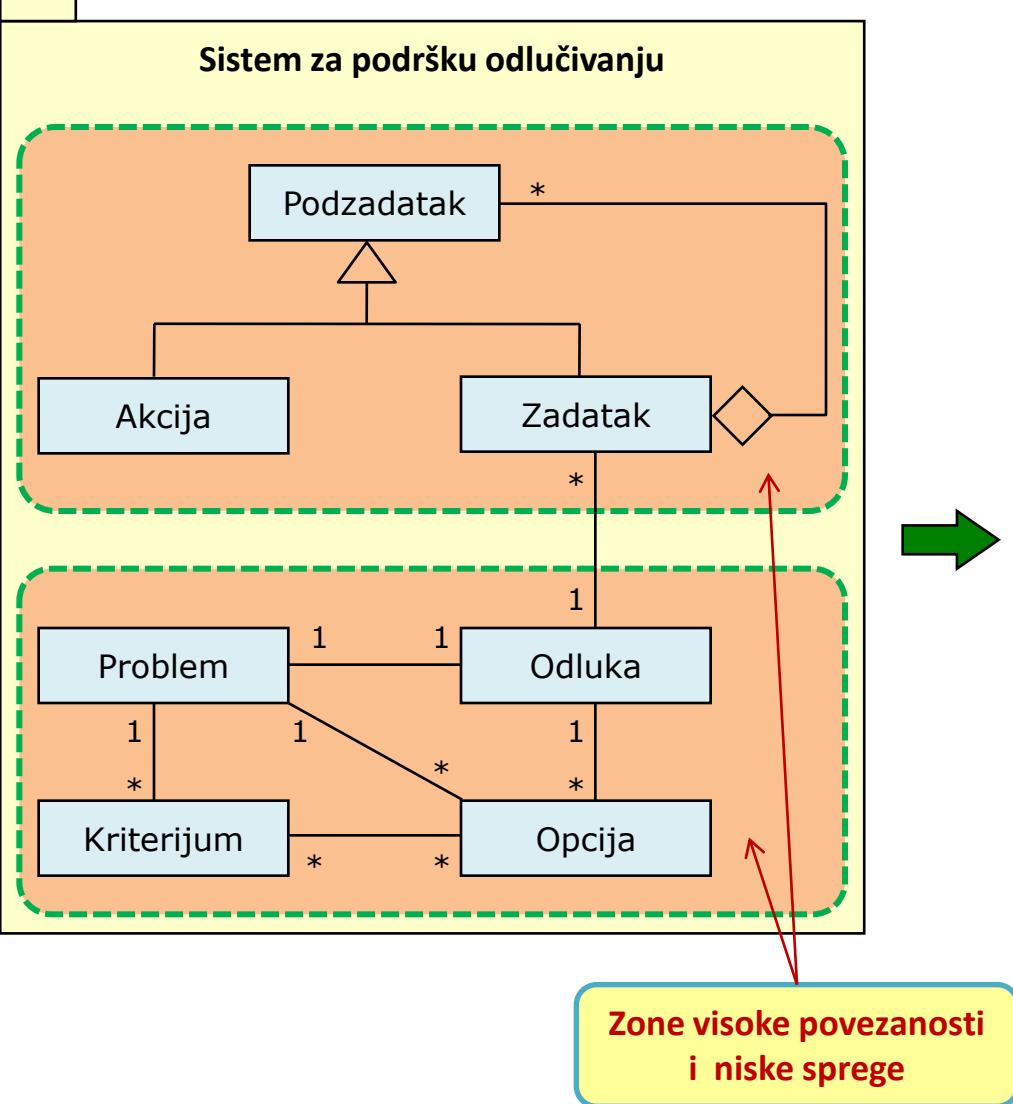
→ **niska sprega (low coupling)**

- promjena u jednom podistem nema uticaj na drugi podistem
- postiže se ako pozivajuća klasa ne mora da zna ništa o implementaciji pozivane klase (princip skrivanja informacija)

Dobar dizajn

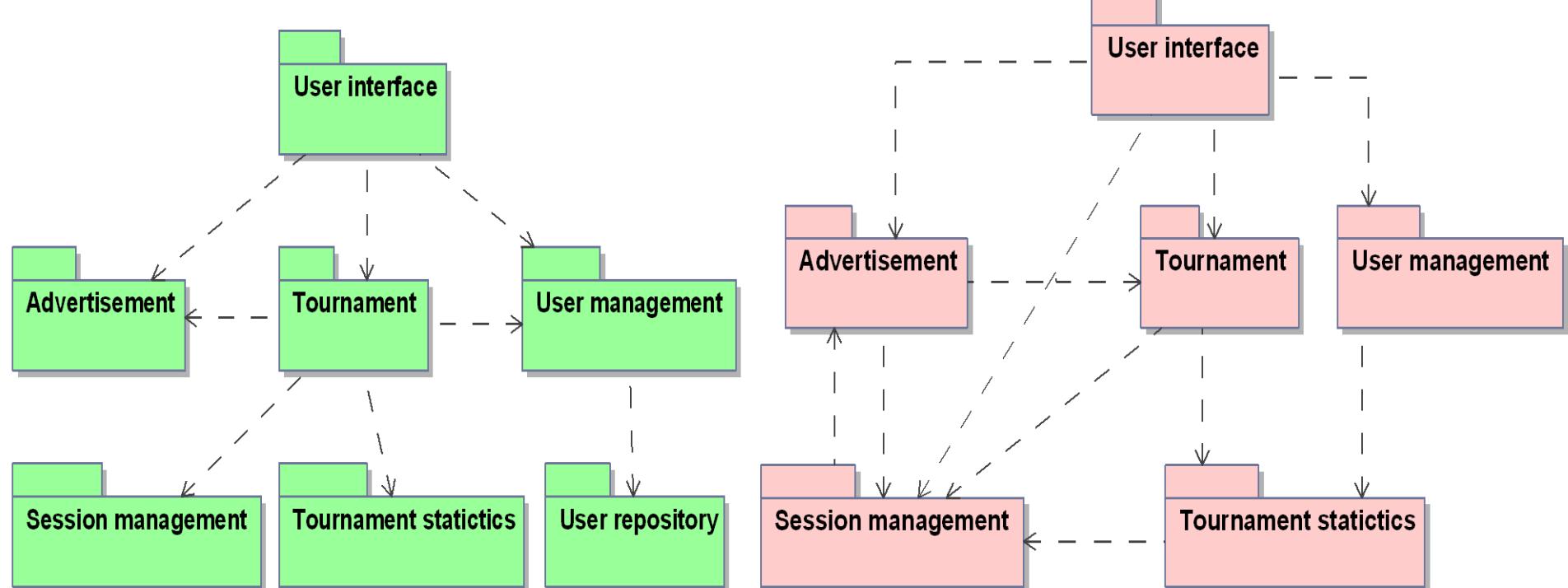
# 2. Dekompozicija sistema (nastavak)

Primjer (Sistem za podršku odlučivanju)



# 2. Dekompozicija sistema (nastavak)

## Primjeri dekompozicije



- 3-slojna dekompozicija
- slojevi nizu uzajamno zavisni
- niska sprega podistema



- visoka sprega podistema
- slojevi uzajamno zavisni
- ...

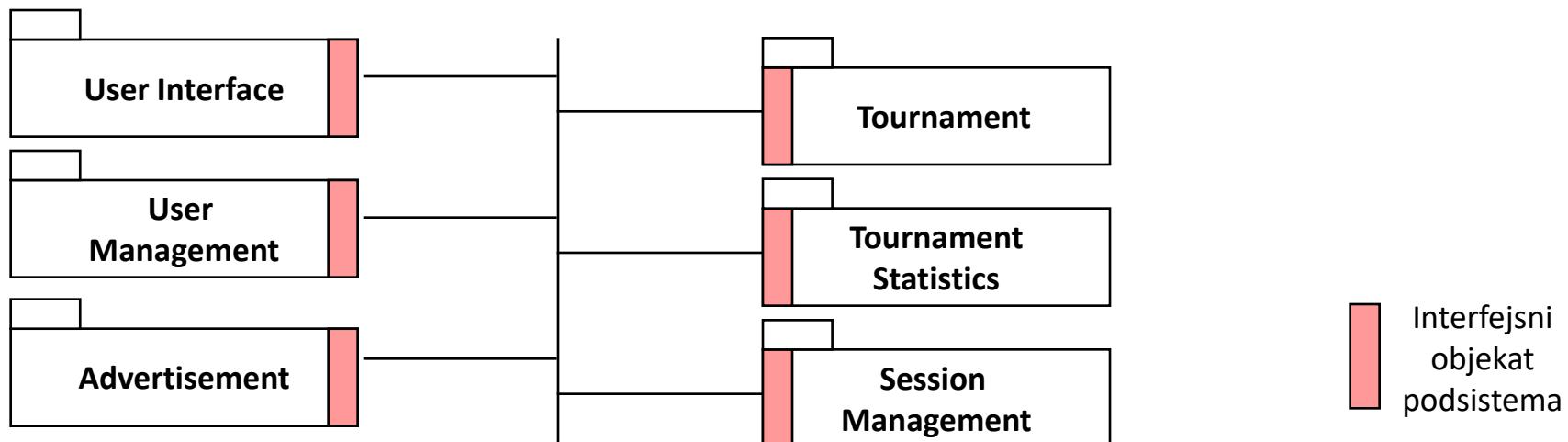
# 2. Dekompozicija sistema (nastavak)

## Servisi

- **Servis** je kolekcija operacija kojom raspolaže neki podsistem.
- Servisi se definišu tokom projektovanja sistema, a rafiniraju kao interfejsi podistema tokom detaljnog objektnog dizajna.
- Heuristika: slučajevi upotrebe često figurišu kao servisi podistema

## Interfejs podistema

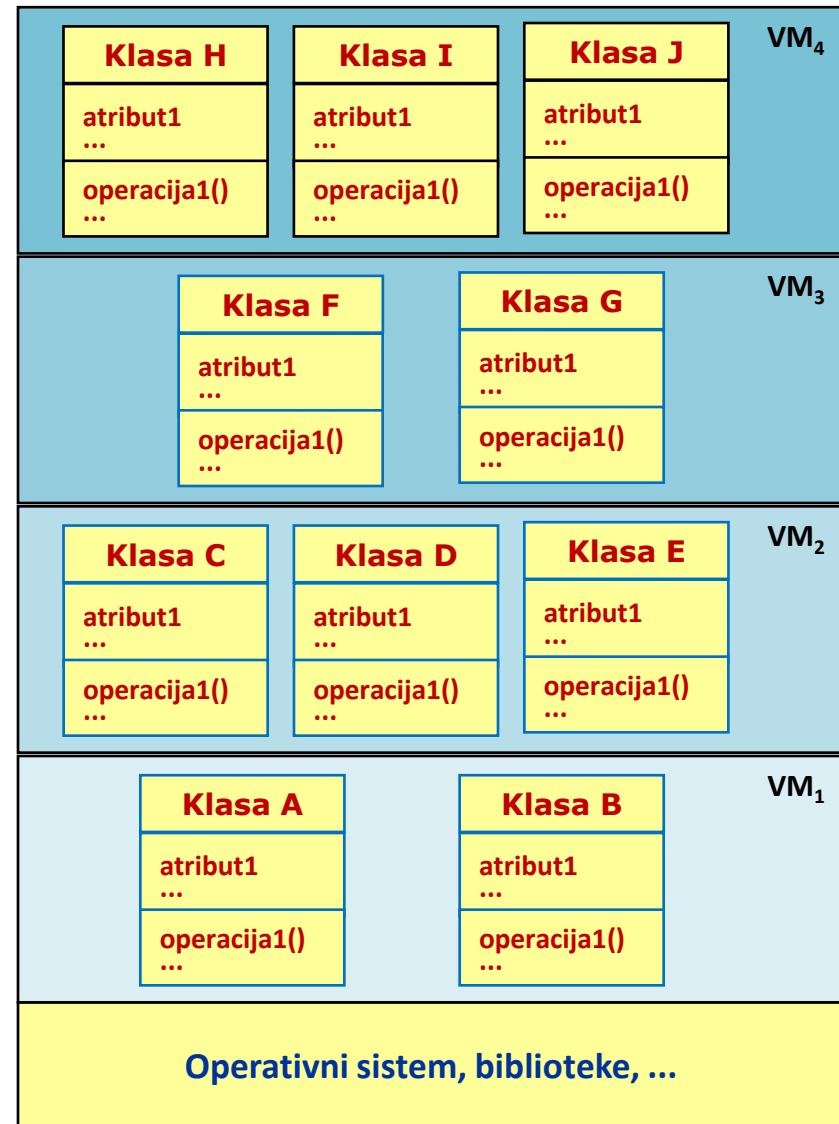
- **Interfejs podistema** je skup operacija kojima se specifikuje interakcija i tok informacija između podistema (ne unutar podistema).
- **Dobar dizajn:** svaki podistem ima jedan **interfejsni objekat** koji reprezentuje sve servise kojima raspolaže dati podistem (**projektni obrazac FASADA**)
- **Dobar dizajn:** sistem kao skup interfejsnih objekata



# 2. Dekompozicija sistema (nastavak)

## Virtuelna mašina (“nivo apstrakcije”)

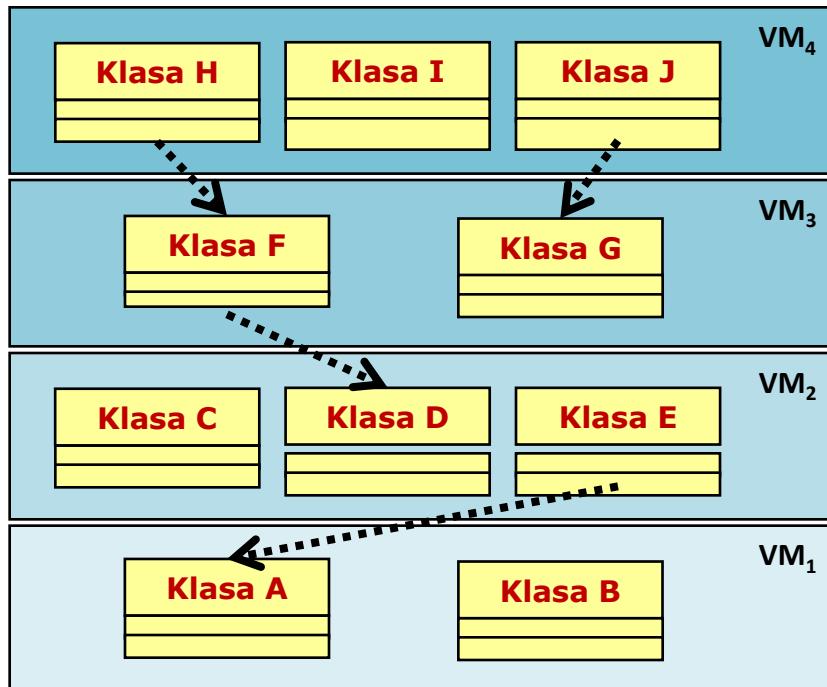
- **Virtuelna mašina** je podsistem koji je sa drugim virtuelnim mašinama (iznad i ispod) povezan vezama tipa “**obezbjeđuje servis za**”.
- Slojevi i virtuelne mašine često se poistovjećuju.
- Virtuelna mašina je kolekcija klasa – **modul** koji koristi servise virtuelnih mašina ispod i **obezbjeđuje servise virtuelnim mašinama iznad**.
- Virtuelna mašina je ključni koncept u apstrakciji sistema (“out of date” za distribuirane sisteme, ali veoma pogodan za jednoprocesorske arhitekture)
- **Sistem kao hijerarhija virtuelnih mašina**, svaka virtuelna mašina koristi jezičke primitive koje obezbjeđuju niže virtuelne mašine



# 2. Dekompozicija sistema (nastavak)

## Zatvorena arhitektura (*opaque layering*)

- Svaka VM može da poziva samo operacije iz VM neposredno ispod.

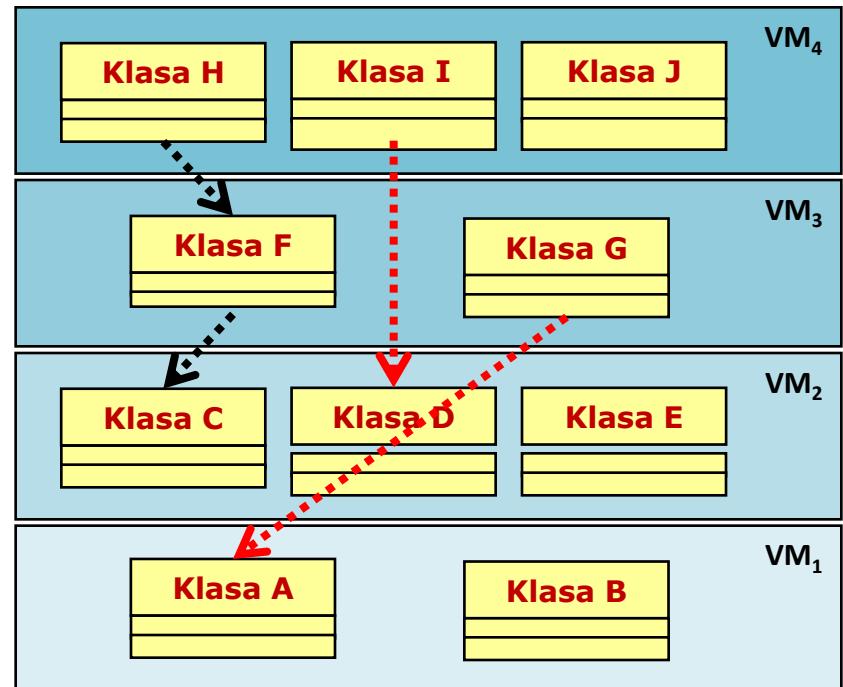


## Projektni ciljevi:

- fleksibilnost
- lako održavanje
- prenosivost

## Otvorena arhitektura (*transparent layering*)

- Svaka VM može da poziva operacije iz bilo koje VM ispod.



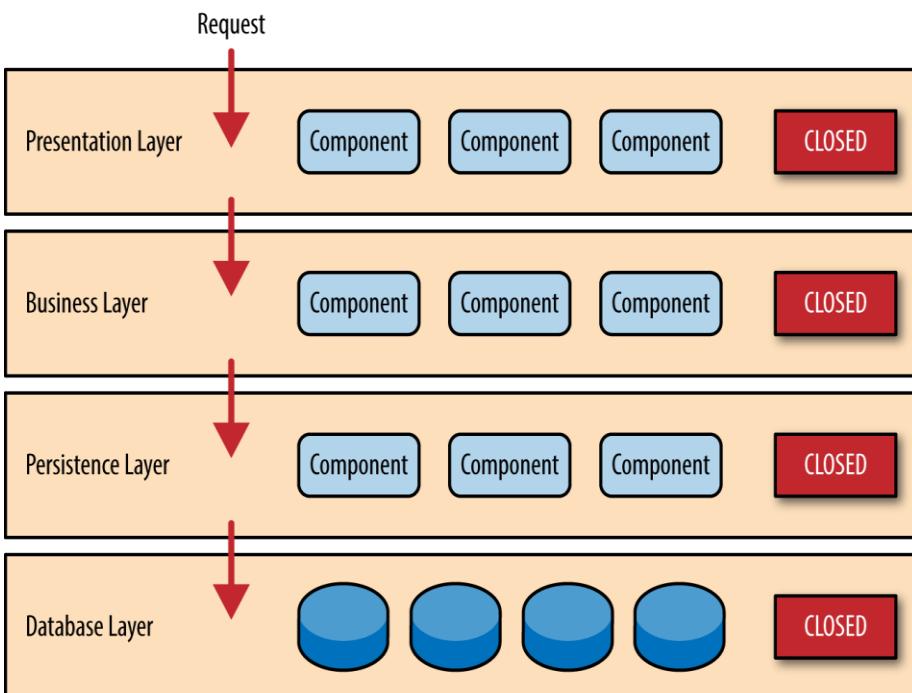
## Projektni ciljevi:

- *run-time* efikasnost

# 2. Dekompozicija sistema (nastavak)

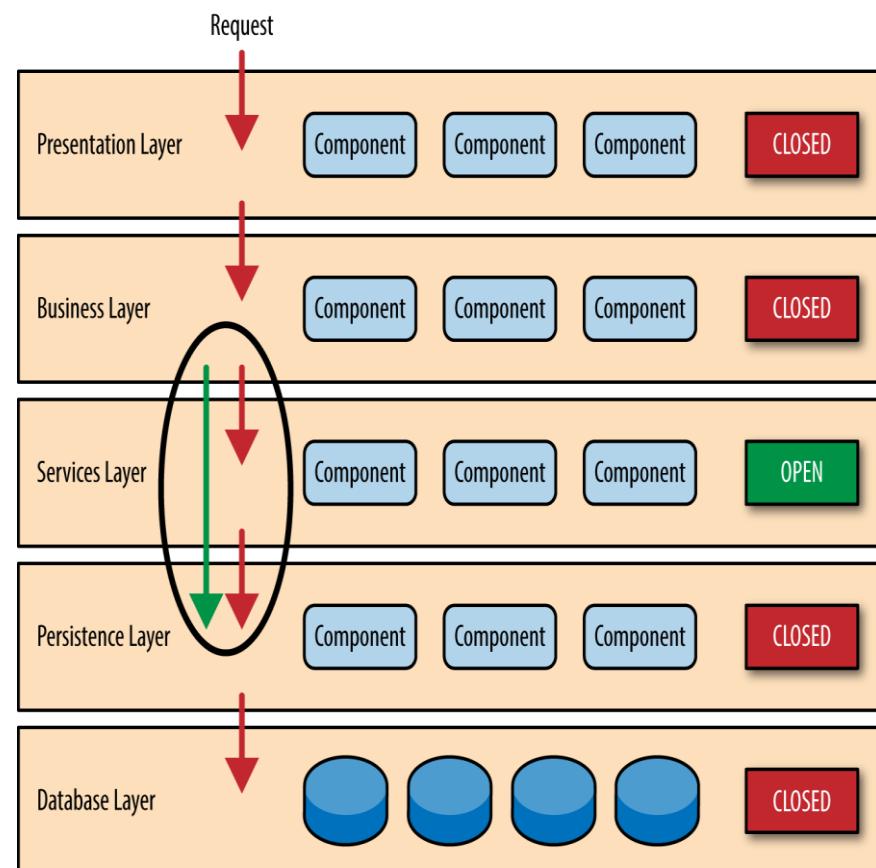
## Zatvorena arhitektura (*opaque layering*)

- Svaka VM može da poziva samo operacije iz VM neposredno ispod.



## Otvorena arhitektura (*transparent layering*)

- Svaka VM može da poziva operacije iz bilo koje VM ispod.



# 2. Dekompozicija sistema (nastavak)

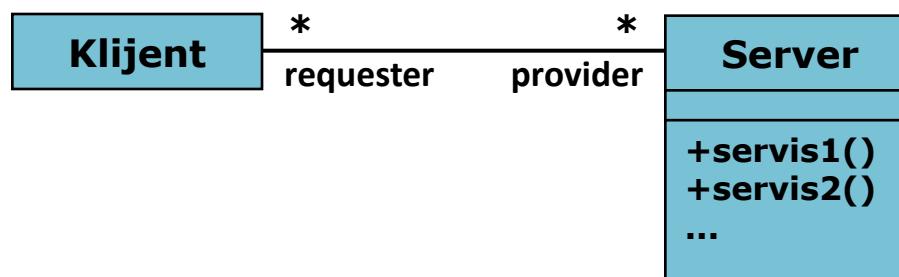
## Tipični arhitekturni stilovi (šabloni za dekompoziciju)

### Klijent/Server arhitekturni stil

- Jedan ili više **servera** obezbjeđuje servise **klijentima**
- **Klijent poziva server, server izvršava odgovarajući servis i vraća rezultat**
  - Klijenti znaju interfejs servera (bez tog da nema ni poziva servisa)
  - Server ne mora da zna interfejs klijenta
- U opštem slučaju, odziv je neposredan
- Krajnji korisnici imaju interakciju samo sa klijentom

### Klijent/Server arhitekture

- Česta upotreba u sistemima sa bazama podataka
  - Front-end: korisnička aplikacija (klijent)
  - Back-end: manipulacija bazom (server)
- Funkcije klijenta:
  - unos podataka (prilagođeni UI interfejs)
  - *front-end* obrada podataka
- Funkcije servera:
  - centralizovano upravljanje podacima
  - integritet i konzistentnost podataka
  - sigurnost
  - ...



# 2. Dekompozicija sistema (nastavak)

## Tipični arhitekturni stilovi

### Projektni ciljevi u klijent/server arhitekturi

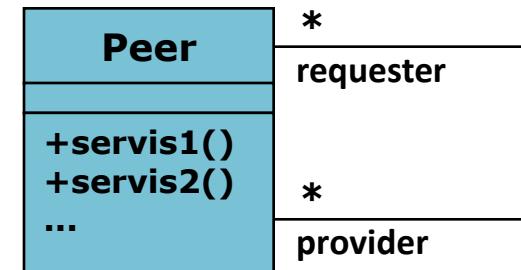
- Prenosivost (portabilnost) –** Server ima mogućnost izvršavanja na različitim operativnim sistemima i u različitim mrežnim okruženjima
- Slojevitost i transparentnost –** Server može da ima distribuiranu arhitekturu, ali ga klijenti vide kao jedinstven “logički” servis
- Visoke performanse –** Klijenti optimizovani za intenzivnu interakciju sa korisnikom  
Server optimizovan za CPU i *storage* procesiranje
- Skalabilnost –** Server ima mogućnost posluživanja velikog broja klijenata
- Fleksibilnost –** Mogućnost implementacije klijenta na različitim terminalnim uređajima (desktop, laptop, PDA, mobile, ...)
- Pouzdanost –** Server ima ugrađene mehanizme koji obezbjeđuju konzistentnost, integritet, transakcije, ....

# 2. Dekompozicija sistema (nastavak)

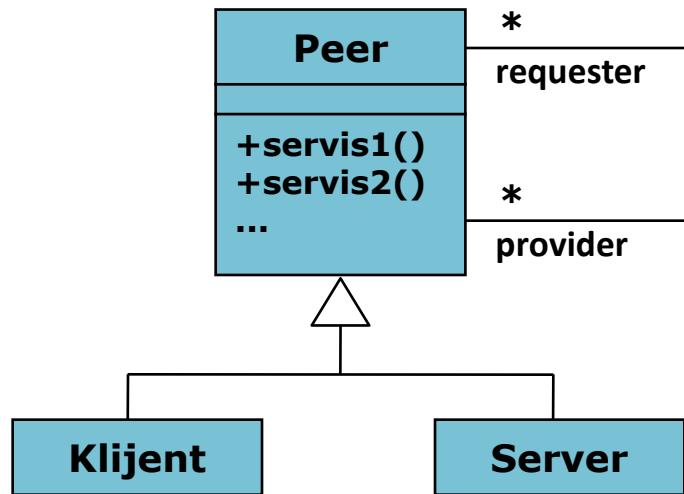
## Tipični arhitekturni stilovi

### Peer-to-Peer arhitekturni stil

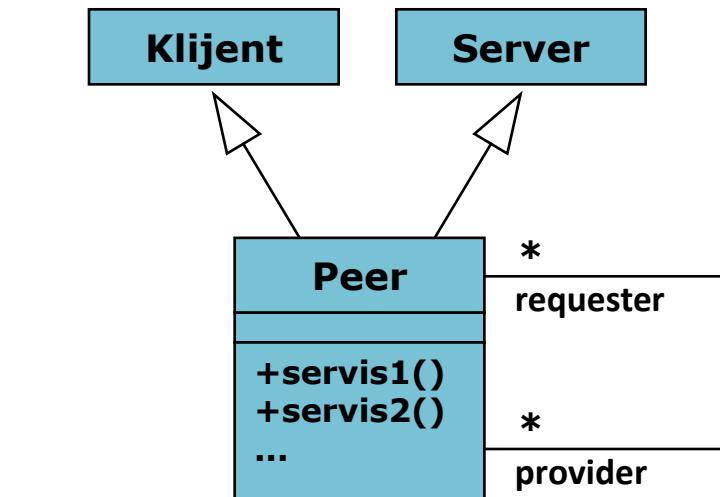
- **peer** = vršnjak, parnjak
- generalizacija klijent/server arhitekturnog stila
- **Klijenti mogu da budu serveri, a serveri mogu da budu klijenti**



### Projektne peer-to-peer alternative?



**✗**  
Peer može da bude klijent ili server



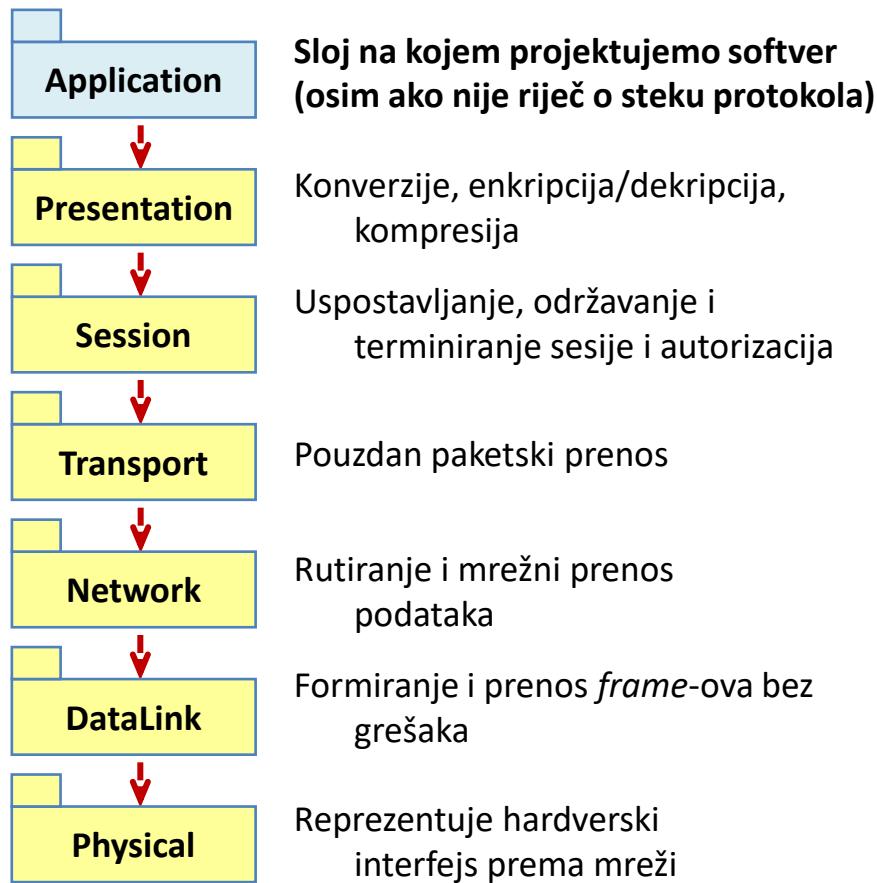
**✓**  
Peer može da bude i klijent i server

# 2. Dekompozicija sistema (nastavak)

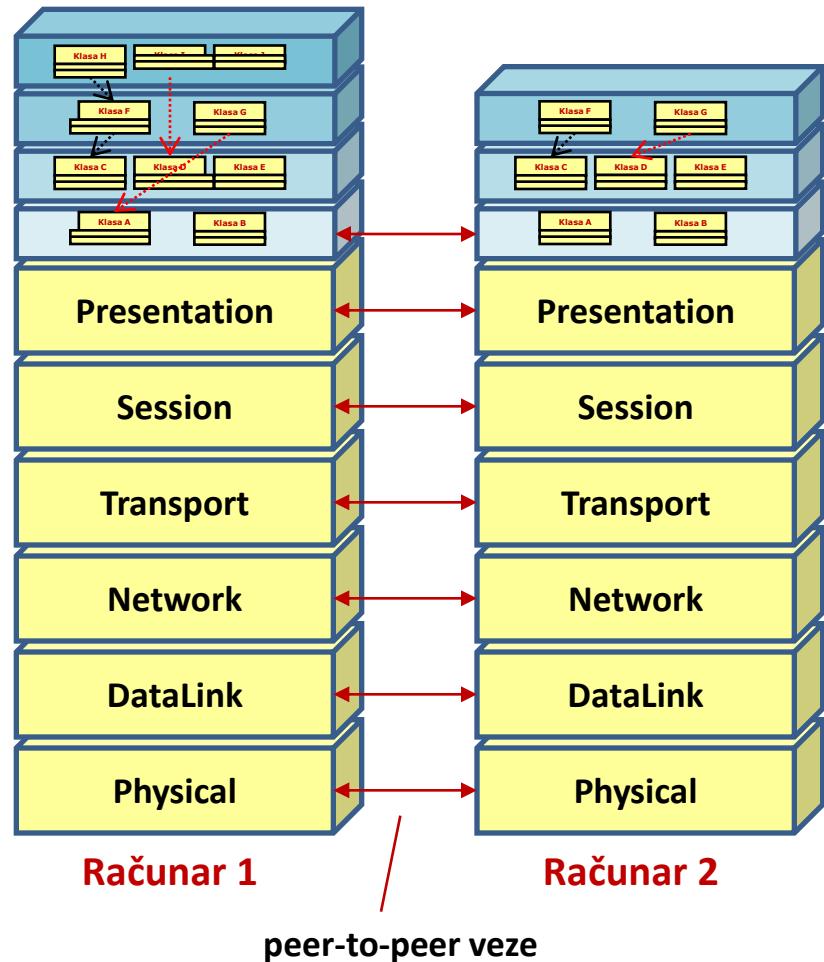
## Tipični arhitekturni stilovi

### Primjer peer-to-peer arhitekture

- **OSI (Open System Interconnection)**
  - 7-slojni referentni komunikacioni model



OSI je primjer zatvorene arhitekture

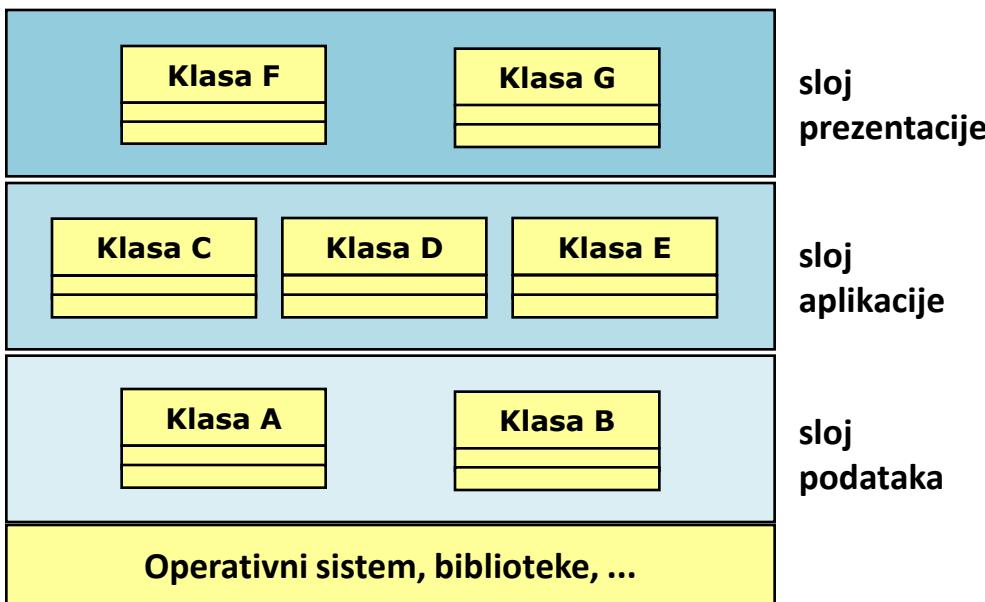


# 2. Dekompozicija sistema (nastavak)

## Tipični arhitekturni stilovi

### 3-slojni (3-Layer) arhitekturni stil

- Arhitekturni stil u kojem je sistem strukturisan na tri hijerarhijska podsistema (sloja)
- Arhitekturni stil kojeg čine tri virtualne mašine (Dijkstra, 1965):
  - **sloj prezentacije**: korisnički interfejs / klijent
  - **sloj aplikacije**: *middleware / business logic*
  - **sloj podataka**: baza podataka



### 3-slojna (3-Tier) arhitektura

- Softverska arhitektura koja podrazumijeva distribuciju tri sloja na tri odvojena hardverska čvora

Termini **Layer** i **Tier** često se koriste bez razlike  
**Layer = tip** (npr. klasa, podsistem)  
**Tier = instanca** (npr. objekat, čvor)

3-slojni arhitekturni stil tipično se koristi u web programiranju:

1. **Web Browser** implementira korisnički interfejs
2. **Web Server** servisira zahtjeve web browsera
3. **DBMS** upravlja podacima

# 2. Dekompozicija sistema (nastavak)

## Tipični arhitekturni stilovi

### 4-slojni (4-Layer) arhitekturni stil

- Arhitekturni stil u kojem je sistem strukturisan na četiri hijerarhijska podsistema (sloja)
- Npr:

- **Prezentacioni sloj**

korisnički interfejs  
(JSF + managed beans)

- **Sloj aplikativne logike:**

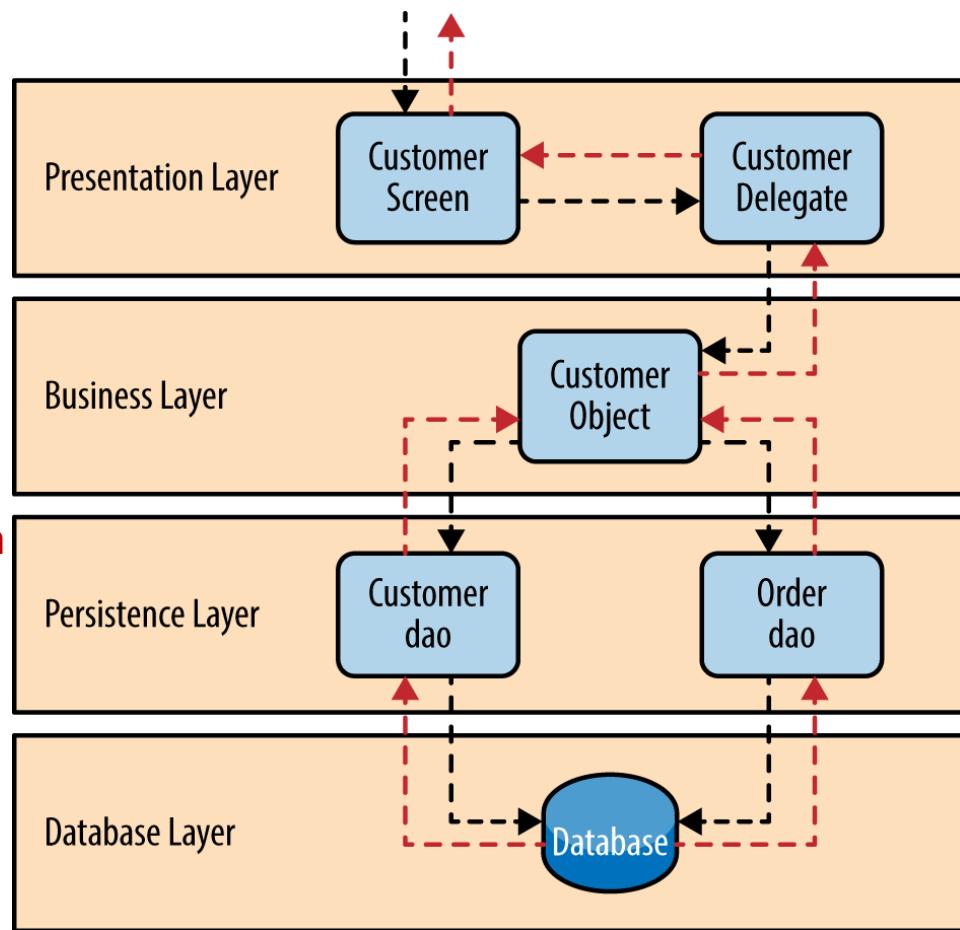
kontroleri za poslovnu logiku  
(local Spring bean / remote EJB3 bean)

- **Sloj pristupa perzistentnim objektima**

upravljanje perzistentnim objektima  
(ORM mapping)

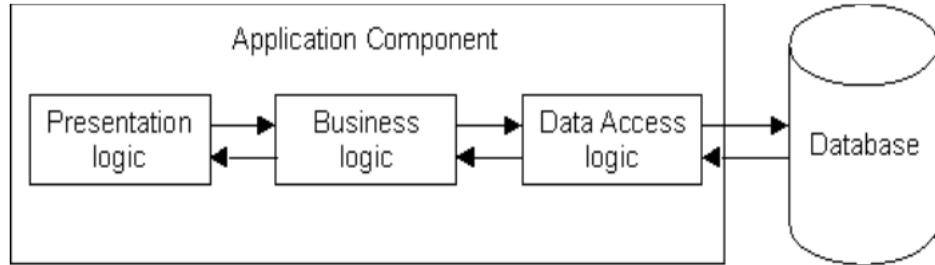
- **Sloj baze podataka**

DBMS + baza podataka



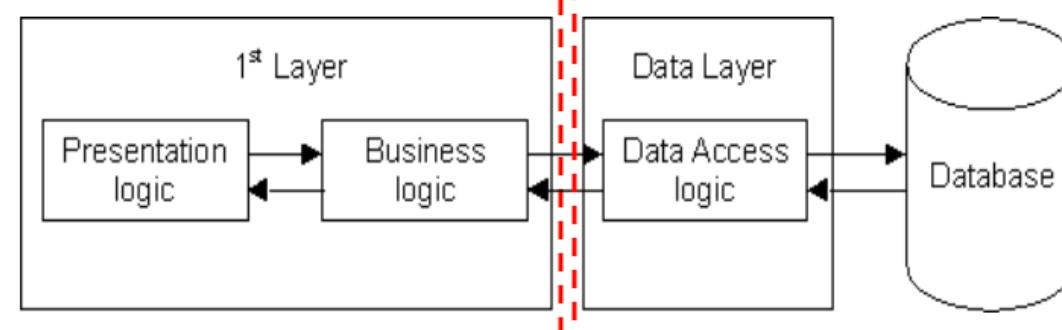
# 2. Dekompozicija sistema (nastavak)

## Realizacija slojeva – različite arhitekture i različiti fizički razmještaji



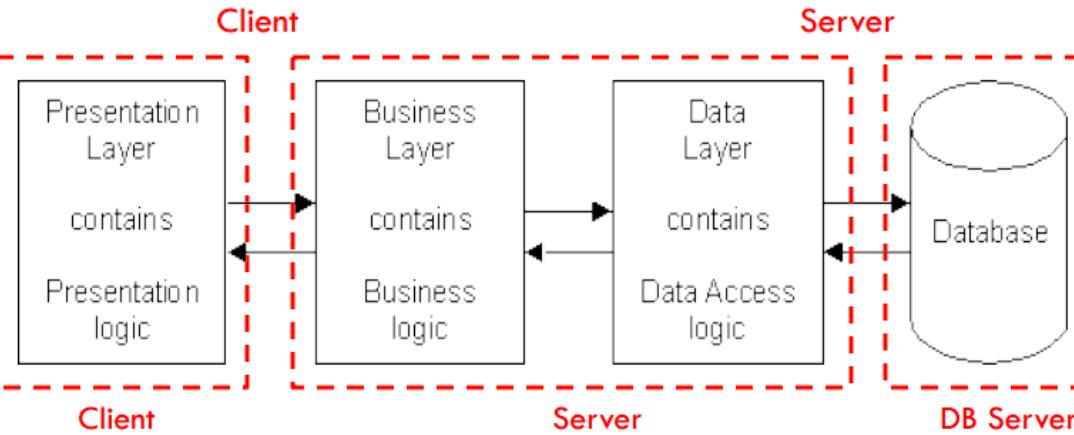
### 1-slojna (1-Tier) arhitektura

Svi slojevi na istom čvoru i jako spregnuti  
Otežana skalabilnost – sve na jednom procesoru  
Otežana portabilnost – nova implementacija?  
Otežano održavanje – slojevi su integrисани



### 2-slojna (2-Tier) arhitektura

Baza podataka i upravljanje podacima na serveru  
Olakšan prelazak na drugi DBMS/bazu podataka  
Prezentacioni i aplikativni sloj jako spregnuti



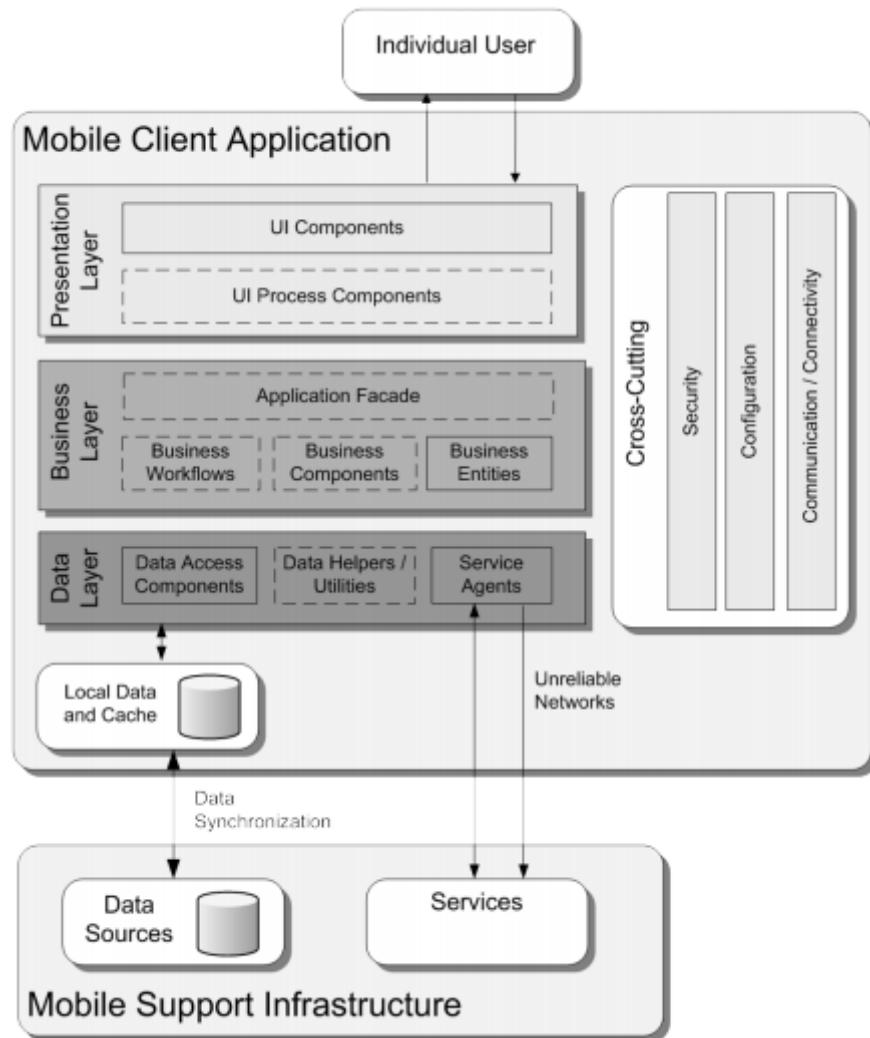
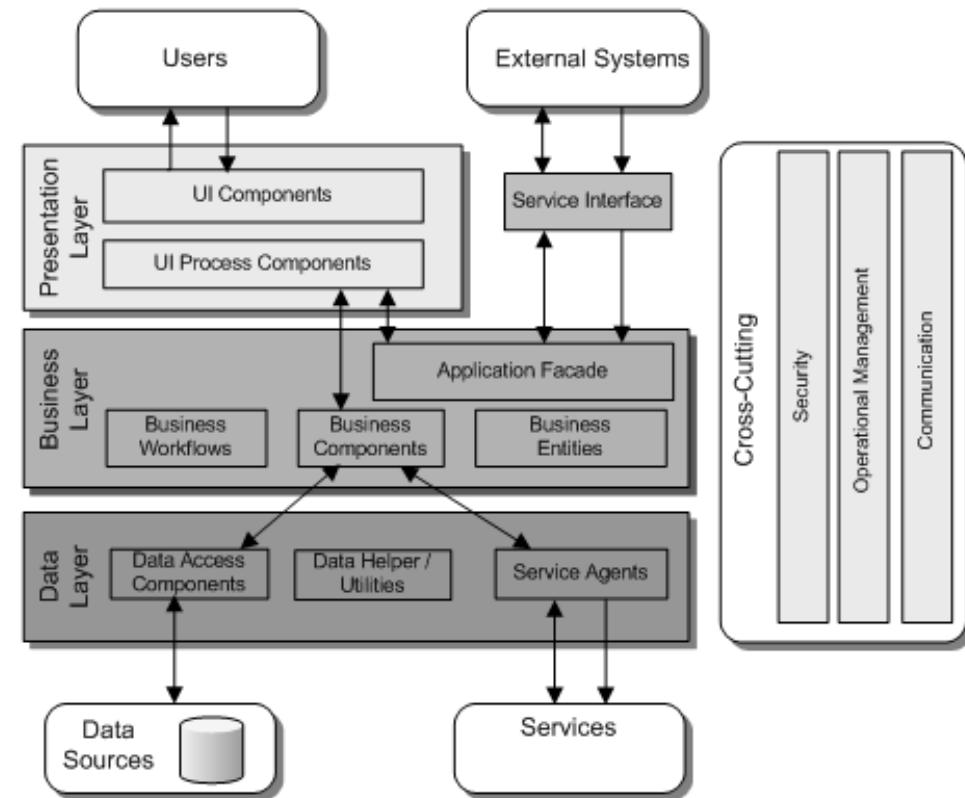
### 3-slojna (3-Tier) arhitektura

Svi slojevi su slabo spregnuti  
Svaki sloj može biti na posebnom čvoru  
Olakšan razvoj, testiranje, održavanje, reuse

# 2. Dekompozicija sistema (nastavak)

## Tipični arhitekturni stilovi

### Primjeri višeslojnih arhitekturnih stilova



## 2. Dekompozicija sistema (nastavak)

### Prednosti višeslojnog stila (visoka kohezija – niska sprega)

- Jednostavnost razvoja –** Veoma popularan arhitekturni stil u industriji  
Razvoj po slojevima – specijalizacija znanja (front-end, back-end, perzistentni sloj)
- Jednostavnost testiranja –** Svaka komponenta pripada specifičnom sloju pa je komponente u drugim slojevima lako zamijeniti korespondentnim stabovima (zamjenskim testnim komponentama) i slojeve testirati nezavisno
- Jednostavnost održavanja –** Slojevi su nezavisni i slabo spregnuti pa je lakše raditi izmjene na pojedinim komponentama i nezavisno ih mijenjati alternativnim
- Višestruka upotreba (reuse) –** Niži slojevi su nezavisni od viših i mogu biti višestruko korišteni

### Mane višeslojnog stila

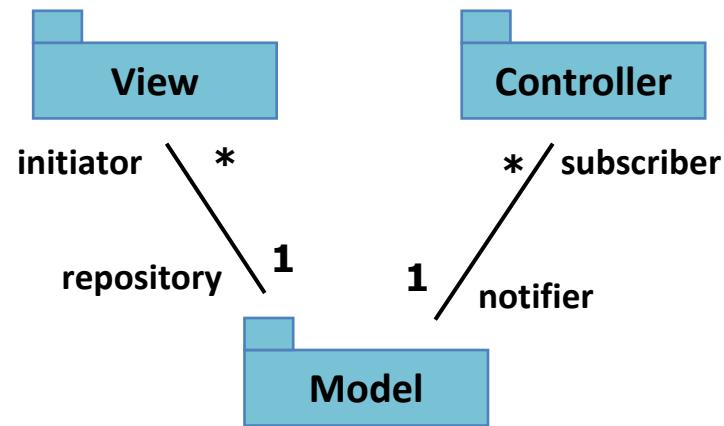
- Zahtjevniji razmještaj –** Veći broj komponenata na različitim čvorovima
- Performanse –** Veći broj slojeva smanjuje efikasnost
- Skalabilnost –** Svaki sloj za sebe je monolitan i nije ga lako skalirati  
Horizontalno – sloj treba višestruko replicirati na više čvorova  
Vertikalno – slojevi na različitim fizičkim čvorovima

# 2. Dekompozicija sistema (nastavak)

## Tipični arhitekturni stilovi

### MVC (Model-View-Controller) arhitekturni stil

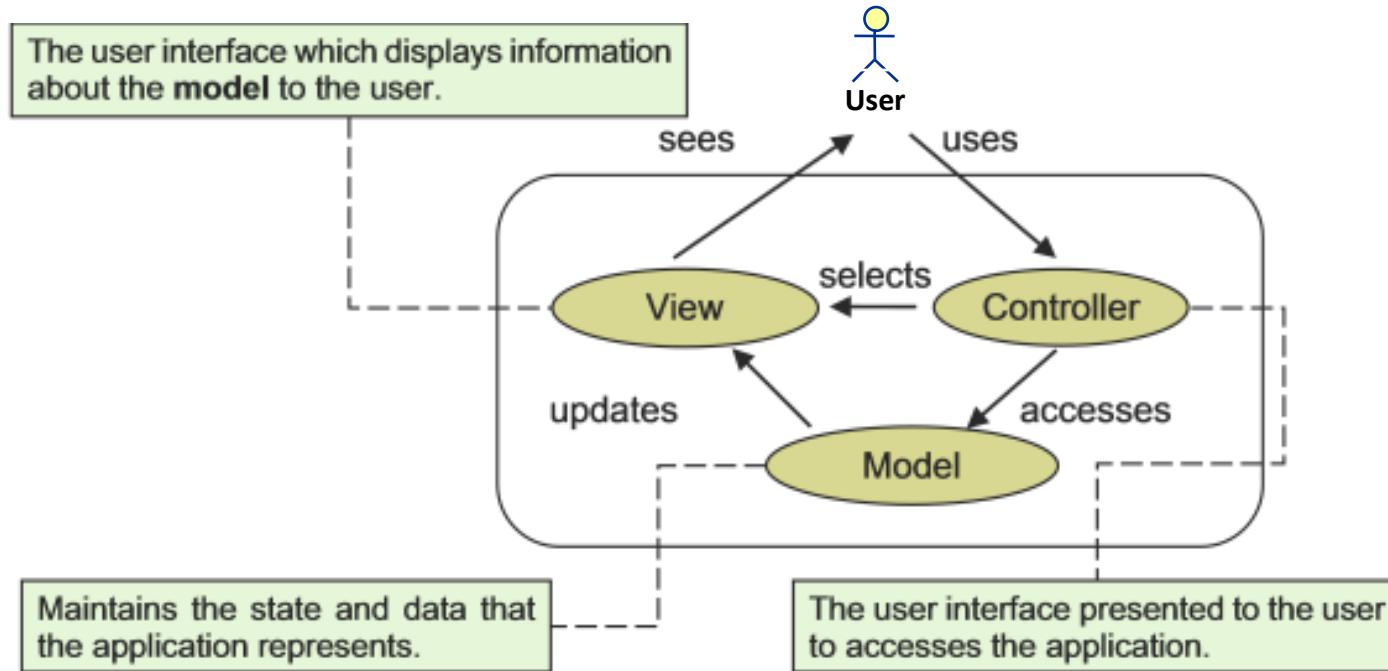
- MVC arhitekturni stil klasificuje podsisteme u tri kategorije:
  - **model** : reprezentacija i pristup podacima (domenskim objektima)
  - **view**: prezentacija podataka korisniku
  - **controller**: aplikativna logika / odgovoran za sekvencu interakcija sistema i korisnika i prosljeđivanje promjena modela prema pogledu
- Motivacija za MVC:
  - Interfejs sistema mijenja se mnogo češće nego aplikativna logika
  - Aplikativna logika mijenja se mnogo češće nego domenski objekti



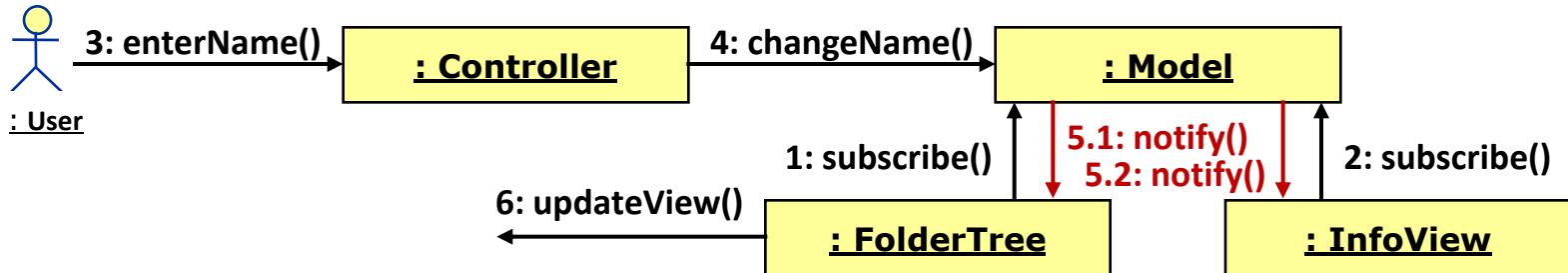
# 2. Dekompozicija sistema (nastavak)

## Tipični arhitekturni stilovi

### MVC – interakcija objekata



### Primjer interakcije:

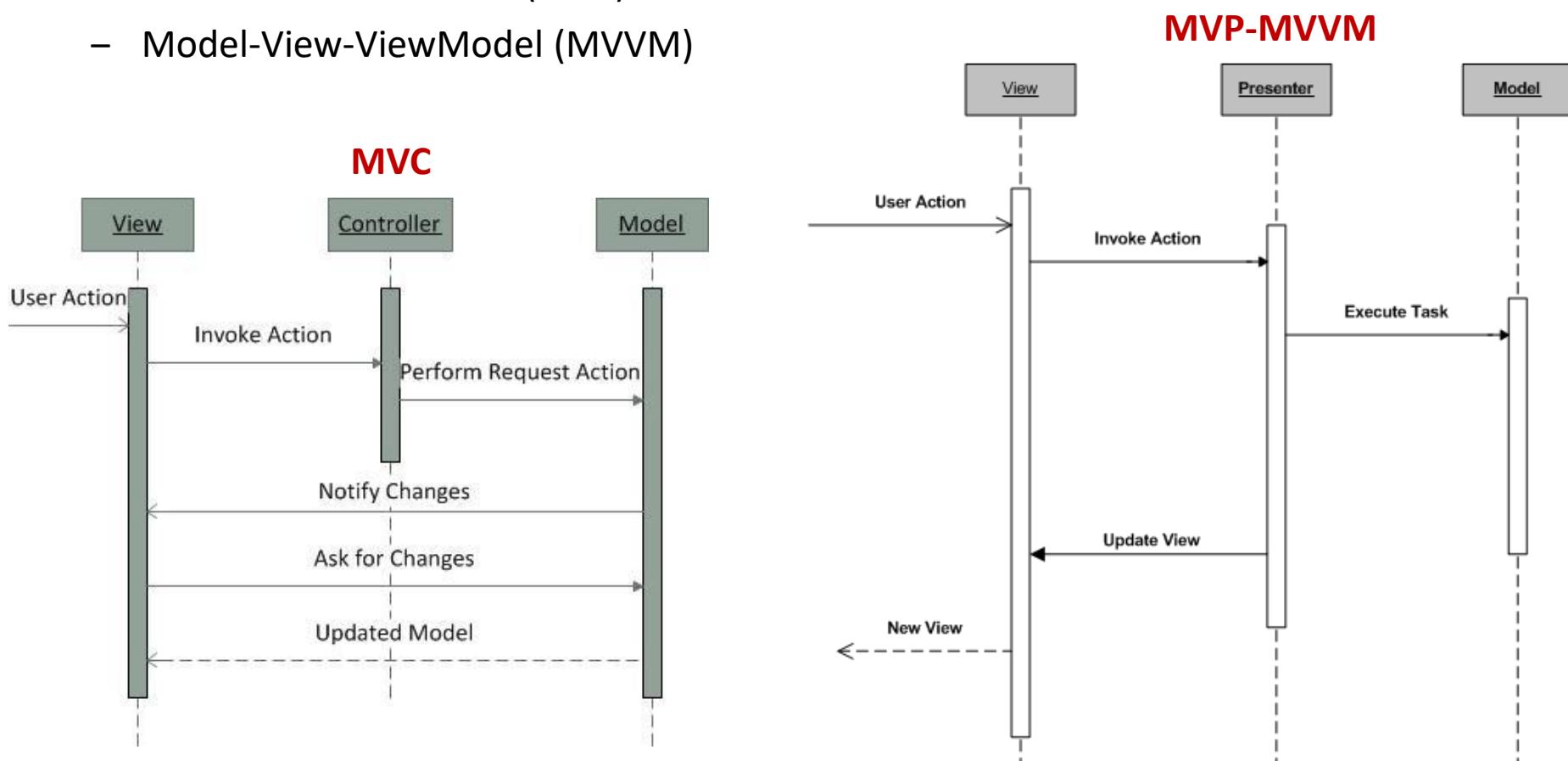


# 2. Dekompozicija sistema (nastavak)

## Tipični arhitekturni stilovi

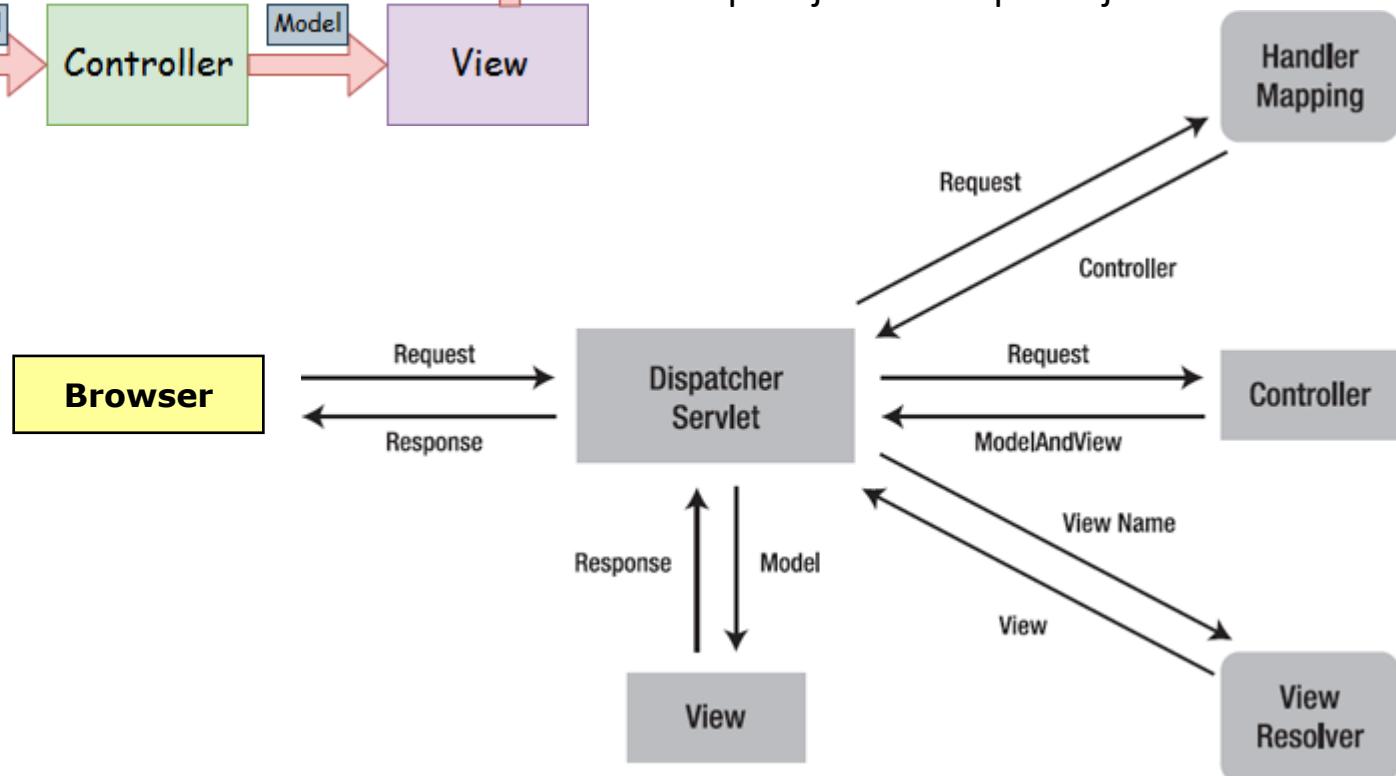
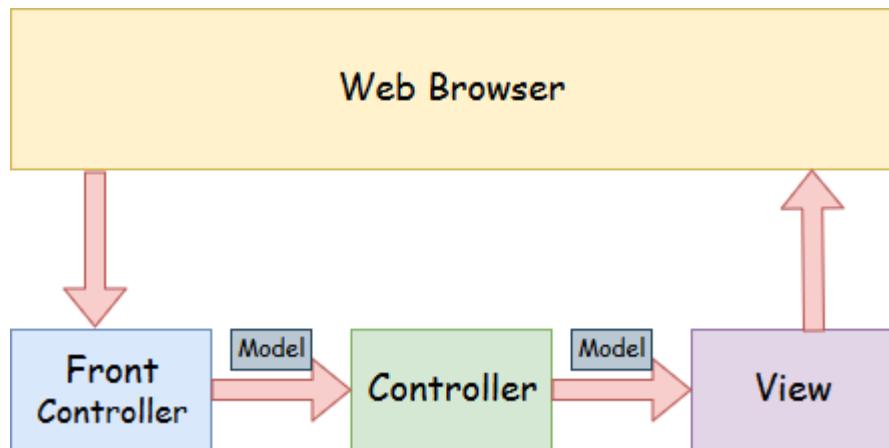
**MVC varijante** – sprega modela i pogleda nije direktna, nego ide preko kontrolera

- Model-View-Adapter (MVA) ili Mediated MVC ili Model-Mediator-View (MMV)
- Model-View-Presenter (MVP)
- Model-View-ViewModel (MVVM)



# 2. Dekompozicija sistema (nastavak)

## MVC – primjene (Spring)



**Model** – sadrži sve podatke ( pojedinačni objekti ili kolekcije)

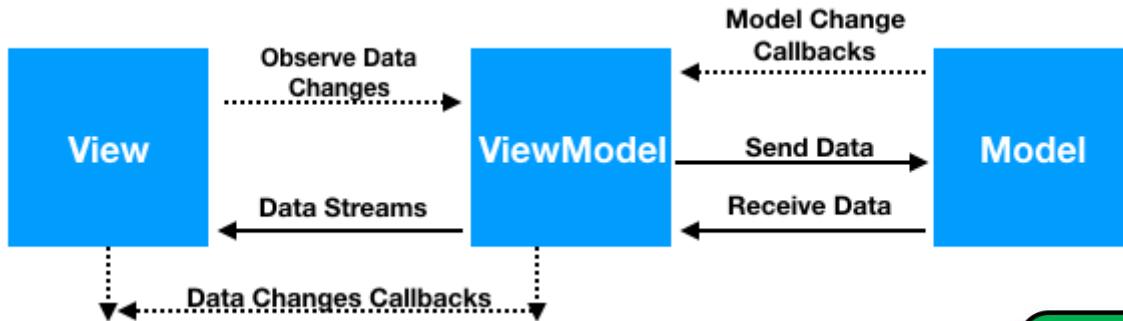
**Controller** – implementira poslovnu logiku

**View** – reprezentuje korisnički interfejs

**Front Controller** - DispatcherServlet klasa upravlja tokom aplikacije

# 2. Dekompozicija sistema (nastavak)

## MVVM – primjene (Andriod app)

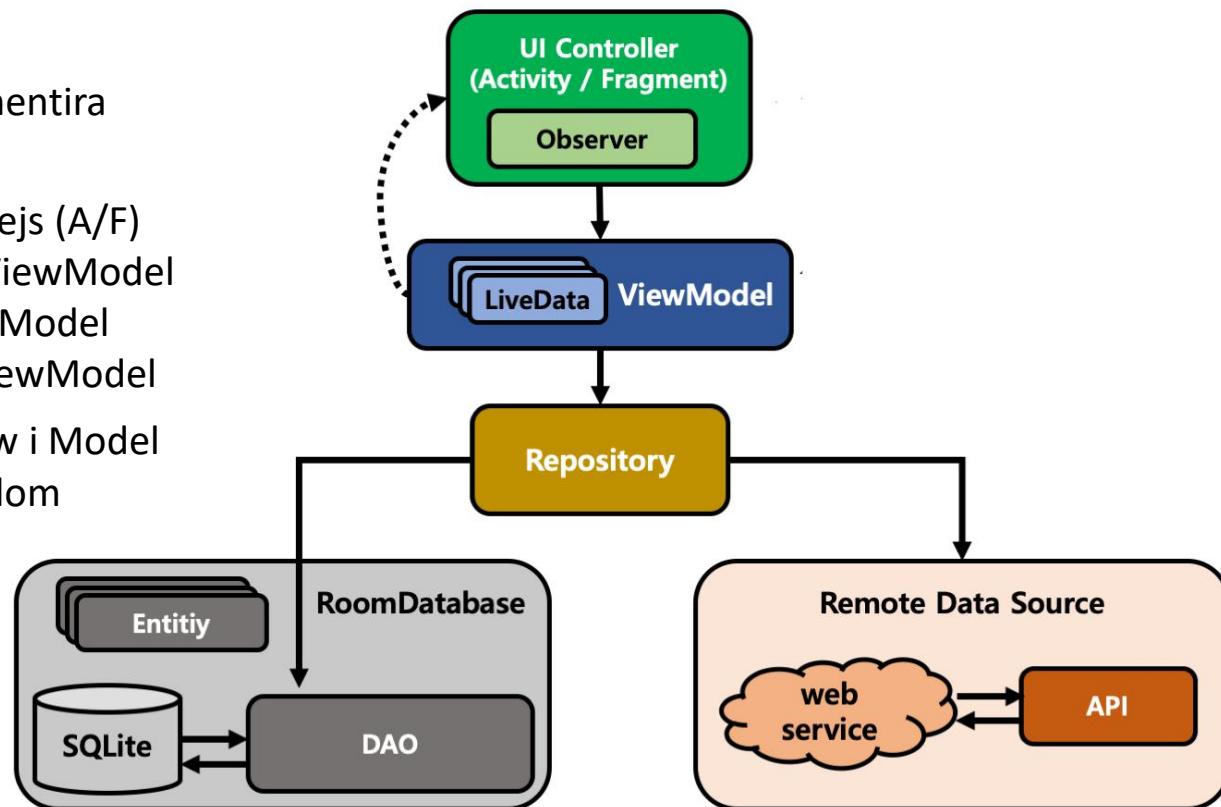


**Model** – sadrži sve podatke i implementira poslovnu logiku Android aplikacije

**View** – reprezentuje korisnički interfejs (A/F) proslijedjuje akcije korisnika prema ViewModel Odgovor ne dobija direktno od ViewModel

Ima opserver koji se prijavljuje na ViewModel

**ViewModel** – medjusloj između View i Model ima sinhronu komunikaciju sa Modelom



# **2. Dekompozicija sistema** (nastavak)

## **Tipični arhitekturni stilovi**

### **MVC arhitekturni stil – prednosti i nedostaci**

#### **Prednosti**

**Skalabilnost – jednostavno proširivanje (dodavanje novih kontrolera i pogleda)**

**Jednostavnost dodavanja novih tipova korisnika (dodatni pogledi i kontroleri)**

**Moguć istovremeni razvoj različitih komponenata**

**Omogućava primjenu principa “Separation of Concerns”**

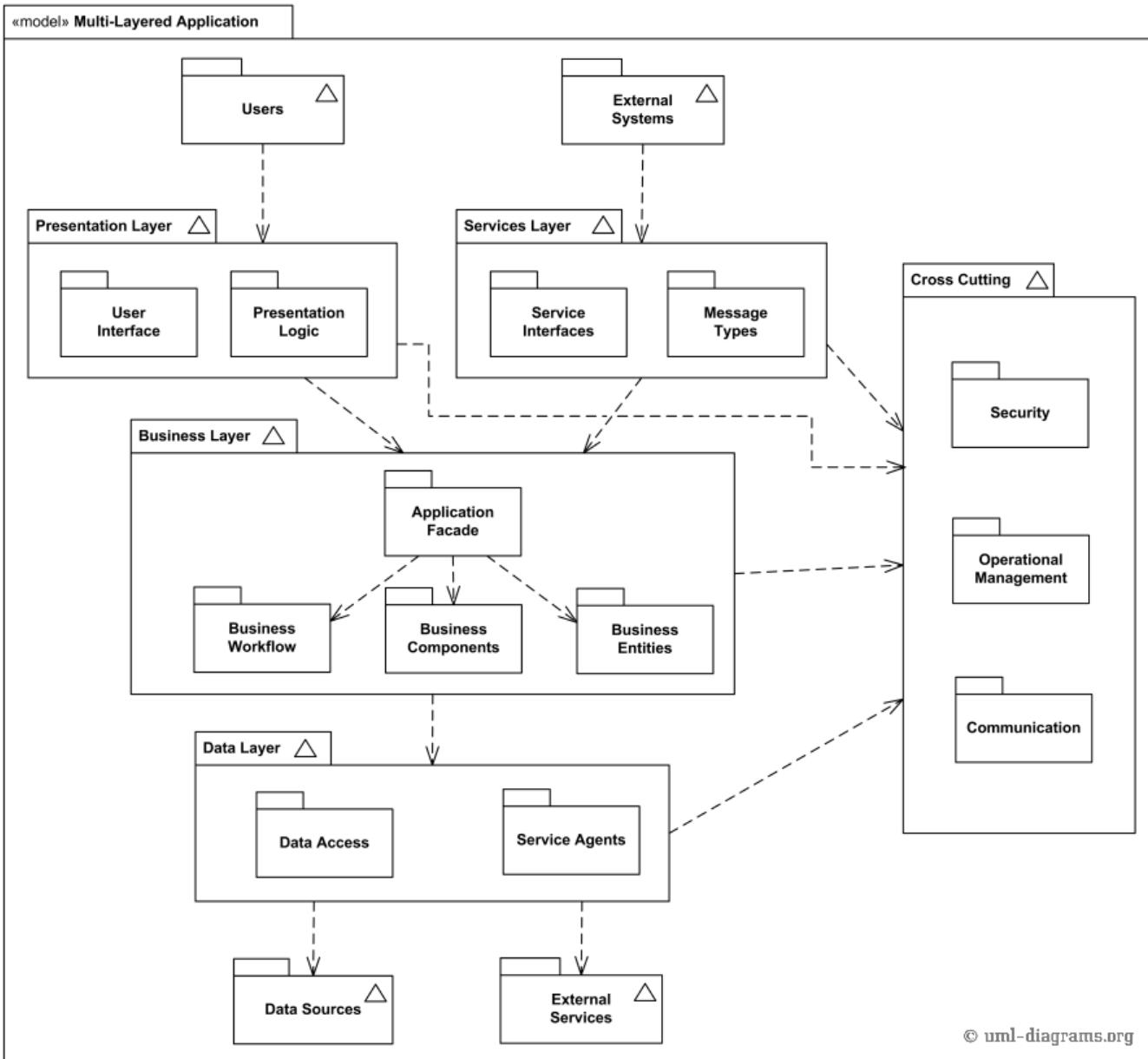
#### **Nedostaci**

**Nije pogodan kod složenijih realizacija pogleda – nove web tehnologije**

**Otežano održavanje – dosta koda u kontrolerima**

**Otežano uvođenje novih nivoa apstrakcije**

# 2. Dekompozicija sistema (nastavak)

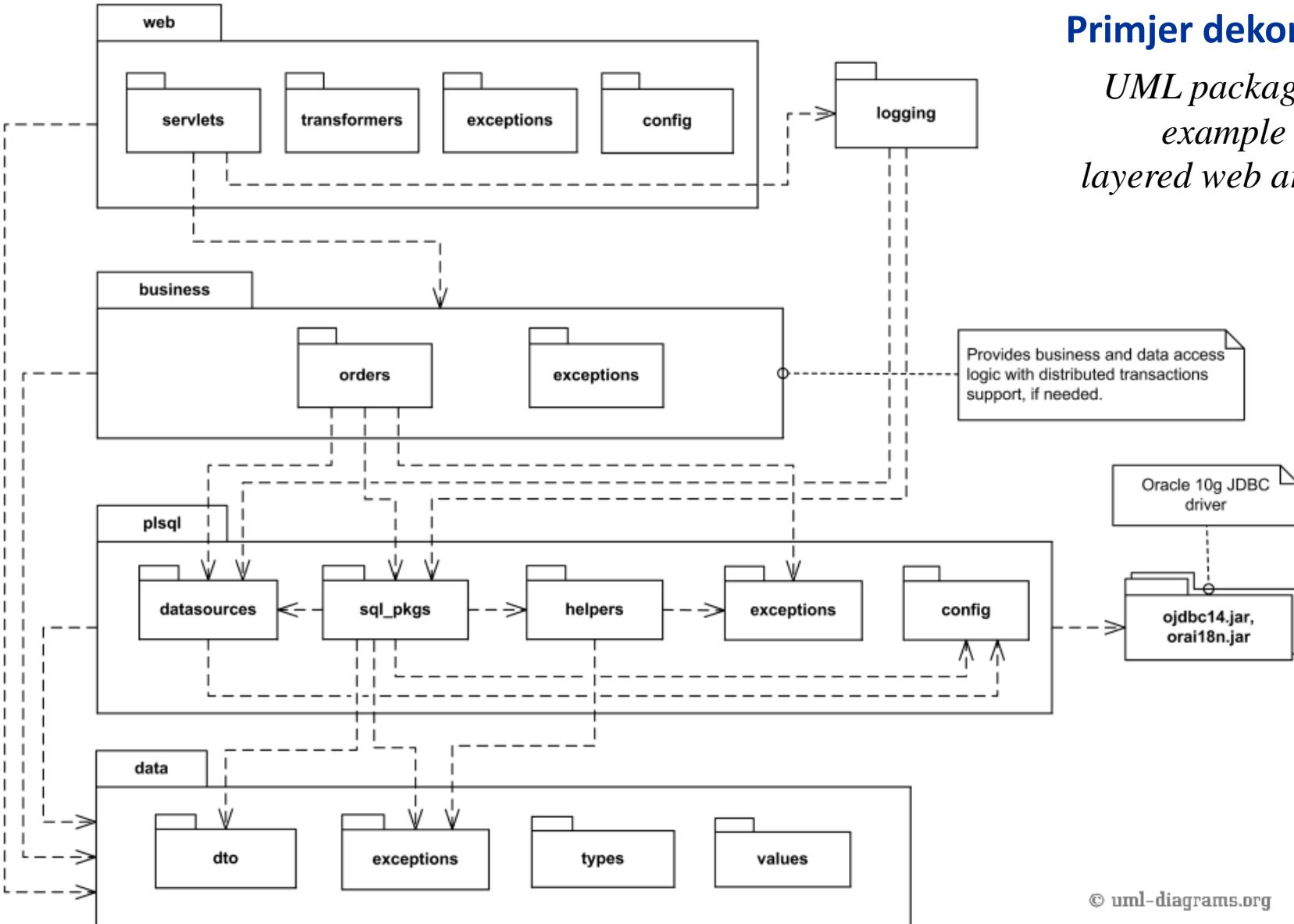


**Primjer dekompozicije**  
*Multi-layered application  
UML model diagram  
example*

# 2. Dekompozicija sistema (nastavak)

Primjer dekompozicije

*UML package diagram  
example of a multi-layered web architecture*



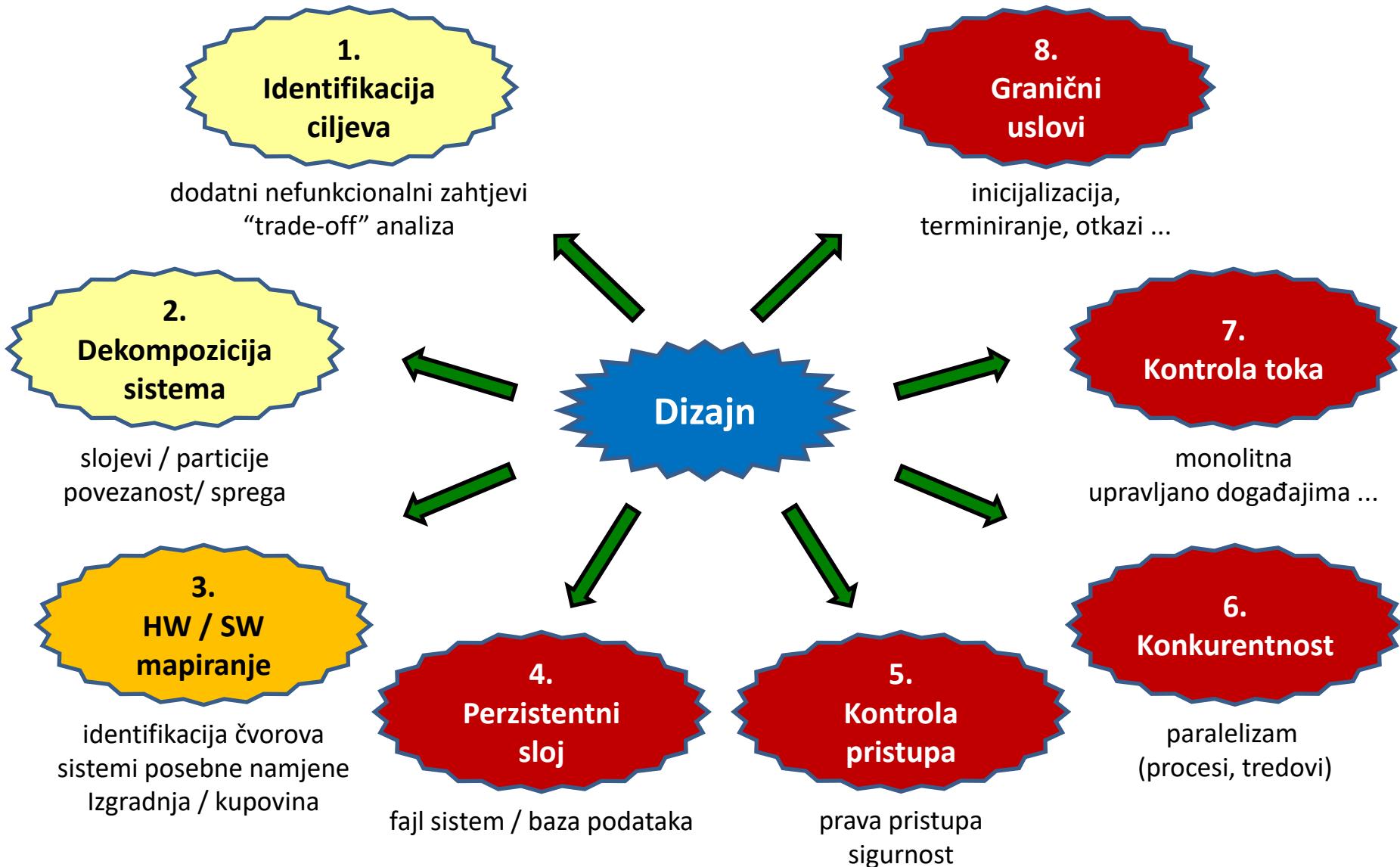
**UNIVERZITET U BANJOJ LUCI  
ELEKTROTEHNIČKI FAKULTET**

**Prof. dr Dražen Brđanin**

**PROJEKTOVANJE SOFTVERA  
/HW-SW mapiranje/**

**Banja Luka  
2024.**

# 8 bitnih aktivnosti u projektovanju



# 3. HW/SW mapiranje

Ova aktivnost u projektovanju sistema treba da odgovori na pitanja:

Kako da se realizuju podsistemi?

- Hardverski ili softverski?

Kako da se objektni model mapira na odabrani hardver i/ili softver?

- Mapiranje objekata: procesor, memorija, U/I.
- Mapiranje asocijacija: konekcije između podistema/hardverskih čvorova.

Mapiranje objekata na hardver:

**kontrolni objekti → procesor**

- Da li je jedan procesor dovoljan za izvršavanje želenog procesa, odnosno da li je željena (očekivana) brzina izvršavanja previše zahtjevna za jedan procesor?
- Možemo li ubrzati izvršavanje ako distribuiramo objekte na više procesora?
- Koliko procesora je potrebno za izvršavanje pri ustaljenom opterećenju?

**domenski objekti → memorija**

- Da li je kapacitet memorije dovoljan da prihvati seriju zahtjeva?
- ...

**granični objekti → U/I**

- Može li se postići željeni odziv pomoću raspoloživog propusnog opsega između podistema?
- ...

# 3. HW/SW mapiranje

## Mapiranje asocijacija:

**Fizička povezanost / veze između hardverskih elemenata** (veze na fizičkom sloju OSI)

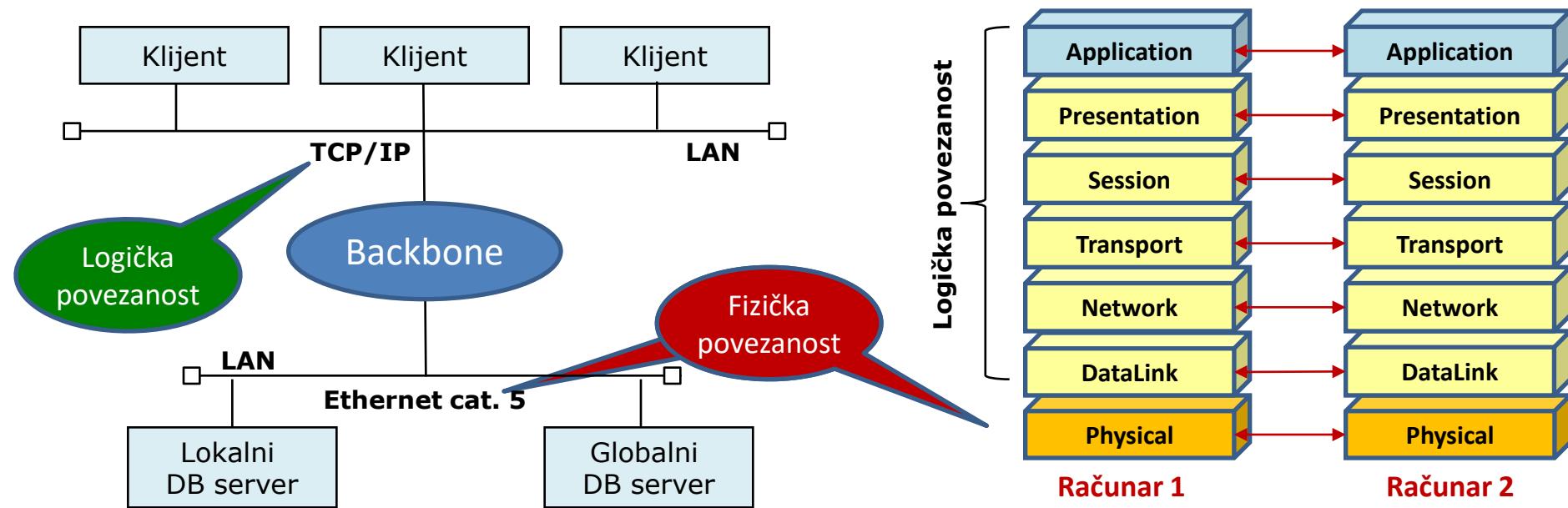
- Koje se asocijacije iz objektnog modela mapiraju u fizičke veze?
- Koje veze tipa *client/supplier* iz objektnog modela korespondiraju fizičkim vezama?

**Logička povezanost / veze između softverskih komponenata**

- Koje asocijacije se ne mapiraju u fizičke veze i u kojim slojevima se implementiraju?

**Reprezentacija fizičkih i logičkih veza**

- Uobičajeno projektanti koriste neformalne notacije za reprezentaciju veza (tipično kombinovana i konfuzna reprezentacija i fizičkih i logičkih veza)



# 3. HW/SW mapiranje

## UML podrška za HW/SW mapiranje

### UML komponenta

- gradivni blok sistema u UML-u
- **Klasifikacija komponenata:**
  - **logička** komponenta: podsistem koji nema eksplisitni run-time ekvivalent
  - **fizička** komponenta: podsistem koji ima eksplisitni run-time ekvivalent, npr. DB server
- **Životni vijek komponenata:**
  - tokom projektovanja (*design time*): npr. asocijacije, klase, ...
  - tokom implementacije (*compile time*): npr. izvorni kod, pointeri, ...
  - tokom povezivanja i/ili eksplotacije (*run time*): npr. adrese, izvršni kod, ...
- Tokom HW/SW mapiranja razmatra se distribucija *design time* komponenata



### UML dijagrami za HW/SW mapiranje

- **dijagram komponenata (component diagram)**
  - za modelovanje zavisnosti između komponenata (*desin time, compile time* i *run time*)
- **dijagram razmještaja (deployment diagram)**
  - za modelovanje rasporeda/razmještaja komponenata u eksplotaciji (*run time*)

# 3. HW/SW mapiranje

## Dijagram komponenata (*component diagram*)

- Strukturalni UML dijagram za modelovanje komponenata i njihovih veza
- Najviši nivo apstrakcije u projektovanju sistema (u pogledu komponenata i njihovih veza)
- Alternativne reprezentacije komponenata

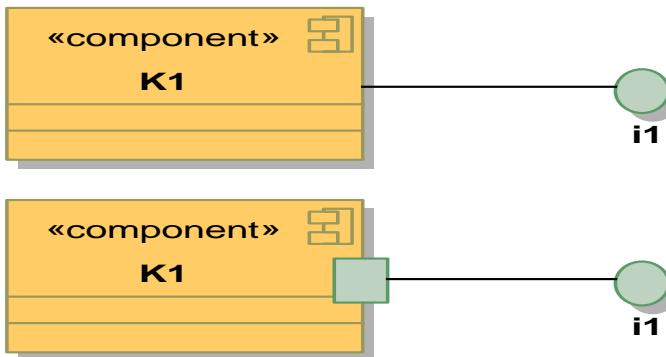


- Veze između komponenata = zavisnosti (isprekidane linije *client → supplier*)
- Dijagram komponenata često se neformalno naziva “*software wiring diagram*”
  - pokazuje kako su komponente uvezane u softverski sistem
- **Interfejs** = skup operacija koje neka komponenta pruža drugim komponentama
  - *provided interface* (*lollipop*)      ———○
  - *required interface* (*socket*)      ———○—
- **Port** = tačka interakcije neke komponente (kvadratić na rubu komponente):
  - sa okolinom komponente (*service port*)
  - sa unutrašnjošću komponente (*behaviour port*)

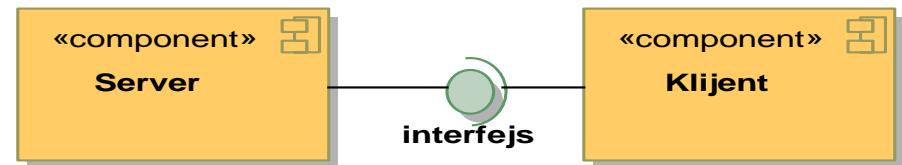


# 3. HW/SW mapiranje

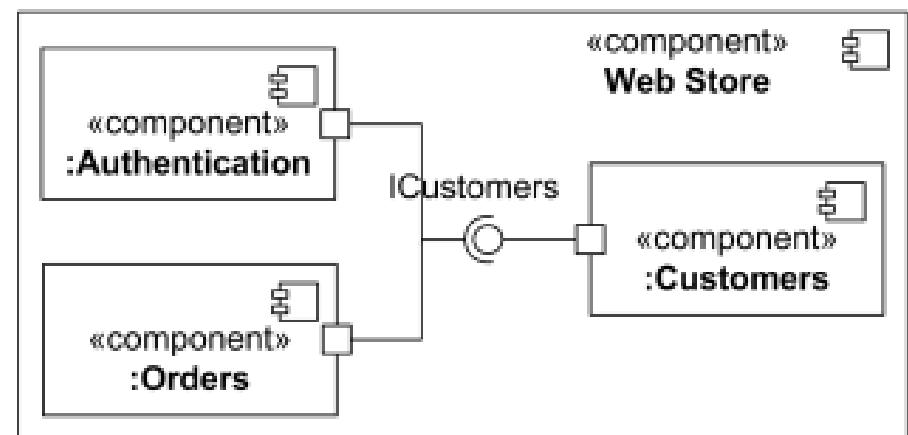
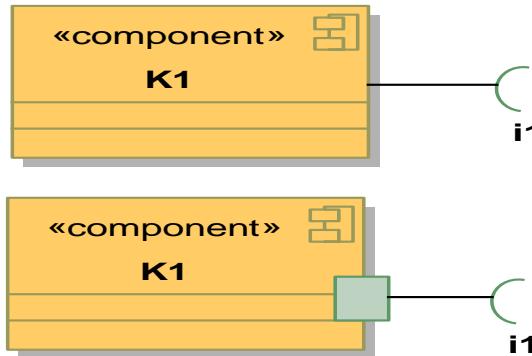
## Provided interface



Primjeri:

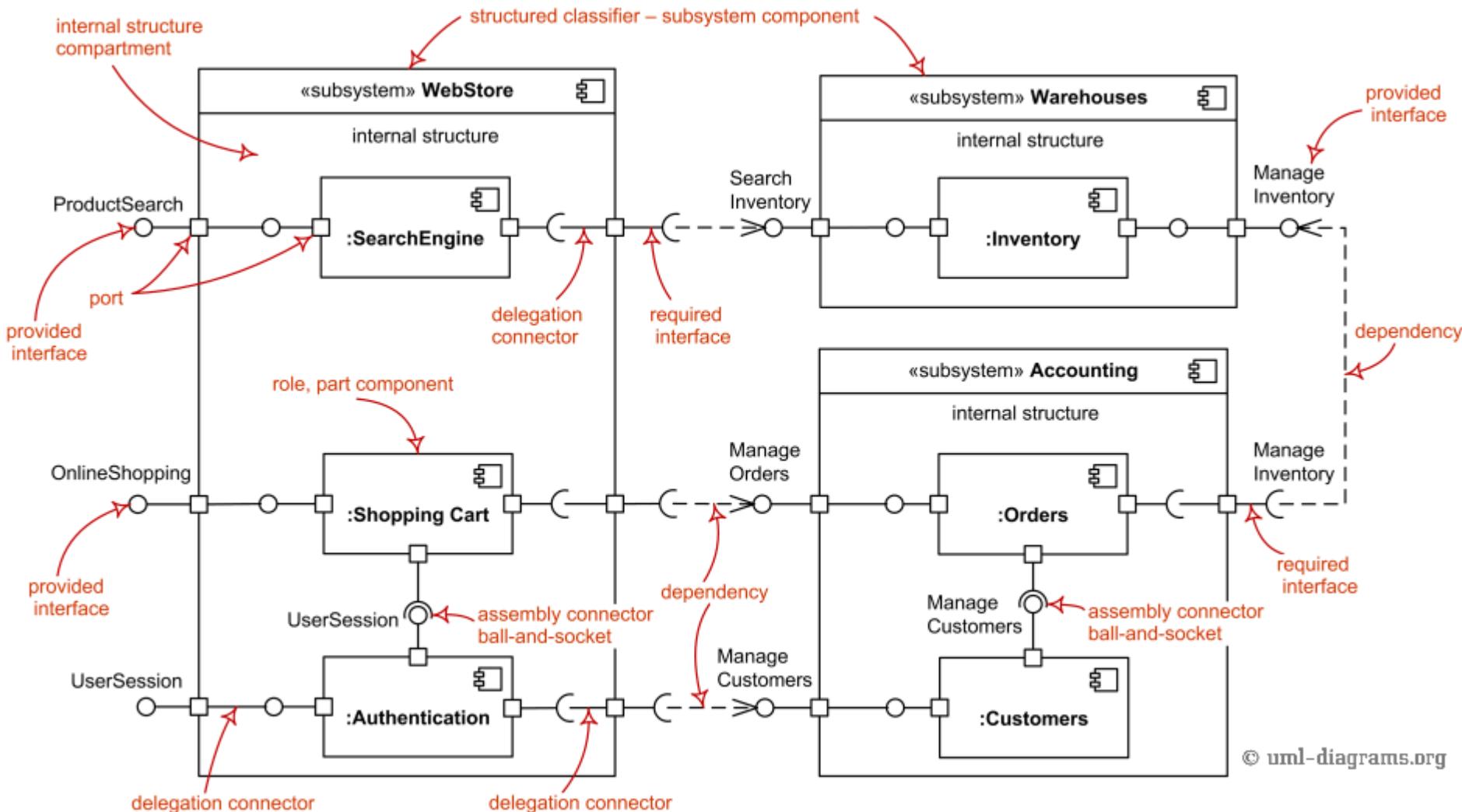


## Required interface



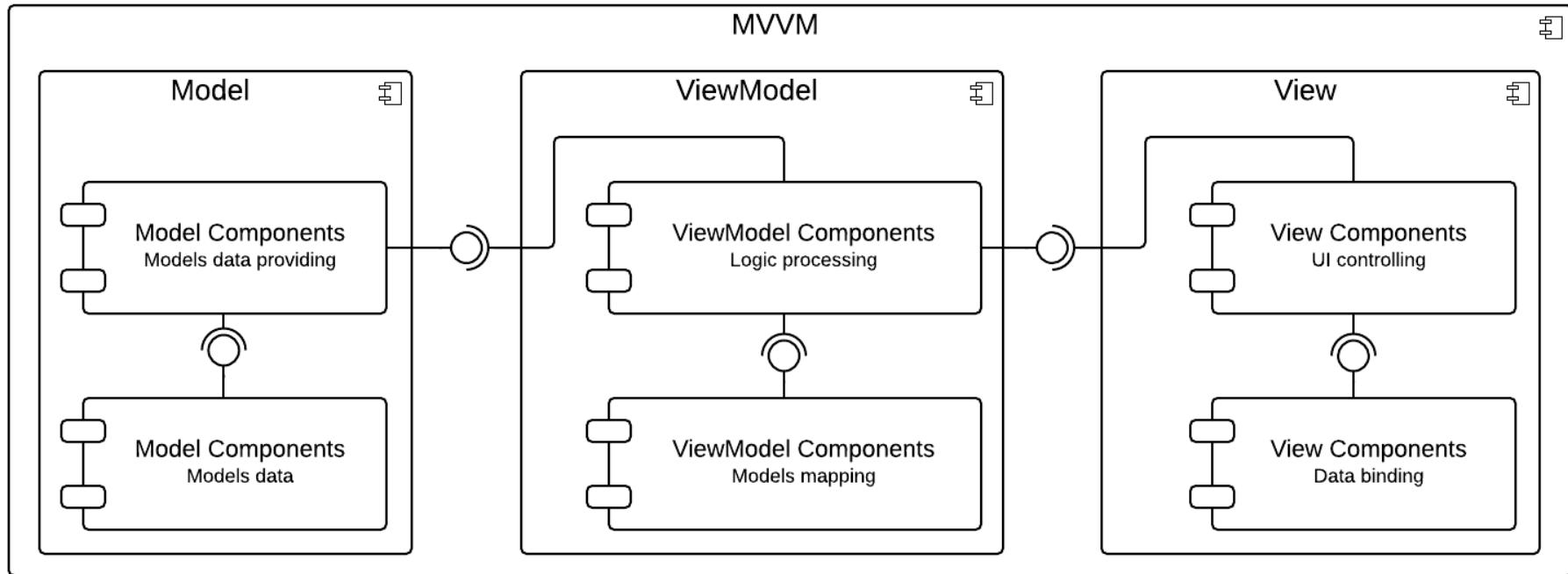
# 3. HW/SW mapiranje

## Primjer dijagrama komponenata



# 3. HW/SW mapiranje

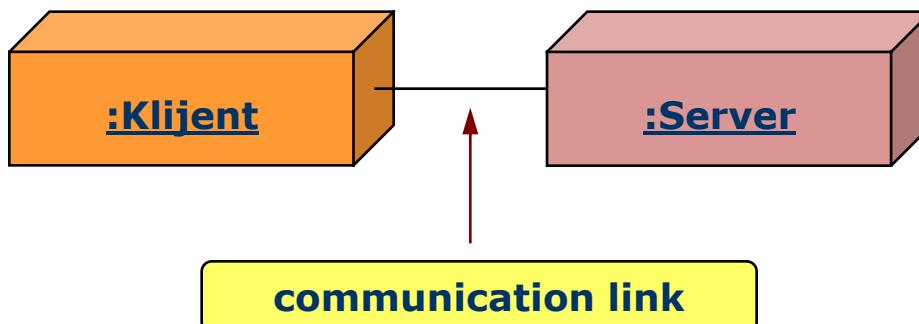
Primjer dijagrama komponenata



# 3. HW/SW mapiranje

## Dijagram razmještaja (*deployment diagram*)

- Strukturalni UML dijagram za modelovanje rasporeda komponenata u eksploraciji sistema
- Pogodan za modelovanje sistema nakon dekompozicije i HW/SW mapiranja
- To je graf kojeg čine:
  - **čvorovi (nodes)**
    - apstrakcije fizičkih objekata – HW i SW (CPU, disk, operativni sistem, ...)
    - reprezentacija kvadrom
    - mogu da sadrže različite softverske artefakte (komponente)
  - **veze čvorova (communication associations)**
    - apstrakcije fizičkih veza između objekata – komunikacioni linkovi
    - reprezentacija punom linijom

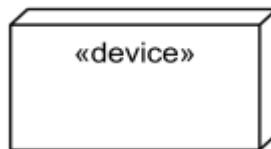


# 3. HW/SW mapiranje

## Vrste čvorova u dijagramu razmještaja

### uredaj («device»)

reprezentacija hardverskih uređaja



Standardna  
notacija



«application server»  
IBM System x3755 M3



«database server»  
Sun SPARC Server



«mobile device»  
smartphone

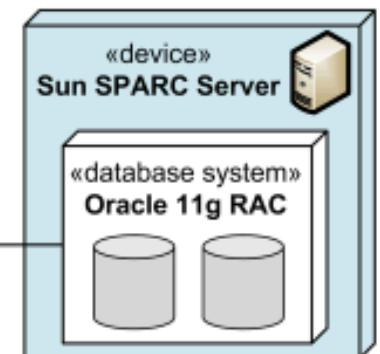
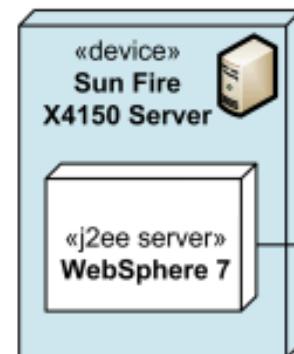
### radno okruženje («executionEnvironment»)

reprezentacija softverskog okruženja u kojem se izvršava neka komponenta



### Primjeri specijalizovane notacije

### Primjer:

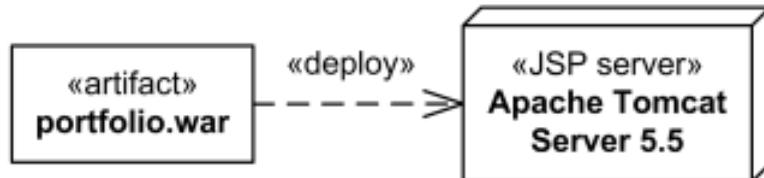


# 3. HW/SW mapiranje

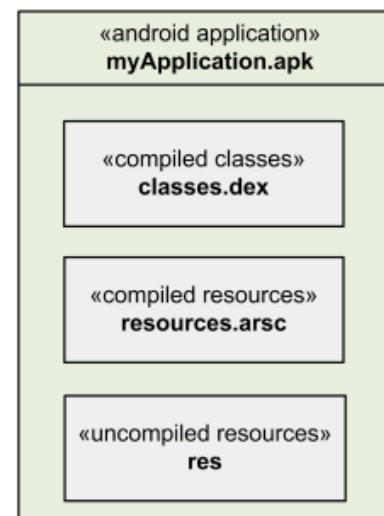
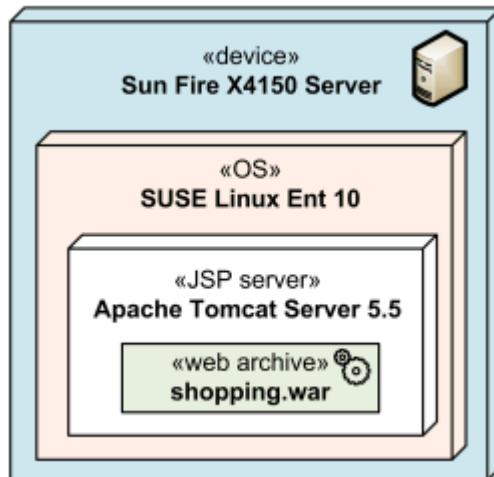
## Razmještaj softverskih artefakata po čvorovima

- Čvorovi mogu da sadrže različite softverske artefakte (komponente, biblioteke, ...)
- Artefakti koji su raspoređeni na čvorovima mogu da se reprezentuju na dva načina:

### zavisnost artefakt → čvor



### artefakt sadržan u čvoru



### Primjer:

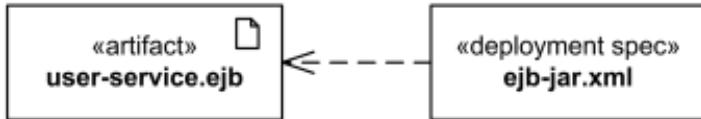


# 3. HW/SW mapiranje

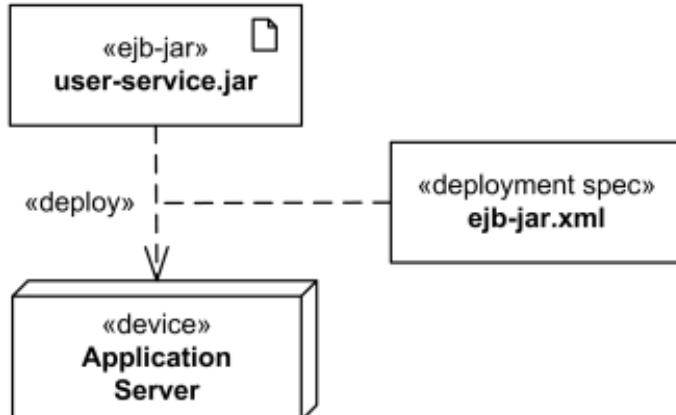
## Veze između softverskih elemenata u čvoru

### Specifikacija razmještaja «deployment spec»

Parametri razmještaja artefakta na čvoru  
(npr. adresa, sesija, konkurentnost, ... )

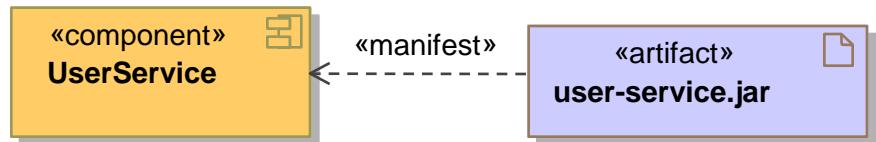


### alternativna reprezentacija:

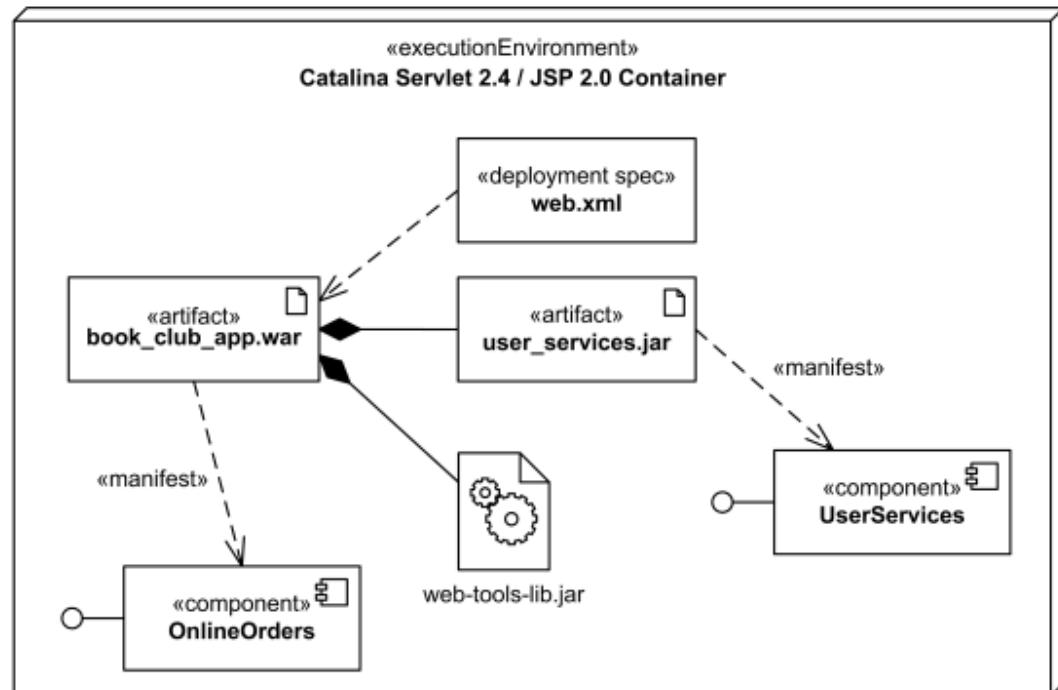


### Implementacija / Manifestacija («manifest»)

Fizička realizacija komponente nekim artefaktom  
(artefakt → komponenta)



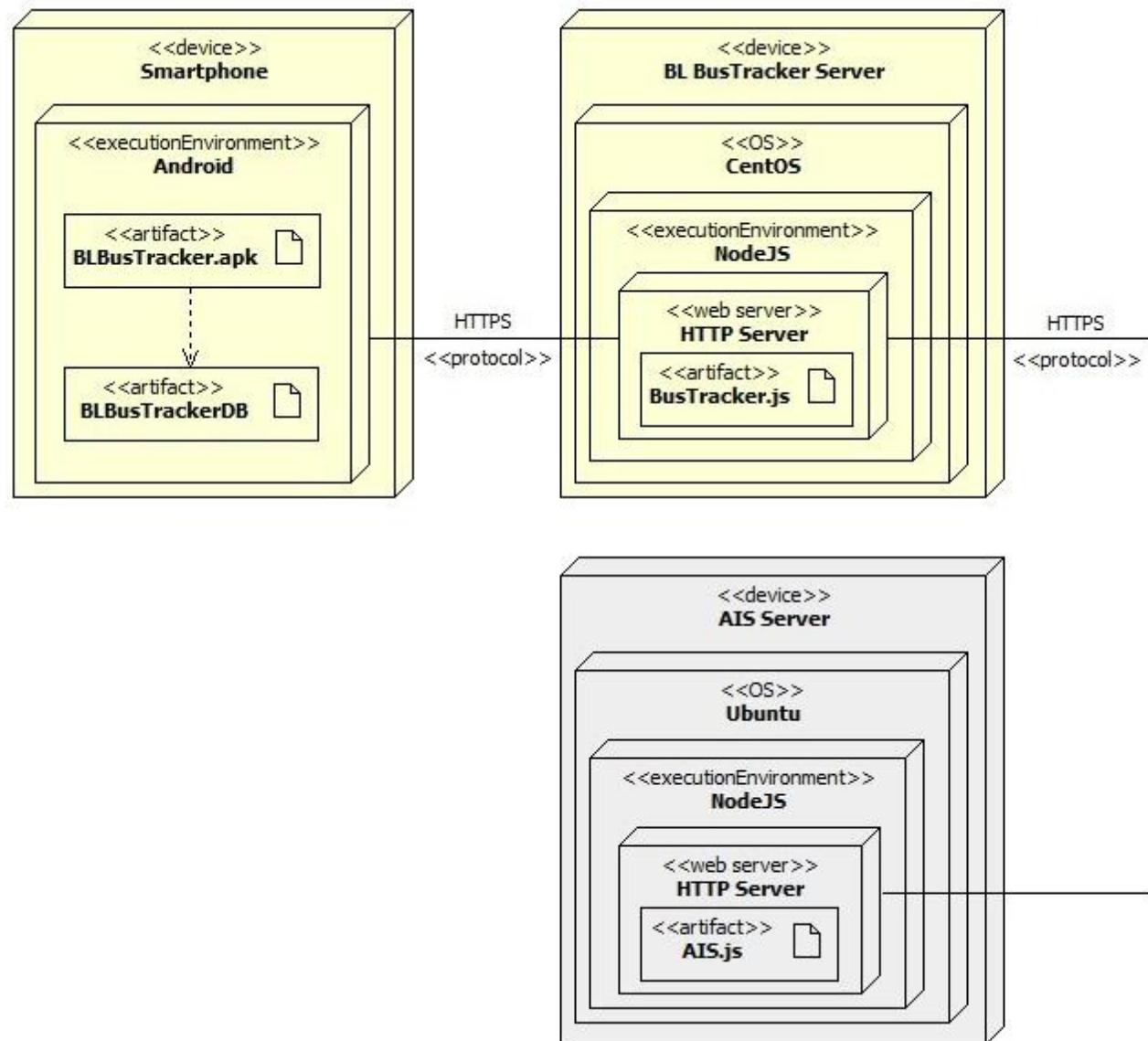
### Primjer:



# 3. HW/SW mapiranje

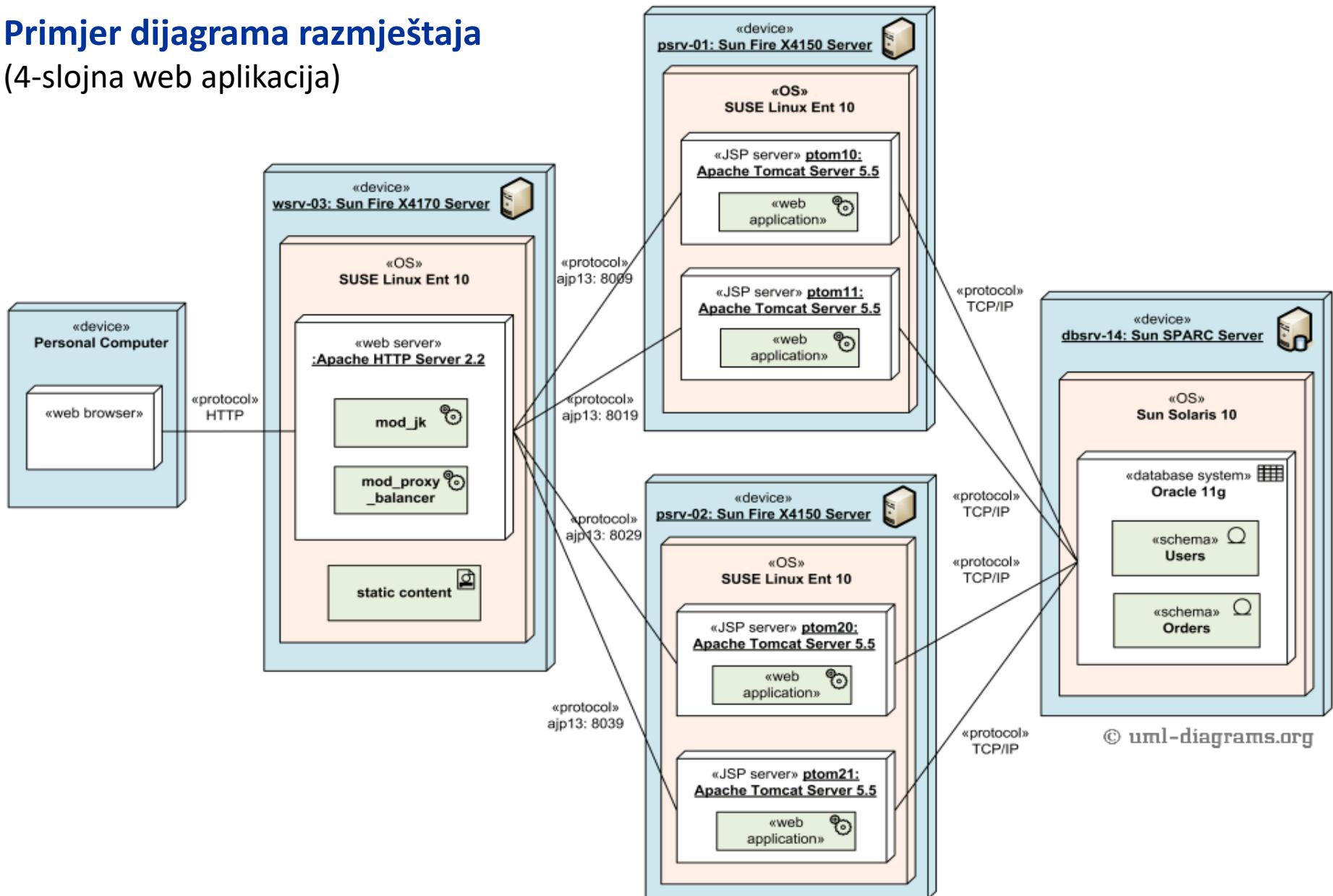
## Primjer dijagrama razmještaja

(BL BusTracker sistem)



# 3. HW/SW mapiranje

Primjer dijagrama razmještaja  
(4-slojna web aplikacija)



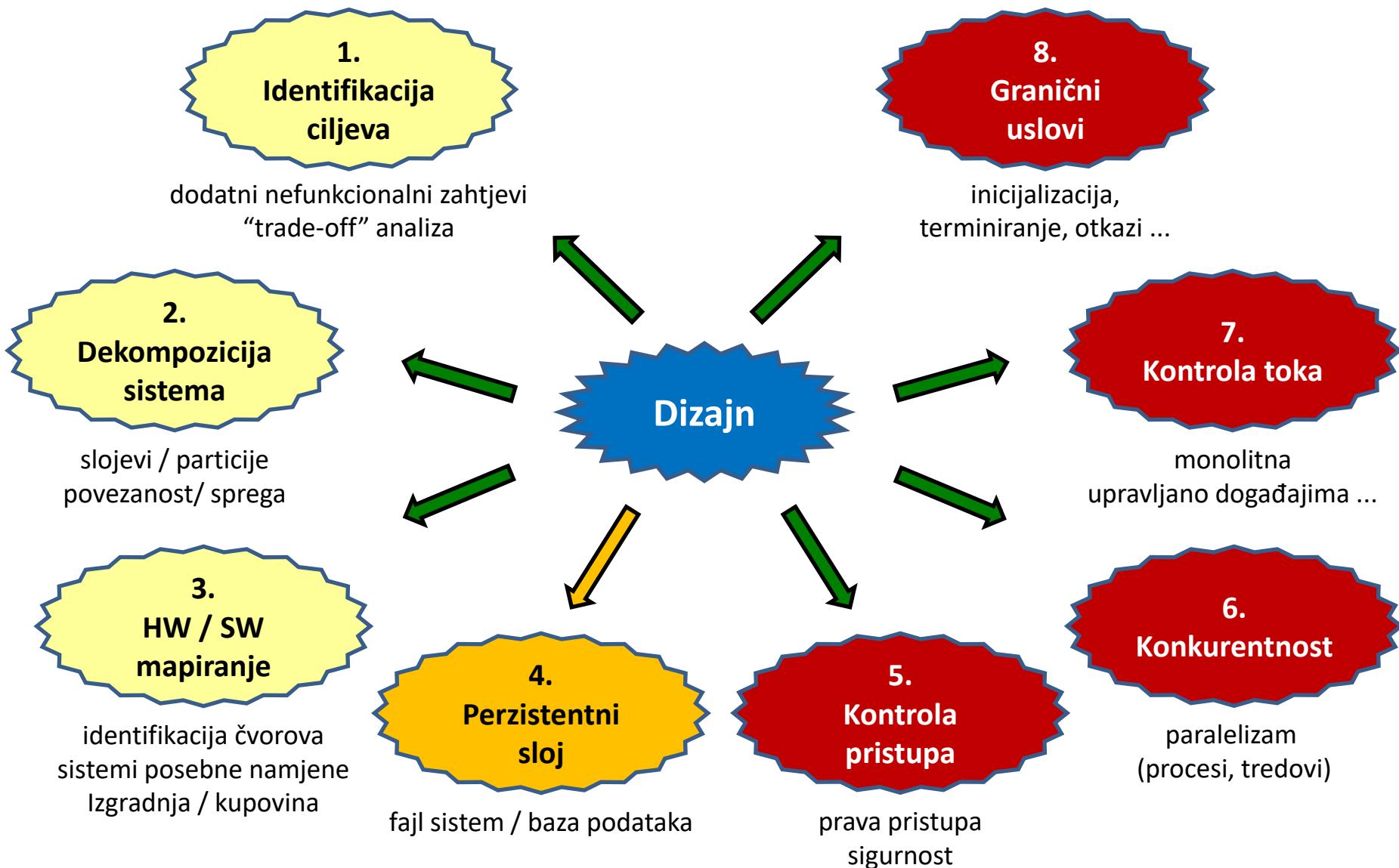
**UNIVERZITET U BANJOJ LUCI  
ELEKTROTEHNIČKI FAKULTET**

**Prof. dr Dražen Brđanin**

**PROJEKTOVANJE SOFTVERA  
/perzistentni sloj/**

**Banja Luka  
2024.**

# 8 bitnih aktivnosti u projektovanju



# 4. Perzistentni sloj

## Perzistentni (trajni) objekti

- Objekti čiji je životni vijek duži od jednog izvršavanja aplikacije i čije stanje mora da se sačuva između dva izvršavanja aplikacije (domenski objekti, korisnička podešavanja, ...).
- **Dobar dizajn:** perzistentni objekti u zasebnom podsistemu sa dobro definisanim interfejsima

## Manipulacija perzistentnim objektima

### Fajl sistem

- **Tipičan slučaj: jedan proces upisuje, a veći broj procesa čita podatke**
- **Osnovne karakteristike:**
  - niska cijena, jednostavnost
  - low-level I/O (read, write)
  - aplikacije moraju da sadrže kôd koji obezbjeđuje odgovarajući nivo apstrakcije

### Baza podataka

- **Tipičan slučaj: više konkurentnih procesa koji čitaju i/ili upisuju podatke**
- **Osnovne karakteristike:**
  - portabilnost, integritet podataka, sigurnost, ...
  - high-level I/O

# 4. Perzistentni sloj

## Neka pitanja vezana za upravljanje podacima

### Učestanost pristupa podacima

- Koliko često se pristupa podacima?
- Kolika je očekivana učestanost postavljanja upita? Najgori slučaj?

### Arhiviranje podataka

- Da li je potrebno arhiviranje podataka?
- Da li je dovoljna reprezentacija trenutnog (posljednjeg) stanja objekata ili mora da se pamti istorija (ranija) stanja objekata?

### Distribuiranost podataka

- Mogu li podaci da se drže centralizovano ili moraju distribuirano?
- Da li kod distribuiranog rasporeda treba obezbijediti lokacijsku transparentnost?  
(korisnik ima percepciju centralizovane organizacije podataka)

### Interfejs za pristup podacima

- Da li je potreban jedinstven interfejs za pristup podacima?
- Da li korisnici imaju različite poglede na podatke?
- Kakav je format upita za pristup podacima?

### Format podataka

- Da li format podataka treba da bude fleksibilan?
- Da li ciljna baza može biti relaciona / ne-relaciona?

# Objektno-Relaciono mapiranje (ORM)

## Mapiranje objektnog modela u relacioni model

### Ekvivalencija E-R i UML objektnog modela (dijagram klasa)

Dijagram klase (bez metoda) = E-R dijagram (MOV)

### Mapiranje UML dijagrama klasa u šemu relacione baze podataka

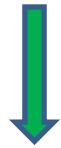
Sličan skup pravila za mapiranje kao i za mapiranje E-R dijagrama

- svaka persistenčna klasa mapira se u korespondentnu tabelu
- svaki atribut klase mapira se u kolonu korespondentne tabele
- asocijacija sa kardinalnostima 1: $*$  mapira se u dodatnu kolonu u tabeli koja odgovara klasi na strani  $*$  i tamo predstavlja strani ključ
- asocijacija sa kardinalnostima  $*:*$  mapira se u tabelu koja sadrži kolone koje reprezentuju strane ključeve prema tabelama koje korespondiraju krajevima date asocijacije
- atributi klase pridružene asocijaciji (*association class*) mapiraju se u korespondentne kolone u odgovarajućoj tabeli
- svaka veza nasljeđivanja mapira se u dodatnu kolonu u tabeli koja korespondira potklasi koja reprezentuje strani ključ (vertikalno mapiranje)

# Objektno-Relaciono mapiranje (ORM)

## Mapiranje klase

Mjesto
-idM : int
-naziv : string



Osoba
-idO : int
-ime : string



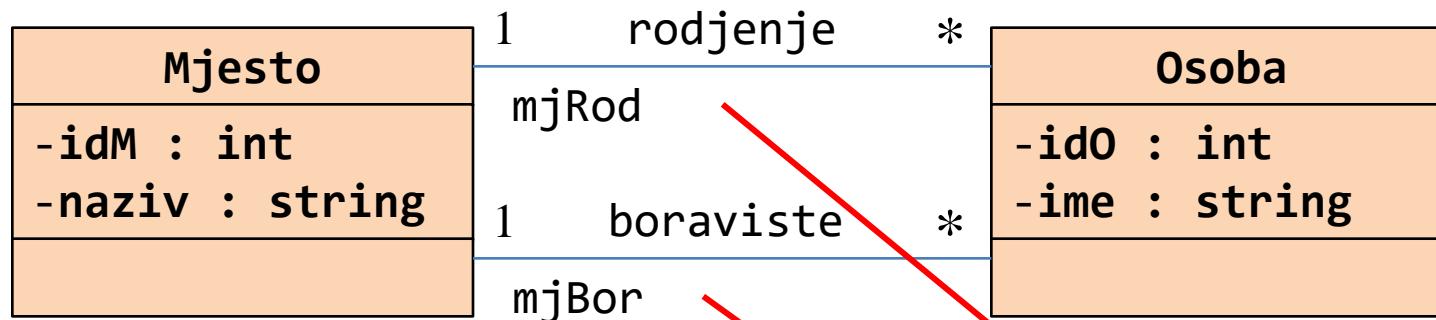
Mjesto	
<b>idM (PK)</b>	naziv

Osoba	
<b>idO (PK)</b>	ime

# Objektno-Relaciono mapiranje (ORM)

## Mapiranje asocijacija (1:\*)

Asocijacija sa kardinalnostima 1:/\* mapira se u dodatnu kolonu u tabeli koja odgovara klasi na strani \* i tamo predstavlja strani ključ



**Mjesto**

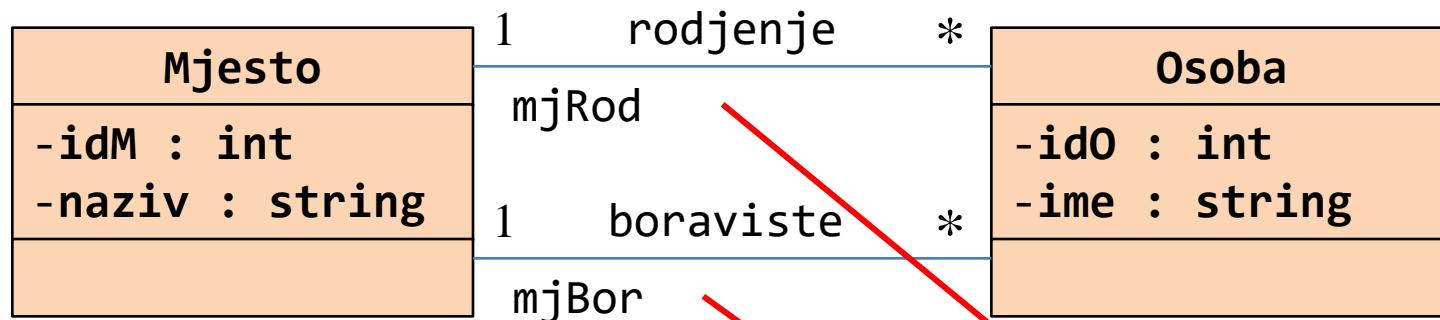
<b>idM (PK)</b>	<b>naziv</b>

**Osoba**

<b>idO (PK)</b>	<b>ime</b>	<b>mjBor (FK)</b>	<b>mjRod (FK)</b>

# Objektno-Relaciono mapiranje (ORM)

## Mapiranje asocijacija (1:\*)

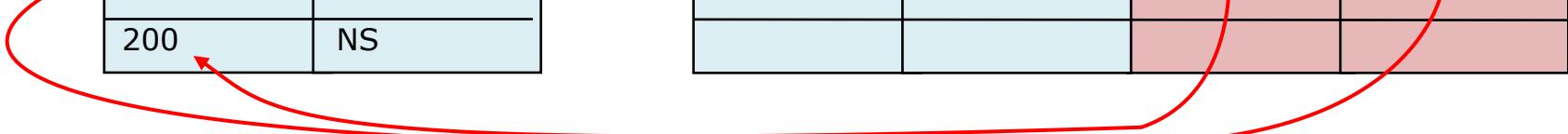


**Mjesto**

<b>idM (PK)</b>	<b>naziv</b>
100	BL
200	NS

**Osoba**

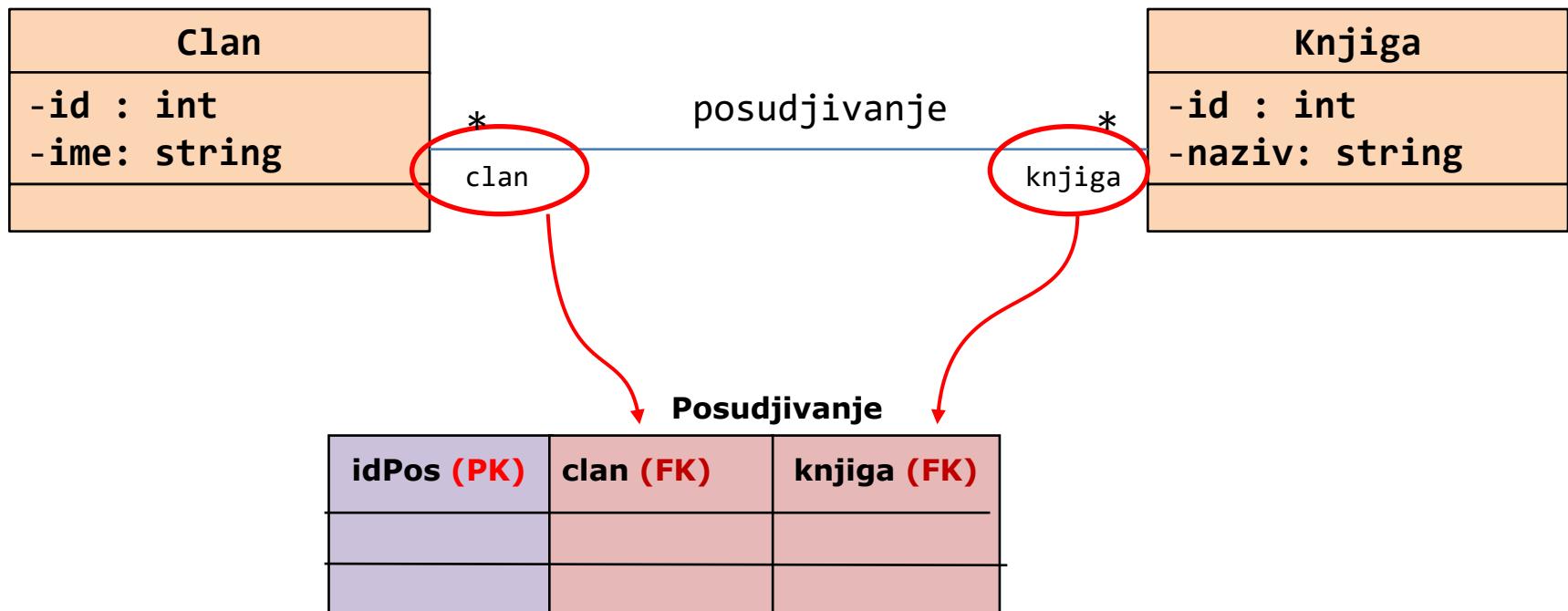
<b>idO (PK)</b>	<b>ime</b>	<b>mjBor (FK)</b>	<b>mjRod (FK)</b>
1001	Marko	200	100



# Objektno-Relaciono mapiranje (ORM)

## Mapiranje asocijacija (\*:\*)

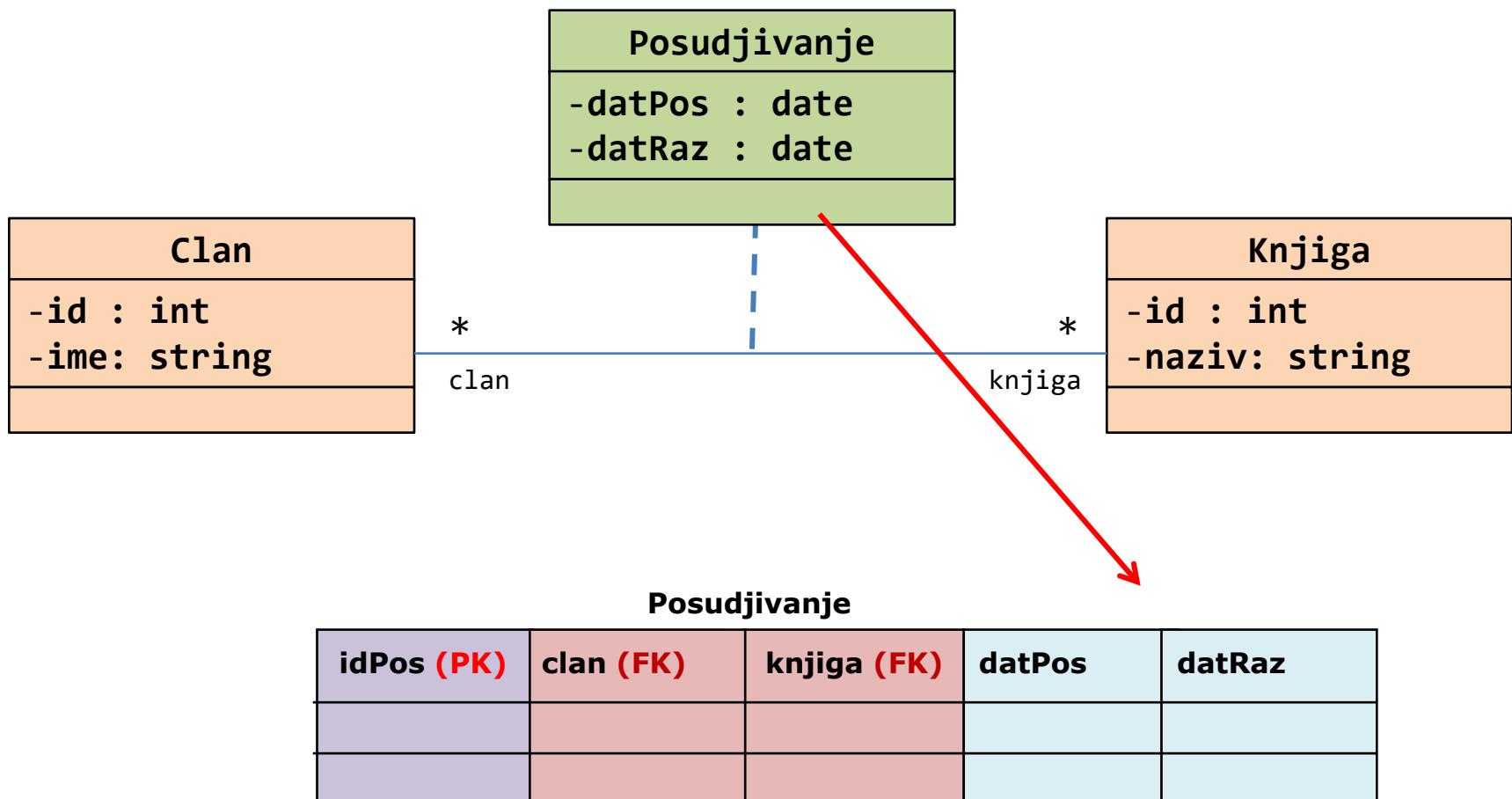
Asocijacija sa kardinalnostima \*:**\*** mapira se u tabelu koja sadrži kolone koje reprezentuju strane ključeve prema tabelama koje korespondiraju krajevima date asocijacije



# Objektno-Relaciono mapiranje (ORM)

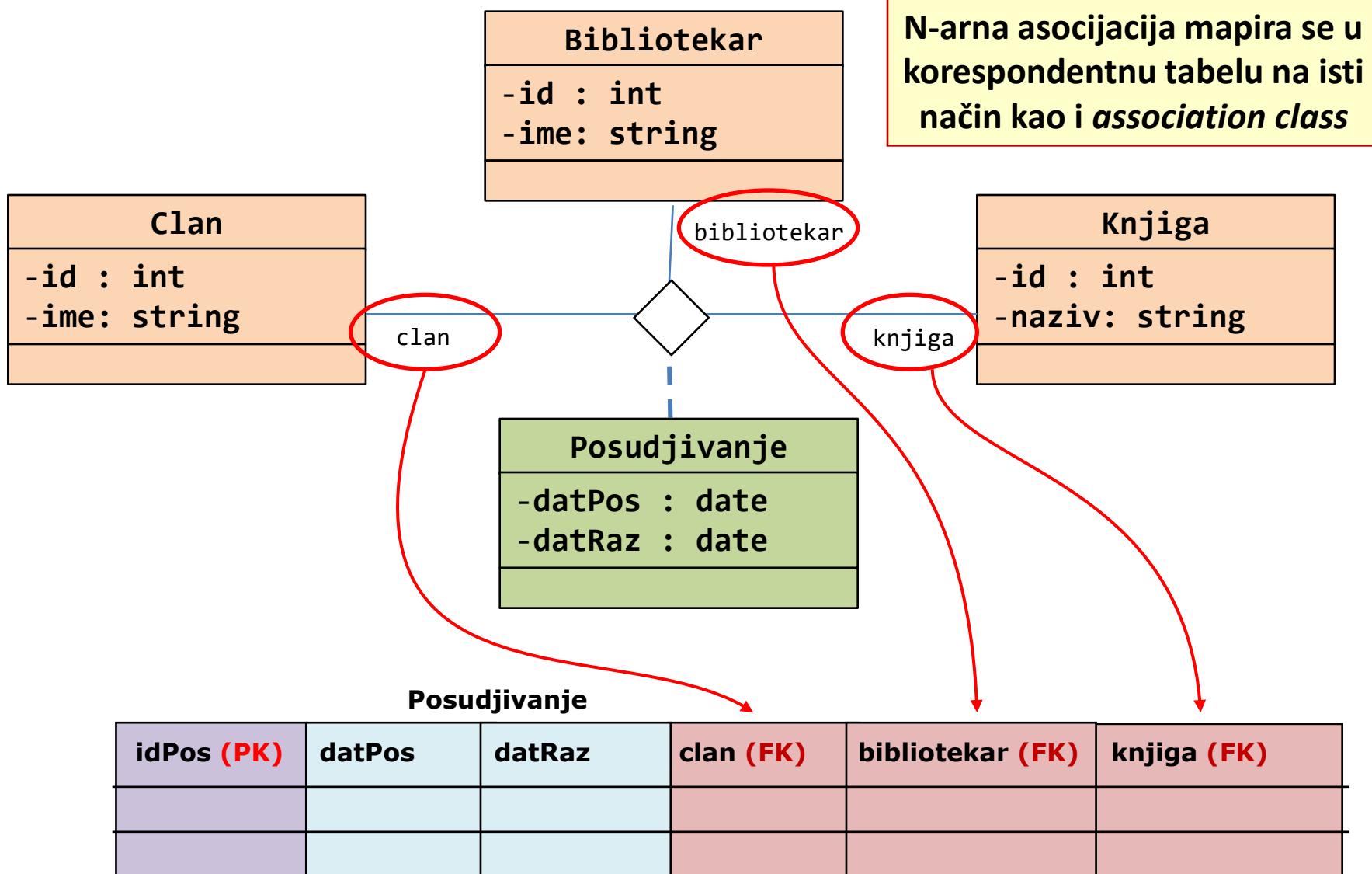
## Mapiranje klase pridružene asocijaciji (*association class*)

Atributi klase pridružene asocijaciji (*association class*) mapiraju se u korespondentne kolone u odgovarajućoj tabeli



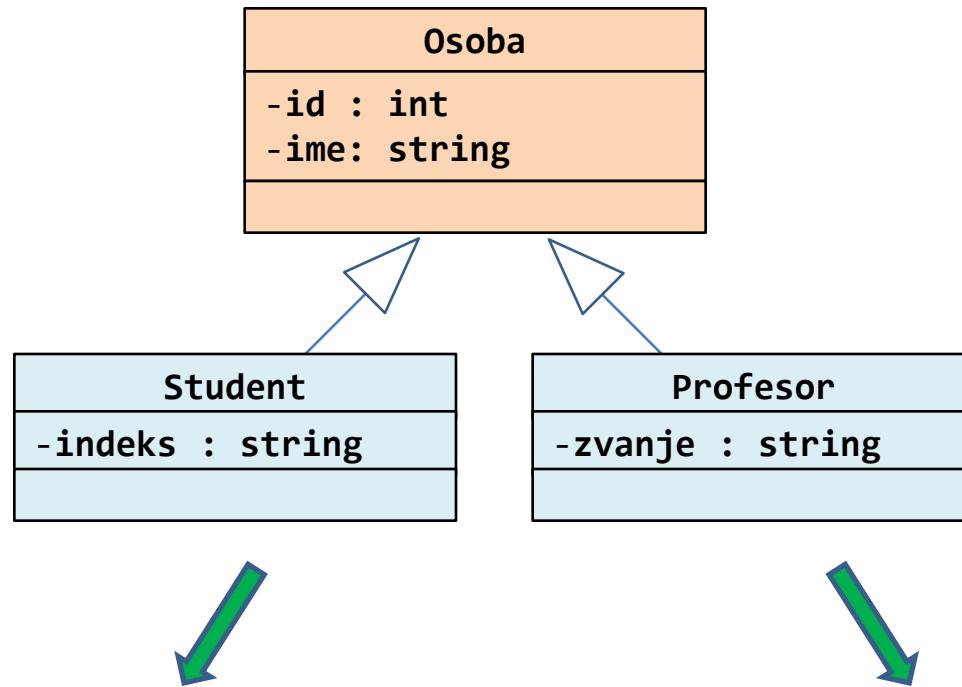
# Objektno-Relaciono mapiranje (ORM)

## Mapiranje *n*-arne asocijacija



# Objektno-Relaciono mapiranje (ORM)

## Mapiranje nasljeđivanja



### Horizontalno mapiranje nasljeđivanja

Svaka potklasa mapira se u korespondentnu tabelu kojoj se dodaju sve kolone koje odgovaraju atributima natklase

**Student**

<b>id (PK)</b>	<b>ime</b>	<b>indeks</b>

**Profesor**

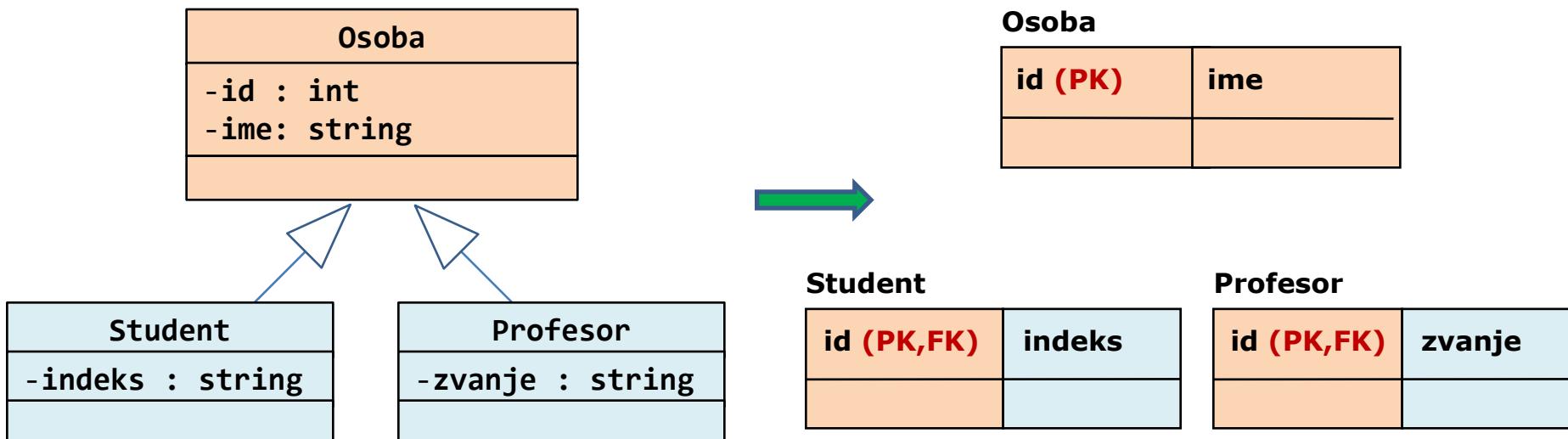
<b>id (PK)</b>	<b>ime</b>	<b>zvanje</b>

# Objektno-Relaciono mapiranje (ORM)

## Mapiranje nasljeđivanja

### Vertikalno mapiranje nasljeđivanja

Svaka veza nasljeđivanja mapira se u dodatnu kolonu u tabeli koja korespondira potklasi koja reprezentuje strani ključ



# Objektno-Relaciono mapiranje (ORM)

## SQL – Strukturni Upitni Jezik / Structured Query Language

- standardni jezik (ANSI: '87, '92, '99, '03, '06, '08) za manipulaciju (relacionim) BP
- **DDL (Data Definition Language)**
  - dio SQL jezika za definisanje baze podataka
  - kreiranje, modifikacija i brisanje tabela i indeksa
  - najznačajnije komande:
    - **CREATE TABLE** – kreira novu tabelu u bazi podataka
    - **ALTER TABLE** – mijenja strukturu postojeće tabele u bazi podataka
    - **DROP TABLE** – briše tabelu iz baze podataka
    - **CREATE INDEX** – kreira indeks za neku tabelu
    - **DROP INDEX** – briše indeks za neku tabelu

# Objektno-Relaciono mapiranje (ORM)

## SQL – DDL

CREATE TABLE – kreiranje tabele

Mjesto
-posta : string
-naziv : string

1      mjBor

boraviste

\*

Osoba
-jmbg : string
-prezime : string
-ime : string
-dat_rod : date

**CREATE TABLE MJESTO**

(

POSTA VARCHAR(5) NOT NULL,  
NAZIV VARCHAR(20),  
**PRIMARY KEY (POSTA)**

);

**CREATE TABLE CLAN**

(

JMBG            VARCHAR(13) NOT NULL,  
MJBOR          VARCHAR(5),  
PREZIME        VARCHAR(20),  
IME             VARCHAR(20),  
DAT\_ROD        DATE,  
**PRIMARY KEY (JMBG),**  
**FOREIGN KEY (MJBOR) REFERENCES MJESTO**

);

# Objektno-Relaciono mapiranje (ORM)

## SQL – DDL

### ALTER TABLE – promjena strukture tabele

Mjesto
-posta : string
-naziv : string

```
CREATE TABLE MJESTO  
(  
    POSTA VARCHAR(5) PRIMARY KEY,  
    NAZIV VARCHAR(20),  
);
```

MJESTO



POSTA	NAZIV

```
ALTER TABLE MJESTO ADD DRZAVA VARCHAR(20);
```

MJESTO



POSTA	NAZIV	DRZAVA

```
ALTER TABLE MJESTO DROP DRZAVA;
```



POSTA	NAZIV

```
DROP TABLE MJESTO;
```

# Objektno-Relaciono mapiranje (ORM)

## SQL – DDL

**CREATE INDEX** – kreiranje indeksa za neku tabelu

**CLAN**

JMBG	PREZIME	IME

```
CREATE INDEX CLAN_JMBG ON CLAN
(
    JMBG
);
```

```
CREATE INDEX CLAN_IME ON CLAN
(
    PREZIME ASC,
    IME DESC
);
```

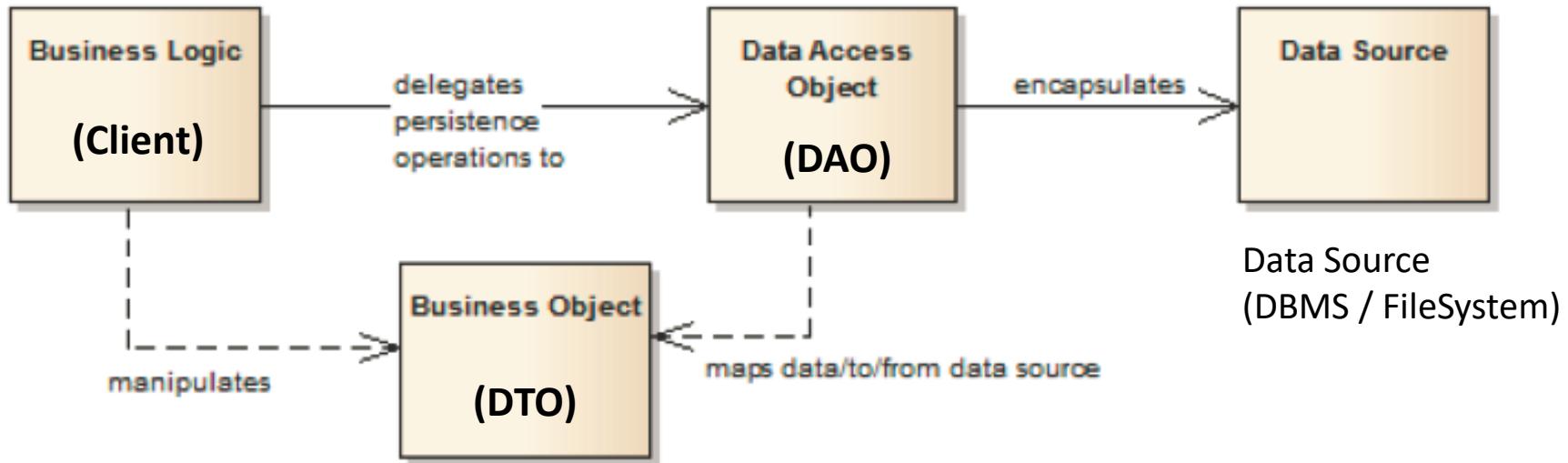
**DROP INDEX** – brisanje indeksa za neku tabelu

```
DROP INDEX CLAN_JMBG ;
```

```
DROP INDEX CLAN_IME ;
```

# Pristup perzistentnom sloju

## DAO obrazac (Data Access Object)



### Business Logic (Client)

- Kontroler koji pristupa (perzistentnom) domenskom objektu
- Kontroler zna kad i zašto mu trebaju podaci, ali ne zna (i ne mora da zna) kako je riješena perzistencija

### Data Access Object (DAO)

- DAO razdvaja logičku i fizičku reprezentaciju domenskih objekata
- DAO zna gdje su i kako se smješteni podaci, ali ne zna kad i zašto treba da im se pristupi

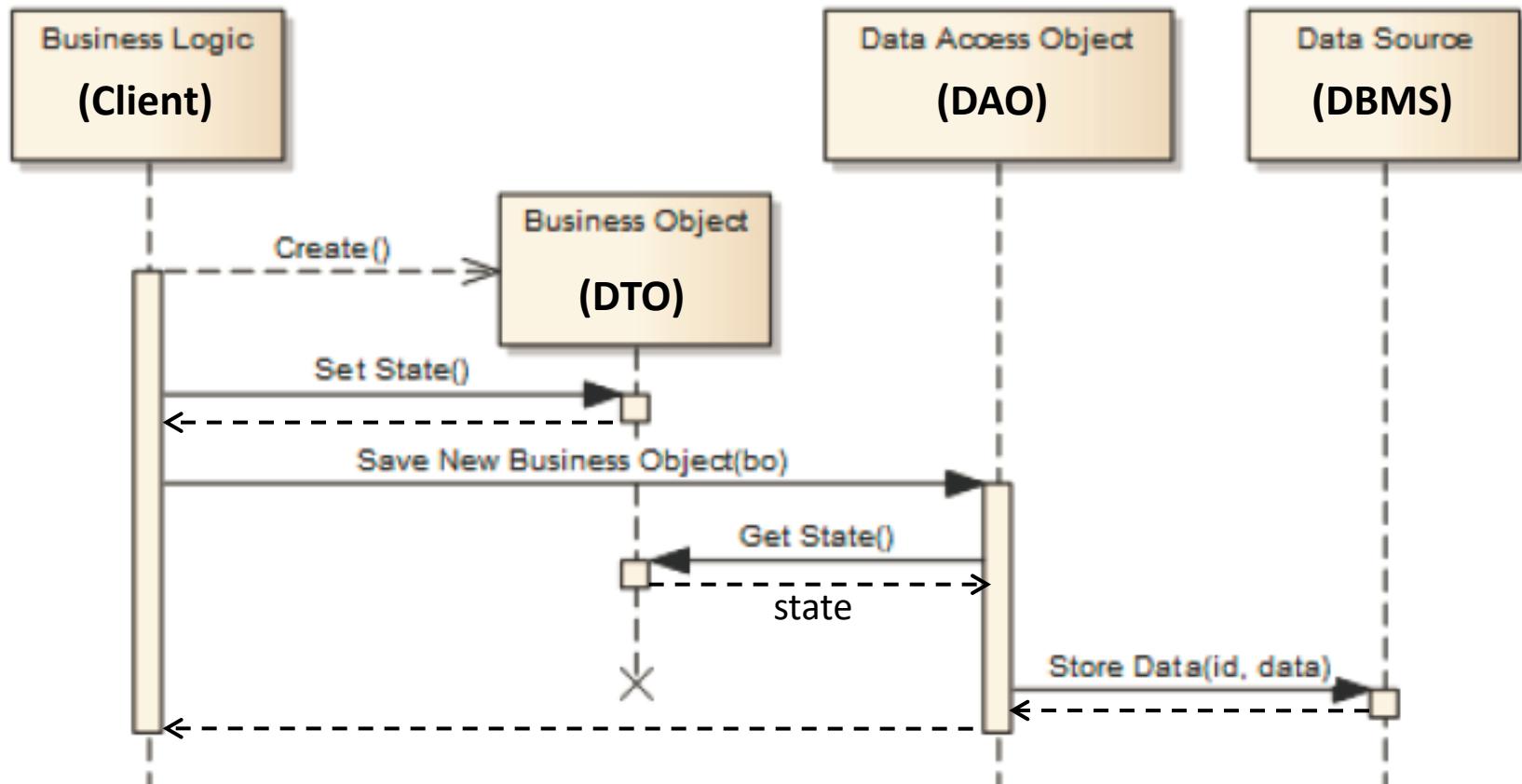
### Business Object (DTO – Data Transfer Object)

- DTO reprezentuje domenski objekat
- DTO je nosilac informacije u komunikaciji Client ↔ DAO

# Pristup perzistentnom sloju

DAO obrazac – komunikacija prilikom pristupa perzistentom sloju

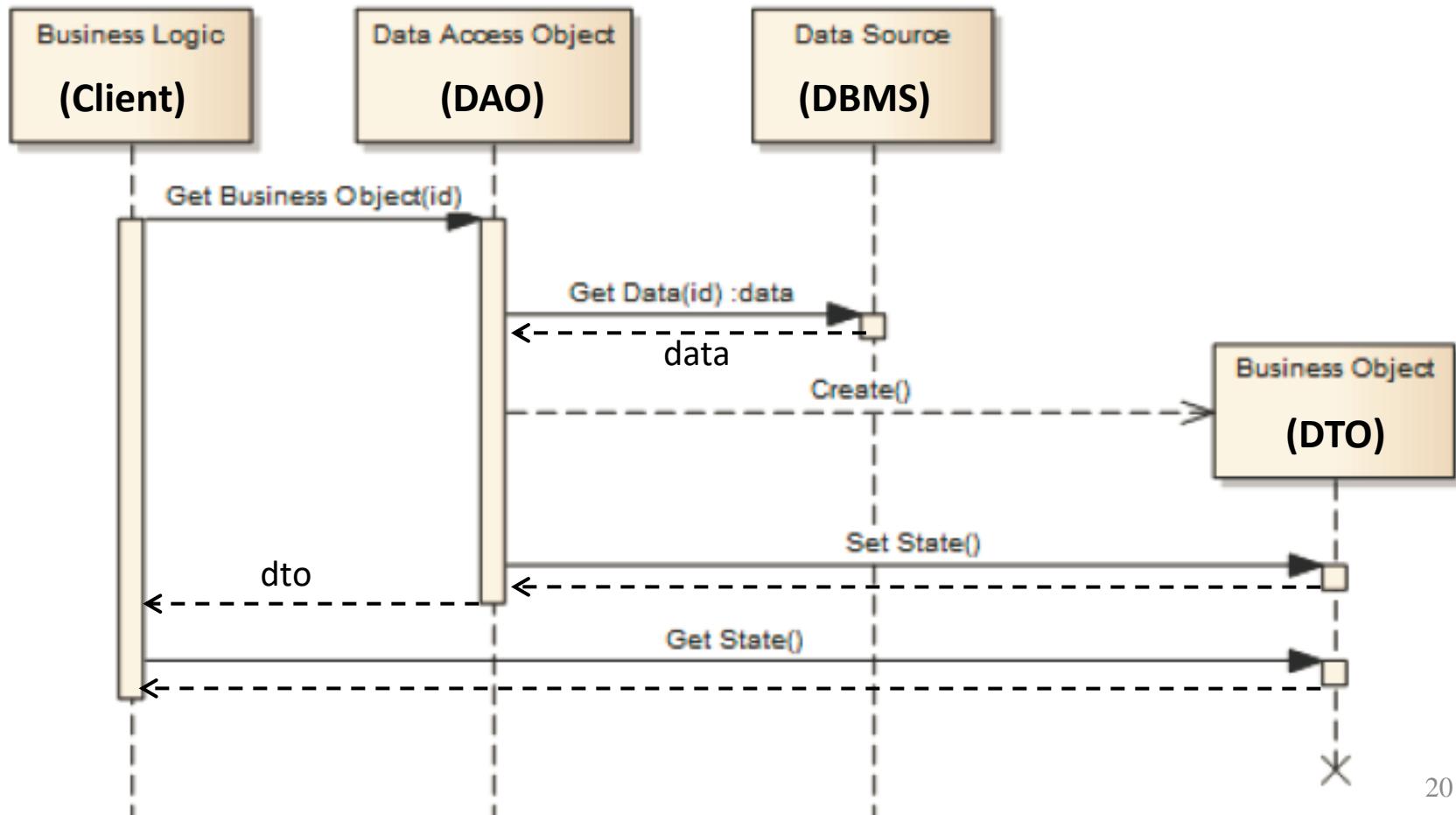
## Kreiranje i perzistencija DTO



# Pristup perzistentnom sloju

DAO obrazac – komunikacija prilikom pristupa perzistentom sloju

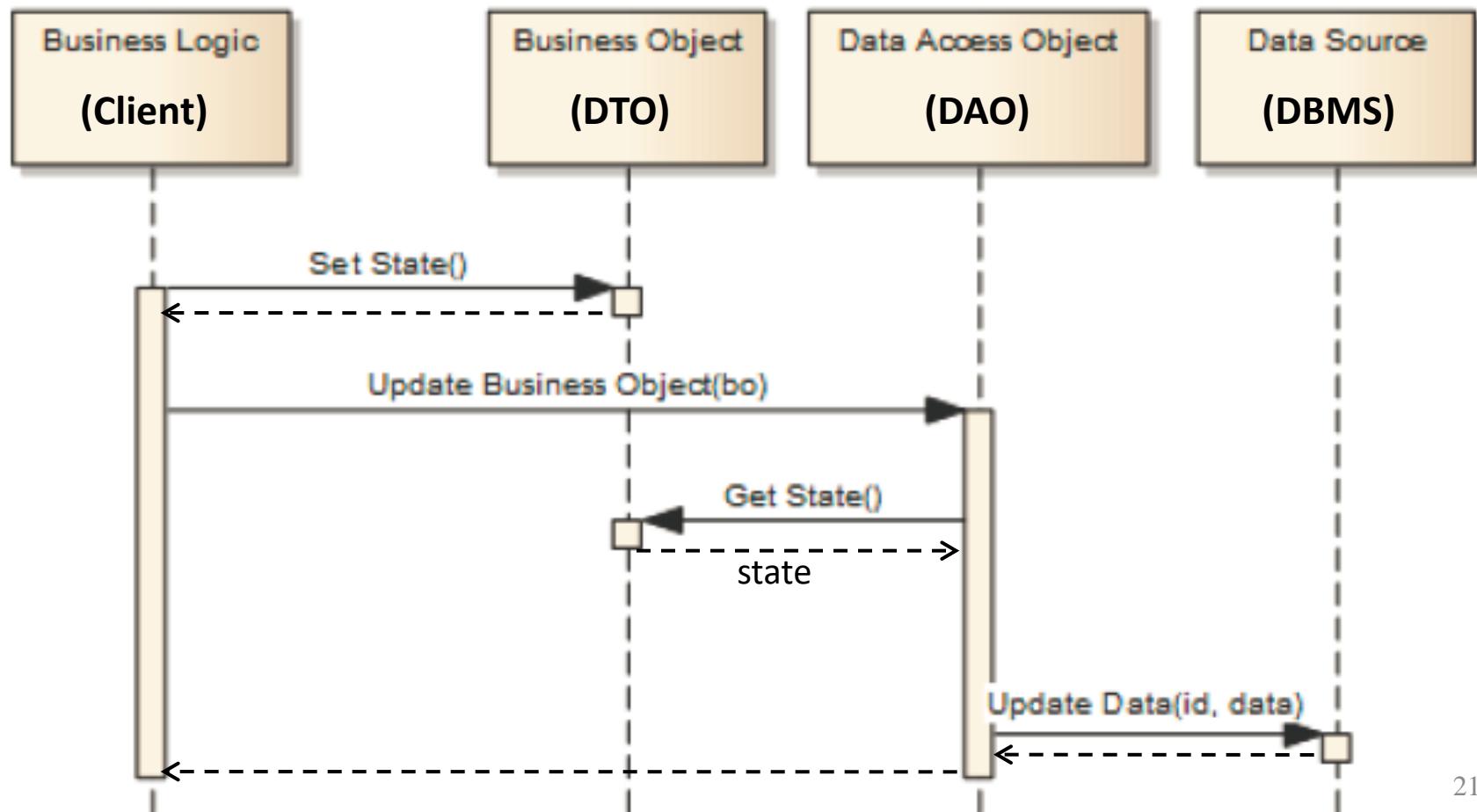
## Pribavljanje DTO



# Pristup perzistentnom sloju

DAO obrazac – komunikacija prilikom pristupa perzistentnom sloju

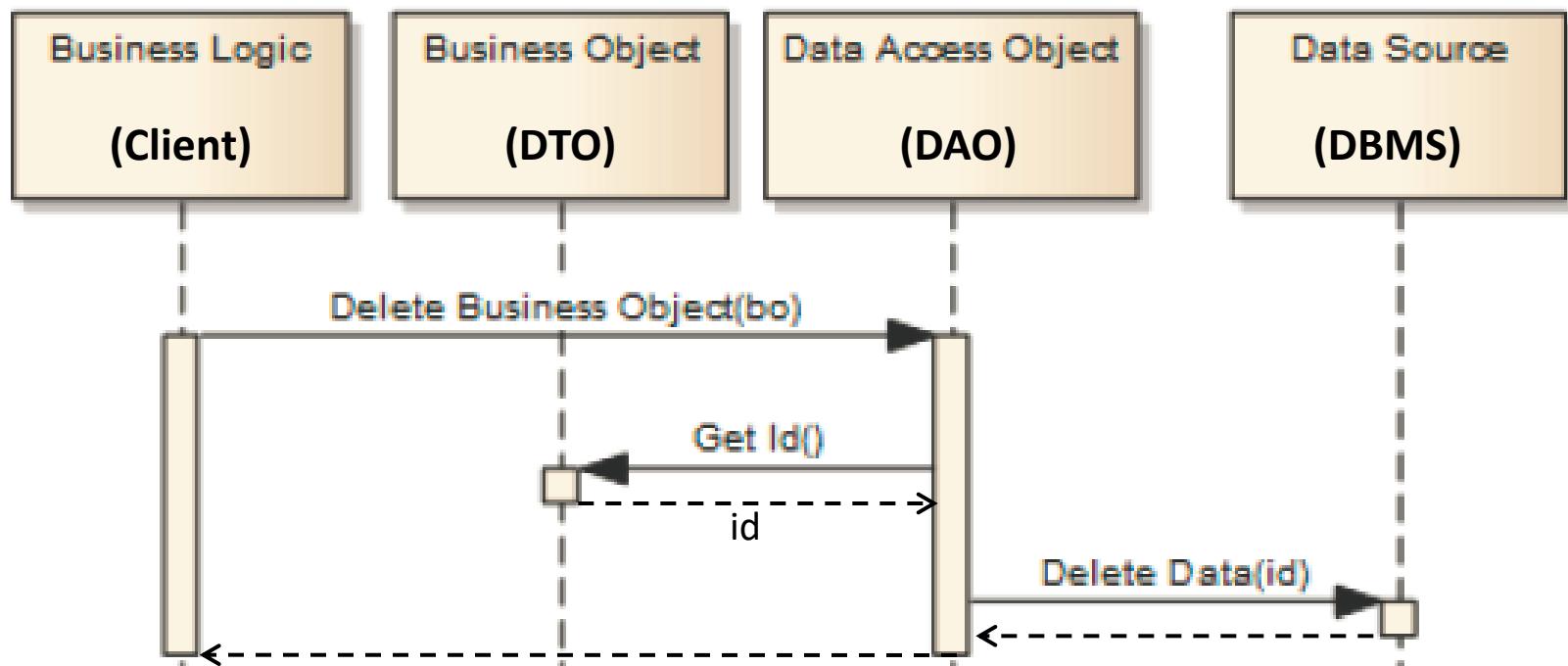
## Modifikacija perzistentnog objekta



# Pristup perzistentnom sloju

DAO obrazac – komunikacija prilikom pristupa perzistentom sloju

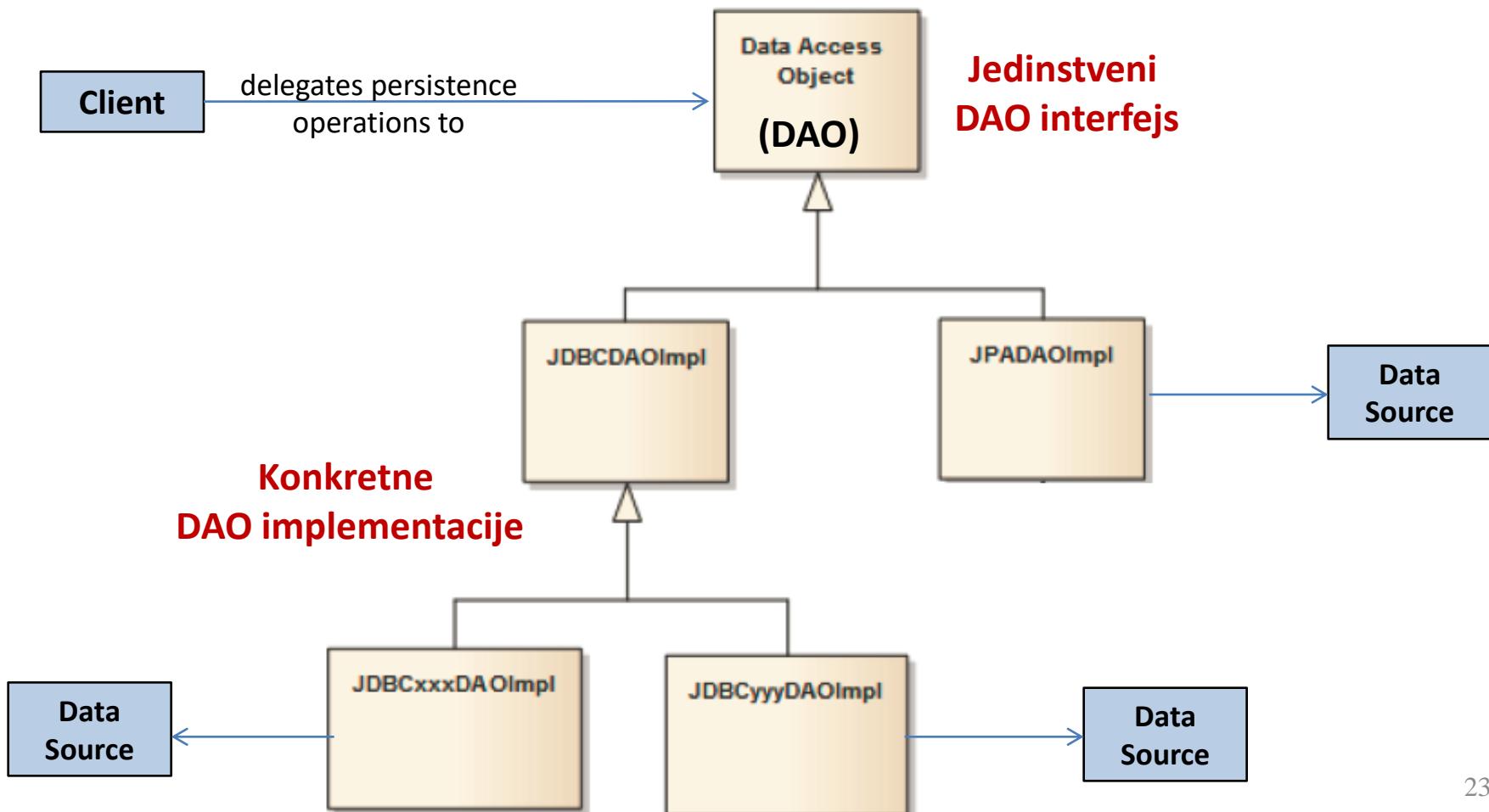
## Brisanje perzistentnog objekta



# Pristup perzistentnom sloju

DAO obrazac – implementacioni detalji

Različite “strategije” za pristup različitim vidovima perzistencije



# Pristup perzistentnom sloju

## DAO obrazac – prednosti korištenja

- Centralizacija svih operacija vezanih za pristup perzistentnim podacima u zaseban podsistem (komponentu)
- Jednostavnije održavanje
- Transparentnost
- Implementacioni detalji za pristup perzistentnom sloju skriveni u DAO klasama
- Lakša migracija i podrška za različite načine pristupa
- **“Separation of concerns”:**
  - Klijent realizuje poslovnu logiku – DAO realizuje pristup podacima
  - Manja kompleksnost koda u poslovnoj logici (nema SQL koda u poslovnoj logici)

# Pristup perzistentnom sloju

## DAO obrazac – implementacioni detalji

### DAO interfejs

- Tehnološki nezavisan i fokusiran na operacije sa DTO
- Deklaracije CRUD operacija (Create-Retrieve-Update-Delete)
- Deklaracije dodatnih agregatnih funkcija
- Deklaracije dodatnih funkcija za specifične slučajeve upotrebe

```
...  
import javax.persistence.PersistenceException;  
  
public interface MjestoDAO  
{  
    Mjesto create(Mjesto m) throws PersistenceException;  
    Mjesto update(Mjesto m) throws PersistenceException;  
    Mjesto retrieve(String p) throws PersistenceException;  
    void delete(Mjesto m) throws PersistenceException;  
    List<Mjesto> readAll() throws PersistenceException;  
}
```

### DAO Exceptions

#### Runtime Exceptions

- neočekivane greške  
(npr. nema konekcije)

#### Checked Exceptions

- očekivane greške  
(npr. pogrešan format,  
nedozvoljena vrijednost)

# Pristup perzistentnom sloju

## DAO obrazac – implementacioni detalji

### DAO implementacija

- Svaka konkretna implementacija prilagođena konkretnom DataStore
- Implementacija CRUD operacija
- Implementacija dodatnih funkcija

```
// primjer DAO implementacije
public class MjestoDAOImp implements MjestoDAO{
    @Override
    public Mjesto create(Mjesto m) { ... }
    ...
}
```

```
// primjer klijenta
public class MjestoDAOTest
{
    protected Mjesto createM()
    {
        MjestoDAO dao =
            new MjestoDAOImp();
        Mjesto m = new Mjesto();
        m = dao.create(m);
        return m;
    }
    ...
}
```

# Pristup bazi podataka – JDBC

## JDBC (*Java Database Connectivity*)

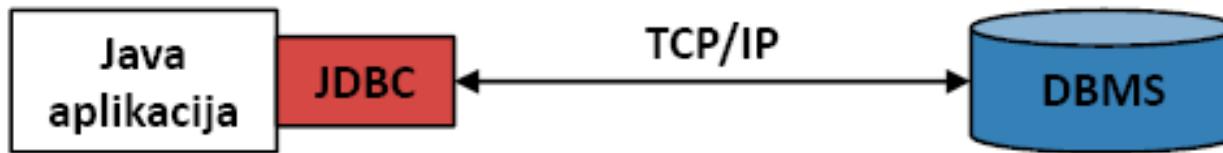
- Java API koji obezbeđuje klase i metode za interakciju sa različitim RDBMS
- Omogućava jednostavan i transparentan rad sa RDBMS
- Cjelokupan podsistem je definisan u **java.sql** paketu

## JDBC drajveri

- Svaki proizvođač obezbeđuje drajver za svoj RDBMS (MS SQL Server, MySQL ...)
- Svi drajveri koriste se na isti način

## Pristup RDBMS iz Java aplikacije

- Klijent-server arhitekturni stil (klijent: java aplikacija | server: RDMBS)
- Klijentska java aplikacija komunicira direktno sa serverom
- JDBC komponente nalaze se u klijentskom sloju



Dvoslojna arhitektura sa JDBC komponentama

# Pristup bazi podataka – JDBC

## Osnovni koraci u radu sa bazom

1. Učitavanje drajvera – automatski
2. Uspostavljanje konekcije
3. Kreiranje iskaza (*Statement*)
4. Izvršavanje iskaza
5. Obrada rezultata
6. Zatvaranje konekcije

## Uspostavljanje konekcije

- Konekcija na DBMS predstavljena je objektom tipa **Connection**
- Metoda **getConnection** klase **DriverManager** vraća objekat tipa **Connection** ako je konekcija sa DBMS-om uspješno uspostavljena:

```
Connection conn = DriverManager.getConnection(  
    "jdbc:mysql://host:port/baza_podataka", "user_ime", "pass");
```

# Pristup bazi podataka – JDBC

## Kreiranje i izvršavanje iskaza

SQL - DML (*Data Manipulation Language*)  
Detaljno na BAZAMA PODATAKA!

- **Statement** – koristi se za implementaciju jednostavnih SQL iskaza
- **ResultSet** – rezultat izvršavanja upita

```
Connection c = DriverManager.getConnection(  
    "jdbc:mysql://localhost:3306/baza", "user", "pass");  
  
Statement s = c.createStatement();  
  
ResultSet rs = s.executeQuery("select * from mjesto");  
  
while (rs.next())  
    System.out.println(rs.getString("posta") + " " +  
                      rs.getString("naziv"));  
  
rs.close();  
s.close();  
c.close();
```

# Pristup bazi podataka – JDBC

## DAO implementacija

```
// primjer DAO implementacije

public class MjestoDAOImp implements MjestoDAO
{
    @Override
    public List<Mjesto> readAll() throws PersistenceException
    {
        List<Mjesto> lista = new ArrayList<Mjesto>();
        try
        {
            Connection c = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/baza", "user", "pass");
            Statement s = c.createStatement();
            ResultSet rs = s.executeQuery("select * from mjesto");
            while (rs.next())
                lista.add(new Mjesto(rs.getString("posta"), rs.getString("naziv")));
            rs.close();
            s.close();
            c.close();
        }
        catch (SQLException e) { e.printStackTrace(); }
        return lista;
    }
}
```

# Tehnologije i alati za automatsko ORM

Prethodno je prikazano:

- objektno-relaciono mapiranje,
- manuelni proces projektovanja perzistentnog sloja na osnovu objektnog modela
- projektovanje sloja za pristup perzistentnom sloju
- aplikativna manipulacija relacionim bazama podataka zasnovana na JDBC

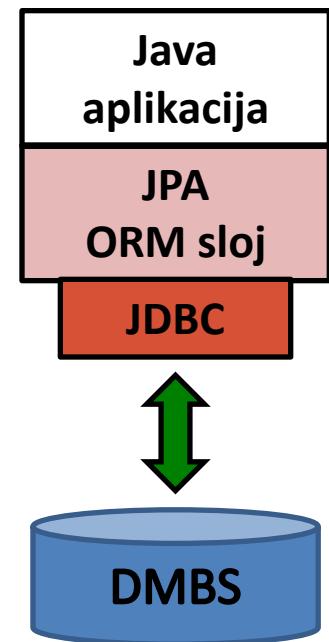
Automatizacija O-R mapiranja i pristupa perzistentnom sloju

- API za ORM
  - Java: **JPA**, EJB, Hibernate, ORMLite (Android), JDO, ...
  - .NET: EntityFramework, ADO.NET, Nhibernate, ...
  - Python: Django, ...

# Pristup perzistentnom sloju – JPA

## JPA (*Java Persistence API*)

- Skup koncepata za manipulaciju perzistentnim slojem iz Java aplikacija
- JPA nije poseban alat niti poseban AF (aplikativni okvir)
- JPA je originalno zasnovan na Hibernate (AF za ORM i pristup RDBMS putem JDBC)
- JPA je originalno namijenjen za rad sa RDBMS, ali postoje implementacije (EclipseLink, Hibernate OGM) koje omogućavaju rad sa NoSQL (nerelacionim) bazama podataka
- JPA i JPA-zasnovani alati formiraju sloj za ORM
- ORM sloj je adapter – međusloj između aplikacije i DBMS, koji programerima stvara osjećaj potpune O-O paradigmе
- **JPA podiže nivo apstrakcije i omogućava programerima da samo definišu mapiranje aplikativnih objekata na odgovarajući DBMS i da se ne bave stvarnim pristupom DBMS-u – JPA završava snimanje i čitanje preko JDBC**



# Pristup perzistentnom sloju – JPA

## JPA (*Java Persistence API*)

- JPA podiže nivo apstrakcije i omogućava programerima da samo definišu mapiranje aplikativnih objekata na odgovarajući DBMS i da se ne bave stvarnim pristupom DBMS-u – JPA završava snimanje i čitanje preko JDBC

```
// primjer JDBC
```

```
public class MjestoDAOImp implements MjestoDAO
{
    public void insert(Mjesto m) throws PersistenceException
    {
        // ...
        Connection c = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/baza", "user", "password");
        String upit =
            "insert into mjesto(posta, naziv) values(?, ?)";
        PreparedStatement ps = c.prepareStatement(upit);
        ps.setString (1, m.getPosta());
        ps.setString (2, m.getNaziv());
        ps.execute();
        c.close();
        // ...
    }
}
```

```
// primjer JPA
```

```
public class MjestoTest
{
    protected Mjesto makeM()
    {
        // ...
        Mjesto m =
            new Mjesto("78000", "BL");
        entityManager.persist(m);
        // ...
        return m;
    }
    // ...
}
```

# Pristup perzistentnom sloju – JPA

## JPA anotacije

- JPA koristi anotacije za mapiranje aplikativnih objekata na perzistentni sloj (pored anotacija mapiranje može da se definiše i u eksternim XML fajlovima)
- Svaka JPA implementacija ima procesor (*engine*) za obradu JPA anotacija

```
@Entity  
public class Mjesto { // ... }
```

Objekti klase Mjesto su perzistentni objekti

```
@Entity  
@Table(name="mjesto")  
public class Mjesto { // ... }
```

Objekti klase Mjesto su perzistentni objekti, koji se nalaze u tabeli "mjesto"

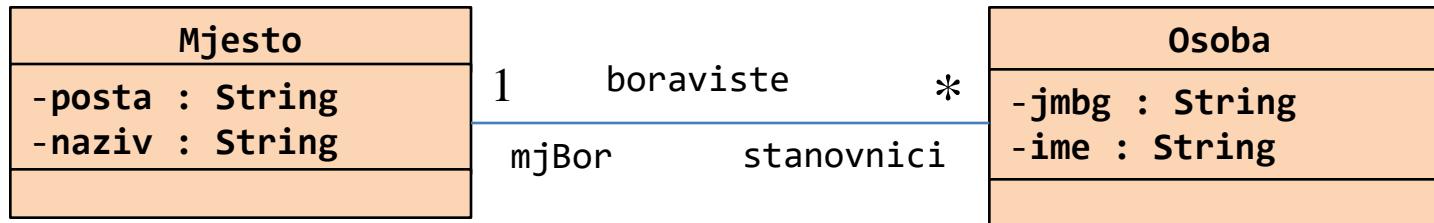
```
@Entity  
@Table(name="mjesto")  
public class Mjesto  
{  
    @Id  
    private String posta;  
}
```

Atribut posta predstavlja identifikator (primarni ključ)

# Pristup perzistentnom sloju – JPA

## JPA anotacije za asocijacije

@ManyToOne @OneToMany	@OneToOne @ManyToMany
--------------------------	--------------------------



```
@Entity
@Table(name="mjesto")
public class Mjesto
{
    @Id
    private String posta;

    @OneToMany(targetEntity=Osoba.class)
    private List stanovnici;

    private String naziv;
    // ...
}
```

```
@Entity
@Table(name="osoba")
public class Osoba
{
    @Id
    private String jmbg;

    @ManyToOne
    private Mjesto mjBor;

    private String ime;
    // ...
}
```

# Pristup perzistentnom sloju – JPA

## JPA (*Java Persistence API*)

- **EntityManager** – pristupna tačka JPA sloja – komunikacija aplikacija  $\leftrightarrow$  JPA sloj

```
public class MjestoTestPersistence
{
    public static void main( String[] args )
    {
        EntityManagerFactory emfactory =
            Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
        EntityManager em = emfactory.createEntityManager();
        em.getTransaction().begin();
        Mjesto m = new Mjesto("78000", "BL");
        em.persist(m);

        Osoba o1 = new Osoba("1111", "Marko", m);
        Osoba o2 = new Osoba("2222", "Janko", m);
        em.persist(o1); em.persist(o2);
        em.getTransaction().commit();
        em.close();
        emfactory.close();
    }
}
```

# Pristup perzistentnom sloju – JPA

## JPA – CRUD operacije

```
public class MjestoTestCRUD
{
    public static void main( String[] args )
    {
        EntityManagerFactory emfactory =
            Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
        EntityManager em = emfactory.createEntityManager();
        Mjesto m = em.find(Mjesto.class, "78000");
        em.getTransaction().begin();
        m.setName("Banja Luka");
        em.getTransaction().commit();

        em.close();
        emfactory.close();
    }
}
```

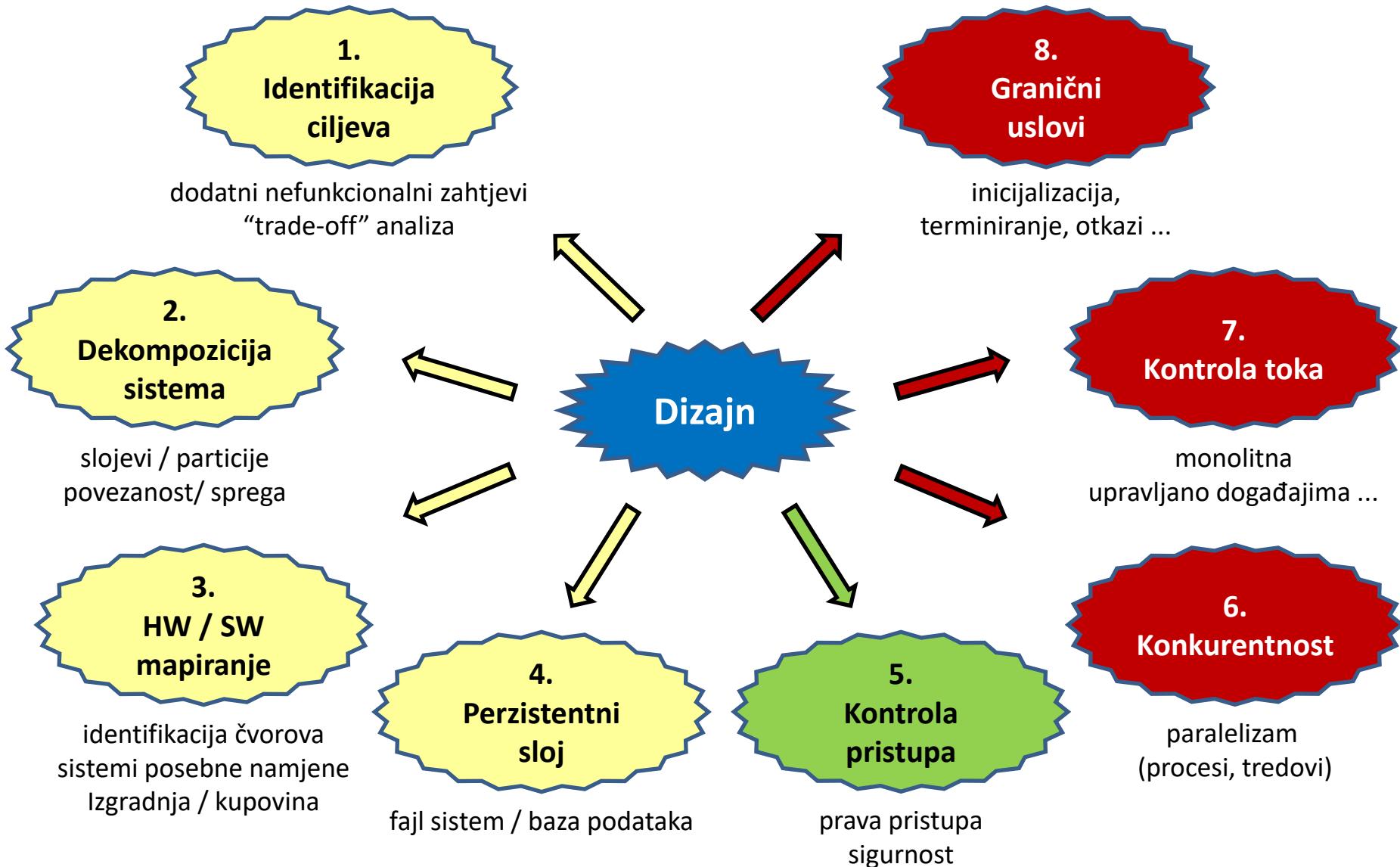
**UNIVERZITET U BANJOJ LUCI  
ELEKTROTEHNIČKI FAKULTET**

**Prof. dr Dražen Brđanin**

**PROJEKTOVANJE SOFTVERA  
/kontrola pristupa/**

**Banja Luka  
2024.**

# 8 bitnih aktivnosti u projektovanju

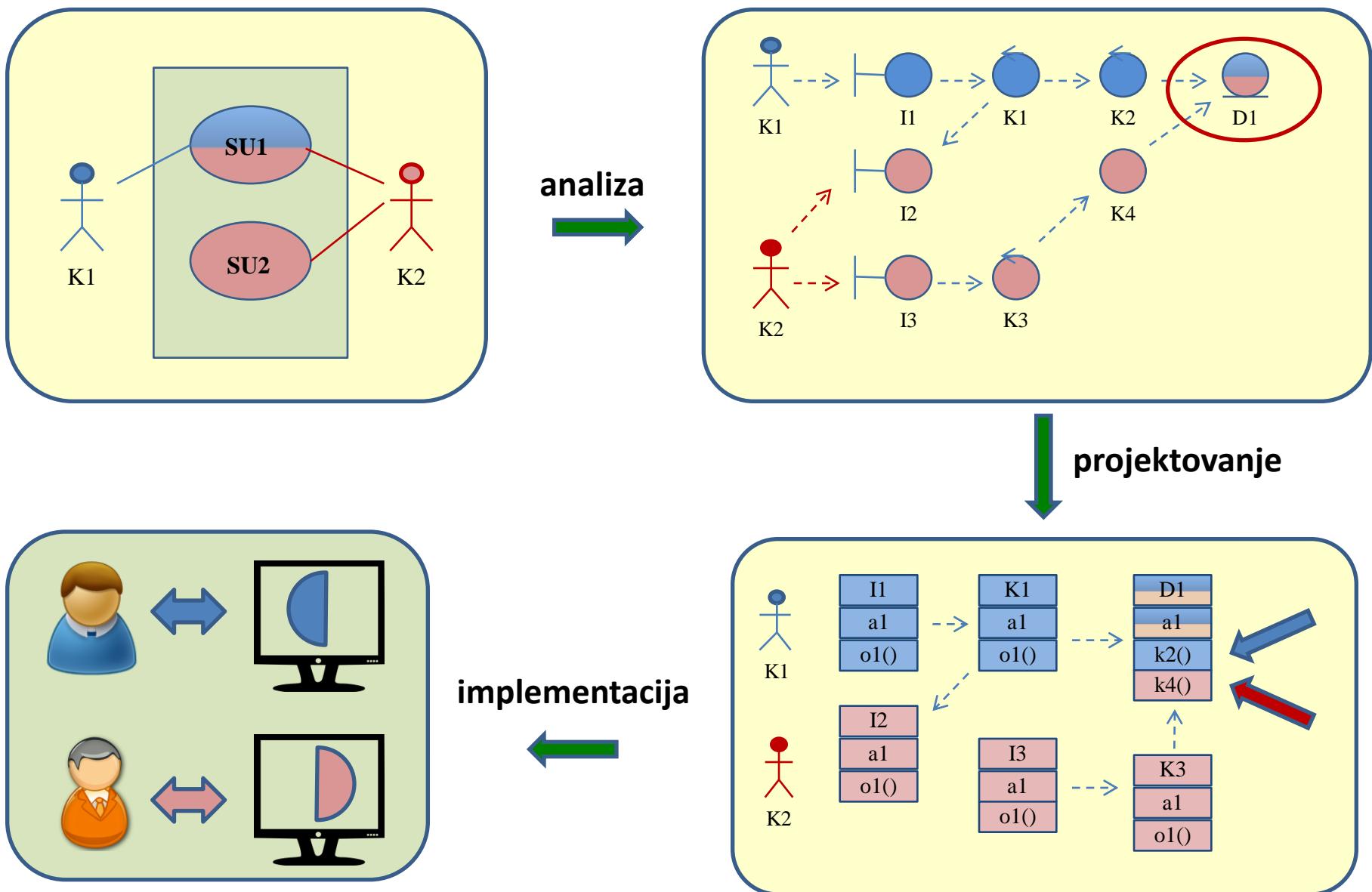


# 5. Kontrola pristupa

## Kontrola pristupa objektima

- U višekorisničkim sistemima **različiti učesnici imaju različita prava pristupa** različitim funkcionalnostima i podacima.
  - **Koji su mehanizmi za autentikaciju korisnika?**
    - kredencijali (*username/password*), hardverski tokeni, ...
  - **Koji korisnici imaju pravo pristupa kojim klasama?**
  - **Kako se objekti štite od neautorizovanog pristupa?**
- **Kako se to modeluje?**
  - **Tokom analize:**
    - različiti učesnici vežu se sa pripadajućim slučajevima upotrebe
  - **Tokom projektovanja sistema:**
    - identifikacija objekata sa dijeljenim pristupom (višekorisnički pristup)
    - zavisno od sigurnosnih zahtjeva, definišu se mehanizmi za autentikaciju učesnika i enkripciju podataka

# 5. Kontrola pristupa



# 5. Kontrola pristupa

## Globalna matrica pristupa (*Global Access Matrix*) [Lampson, 1971]

- Služi za modelovanje statičke kontrole prava pristupa učesnika pojedinim klasama
- **Tipično: red = učesnik, kolona = klasa** (za koju definišemo prava pristupa)
- **Pravo pristupa:** lista operacija (u ćeliji matrice) koje učesnik može da izvrši nad objektima date klase

učesnici	klase			
	Klasa 1	Klasa 2	...	Klasa $m$
Učesnik 1	operacija1() operacija2() ...	operacija3()		x
...				
Učesnik $n$	operacija3() operacija5()	operacija1()		operacija1()

- **Mehanizmi za reprezentaciju (implementaciju) matrice pristupa:**
  - **globalna tabela prava pristupa** (*global access table – GAT*)
  - **lista prava pristupa** (*access control list – ACL*)
  - **lista mogućnosti korisnika** (*user capability list – UCL*)

# 5. Kontrola pristupa

## Mehanizmi za reprezentaciju (implementaciju) matrice pristupa

### 1. Globalna tabela prava pristupa (Global Access Table – GAT)

- Svaka ćelija matrice pristupa reprezentuje se trojkom  
*< učesnik, klasa, operacija >*

učesnici	klase	
	Klasa 1	Klasa 2
Učesnik 1	operacija1() operacija3()	operacija2()
Učesnik 2	operacija2() operacija3()	operacija1()



učesnik	klasa	operacija
Učesnik 1	Klasa 1	operacija1()
Učesnik 1	Klasa 1	operacija3()
Učesnik 1	Klasa 2	operacija2()
Učesnik 2	Klasa 1	operacija2()
Učesnik 2	Klasa 1	operacija3()
Učesnik 2	Klasa 2	operacija1()

- Provjera da li učesnik ima pravo pristupa nekoj operaciji vrši se pretragom trojki u globalnoj tabeli – ako nema odgovarajuće trojke, učesnik nema pravo pristupa
- Implementacija globalne tabele (tipično) zahtijeva mnogo prostora (zavisi od broja učesnika, klase i operacija u klasama)

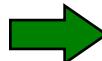
# 5. Kontrola pristupa

## Mehanizmi za reprezentaciju (implementaciju) matrice pristupa

### 2. Lista prava pristupa (Access Control List – ACL)

- Svaku klasu karakteriše lista parova <učesnik, operacija>

učesnici	klase	
	Klasa 1	Klasa 2
Učesnik 1	operacija1() operacija3()	operacija2()
Učesnik 2	operacija2() operacija3()	operacija1()



Klasa 1	
Učesnik 1	operacija1()
Učesnik 1	operacija3()
Učesnik 2	operacija2()
Učesnik 2	operacija3()

Klasa 2	
Učesnik 1	operacija2()
Učesnik 2	operacija1()

- Svaki put kad se pristupa nekom objektu, provjerava se da li pripadajuća lista prava pristupa sadrži odgovarajući par <učesnik, operacija> – ako ne sadrži, učesnik nema pravo pristupa (kao što npr. recepcioner provjerava da li se na spisku rezervacija nalazi ime nekog gosta – ako se nalazi, gost će biti smješten)
- Kontrolne liste omogućavaju brzo dobijanje odgovora na pitanje “Ko ima pravo pristupa?”
- Tipične primjene:
  - operativni sistemi (kontrola pristupa datotekama, *active directory*)
  - DBMS (kontrola pristupa tabelama)

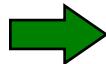
# 5. Kontrola pristupa

## Mehanizmi za reprezentaciju (implementaciju) matrice pristupa

### 3. Lista mogućnosti korisnika (user capability list – UCL)

- Svakog učesnika karakteriše lista parova `< klasa, operacija >`

učesnici	klase	
	Klasa 1	Klasa 2
Učesnik 1	operacija1() operacija3()	operacija2()
Učesnik 2	operacija2() operacija3()	operacija1()



Učesnik 1		Učesnik 2	
Klasa 1	operacija1()	Klasa 1	operacija2()
Klasa 1	operacija3()	Klasa 1	operacija3()
Klasa 2	operacija2()	Klasa 2	operacija1()

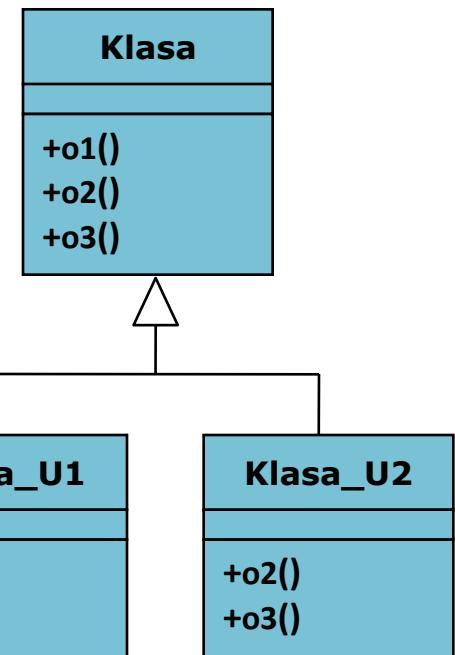
- Učesnik može da izvrši operaciju nad nekim objektom ako lista njegovih mogućnosti sadrži odgovarajući par `<klasa, operacija>` – ako ne sadrži, učesnik nema pravo pristupa (kao što npr. obezbjeđenje na stadionu omogućava ulaz na stadion posjetiocima koji imaju ulaznicu)
- Liste mogućnosti omogućavaju brzo dobijanje odgovora na pitanje “Kojim objektima učesnik ima pravo pristupa?”

# 5. Kontrola pristupa

## Strukturalna (statička) implementacija prava pristupa

- Svaki red u globalnoj matrici pristupa predstavlja pogled na sistem iz perspektive jednog učesnika – **koliko učesnika  
toliko pogleda**. Svi pogledi moraju da budu konzistentni.
- **Pogledi se često implementiraju specijalizacijom klasa** za svaki različit tip para **<učesnik, operacija>**
- **Prednosti:**
  - manja vjerovatnoća za neautorizovani pristup
- **Nedostaci:**
  - nefleksibilnost, potrebne izmjene modela i aplikacije za svaki novi tip para **<učesnik, operacija>**

učesnici	klase
	Klasa
U1	o1() o3()
U2	o2() o3()



# 5. Kontrola pristupa

## Dinamička kontrola prava pristupa

- Često učesnici istog tipa nemaju ista prava pristupa objektima iste klase!

Npr. u IS banke, brokeri (lični bankari) imaju pristup većem broju klijentskih računa.

Pravo na transakcije (uplate, isplate) na nekom klijentskom računu ima samo jedan broker (tačno određeni broker), dok drugi brokeri nemaju pravo transakcija na tom računu.

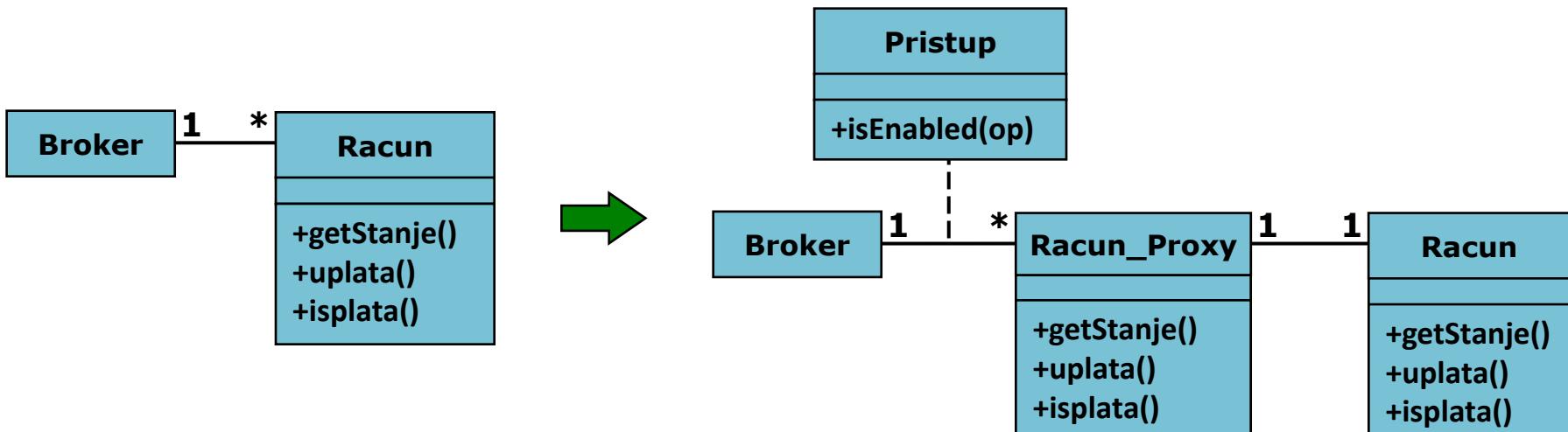
U ovom slučaju treba **dinamička kontrola prava pristupa**.

- Moguće rješenje: **proxy šablon**

Za svaki račun kreira se odgovarajući proksi koji ima ulogu kontrole pristupa.

Asocijacija Pриступ između brokera i proksija pokazuje kojem računu broker ima pravo pristupa.

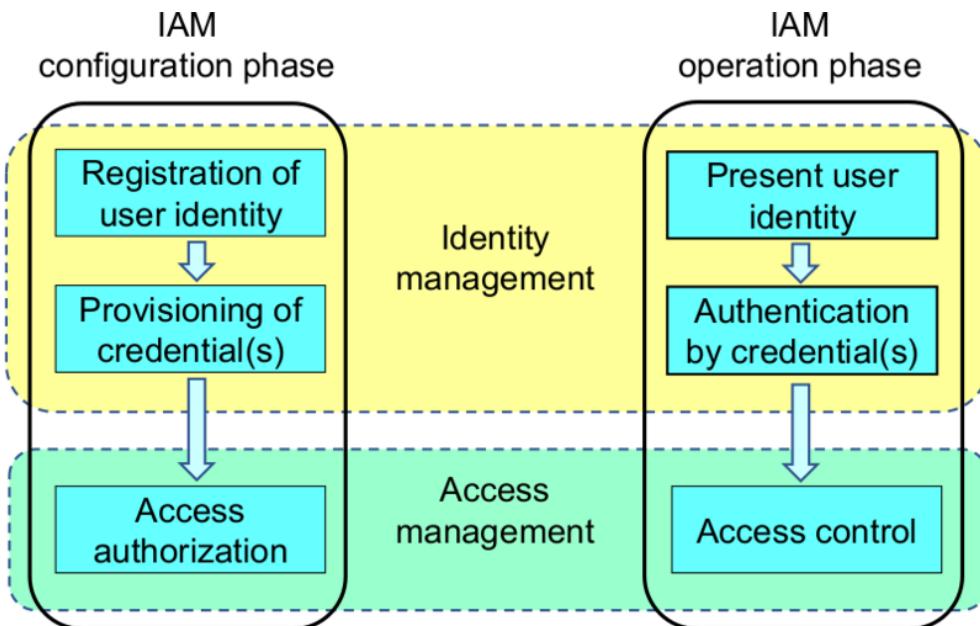
Da bi pristupio računu, broker prvo šalje poruku odnosnom proksiju. Potom proksi provjerava da li dati broker ima odgovarajuću asocijaciju sa proksijem (pristup sadrži listu dozvoljenih operacija), odnosno da li ima pravo izvršavanja tražene operacije. Ako ima broker to pravo, proksi proslijeđuje poruku računu.



# 5. Kontrola pristupa

## Napredna rješenja za kontrolu pristupa

- **Identity Management (IdM) / Identity and Access Management (IAM or IdAM)**
  - is a framework of policies and technologies to ensure that the right users have the appropriate access to technology resources



- **Configuration phase**
  - first registering and authorizing access rights (assisted / self-service)
- **Operation phase**
  - identifying, authenticating and controlling individuals or groups of people to have access to applications, systems or networks based on previously authorized access rights

## Typical IAM System capabilities

- Authentication & Authorization & Roles & Delegation

# 5. Kontrola pristupa

## Napredna rješenja za kontrolu pristupa

- **Identity Management (IdM) / Identity and Access Management (IAM or IdAM)**

### Standardization

- ISO/IEC 24760-1 A framework for identity management –  
Part 1: Terminology and concepts
- ISO/IEC 24760-2 A Framework for Identity Management –  
Part 2: Reference architecture and requirements
- ISO/IEC 24760-3 A Framework for Identity Management –  
Part 3: Practice
- ISO/IEC 29115 Entity Authentication Assurance
- ISO/IEC 29146 A framework for access management
- ISO/IEC CD 29003 Identity Proofing and Verification
- ISO/IEC 29100 Privacy framework
- ISO/IEC 29101 Privacy Architecture
- ISO/IEC 29134 Privacy Impact Assessment Methodology

### Tools

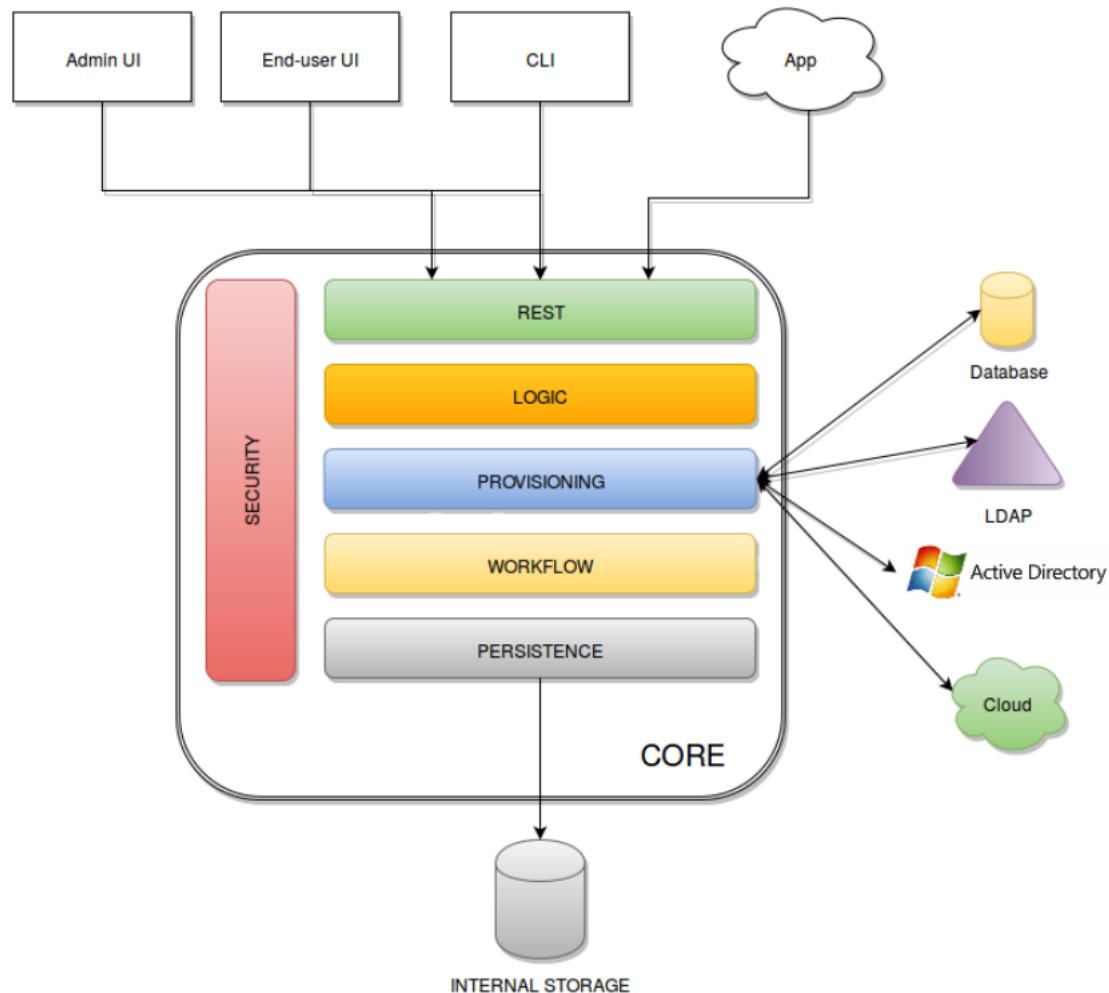
- Auth0, SpectralOPS
- AWS Identity & Access Management
- Microsoft Azure Active Directory
- Google Cloud IAM
- IBM IAM
- Oracle Identity Management
- ...
- Apache Syncope (open-source)

# 5. Kontrola pristupa

## Napredna rješenja za kontrolu pristupa

### Apache Syncpe

- Open-source IAM system
- Admin UI / End-user UI
- Third-party applications
  - Eclipse IDE plug-in
  - Netbeans IDE plug-in
- Višeslojna arhitektura
- STORAGE (različiti DBMSs)
- API



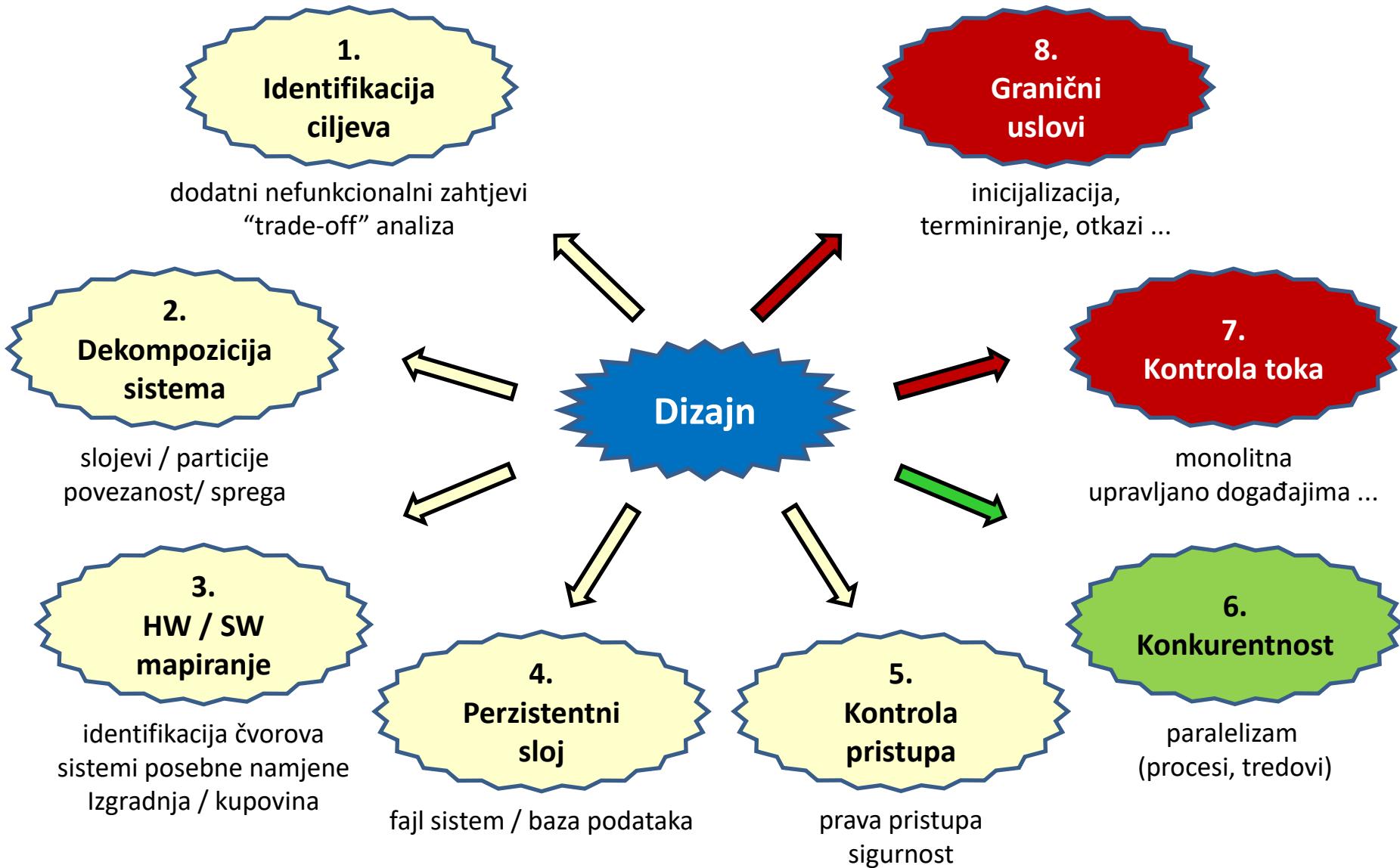
**UNIVERZITET U BANJOJ LUCI  
ELEKTROTEHNIČKI FAKULTET**

**Prof. dr Dražen Brđanin**

**PROJEKTOVANJE SOFTVERA  
/konkurentnost/**

**Banja Luka  
2024.**

# 8 bitnih aktivnosti u projektovanju



# 6. Konkurentnost

## Sadržaj prezentacije

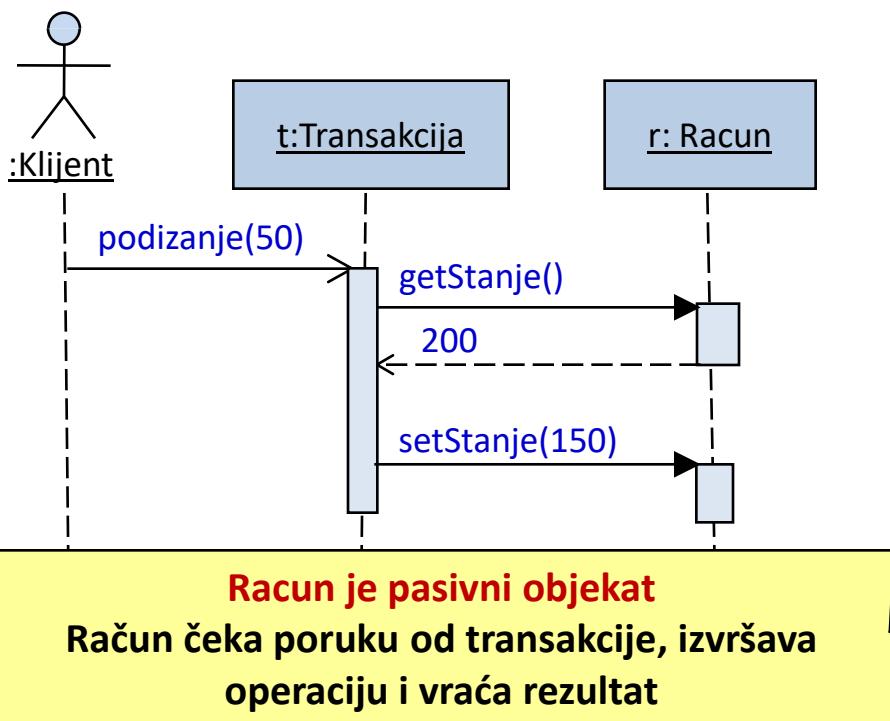
- pasivni i aktivni objekti
- sekvencijalne i konkurentne aplikacije
- problemi u komunikaciji konkurentnih procesa
- sinhronizacija konkurentnih procesa
- razmjena podataka između konkurentnih procesa
- identifikacija konkurentnosti u dinamičkom modelu sistema
- mapiranje dinamičkog modela
- implementacija konkurentnosti

# 6. Konkurentnost

## Pasivni i aktivni objekti

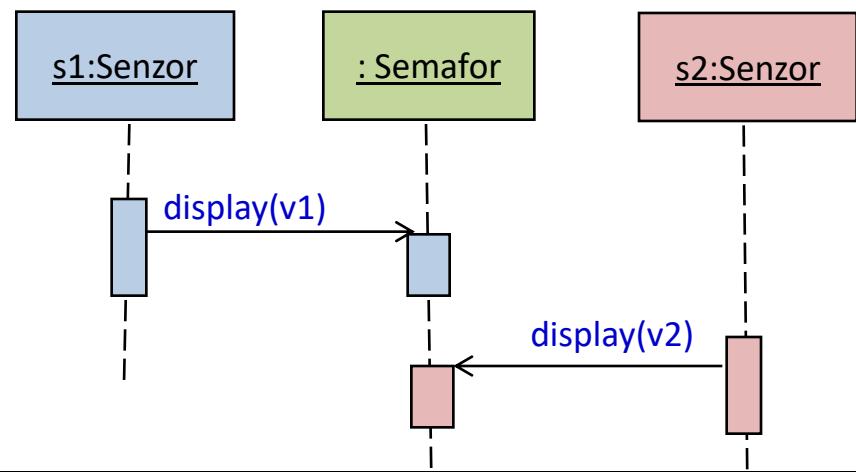
### Pasivni objekti

- objekti koji čekaju poruku od drugog objekta i izvršavaju operaciju na zahtjev
- nikad ne iniciraju akcije



### Aktivni objekti

- objekti koji iniciraju akcije i koji se izvršavaju nezavisno od drugih objekata
- još se nazivaju i konkurentni objekti (konkurentni procesi)



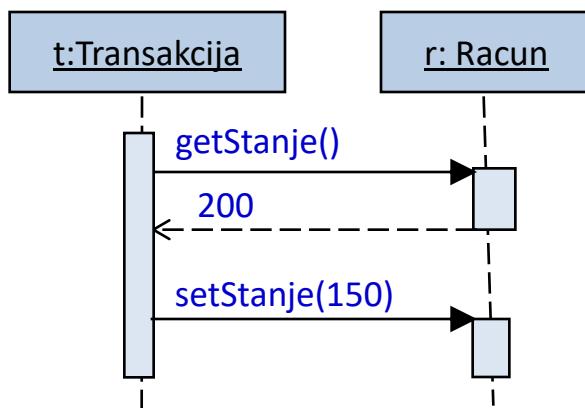
**Senzori su aktivni objekti**  
Senzori mogu istovremeno, nezavisno jedan od drugog, da mijene vrijednosti i šalju semaforu

# 6. Konkurentnost

## Sekvencijalne i konkurentne aplikacije

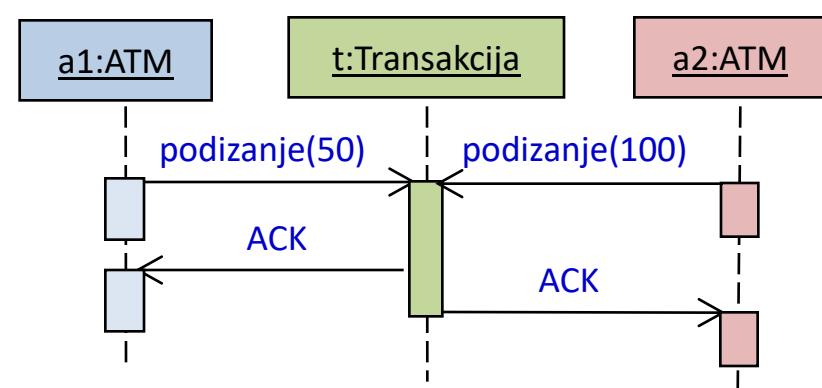
### Sekvencijalne aplikacije

- sekvencijalni program čine **samo pasivni objekti**
- Kad neki objekat pozove operaciju na drugom objektu, kontrola se prenosi u pozvanu operaciju i pozivajući objekat čeka rezultat da bi nastavio izvršavanje
- U sekvencijalnim programima u komunikaciji se primjenjuje komunikacioni obrazac **sinhrona poruka sa odgovorom**



### Konkurentne aplikacije

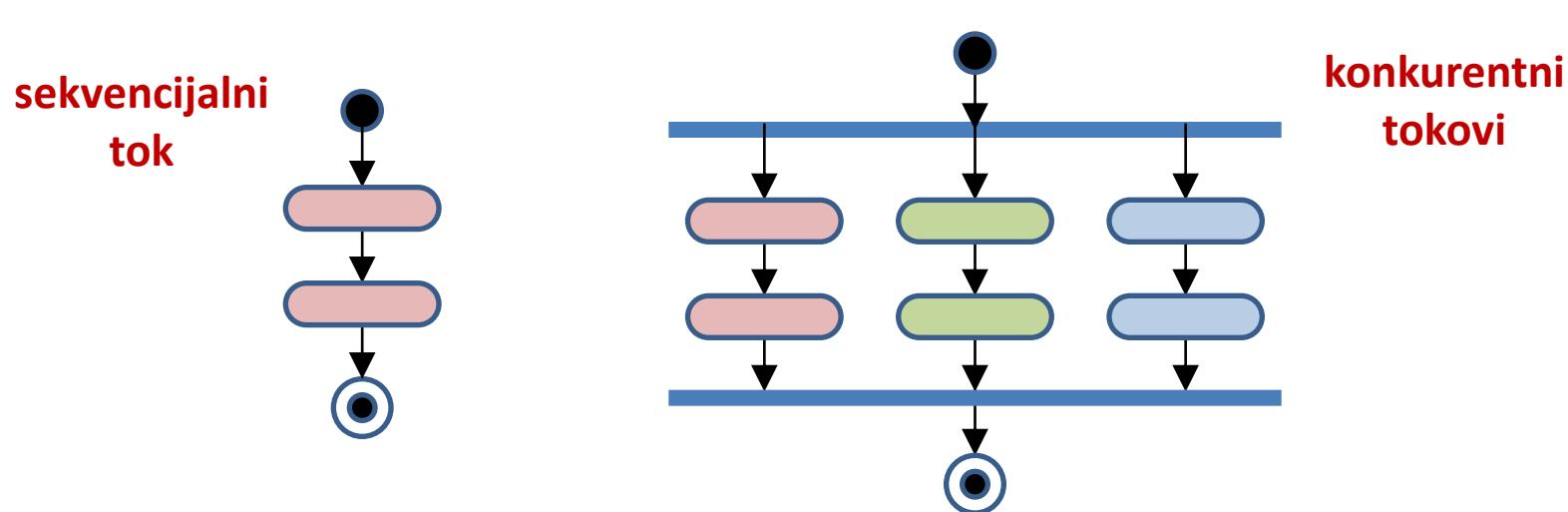
- konkurentni program ima više aktivnih (konkurentnih) objekata, pri čemu svaki aktivni objekat ima sopstveni kontrolni tok
- U konkurentnim programima **primjenjuje se asinhrona komunikacija**
- Konkurentni objekat šalje asinhronu poruku drugom objektu i nastavlja izvršavanje (odredišni objekat ima bafer u kojem drži sve primljene poruke, koje procesira kad bude raspoloživ)



# 6. Konkurentnost

## Konkurentni objekti

- Alternativno se nazivaju: **aktivni objekti, konkurentni procesi, konkurentni zadaci, niti**
- **Konkurentni objekat ima sopstvenu nit kontrole toka i može da se izvršava nezavisno od drugih objekata**
- **Pasivni objekti nemaju sopstvenu nit kontrole toka, već se izvršavaju unutar niti nekog aktivnog objekta**
- **Konkurentni objekat predstavlja izvršavanje jednog sekvencijalnog programa ili jedne sekvencijalne komponente konkurentnog programa**
- Svaki konkurentni objekat ima svoj sekvencijalni tok



# 6. Konkurentnost

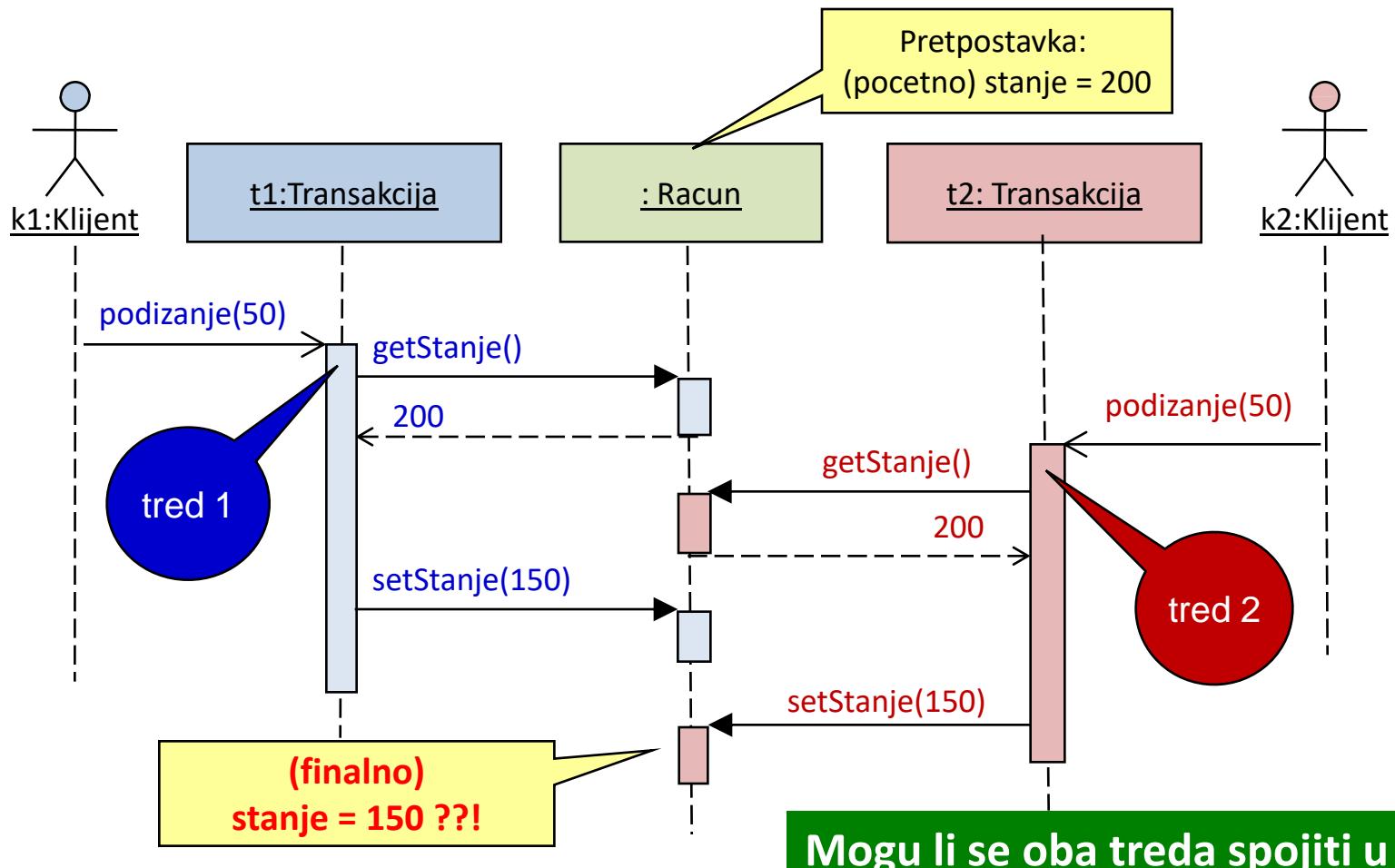
## Problemi u komunikaciji konkurentnih procesa

- Konkurentni procesi izvršavaju se asinhrono (razlicitim brzinama)
- Konkurentni objekti međusobno asinhrono komuniciraju (s vremena na vrijeme) i moraju da se sinhronizuju
- U komunikaciji konkurentnih objekata javljaju se problemi koji ne postoje u sekvensijalnim procesima:
  - **uzajamna isključivost (*mutual exclusion*)**
    - kad više konkurentnih objekata treba ekskluzivan pristup istom resursu (npr. pristup istom dijeljenom podatku ili fizičkom uređaju)
  - **proizvođač-potrošač (*producer-consumer*)**
    - kad jedan aktivni objekat šalje podatke drugom aktivnom objektu
  - **sinhronizacija**
    - kad izvršavanje više konkurentnih objekata treba da se sinhronizuje

# 6. Konkurentnost

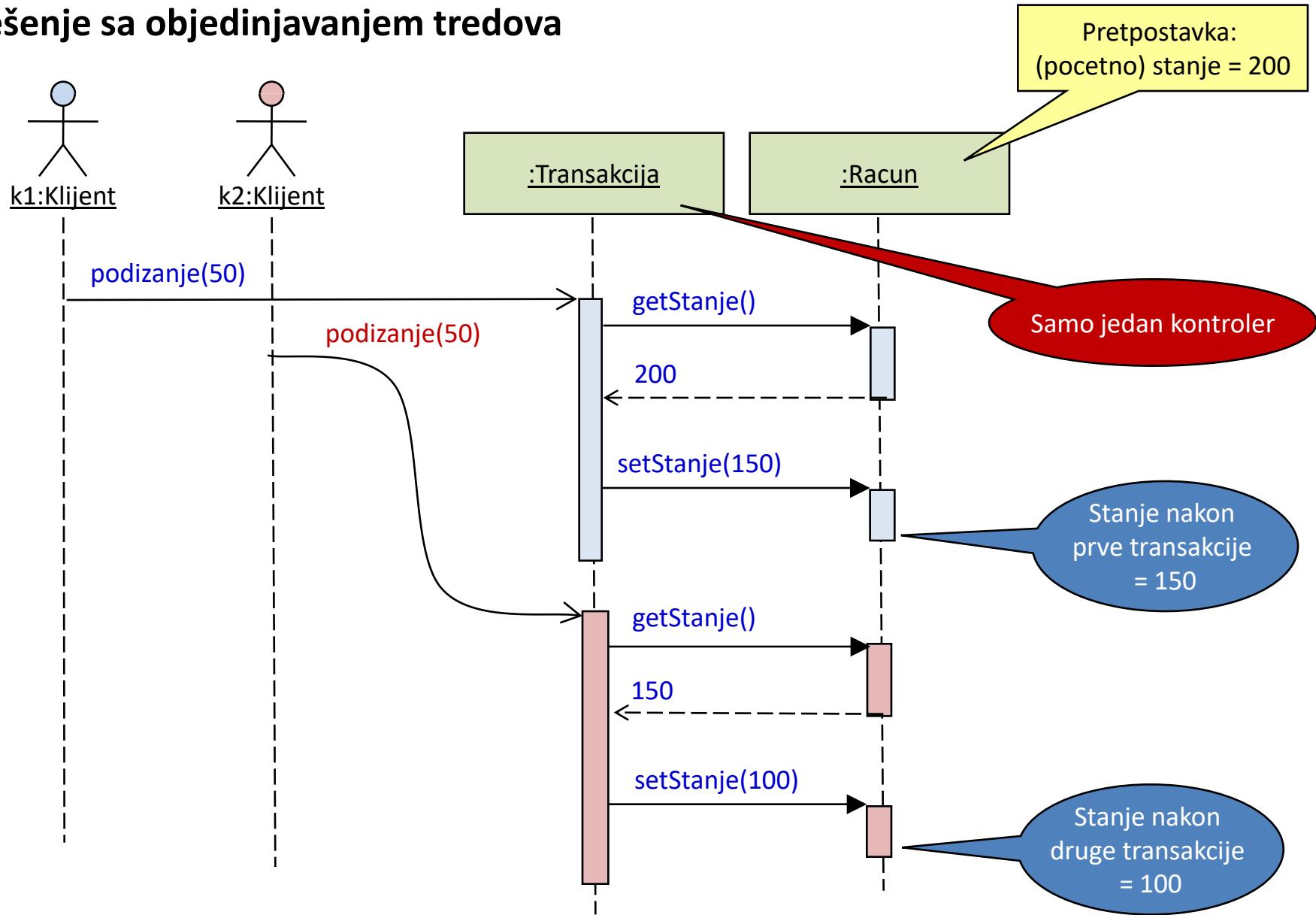
## Uzajamna isključivost konkurentnih procesa

- više konkurentnih objekata treba ekskluzivan pristup istom resursu  
(npr. pristup istom dijeljenom podatku ili fizičkom uređaju)



# 6. Konkurentnost

## Rješenje sa objedinjavanjem tredova



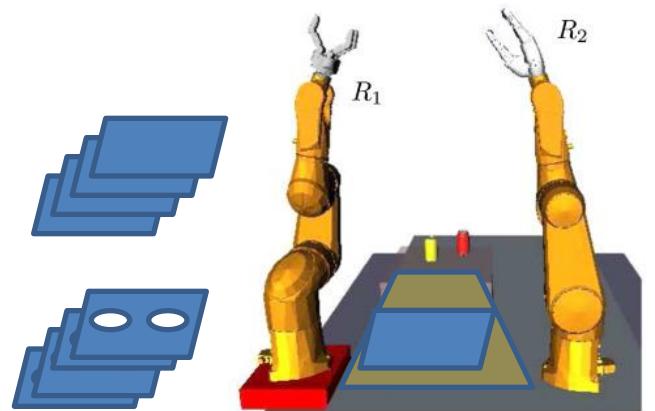
# 6. Konkurentnost

## Sinhronizacija konkurentnih procesa

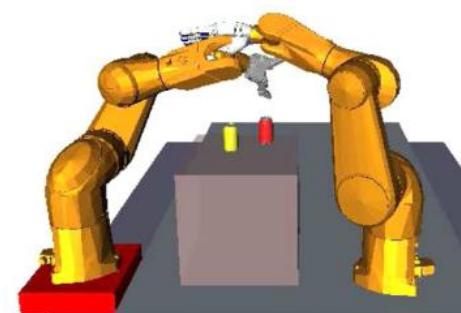
- Sinhronizacija više konkurentnih procesa bez međusobne razmjene podataka
- Mehanizam za sinhronizaciju – signaliziranje događaja
  - Odredišni objekat je u stanju čekanja signala (operacija wait)
  - Izvorni objekat signalizira događaj – šalje asinhronu poruku (signal, notify)

### Primjer: sinhronizacija dva robota

- ① robot  $R_1$  postavlja objekat
- ② robot  $R_2$  buši rupe
- ③ robot  $R_1$  odnosi objekat i postavlja novi



Potencijalni problem ako nema sinhronizacije, jer se radni prostor oba robota preklapa pa može da dođe do havarije



# 6. Konkurentnost

## Primjer: sinhronizacija dva robota

- ① robot  $R_1$  postavlja objekat
- ② robot  $R_2$  buši rupe
- ③ robot  $R_1$  odnosi objekat i postavlja novi

## Rješenje: signalizacija događaja

- $R_1$  postavi objekat, izmakne se i signalizira da je objekat spremna (*ObjectReady*)
- $R_2$  prima signal *ObjectReady* i iz stanja čekanja započinje manipulaciju
- $R_2$  završi bušenje, izmakne se i signalizira da je objekat završen (*ObjectCompleted*)
- $R_1$  prima signal *ObjectCompleted* i odnosi završeni objekat

### Proces $R_1$

```
while (true)
{
    postaviNoviDio(),
    odmakniSe()
signal(ObjectReady)
wait(ObjectCompleted)
    odnesiZavrseniDio()
}
```

### Proces $R_2$

```
while (true)
{
    wait(ObjectReady)
    busenje(),
    odmakniSe()
signal(ObjectCompleted)
}
```

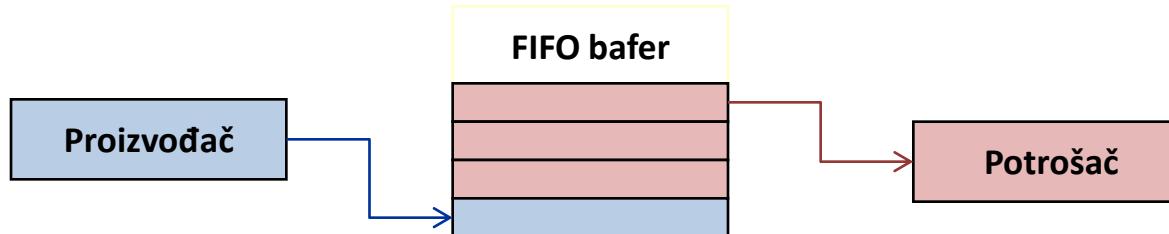
## Dva sinhronizaciona problema (kolizije):

- $R_1$  mora da postavi objekat i da se izmakne prije nego što  $R_2$  uđe u radni p.
- $R_2$  mora da završi bušenje i da se izmakne prije nego što  $R_1$  uđe u radni p.

# 6. Konkurentnost

## Proizvođač-potrošač (*producer-consumer*)

- Proizvođač generiše podatke koje prima potrošač
- Potrebna je odgovarajuća sinhronizacija, kako bi došlo do **razmjene podataka**
  - proizvođač mora biti spremjan da šalje podatke, a potrošač mora biti spremjan da ih primi
  - ako je potrošač spremjan za prijem, a proizvođač još ne može da ih pošalje, potrošač mora da čeka
  - ako proizvođač može da pošalje podatke prije nego što ih potrošač može primiti, tada proizvođač mora da čeka ili podaci moraju biti baferovani prije nego što ih potrošač preuzme
- Tipična rješenja:
  - asinhrone poruke proizvođač → potrošač (sa gubitkom informacija)
  - FIFO bafer između proizvođača i potrošača (bez gubitka informacija)



# 6. Konkurentnost

## Identifikacija konkurentnosti u dinamičkom modelu sistema

- Za dva objekta kažemo da su **inherentno konkurentna** ako istovremeno mogu da prime neku poruku ili signal, nezavisno jedan od drugog:
  - to može da bude ista poruka/signal ili nezavisne poruke/signali
- **Osnov za identifikaciju konkurentnosti:**
  - objekti u dijagramu sekvene koji mogu simultano da primaju poruke/signale

## Pitanja prilikom identifikacije konkurentnosti

- **Da bismo identifikovali konkurentne tredove treba postavljati sljedeća pitanja:**
  - Da li sistem omogućava (istovremeni) pristup većem broju korisnika?
  - Koji kontrolni objekti mogu da se izvršavaju nezavisno jedan od drugog?
  - Može li se neki upit prema sistemu dekomponovati u više upita i da li se ti upiti mogu rješavati u paraleli?
    - npr. pretraživanje u distribuiranoj bazi podataka
    - npr. segmentirano pretraživanje slika
    - ...

# 6. Konkurentnost

## Mapiranje dinamičkog modela u tredove

- Inherentno konkurentni objekti tipično se raspoređuju u različite upravljačke niti (tredove)
- Objekti sa uzajamno isključivom (*mutual exclusive*) aktivnošću raspoređuju se u isti tred (npr. pristup istom objektu koji ima za cilj promjenu stanja)
- **Tred** je putanja kroz stanja u dijagramu stanja u kojima je neki objekat stalno aktivan:
  - **Tred se zadržava** u nekom stanju sve dok objekat čeka:
    - rezultat poruke koju je poslao drugom objektu, ili
    - dok čeka neku poruku drugog objekta.
  - **Cijepanje treda:** kad objekat šalje ne-blokirajuću poruku (signal) drugom objektu

# 6. Konkurentnost

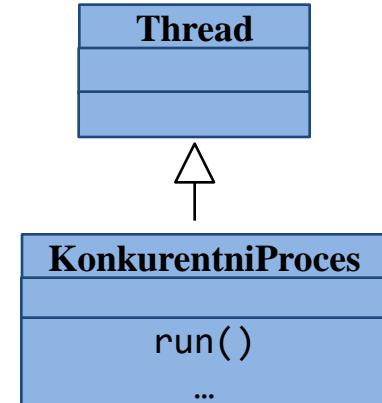
## Implementacija konkurentnosti

- Konkurentnost može da se realizuje u sistemima kod kojih je omogućena:
  - **fizička konkurentnost**
    - tredovi se raspoređuju na različite hardverske resurse
    - višeprocesorske mašine, klasteri, ...
  - **logička konkurentnost**
    - softverska realizacija tredova (neki jezici imaju mogućnost, npr. java)

---

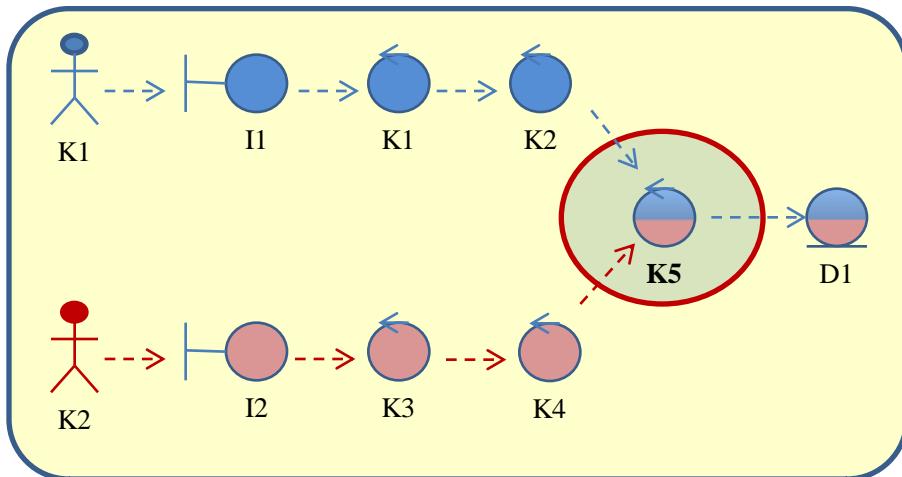
```
public class KonkurentniProces extends Thread
{
    public void run()
    {
        while (true)
            // task body
    }
}
```

---

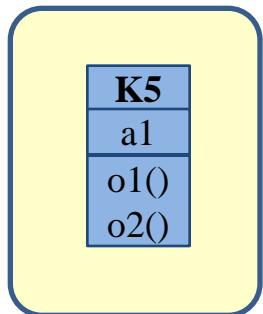


```
// instanciranje objekta i pokretanje niti
KonkurentniProces kp = new KonkurentniProces();
kp.start(); // poziva se run() metoda i sad imamo dvije niti
```

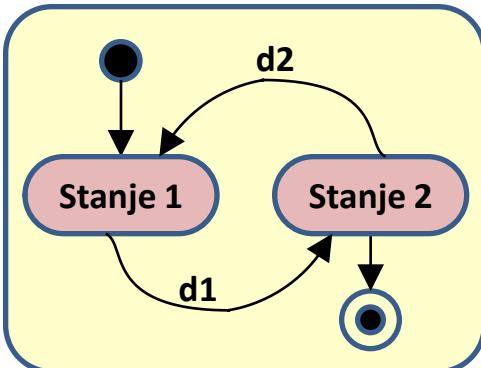
# 6. Konkurentnost



dijagram  
klasa

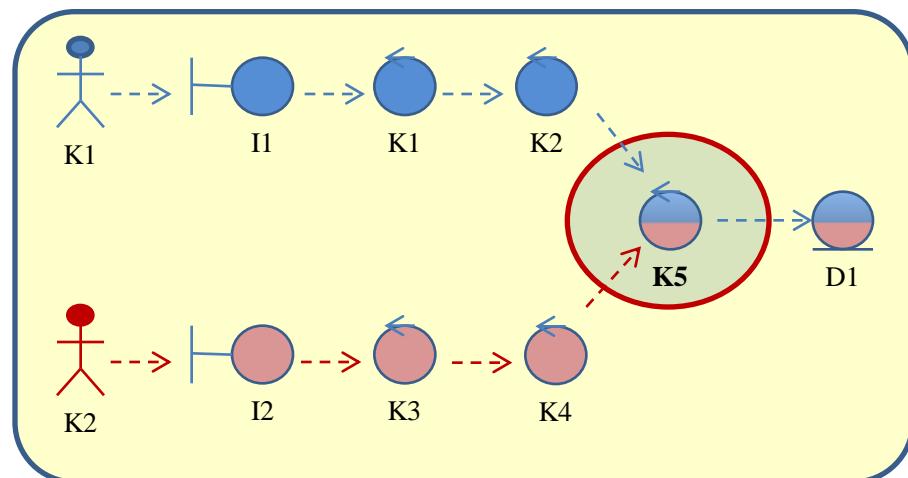


dijagram stanja



```
public class K5 extends Thread
{
    private int state=1;
    //...
    public void run()
    {
        while (true)
        {
            if (baf.get()==1 && state==1)
            {
                stanje=2;
                // ...
            }
            if (baf.get()==2 && state==2)
            {
                stanje=1;
                // ...
            }
        }
    }
}
```

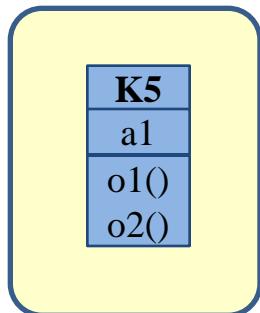
# 6. Konkurentnost



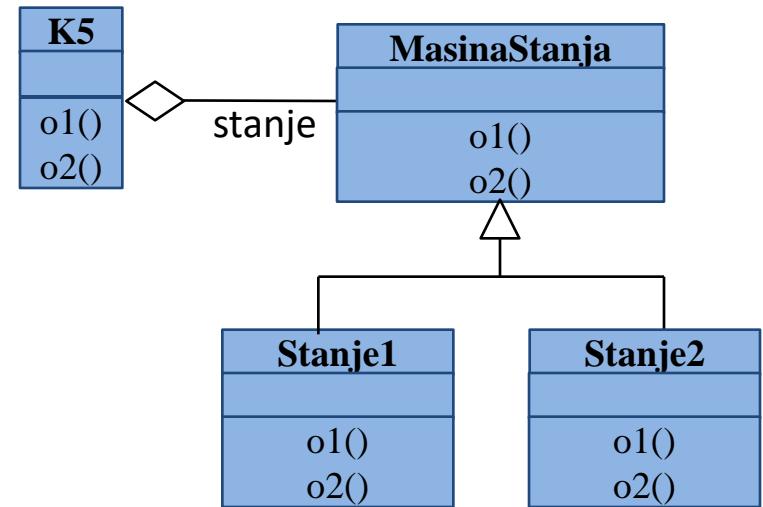
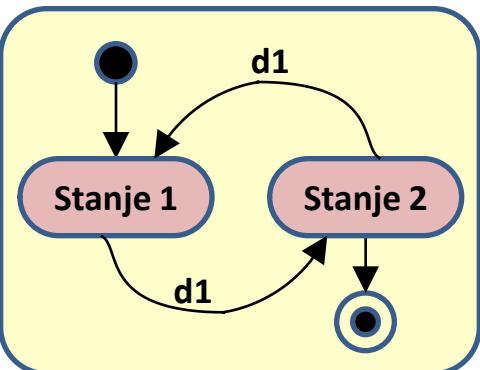
**STATE**  
projektni obrazac

Detaljnije će biti prikazano  
kod projektnih obrazaca

dijagram  
klasa



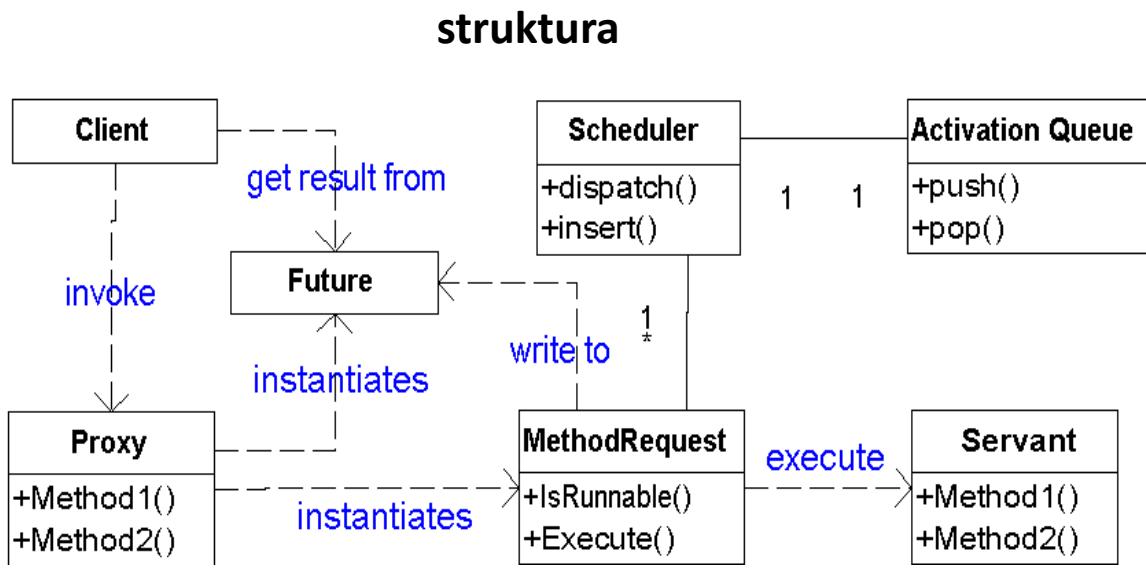
dijagram stanja



# 6. Konkurentnost

## Projektovanje i implementacija aktivnih objekata

- Projektni obrazac: **Active object (Actor Pattern)**
  - uobičajena primjena u distribuiranim sistemima koji zahtijevaju višenitne servere
  - omogućava razdvajanje aplikativne logike za poziv i izvršavanje metode



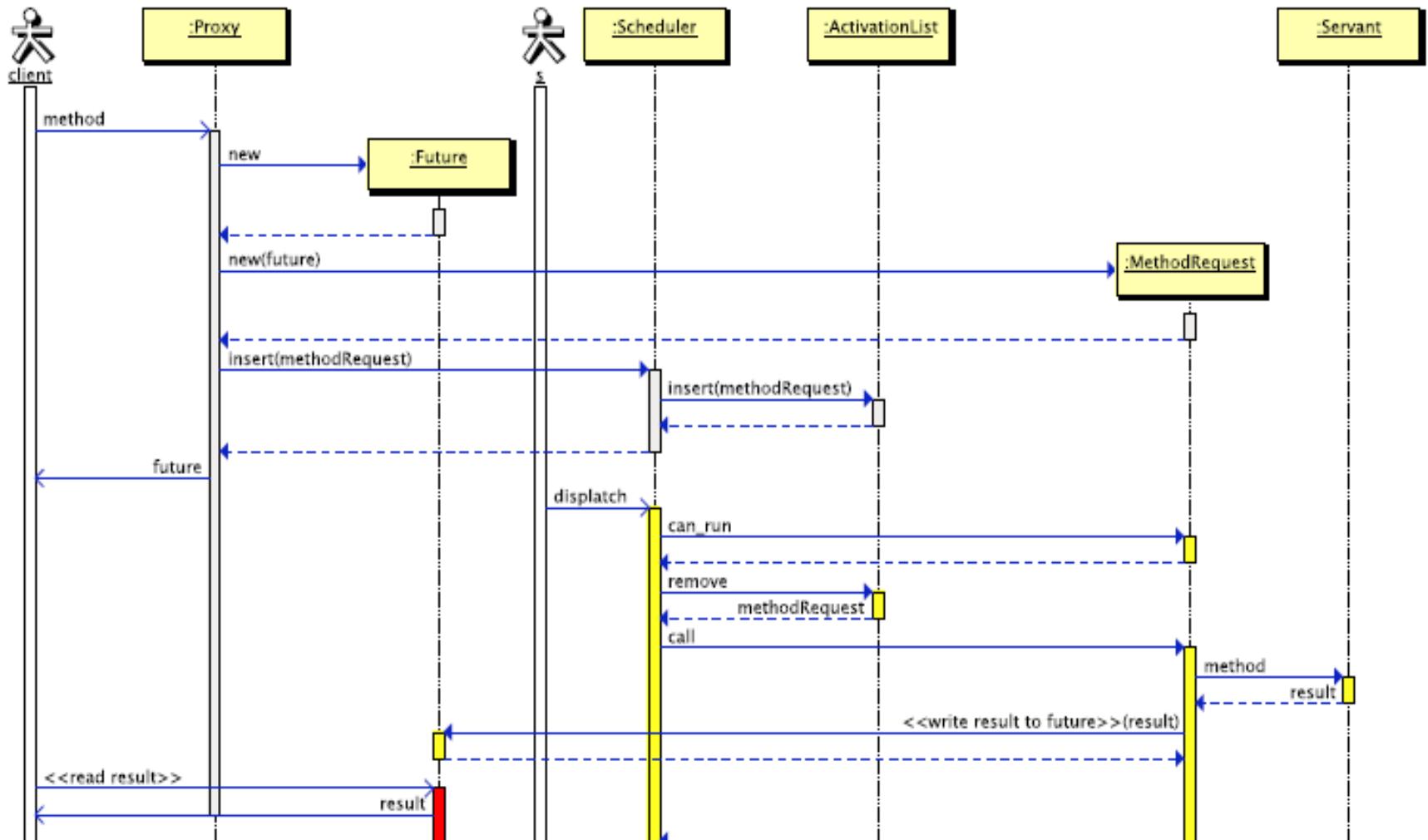
- **Proxy** pruža odgovarajući interfejs klijentu – omogućava poziv metoda
- **MethodRequest** je objekat koji reprezentuje poziv metode (svaki poziv metode inkapsulira se u objekat)
- **Scheduler** upravlja izvršavanjem zahtjeva (raspoređuje ih na servanta kad su spremni za izvršavanje)
- **ActivationQueue** sadrži sve zahtjeve koji čekaju na izvršavanje
- **Servant** implementira metode
- **Future** omogućava klijentu da preuzme rezultat poziva

# 6. Konkurentnost

## Projektovanje i implementacija aktivnih objekata

- Projektni obrazac: *Active object (Actor Pattern)*

Interakcija objekata

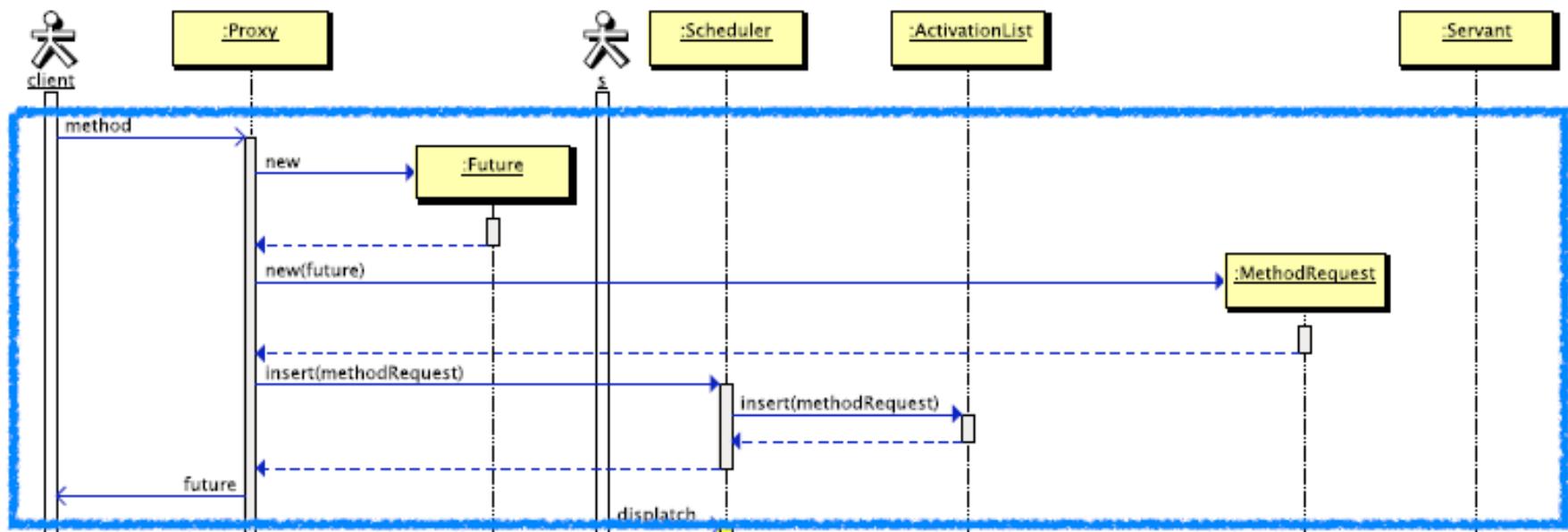


# 6. Konkurentnost

## Projektovanje i implementacija aktivnih objekata

- Projektni obrazac: *Active object (Actor Pattern)*

Method scheduling



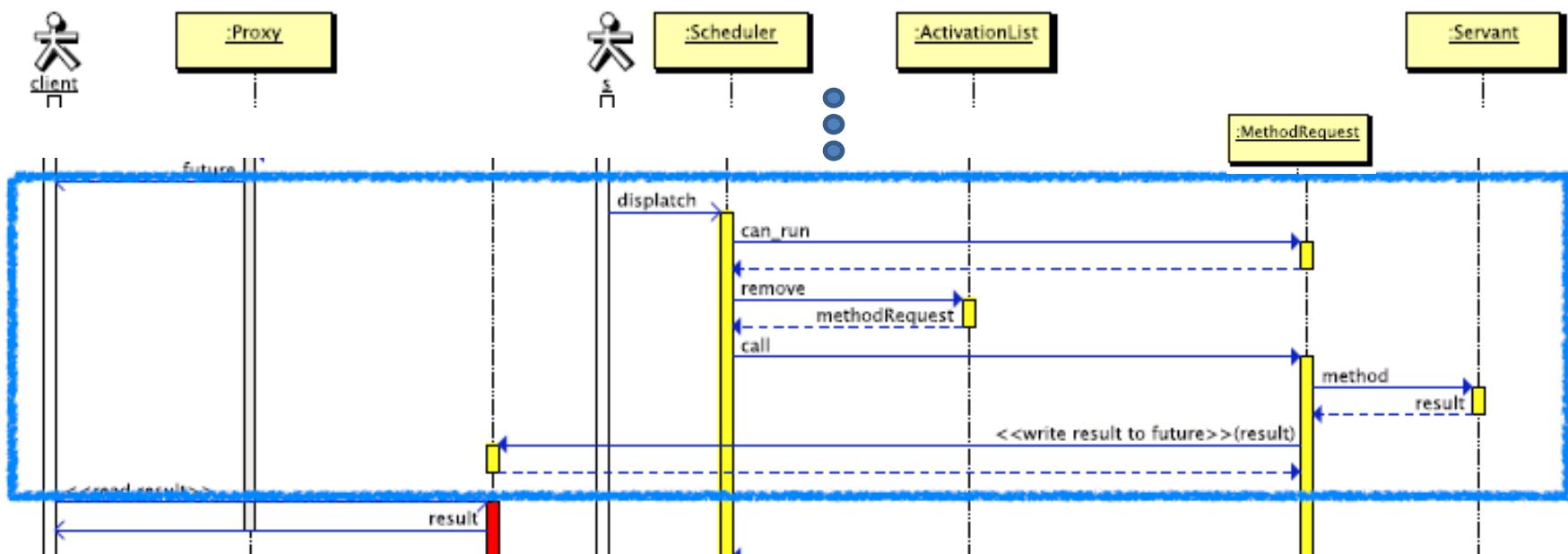
1. Klijent poziva željenu metodu Proxy objekta
2. Proxy kreira Future objekat koji će vratiti rezultat klijentu (ako bude rezultata)
3. Proxy kreira MethodRequest objekat i šalje mu referencu na Future (kako bi se u Future kasnije mogao smjestiti rezultat)
4. Proxy šalje zahtjev Scheduler-u, koji smješta taj zahtjev u ActivationList (Queue)
5. Ako poziv metode treba da vrati rezultat, Future objekat se vraća klijentu

# 6. Konkurentnost

## Projektovanje i implementacija aktivnih objekata

- Projektni obrazac: *Active object (Actor Pattern)*

Method dispatch



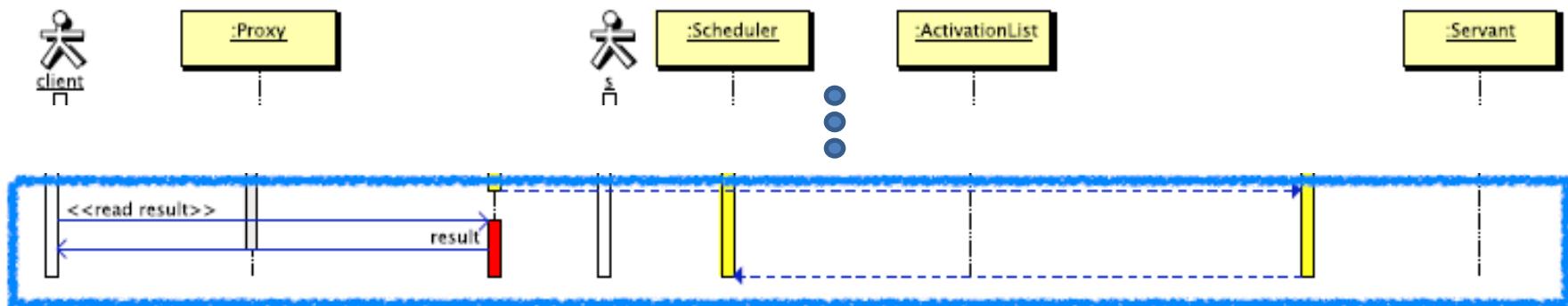
1. Scheduler provjerava da li se može izvršiti zahtjev
2. Ako zahtjev može da se izvrši, zahtjev se uzima iz reda (brisanje iz reda) i pokreće izvršavanje
3. Zahtjev poziva korespondentnu metodu servanta
4. Ako servant vrati rezultat, MethodRequest upisuje rezultat u korespondentni Future objekat

# 6. Konkurentnost

## Projektovanje i implementacija aktivnih objekata

- Projektni obrazac: *Active object (Actor Pattern)*

Results retrieving



1. Nakon što je rezultat upisan u Future, klijent može da ga pročita

Čitanje rezultata je asinhrono u odnosu na poziv metode

# 6. Konkurentnost

## Implementacija konkurentnosti – definicija aktivnih objekata

- Dvije vrste aktivnih objekata: **Runnable** i **Callable**

### **Runnable – komanda**

- ne može da vrati rezultat

```
public class Task implements Runnable
{
    // deklaracija atributa

    // definicija operacija

    @Override
    public void run()
    {
        try
        {
            // aplikativna logika
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

### **Callable**

- može da vrati rezultat

```
import java.util.concurrent.Callable;

public class Task implements Callable<T>
{
    private T atr;

    public Task(T arg)
    {
        atr = arg;
    }

    @Override
    public T call() throws Exception
    {
        // izracunaj rezultat
        return rezultat;
    }
}
```

# 6. Konkurentnost

## Primjer implementacije Producer-Consumer

```
// producer
class P implements Runnable
{
    private List<Integer> red;

    public P(List<Integer> r)
    {
        this.red = r;
    }

    @Override
    public void run()
    {
        int brojac = 0;
        while (true)
        {
            try { puni(brojac++); }
            catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

```
private void puni(int i) throws
    InterruptedException
{
    synchronized (red) // zakljucavanje reda
    {
        while (red.size() == 5)
        {
            System.out.println("Red je pun. " +
                Thread.currentThread().getName() +
                " ceka, size: " + red.size());
            red.wait(); // ceka dok je red pun
            // otključava red i "spava" dok ne
            // dobije notifikaciju da je
            // oslobođen prostor
        }

        Thread.sleep(1000); // ceka 1s
        red.add(i);
        System.out.println("Upisano: " + i);
        red.notifyAll(); // budi sve tredove
        // koji su cekali
    }
}
```

# 6. Konkurentnost

## Primjer implementacije Producer-Consumer

```
// consumer
class C implements Runnable
{
    private List<Integer> red;

    public C(List<Integer> r)
    {
        this.red = r;
    }

    @Override
    public void run()
    {
        while (true)
        {
            try { prazni(); }
            catch (InterruptedException ex)
            {
                ex.printStackTrace();
            }
        }
    }
}
```

```
private void prazni() throws
    InterruptedException
{
    synchronized (red) // zaključava red
    {
        while (red.isEmpty())
        {
            System.out.println("Red je prazan. "
                + Thread.currentThread().getName()
                + " ceka, size: " + red.size());
            red.wait(); // ceka dok je prazan
            // otključava red i "spava" dok ne
            // dobije notifikaciju da je
            // nesto upisano u bafer
        }
        Thread.sleep(1000);
        int i = (Integer) red.remove(0);
        System.out.println("Procitano: " + i);
        red.notifyAll(); // budi sve tredove
        // koji su cekali
    }
}
```

# 6. Konkurentnost

## Primjer implementacije Producer-Consumer

```
// Demonstracija producer-consumer
public class PCdemo
{
    public static void main(String[] args)
    {

        List<Integer> red = new ArrayList<Integer>();

        System.out.println("Start.");

        Thread tP = new Thread(new P(red), "P");
        Thread tC = new Thread(new C(red), "C");
        tP.start();
        tC.start();
    }
}
```

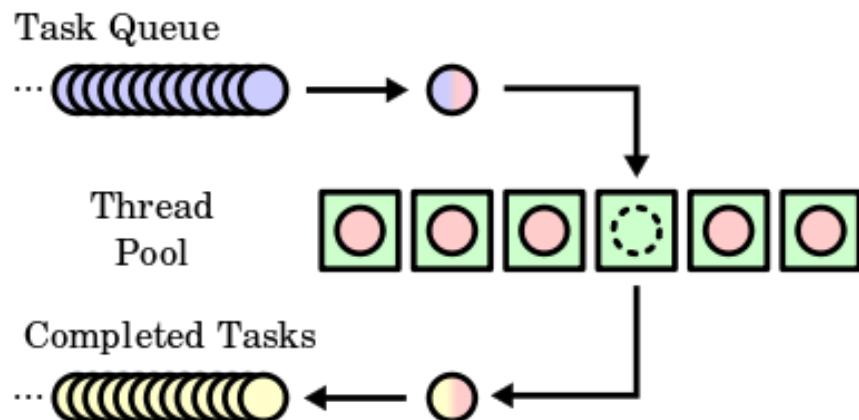
```
Start.
Upisano: 0
Procitano: 0
Red je prazan. C ceka, size: 0
Upisano: 1
Upisano: 2
Procitano: 1
Procitano: 2
Red je prazan. C ceka, size: 0
Upisano: 3
Upisano: 4
Procitano: 3
Upisano: 5
Procitano: 4
Upisano: 6
Procitano: 5
Procitano: 6
Red je prazan. C ceka, size: 0
Upisano: 7
Procitano: 7
Red je prazan. C ceka, size: 0
...
...
```

# 6. Konkurentnost

## Implementacija konkurentnosti – upravljanje nitima u Javi

### – Executor framework

- Svako kreiranje nove niti je skupa operacija i (može da) degradira performanse
- *Executor framework* koristi **thread pool** za upravljanje konkurentnim taskovima



*Thread pool* je kolekcija niti koju executor koristi za izvršavanje konkurentnih taskova

*Thread pool* može imati fiksnu ili promjenljivu veličinu

Ako ima više taskova nego niti, taskovi se smještaju u *Task Queue* (FIFO bafer) i čekaju na izvršavanje

Kad neka nit završi izvršavanje taska, iz bafera može da uzme i izvrši sljedeći task

Kad se izvrše svi taskovi, niti ostaju aktivne i čekaju sljedeće taskove

# 6. Konkurentnost

## Implementacija konkurentnosti – upravljanje nitima u Javi

- **ThreadPoolExecutor** klasa (implementira Executor i ExecutorService interfejse)
- **ThreadPoolExecutor razdvaja kreiranje i izvršavanje konkurentnog objekta**
- **U korisničkoj aplikaciji treba:**
  - instancirati *thread pool*,
  - kreirati konkurentne objekte (tipa Runnable ili Callable)
  - poslati kreirane objekte u *thread pool*
  - na kraju zatvoriti *thread pool*
- **ThreadPoolExecutor vodi računa o performansama i opterećenju niti** (nije preporučljivo razvijati sopstveni sistem za upravljanje nitima i konkurentnim taskovima)

```
import
    java.util.concurrent.Executors;
import
    java.util.concurrent.ThreadPoolExecutor;

public class ThreadPoolExample
{
    public static void main(String[] args)
    {
        ThreadPoolExecutor executor =
            (ThreadPoolExecutor)
            Executors.newFixedThreadPool(2);

        Task t = new Task();
        executor.execute(t);

        executor.shutdown();
    }
}
```

# 6. Konkurentnost

## Implementacija konkurentnosti – upravljanje nitima u Javi

- Postoji pet vrsta **ThreadPoolExecutor** objekata:
  - **Fixed thread pool executor**
    - fiksan broj niti – prekobrojni taskovi čekaju u redu
    - executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(10);
  - **Cached thread pool executor**
    - broj niti nije fiksan – kreiraju se dodatne niti po potrebi (nije preporučljivo za vremenski zahtjevne taskove niti za mnogo taskova – može da naruši performanse)
    - executor = (ThreadPoolExecutor) Executors.newCachedThreadPool();
  - **Scheduled thread pool executor**
    - omogućava odgođeno ili periodično izvršavanje
    - executor = (ThreadPoolExecutor) Executors.newScheduledThreadPool(10);
  - **Single thread pool executor**
    - samo jedna nit – kad imamo samo jedan konkurentni task
    - executor = (ThreadPoolExecutor) Executors.newSingleThreadExecutor();
  - **Work stealing thread pool executor**
    - Kreira dovoljno niti da obezbijedi projektovani nivo paralelizma (multiprocessing)
    - executor = (ThreadPoolExecutor) Executors.newWorkStealingThreadPool(4);

# 6. Konkurentnost

## Implementacija konkurentnosti – definicija aktivnih objekata

Definicija aktivnog objekta koji se izvršava asinhrono i vraća rezultat

```
import java.util.concurrent.Callable;  
  
public class AktObj implements Callable<T>  
{  
    private T atr;  
    public AktObj(T arg)  
    { atr = arg; }  
    @Override  
    public T call() throws Exception  
    {  
        // izracunaj rezultat  
        return rezultat;  
    }  
}
```

Klasa AktObj implementira interfejs Callable.

Callable reprezentuje **asinhroni proces** koji vraća rezultat.

Klasa AktObj definiše metodu call() kojom implementira aplikativnu logiku.

# 6. Konkurentnost

## Implementacija konkurentnosti

### Poziv asinhronog procesa (**Future**)

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ThreadPoolExecutor;
public class AktObjTest
{
    public static void main(String[] args)
        throws ExecutionException,
               InterruptedException
    {
        // ...
        ThreadPoolExecutor executor =
            (ThreadPoolExecutor)
            Executors.newFixedThreadPool(2);
        AktObj aobj = new AktObj(arg);
        Future<T> f = executor.submit(aobj);
        // ...
        T rez = f.get();
    }
}
```

Klasa AktObjTest ilustruje korištenje AktObj objekata.

executor upravlja životnim ciklusom asinhronih procesa.

Pomoću metode submit() executor prima i inicira izvršavanje Callable objekta.

Future (implementira Runnable) reprezentuje rezultat izvršavanja asinhronog procesa.

Objekat tipa Future dobija se odmah, a vrijednost rezultata biće naknadno pribavljen.

# 6. Konkurentnost

## Primjer implementacije – ThreadPool & Future

```
public class Fact implements Callable<int>
{
    private int n;
    public Fact(int n) { this.n = n; }
    @Override
    public int call() throws Exception
    {
        int f = 1;
        if (n>1) for (int i=2; i<=n; f*=i++);
        System.out.println(n + "! -> " + f);
        return f;
    }
}
```

```
4! -> 24
6! -> 720
Future: 720 Task done: is true
Future: 24 Task done: is true
2! -> 2
6! -> 720
Future: 720 Task done: is true
Future: 2 Task done: is true
```

```
public class CallableDemo
{
    public static void main(String[] args)
    {
        ThreadPoolExecutor executor =
            (ThreadPoolExecutor)
            Executors.newFixedThreadPool(2);
        List<Future<int>> fl=new ArrayList<>();
        Random random = new Random();
        for (int i=0; i<4; i++) {
            int n=random.nextInt(10);
            Fact f = new Fact(n);
            Future<int> r=executor.submit(f);
            fl.add(r);
        }
        for(Future<int> f:fl) {
            try {
                System.out.println("Future: " +
                    f.get()+" Task done:" + f.isDone());
            }
            catch (...) { e.printStackTrace(); }
        }
        executor.shutdown();
    }
}
```

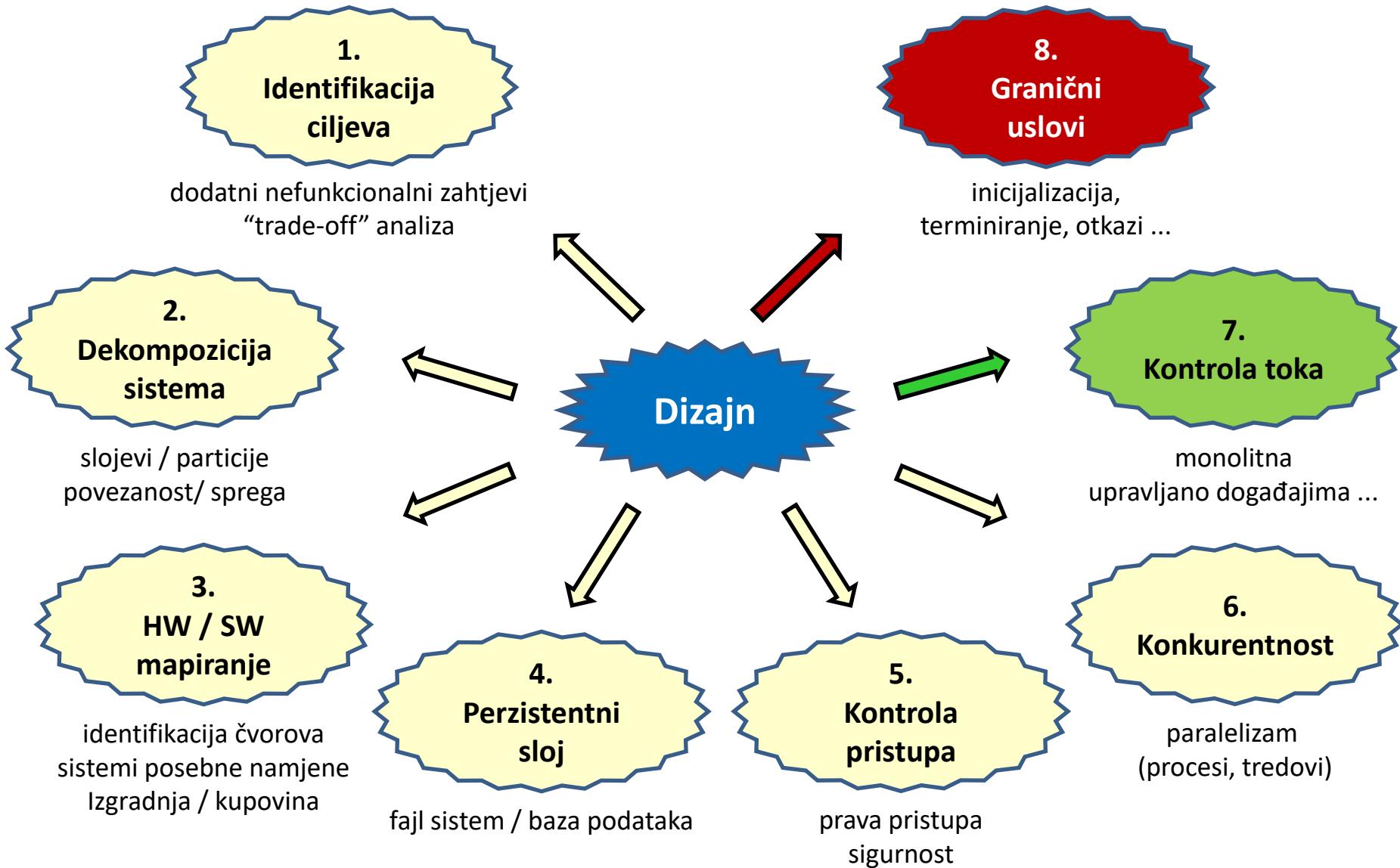
**UNIVERZITET U BANJOJ LUCI  
ELEKTROTEHNIČKI FAKULTET**

**Prof. dr Dražen Brđanin**

**PROJEKTOVANJE SOFTVERA  
/kontrola toka/**

**Banja Luka  
2024.**

# 8 bitnih aktivnosti u projektovanju



# 7. Kontrola toka

## Osnovni oblici kontrole toka

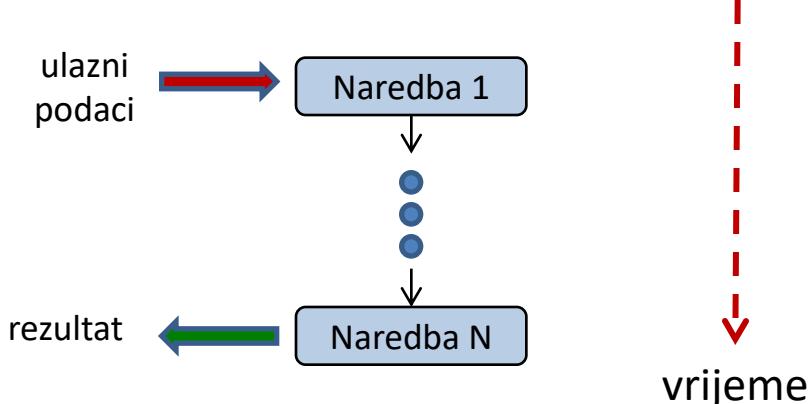
- Kontrola toka predstavlja redoslijed izvršavanja akcija/naredbi
- U O-O sistemima:
  - kontrola toka uključuje redoslijed izvršavanja metoda
  - kontrola toka zavisi od:
    - eksternih događaja (poruka) koje generišu učesnici,
    - vremenskih događaja (istek intervala, ...)
- Kontrola toka u OOA&D:
  - Analiza:
    - kontrola toka nije predmet istraživanja
    - podrazumijeva da svi objekti simultano izvršavaju operacije kad god je potrebno
  - Dizajn:
    - U obzir mora da se uzme činjenica da svaki objekat nema procesor 100% za sebe
- Dva oblika kontrole toka:
  - **centralizovana**
  - **distribuirana (decentralizovana)**

# 7.1. Centralizovana kontrola toka

## Centralizovana kontrola toka

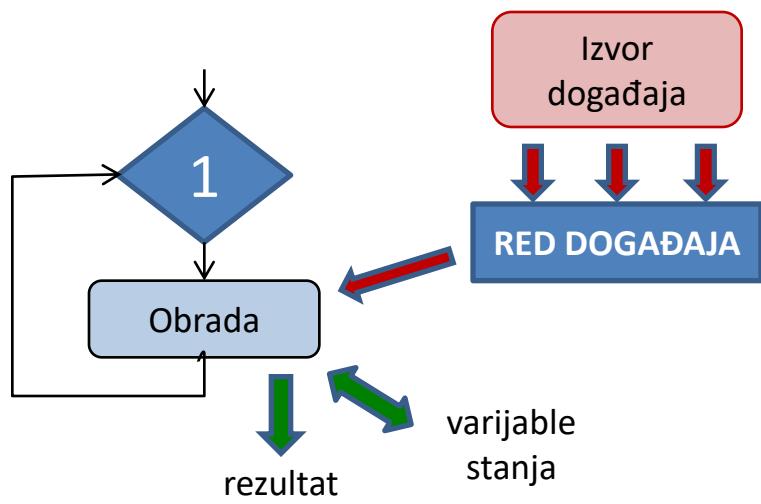
### proceduralna (*procedure-driven*)

- kontrola je u programskom kodu
  - Operacije čekaju na unos podataka kad god trebaju podatke od učesnika
  - Uglavnom u starijim sistemima implementiranim u proceduralnim jezicima
  - Nije pogodna za implementaciju konkurentnih aplikacija (koji je redoslijed sekvensiranja u slučaju većeg broja istovremenih događaja?)



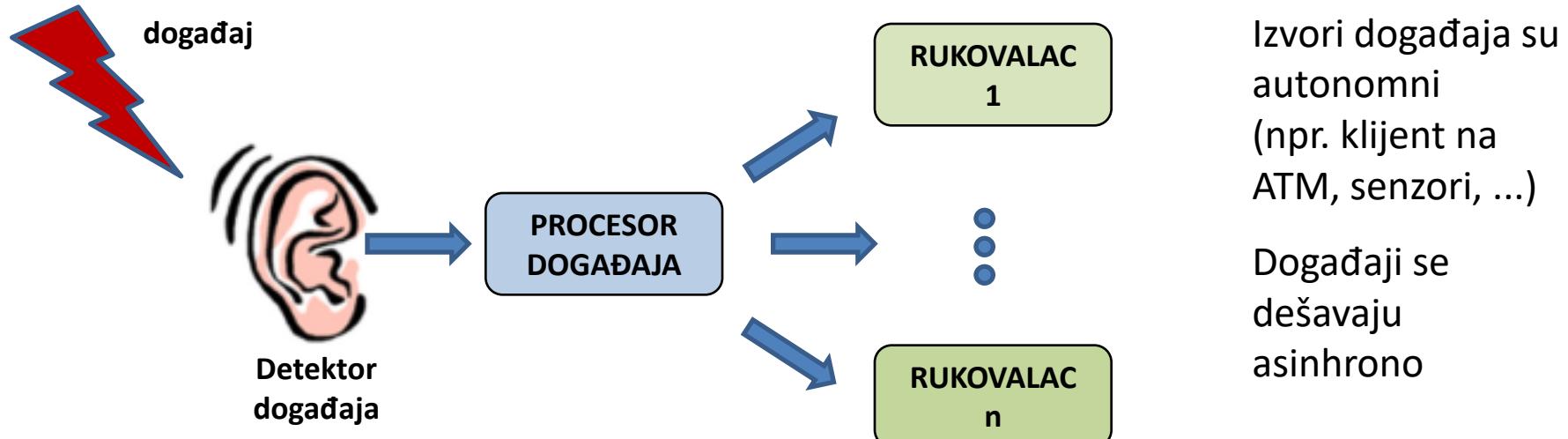
### upravljana događajima (*event-driven*)

- kontrolu ima dispečer koji poziva funkcije (*event handler*)
  - Glavna petlja čeka na spoljne događaje
    - kad se pojavi neki spoljni događaj, on se proslijeđuje odgovarajućem objektu, koji ga potom obrađuje
  - Ovakva glavna kontrolna petlja ima veoma jednostavnu strukturu

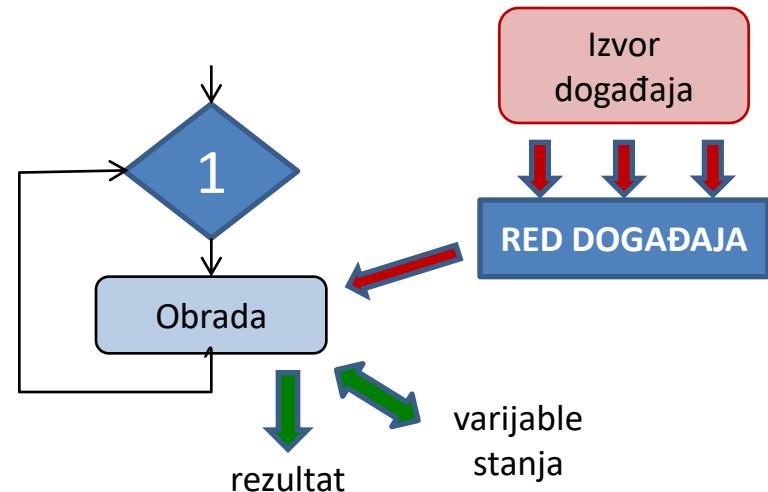


# 7.1. Centralizovana kontrola toka

Kontrola toka upravljana događajima (*event-driven*)



```
// upravljacka nit
while (true)
{
    e = red.getNew();
    if (e == quit) break;
    proces(e);
}
```



# 7.1. Centralizovana kontrola toka

## Kontrola toka upravljana događajima (*event-driven*)

### Tipična primjena: GUI-bazirane aplikacije

Korisnici imaju interakciju sa kontrolnim elementima (npr. dugme) i generišu događaje.

Događaje obrađuju rukovaoci događajima (*event handler*, u javi *event listener*).

Rukovalac je “zakačen” na odgovarajuću komponentu.

Rukovaoci implementiraju odgovarajući interfejs.

```
class Panel extends JPanel implements ActionListener
{
    public Panel()
    {
        super();
        JButton btn = new JButton("OK");
        btn.addActionListener(this);
        this.add(btn);
    }

    @Override
    public void actionPerformed(ActionEvent e)
    { // obrada dogadjaja }
}
```

The diagram illustrates the event-handling mechanism. It shows a Java code snippet within a red-bordered box. A yellow speech bubble labeled "rukovalac" points to the line `btn.addActionListener(this);`. Another yellow speech bubble labeled "događaj" points to the line `@Override public void actionPerformed(ActionEvent e)`.

# 7.1. Centralizovana kontrola toka

## Kontrola toka upravljana događajima (*event-driven*)

### Rukovanje događajima

Događaj (*event*) se prenosi od **izvora događaja** (npr. dugme) do odgovarajućeg **rukovaoca događajima** (*event listener*)

Izvor događaja je objekat koji **zna kada/kako** se dogodio događaj i o tome **obavještava** zainteresovane objekte – **rukovaoce objektima**

Rukovalac događajima je objekat koji **treba biti obaviješten** da se desio neki događaj da bi reagovao na događaj, tj. izvršio odgovarajuću akciju

Informacija o događaju inkapsulirana je u odgovarajućem objektu (**ActionEvent**)

Izvor događaja mora biti u mogućnosti da **registruje rukovaoce** i da im šalje objekte događaja

Rukovalac događajima implementira odgovarajući **interfejs** (**ActionListener**)

Kad se desi događaj, **izvor obavještava** sve **registrovane rukovaoce** pozivom odgovarajuće metode (**actionPerformed**)

Kad rukovalac **dobije poruku** da se desio događaj, on **izvršava** odgovarajuće akcije ili **ignoriše događaj**

# 7.1. Centralizovana kontrola toka

## Primjer: (implementacija)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonCounter
{
    public static void main(String[] args)
    {
        ButtonCounterFrame frame = new ButtonCounterFrame();
        frame.setVisible(true);
    }
}

class ButtonCounterFrame extends JFrame
{
    public ButtonCounterFrame()
    {
        // ... set up the frame and center it on the screen
        // ... create panels for the button and label

        ClickCountingLabel label = new ClickCountingLabel();
        labelPanel.add(label);

        JButton button = new JButton("Click Me!");
        button.addActionListener(label);
        buttonPanel.add(button);

        add(buttonPanel, BorderLayout.NORTH);
        add(labelPanel, BorderLayout.CENTER);

        // ... add panels to frame (to its content pane)
    }
}
```

```
class ClickCountingLabel extends JLabel implements ActionListener
{
    private static final String LABEL_TEXT_PREFIX =
        "Number of button clicks = ";

    private int numClicks = 0;

    public ClickCountingLabel()
    {
        super();
        setText(LABEL_TEXT_PREFIX + numClicks);
    }

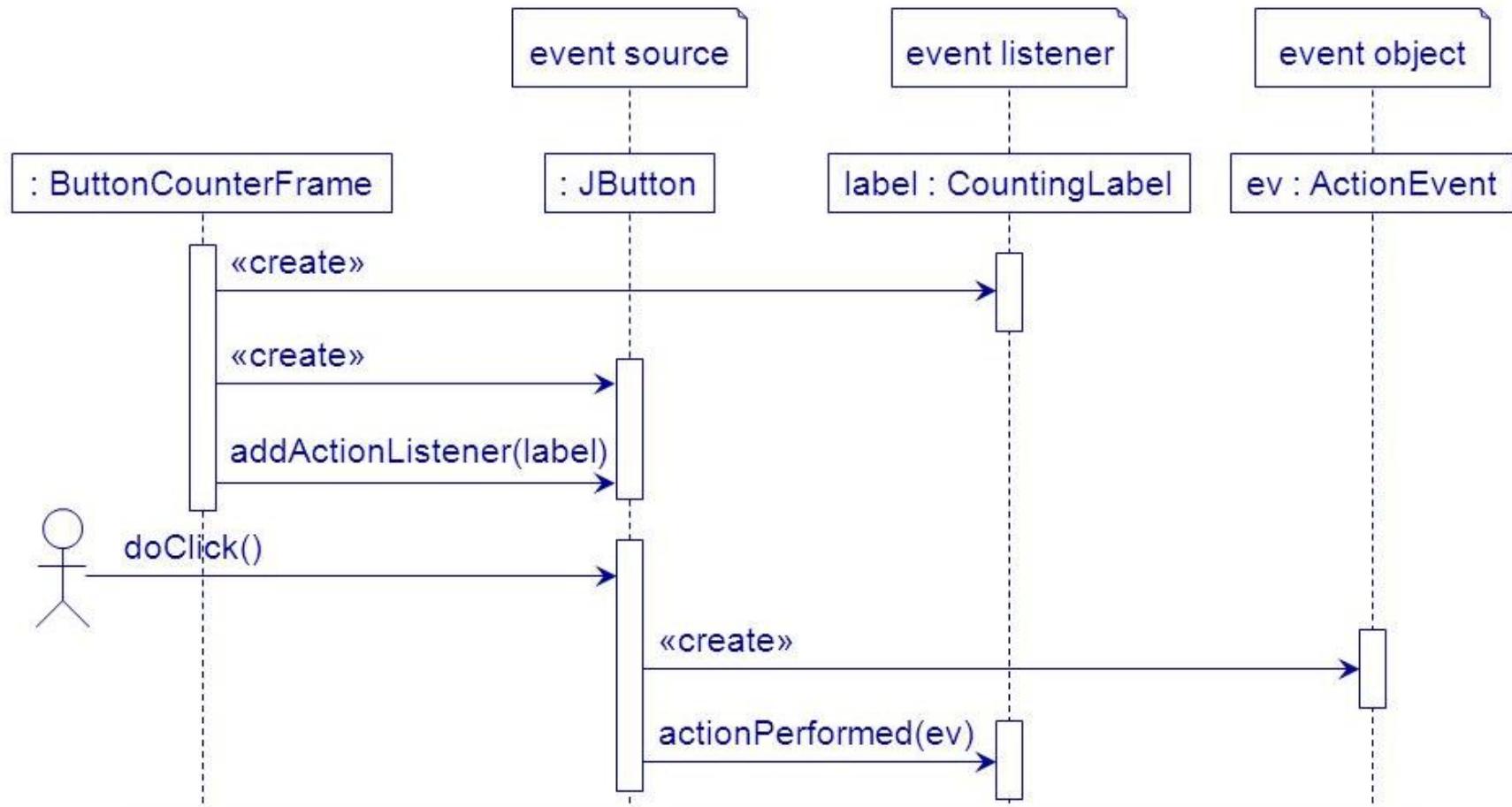
    public void actionPerformed(ActionEvent event)
    {
        ++numClicks;
        setText(LABEL_TEXT_PREFIX + numClicks);
    }
}
```



button – izvor događaja  
label – rukovalac događajima

# 7.1. Centralizovana kontrola toka

Primjer: (interakcija objekata)

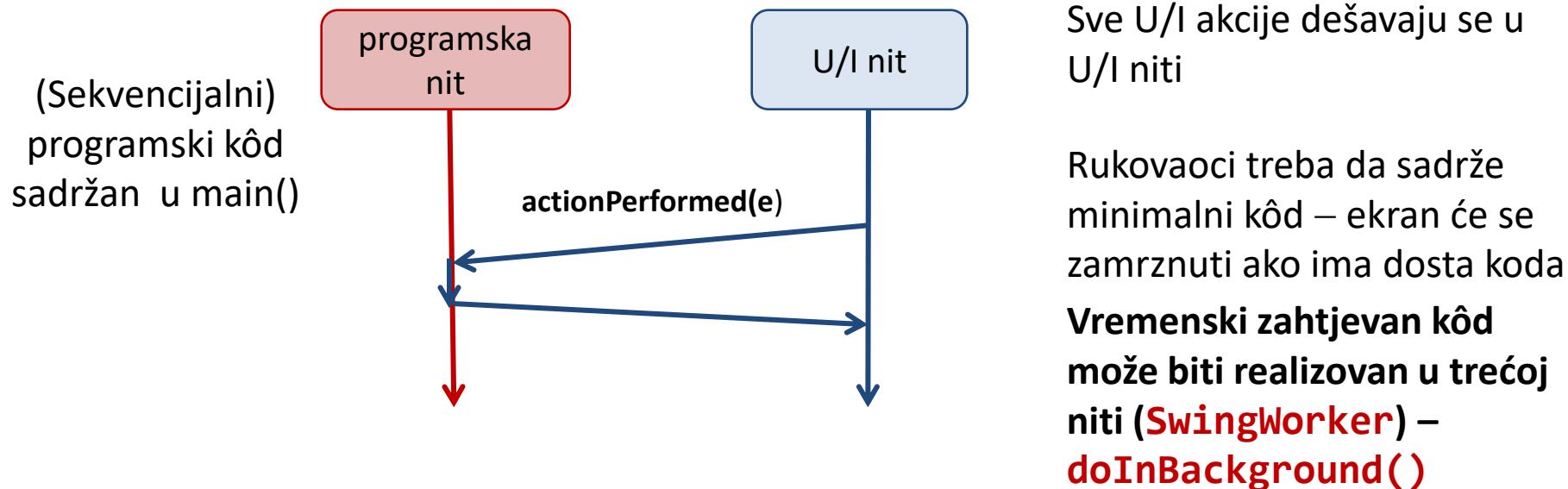


# 7.1. Centralizovana kontrola toka

## Kontrola toka upravljana događajima (*event-driven*)

Programska nit  $\leftrightarrow$  U/I nit

Program i korisnički interfejs izvršavaju se u konkurentnim nitima



## Java GUI

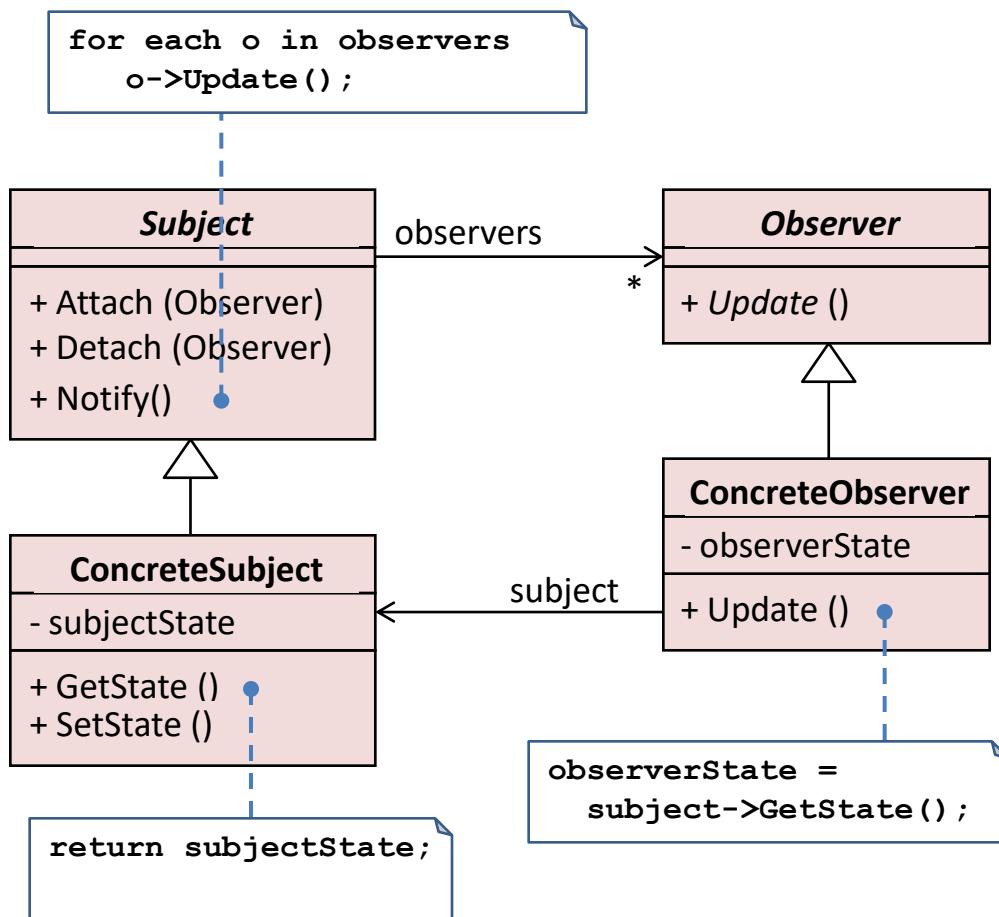
Implementacija slijedi **opserver (publish/subscribe)** projektni obrazac

Komponente su *publisher-i*, a rukovaoci su posmatrači (*subscriber-i*) koji su registrovani na komponente od interesa

Kad se desi neki događaj, komponenta obavještava rukovaće da se desio događaj, a rukovaoci reaguju na sebi svojstven način

# 7.1. Centralizovana kontrola toka

**Observer**  
(Nadzornik, Posmatrač)



- Definiše zavisnosti tipa jedan:više među različitim objektima i obezbjeđuje da se promjena stanja u jednom objektu automatski reflektuje u svim zavisnim objektima.

## Subject

- čuva reference prema opserverima
- obezbjeđuje interfejs za dodavanje i uklanjanje opservera

## ConcreteSubject

- čuva stanje od interesa za konkretnе opservere
- šalje notifikaciju opserverima kad promijeni stanje

## Observer

- definiše interfejs za ažuriranje opservera kad se subject promjeni

## ConcreteObserver

- čuva referencu na ConcreteSubject objekte
- čuva stanje koje treba da ostane konzistentno sa stanjem konkretnog subjekta
- implementira interfejs za ažuriranje objekata koji je definisan u klasi Observer

# 7.1. Centralizovana kontrola toka

## Dobre i loše strane centralizovane kontrole toka

- Centralizovani dizajn (jedan upravljački objekat ima kontrolu)
  - dobre strane:
    - jednostavne izmjene u kontrolnoj strukturi
    - pogodno za testiranje podsistema
  - loše strane:
    - jedan kontrolni objekat je potencijalno “usko grlo”

## Heuristika za centralizovanu/distribuiranu kontrolu

- Kontrola može da se implementira pomoću jednog ili više kontrolnih objekata (kontrolni objekti registruju spoljne događaje, vode računa o privremenim stanjima objekata, adekvatno sekvenciraju operacije, ...)
- Lokalizacija kontrole za jedan slučaj upotrebe u jedan kontrolni objekat kod čini fleksibilnijim, lakšim za pregled i održavanje...
- Ako **dijagram sekvence (ili dijagram komunikacije)** izgleda “zvjezdasto”  
→ distribuirana kontrola

## 7.2. Distribuirana kontrola toka

### Distribuirani (decentralizovani) sistemi

- “*Distribuirani softverski sistem predstavlja kolekciju autonomnih softverskih komponenata, koju korisnici vide kao jedinstven sistem*” (Tannebaum)
- “*Distribuirani sistem predstavlja kolekciju softverskih komponenata raspoređenih na različitim hardverskim čvorovima, koje mogu međusobno da komuniciraju i usklađuju svoj rad samo razmjenom poruka*” (Coulouris et al.)
- “*Distribuirani sistem je onaj sistem u kojem otkaz nekog računara, za kojeg i ne znate da postoji, može da uzrokuje neupotrebljivost i vašeg računara*” (Lamport)
- **Svi današnji veliki softverski sistemi (*large scale*) su distribuirani.**
- Obrada podataka nije ograničena na jedan hardverski čvor, nego je distribuirana na više računara.
- Krajnji korisnici imaju doživljaj jedinstvenog sistema, bez obzira na fizičku distribuciju hardverskih i softverskih komponenata.

## 7.2. Distribuirana kontrola toka

### Osnovne karakteristike distribuiranih sistema

- **Dijeljenje resursa** (hardverskih i softverskih)
- **Otvorenost /Interoperabilnost** (standardni protokoli omogućavaju komunikaciju hardverskih i softverskih komponenata različitih proizvođača, bez obzira na korištene jezike, tehnologije i alate)
- **Konkurentnost** (istovremeno izvršavanje različitih softverskih komponenata)
- **Skalabilnost** (povećavanje propusne moći dodavanjem novih hardverskih i softverskih komponenata)
- **Transparentnost** (korisnici imaju doživljaj jedinstvenog sistema, bez obzira na fizičku distribuciju hardverskih i softverskih komponenata)
- **Složenost** (distribuirani sistemi su mnogo kompleksniji nego monolitni sistemi)
- **Nema “glavne” komponente** (u opštem slučaju nema “nadređene” komponente” u sistemu i kontrola odozgo prema dolje nije moguća)

## 7.2. Distribuirana kontrola toka

### Izazovi u projektovanju distribuiranih sistema

- **Transparentnost** (korisnici imaju doživljaj jedinstvenog sistema, bez obzira na fizičku distribuciju hardverskih i softverskih komponenata)
  - **Gdje je granica transparentnosti** (do kog nivoa korisnik treba da ima osjećaj da pristupa jedinstvenom sistemu)?
  - **Idealno**, korisnici ne bi trebalo da imaju osjećaj da je sistem distribuiran, a servisi bi trebalo da su nezavisni od načine distribucije
  - **Praktično**, to je nemoguće (kontrola nije jedinstvena, postoje mrežna kašnjenja) – zato je bolje da su korisnici toga svjesni i da treba da očekuju nedostatke
  - Da bi se ostvarila transparentnost treba koristiti **apstrakciju resursa i logičko adresiranje** (a ne fizičko) – **za logičko↔fizičko mapiranje zadužen je middleware**

# 7.2. Distribuirana kontrola toka

## Izazovi u projektovanju distribuiranih sistema

### – Skalabilnost

- Sposobnost sistema da pruža servise (odgovarajućeg kvaliteta) bez obzira na:
  - povećavanje broja korisnika,
  - geografsku distribuiranost resursa (npr. nova poslovna jedinica)
- “**Scaling up**” (**vertikalno skaliranje**) – zamjena resursa resursom sa većom propusnom moći
- “**Scaling out**” (**horizontalno skaliranje**) – dodavanje dovih resursa
- Horizontalno skaliranje ima bolji odnos uloženo-dobijeno, ali se zahtijeva podrška za konkurentan rad

### – Kvalitet usluge (*QoS – Quality of Service*)

- Sposobnost sistema da servise pruža:
  - pouzdano,
  - sa prihvatljivim vremenom odziva i
  - sa prihvatljivim propusnim opsegom
- Posebno kritično: **vremenski odziv i propusni opseg** (audio/video)

# 7.2. Distribuirana kontrola toka

## Izazovi u projektovanju distribuiranih sistema

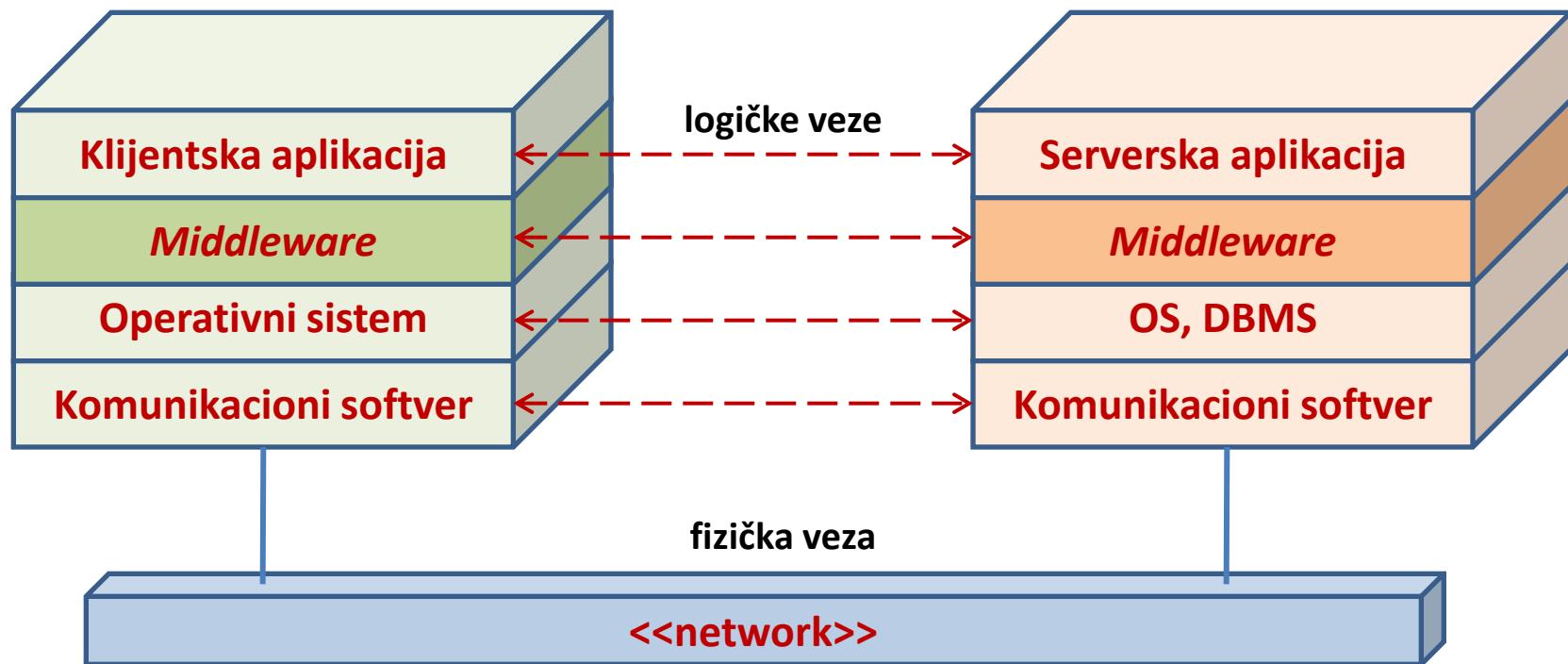
- **Sigurnost**
  - Mnogo više sigurnosnih prijetnji nego u centralizovanim sistemima:
    - **presretanje komunikacije,**
    - **nedostupnost servisa** (*denial of service*) uslijed preopterećenja čvora lažnim zahtjevima,
    - **modifikacija podataka ili servisa,**
    - neautorizovani pristup (krivotvorene kredencijale, krađa identiteta, ...).
  - Dijelovi sistema mogu da pripadaju različitim organizacijama, koje mogu da imaju različite sigurnosne politike
- **Upravljanje otkazima (*Failure Management*)**
  - Otkazi su neizbjegni (pogotovo u distribuiranim sistemima), a sistem treba da bude što manje osjetljiv na otkaze:
    - **postojanje mehanizama za otkrivanje komponenata koje su otkazale**
    - **nastavak pružanja maksimalnog broja usluga** (koji ne zavise od komponenata koje su otkazale)
    - **automatski oporavak od otkaza, ako je moguće**

## 7.2. Distribuirana kontrola toka

Interakcija softverskih komponenata u distribuiranim sistemima

Komunikacija u distribuiranim sistemima: **sinhrona / asinhrona**

Komunikacija se izvodi posredstvom softverskog međusloja – **middleware**



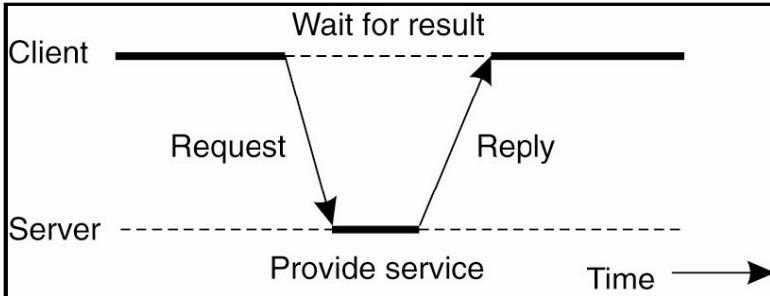
# 7.2. Distribuirana kontrola toka

## Modeli interakcije komponenata u distribuiranim sistemima

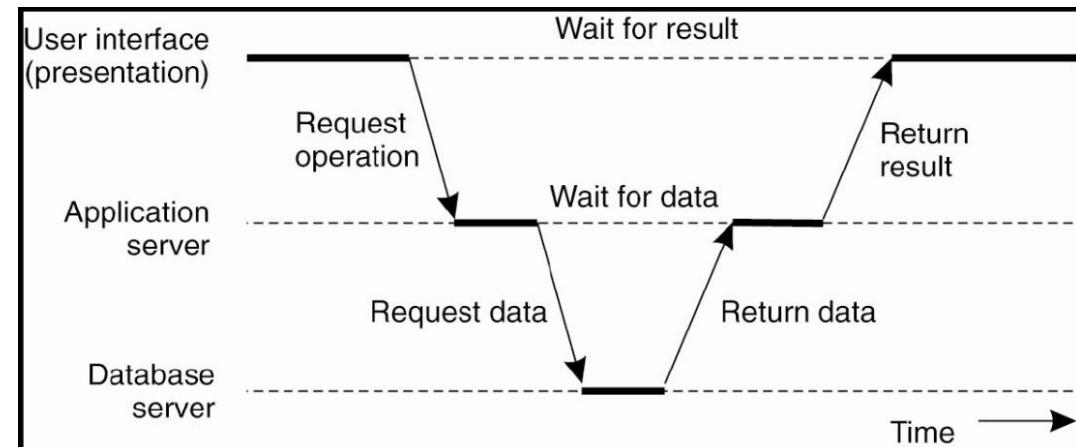
### Sinhrona interakcija

#### – Proceduralna komunikacija

- Jedna komponenta poziva (poznati) servis druge komponente (na istom ili drugom hardverskom čvoru) i čeka odgovor (kao da je riječ o objektima unutar iste softverske komponente)
- Tipična interakcija u  $n$ -slojnim arhitekturama
- Implementacija: RPC (*Remote Procedure Call*)



Proceduralna komunikacija u  
2-slojnoj arhitekturi



Proceduralna komunikacija u  
3-slojnoj arhitekturi

# 7.2. Distribuirana kontrola toka

## Modeli interakcije komponenata u distribuiranim sistemima

### Sinhrona interakcija – RPC (*Remote Procedure Call*)

#### **Request–Response** protokol

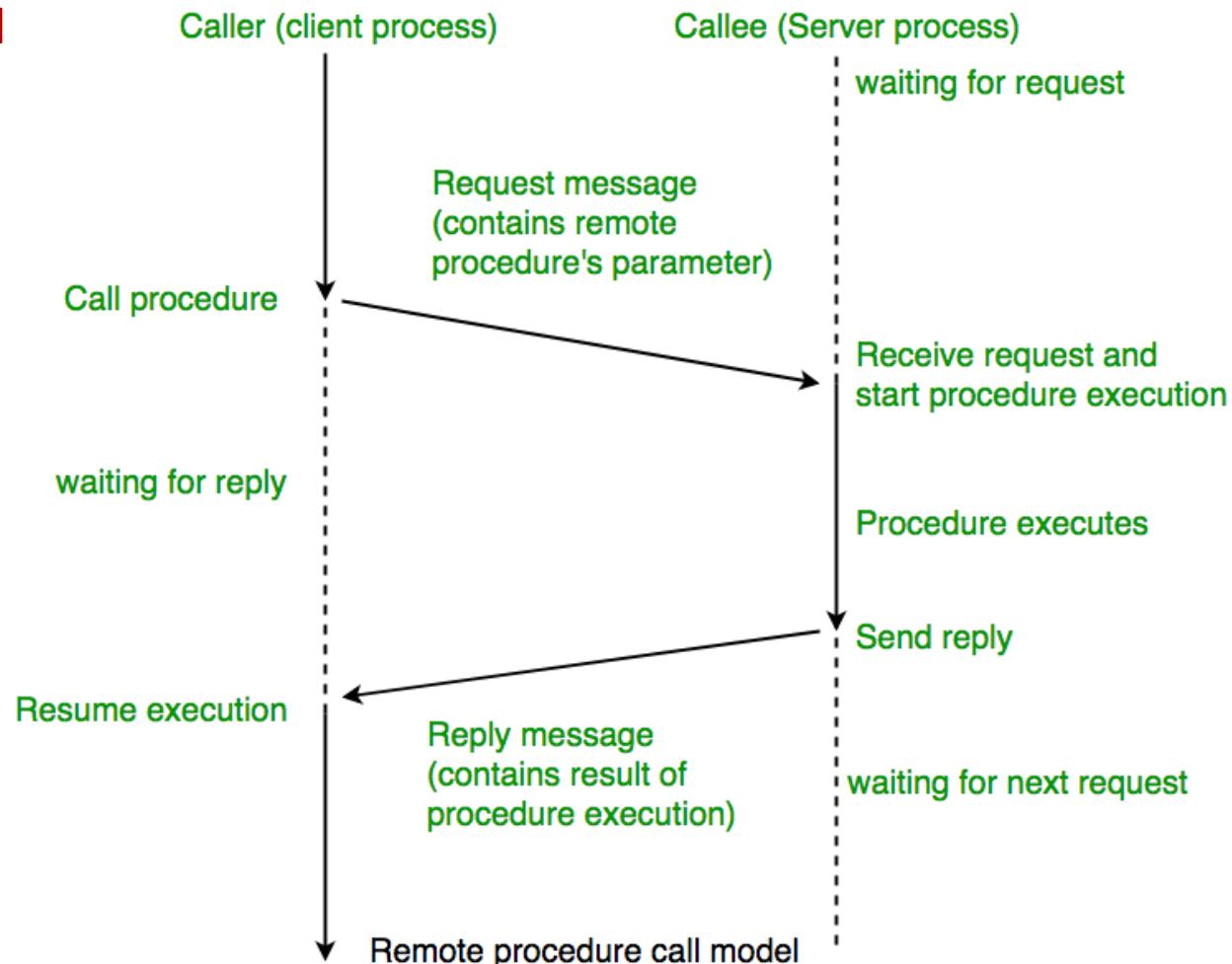
**Pozivalac** (klijentski proces) poziva poznati servis (serverski proces) na istom ili drugom hardverskom čvoru

Nakon što pošalje poziv udaljenom servisu, **pozivalac čeka odgovor** (suspendovano izvršavanje klijentskog procesa)

**Serverski proces**, nakon što završi izvršavanje, **vraća rezultat** klijentskom procesu

**Klijentski servis prima rezultat i nastavlja izvršavanje**

Primjeri: **HTTPRequest, JDBC**



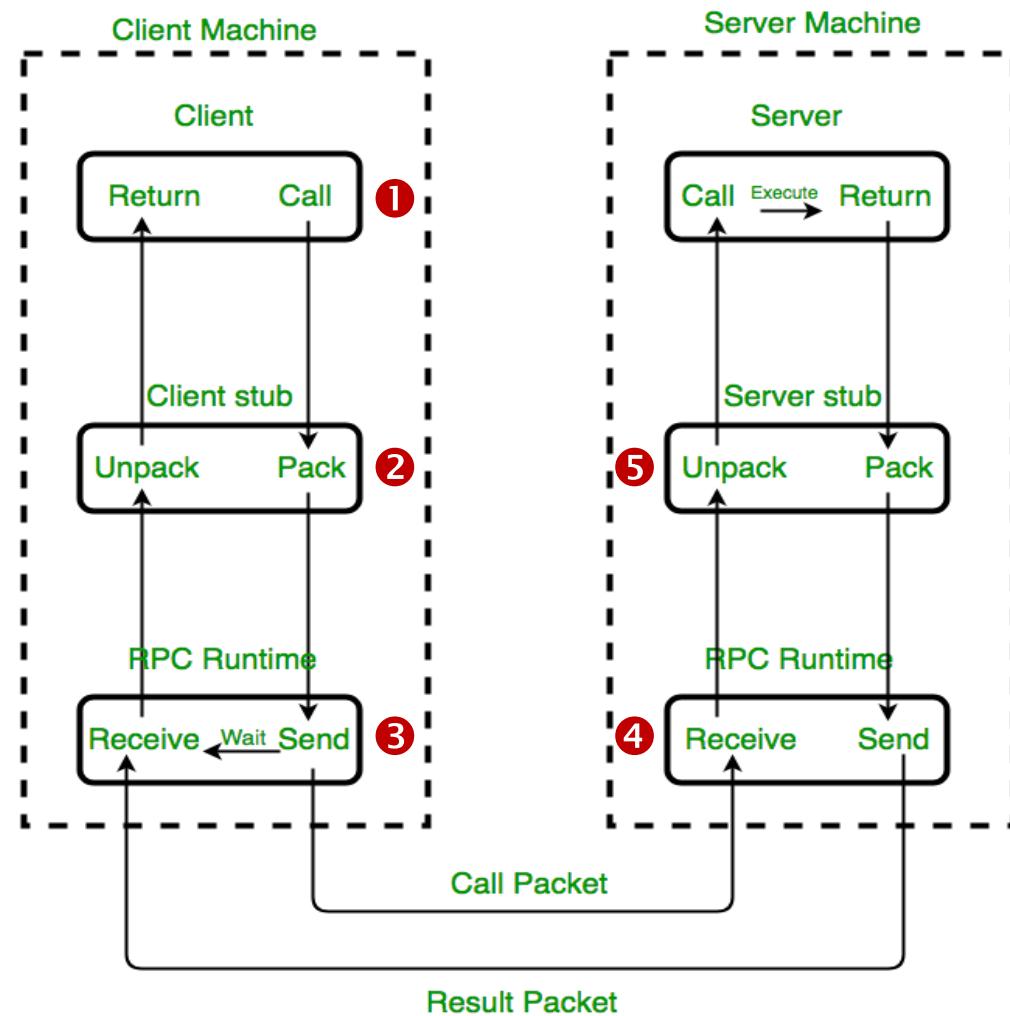
# 7.2. Distribuirana kontrola toka

## Modeli interakcije komponenata u distribuiranim sistemima

### Sinhrona interakcija – RPC (*Remote Procedure Call*)

#### Realizacija RPC mehanizma

1. Klijent poziva operaciju klijentskog proksi objekta (*client stub*) – klijent i klijentski proksi nalaze se u istom adresnom prostoru
2. Klijentski proksi pakuje parametre u poruku (*marshalling*) i šalje je transportnom sloju (*middleware*)
3. Transportni sloj proslijeđuje poruku (udaljenom) serveru
4. Na serverskoj strani, transportni sloj prima i proslijeđuje poruku serverskom proksi objektu (*server stub*)
5. Serverski proksi raspakuje poruku (*unmarshalling*) i proslijeđuje parametre serverskom procesu



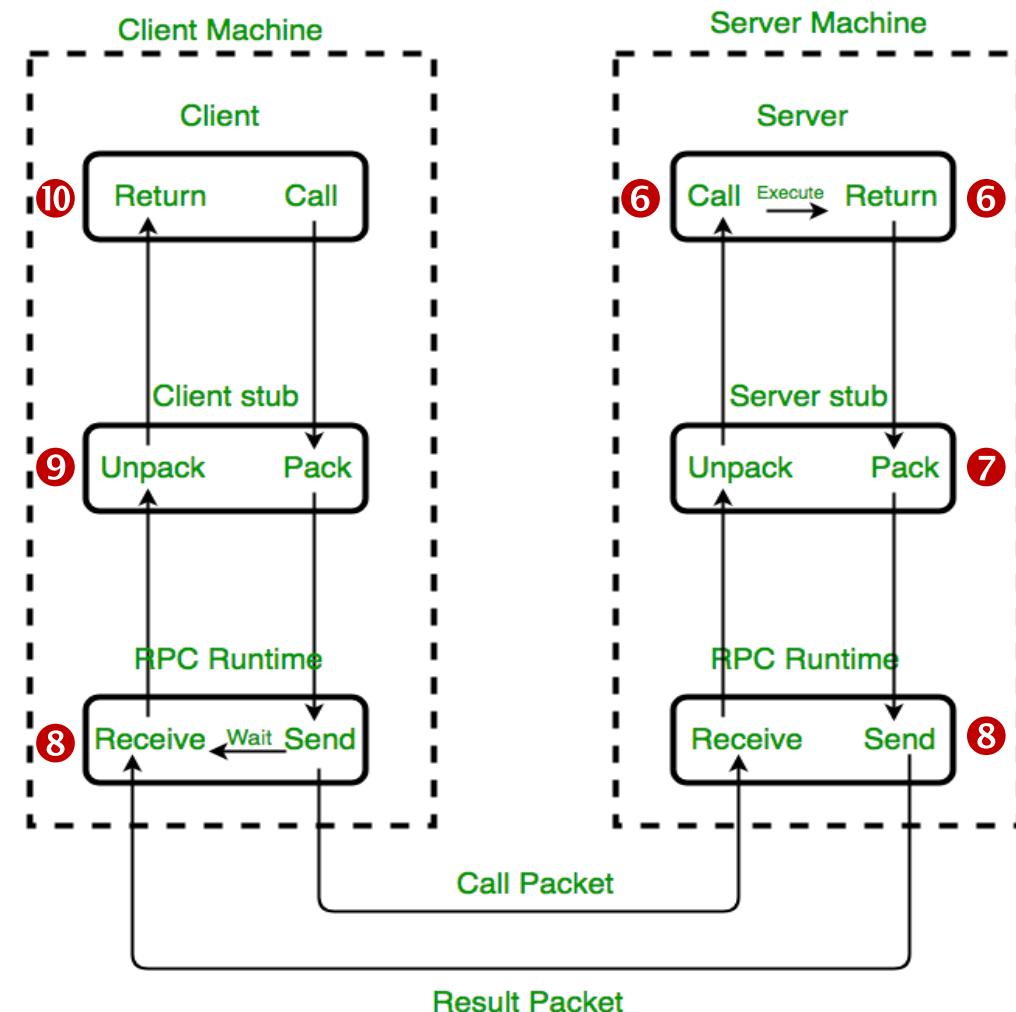
# 7.2. Distribuirana kontrola toka

## Modeli interakcije komponenata u distribuiranim sistemima

### Sinhrona interakcija – RPC (*Remote Procedure Call*)

#### Realizacija RPC mehanizma

6. Server izvršava pozvanu proceduru i vraća rezultat serverskom proksiju kao da je riječ o lokalnom pozivu
7. Serverski proksi pakuje rezultat u poruku (*marshalling*) i šalje je transportnom sloju (*middleware*)
8. Transportni sloj šalje poruku klijentskoj mašini, koja prima poruku i proslijeđuje je klijentskom proksiju
9. Klijentski proksi raspakuje poruku, ekstrahuje rezultat i šalje ga klijentu
10. Klijent prima rezultat od klijentskog proksija, kao da je u pitanju bio lokalni poziv



# 7.2. Distribuirana kontrola toka

## Modeli interakcije komponenata u distribuiranim sistemima

### Sinhrona interakcija – RPC (*Remote Procedure Call*)

#### RPC middleware

- Kolekcija servisa koji omogućavaju realizaciju *request-response* protokola i implementiraju RPC mehanizam
- Tokom RPC poziva, RPC komponente uspostavljaju komunikaciju preko odgovarajućeg protokola, obezbjeđuju odgovarajuće vezivanje (***binding***), prenose podatke i rješavaju komunikacione greške.

#### Proksi objekti

- Proksi objekti obezbjeđuju transparentnost u klijentskom aplikativnom kodu
- **Klijentski proksi** predstavlja interfejs između lokalnog klijenta i RPC sistema
- **Serverski proksi** predstavlja interfejs između servisa i RPC sistema

#### Vezivanje (***binding***) klijentskog i serverskog koda

- Kako klijent zna koga zove i gdje se servis nalazi?

## 7.2. Distribuirana kontrola toka

### Modeli interakcije komponenata u distribuiranim sistemima

#### Sinhrona interakcija – Java RMI

##### Java RMI (*Remote Method Invocation*)

- RMI je objektno-orientisana verzija RPC mehanizma
- osnovne komponente:
  - **serverska strana:**
    - interfejs udaljenog objekta
    - udaljeni objekat – instanca klase koja implementira interfejs
    - serverska aplikacija koja instancira i objavljuje udaljeni objekat
    - registar imena
  - **klijentska strana**

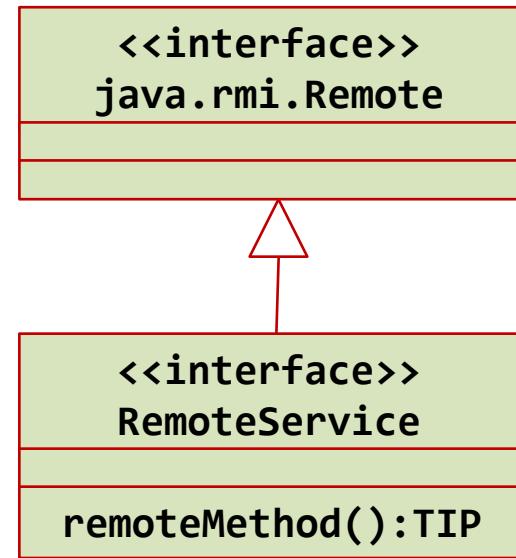
# 7.2. Distribuirana kontrola toka

## Modeli interakcije komponenata u distribuiranim sistemima

### Java RMI – Interfejs udaljenog objekta

- proširuje `java.rmi.Remote` interfejs
- deklariše udaljene metode
- svaka udaljena metoda mora da deklariše dizanje izuzetka tipa `RemoteException`

```
package example.rmi;  
import java.rmi.Remote;  
import java.rmi.RemoteException;  
public interface RemoteService extends Remote  
{  
    TIP remoteMethod() throws RemoteException;  
}
```



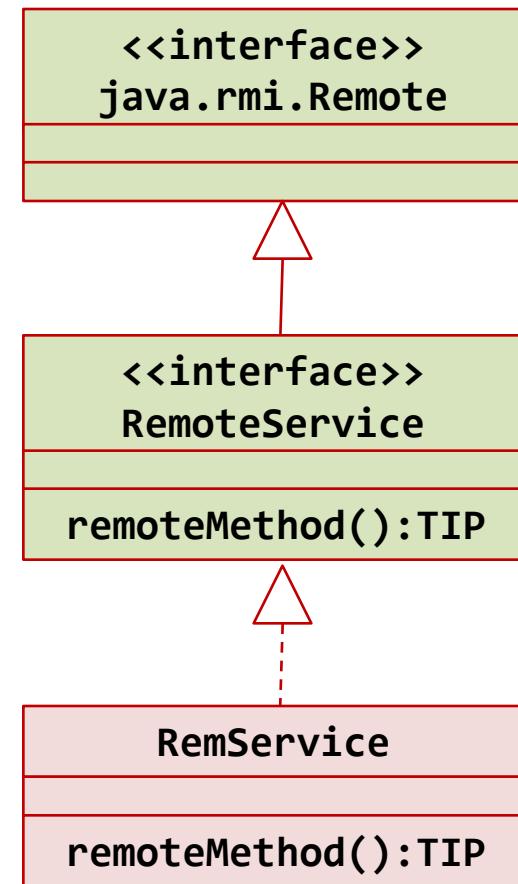
# 7.2. Distribuirana kontrola toka

## Modeli interakcije komponenata u distribuiranim sistemima

### Java RMI – Udaljeni objekat (*Remote object*)

- udaljeni servis koji implementira udaljeni interfejs
- implementira udaljene metode
- može biti implementiran u zasebnoj klasi ili u serverskoj klasi

```
package example.rmi;  
public class RemService implements RemoteService  
{  
    public RemService() {}  
    public TIP remoteMethod()  
    { // return rezultat; }  
}
```



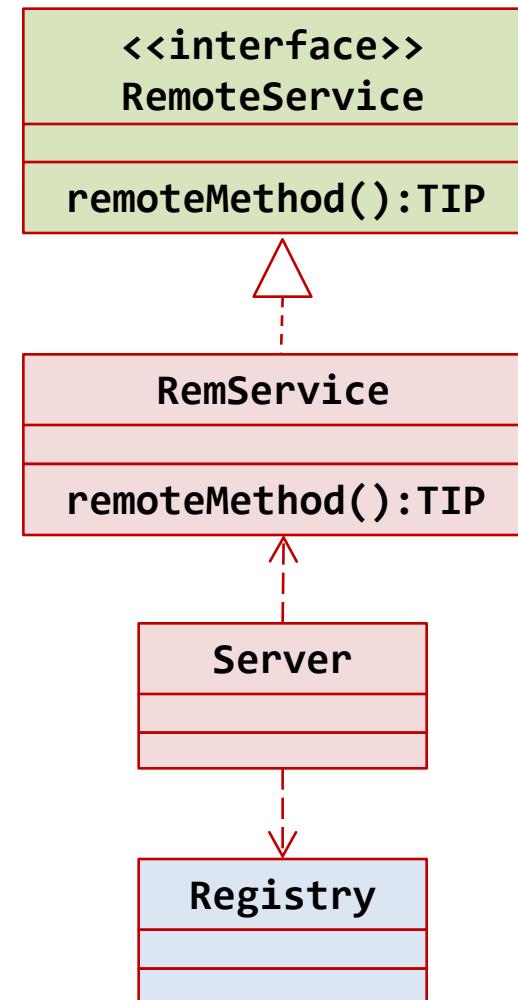
# 7.2. Distribuirana kontrola toka

## Modeli interakcije komponenata u distribuiranim sistemima

### Java RMI – Serverska aplikacija

- u main metodi instancira jedan udaljeni objekat, eksportuje ga i u registru udaljenih objekata veže za ime u Java RMI registru
- udaljeni objekat nakon eksportovanja očekuje pozive na odgovarajućem TCP portu
- eksportovanjem se dobija proksi objekat koji se u registru veže za ime i kojeg kasnije klijent može da preuzme, kako bi pozivao metode udaljenog objekta

```
package example.rmi;  
import java.rmi.registry.Registry;  
import java.rmi.registry.LocateRegistry;  
import java.rmi.RemoteException;  
import java.rmi.server.UnicastRemoteObject;  
public class Server { // }
```



# 7.2. Distribuirana kontrola toka

Modeli interakcije komponenata u distribuiranim sistemima

Java RMI – Serverska aplikacija

```
public class Server
{
    // ...
    public static void main(String args[])
    {
        try
        {
            // instanciranje i eksportovanje udaljenog objekta
            RemService rs = new RemService();
            RemoteService stub = (RemoteService)
                UnicastRemoteObject.exportObject(rs, 0);
            // objavljivanje serverskog proksija u registru
            Registry registar = LocateRegistry.getRegistry();
            registar.bind("RemoteService", stub);
        } catch (Exception e) { // }
    }
}
```

# 7.2. Distribuirana kontrola toka

## Modeli interakcije komponenata u distribuiranim sistemima

### Java RMI – Klijentska aplikacija

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Klijent
{
    private Klijent() {}
    public static void main(String args[])
    {
        try
        {
            // ...
            Registry registar = LocateRegistry.getRegistry(host);
            RemoteService stub = (RemoteService) registar.lookup("RemoteService");
            TIP rezultat = stub.remoteMethod();
            // ...
        }
        catch (Exception e) { // ... }
    }
}
```

**LocateRegistry.getRegistry(host)**  
vraća registar stub za pristup registru na hostu  
ako je host==null, gleda lokalno  
Na osnovu registar stub-a pribavlja se stub za  
udaljeni objekat

## 7.2. Distribuirana kontrola toka

### Modeli interakcije komponenata u distribuiranim sistemima

#### Asinhrona interakcija – Razmjena poruka (*message-based*)

- Jedna komponenta kreira poruku sa zahtjevom prema drugoj komponenti
- Poruka se prosljeđuje putem middleware odredišnoj komponenti
- Odredišna komponenta raspakuje poruku i izvršava odgovarajući servis i po potrebi šalje povratnu poruku sa rezultatom
- Poruke se primaju i privremeno drže u FIFO baferu, sve dok primalac nije raspoloživ
- Komponente mogu biti implementirane različitim jezicima i tehnologijama
- **Middleware je zadužen za rutiranje poruka i uspješnu komunikaciju** različitih komponenata bez obzira na njihove specifičnosti
- **Middleware obezbjeđuje lokacijsku transparentnost** – komponenta ne mora da zna na kojoj fizičkoj lokaciji se nalazi neka druga komponenta
- **Midleware obezbjeđuje neke zajedničke servise** koje mogu da koriste različite komponente (npr. upravljanje transakcijama, sigurnost ...)
- Primjeri middleware-a: JMS, CORBA, DCOM, .NET

## 7.2. Distribuirana kontrola toka

### Modeli interakcije komponenata u distribuiranim sistemima

#### Asinhrona interakcija – JMS (*Java Message Service*)

- JMS – Java API za realizaciju asinhronne komunikacije
- Dva modela asinhronne interakcije: **Point-to-Point, Publisher-Subscriber**

#### Komponente JMS sistema

- **JMS Client:** Java program koji
  - šalje poruke (*sender/producer/publisher*) ili
  - prima poruke (*receiver/consumer/subscriber*).
- **JMS Provider/Middleware**
  - JMS Provider obezbeđuje API za realizaciju asinhronne komunikacije
  - Uobičajeni nazivi: MOM, *Message Broker*, *Messaging Server*, *JMS Server*
  - Neke implementacije imaju dodatni UI interfejs za administriranje MOM sistema
- **JMS Administered Objects:** komponente za
  - uspostavljanje komunikacije Client↔MOM (**ConnectionFactory**)
  - realizaciju skladišta poruka (**Queue / Topic**)
- **JMS Message:** objekat koji sadrži podatke koji se razmjenjuju između klijenata

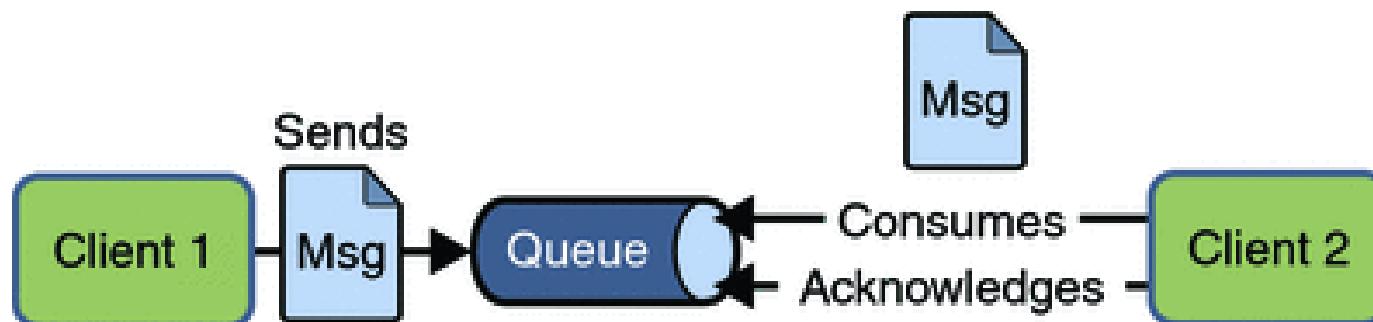
# 7.2. Distribuirana kontrola toka

## Modeli interakcije komponenata u distribuiranim sistemima

### Asinhrona interakcija – JMS (*Java Message Service*)

#### Point-to-Point model (P2P)

- Jedna poruka dostavlja se samo jednom primaocu (*unicasting*)
- **Queue** je message-oriented middleware (MOM) zadužen za prijem i čuvanje poruka sve dok primalac nije raspoloživ
- Ne postoji vremenska zavisnost između pošiljaoca i primaoca (poruka će čekati u redu sve dok primalac ne bude raspoloživ)



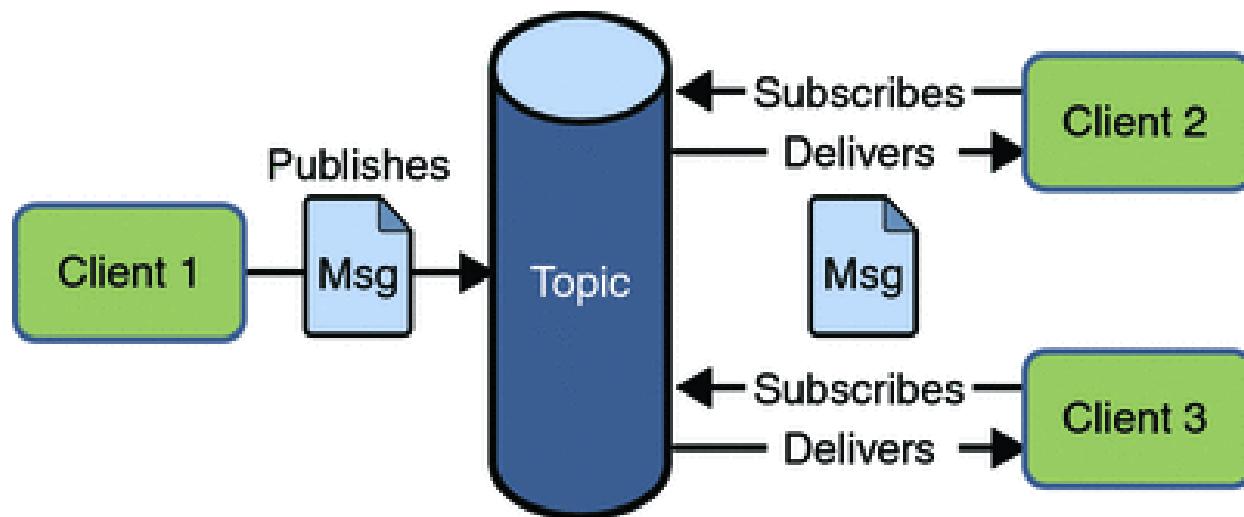
# 7.2. Distribuirana kontrola toka

## Modeli interakcije komponenata u distribuiranim sistemima

### Asinhrona interakcija – JMS (*Java Message Service*)

#### Publisher-Subscriber model (Pub-Sub)

- Jedna poruka dostavlja se većem broju primaoca (*broadcasting*)
- **Topic** je MOM middleware zadužen za prijem i distribuciju poruka
- Poruke se distribuiraju primaocima koji su zainteresovani (*subscribers*) za prijem poruke
- Može biti više izvornih objekata (pošiljaoca) koji šalju poruke u MOM
- Postoje vremenske zavisnosti između pošiljaoca i primalaca (MOM ne mora da drži poruke, ako nema prijavljenih primalaca)



# 7.2. Distribuirana kontrola toka

## Modeli interakcije komponenata u distribuiranim sistemima

### Aシンクロな通信 – JMS (*Java Message Service*)

#### JMS programski model

① ConnectionFactory kreira P2P/P-S konekciju (pošiljaoca/primaoca) sa JMS serverom

② Kreiranje P2P/P-S sesije (pošiljaoca/primaoca)

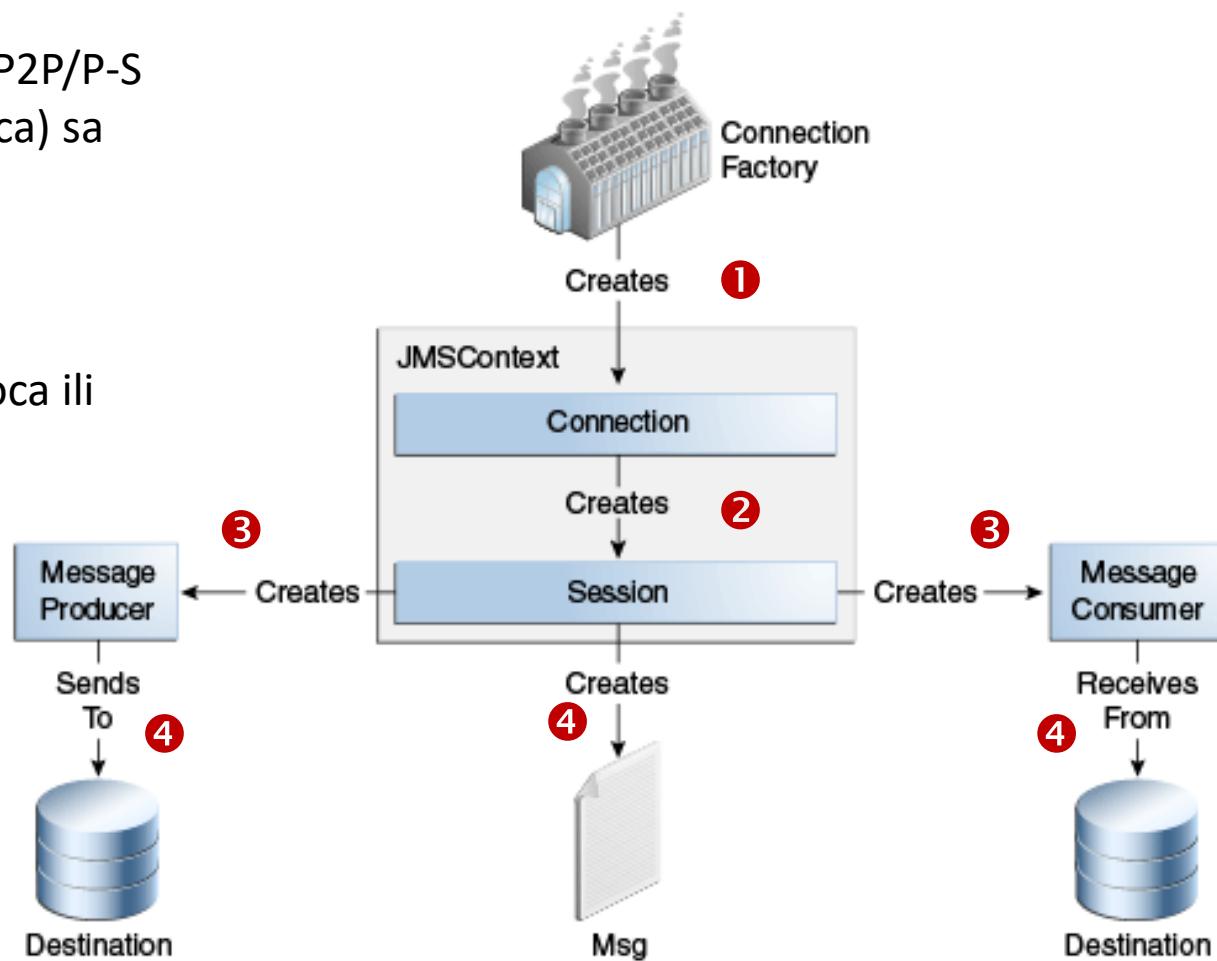
③ Instanciranje P2P/P-S pošiljaoca ili primaoca

=====

④ Kreiranje poruke i slanje u skladište (pošiljalac)

=====

④ Čitanje poruke iz skladišta i slanje primaocu



## 7.2. Distribuirana kontrola toka

### Asinhrona interakcija – JMS (*Java Message Service*)

```
import javax.naming.*;
import javax.jms.*;
public class Sender {
    public static void main(String[] args) {
        try
        { //Create and start connection
            InitialContext c=new InitialContext();
            QueueConnectionFactory f=(QueueConnectionFactory)c.lookup("myQCFactory");
            QueueConnection con=f.createQueueConnection();
            con.start();
            // create queue session
            QueueSession ses=con.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
            // get the Queue object
            Queue t=(Queue)c.lookup("myQueue");
            // create QueueSender object
            QueueSender sender=ses.createSender(t);
            // create TextMessage object
            TextMessage msg=ses.createTextMessage();
            // send message
            sender.send(msg);
            // connection close
            con.close();
        }catch(Exception e){System.out.println(e);}
    }
}
```

Primjer PTP asinhronne komunikacije - pošiljalac

# 7.2. Distribuirana kontrola toka

## Asinhrona interakcija – JMS (*Java Message Service*)

```
import javax.naming.*;
import javax.jms.*;
public class Receiver {
    public static void main(String[] args) {
        try
        { //Create and start connection
            InitialContext c=new InitialContext();
            QueueConnectionFactory f=(QueueConnectionFactory)c.lookup("myQCFactory");
            QueueConnection con=f.createQueueConnection();
            con.start();
            // create queue session
            QueueSession ses=con.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
            // get the Queue object
            Queue t=(Queue)c.lookup("myQueue");
            // create QueueReceiver
            QueueReceiver r=ses.createReceiver(t);
            // create listener object
            MyListener ls=new MyListener();
            // register the listener with receiver
            r.setMessageListener(ls);
            // ...
            con.close();
        }catch(Exception e){ // ... }
    }
}
```

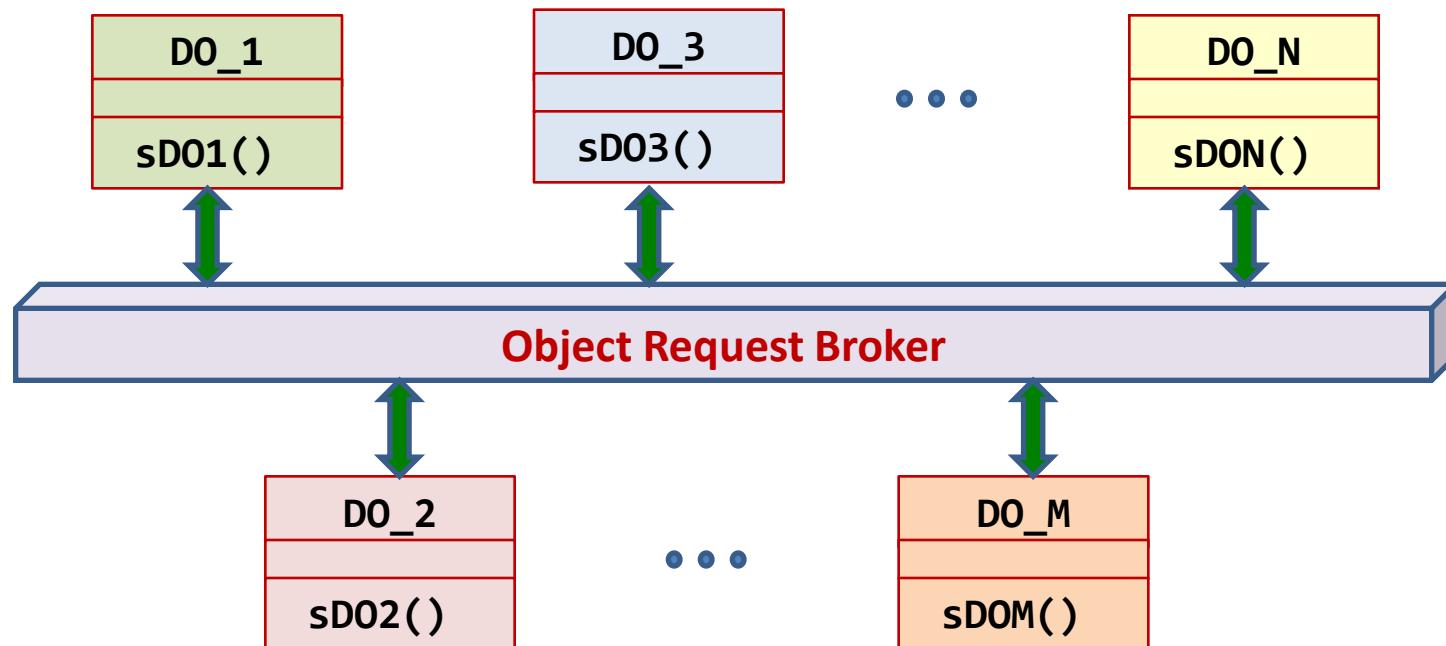
Primjer PTP asinhronne komunikacije - primalac

```
import javax.jms.*;
public class MyListener
    implements MessageListener
{
    public void onMessage(Message m)
    {
        try{
            TextMessage mm=(TextMessage)m;
            System.out.println(mm.getText());
        }
        catch(JMSException e) { // ... }
    }
}
```

## 7.2. Distribuirana kontrola toka

### Arhitektura distribuiranih objekata (*Distribute Objects Architecture*)

- Nema eksplisitne podjele na klijente i servere
- Svaki distribuirani objekat/komponenta pruža servise drugim objektima/komponentama
- Komunikacija između distribuiranih objekata/komponenata odvija se posredstvom middleware-a koji se naziva *Object Request Broker – ORB*



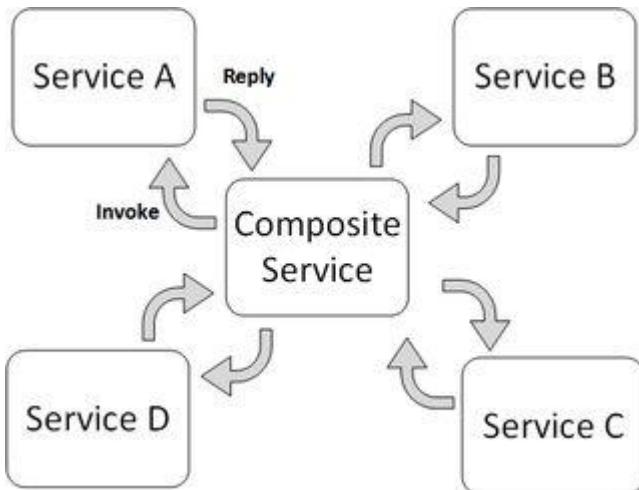
# 7.2. Distribuirana kontrola toka

## Servisno-orientisana arhitektura (SOA)

- Distribuirana softverska arhitektura koji čine autonomne softverske komponente (servisi) koje mogu biti raspoređene na različitim hardverskim čvorovima
- Funkcionalnost sistema ostvaruje se odgovarajućim kombinovanjem funkcionalnosti pojedinih servisa: **orkestracija / koreografija**

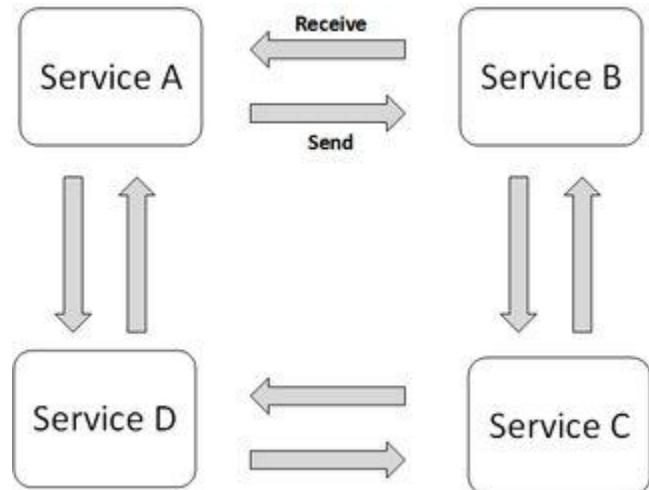
### Orkestracija servisa

- Funkcionalnost se ostvaruje centralizovanim upravljanjem (**orkestratorski servis**)
- **orkestrator** ostvara željenu funkcionalnost odgovarajućim kombinovanjem aktivnosti pojedinih servisa



### Koreografija servisa

- Funkcionalnost se ostvaruje decentralizovanim upravljanjem
- Različiti servisi imaju mogućnost da prime poziv, realizuju dio funkcionalnosti i pozovu drugi servis

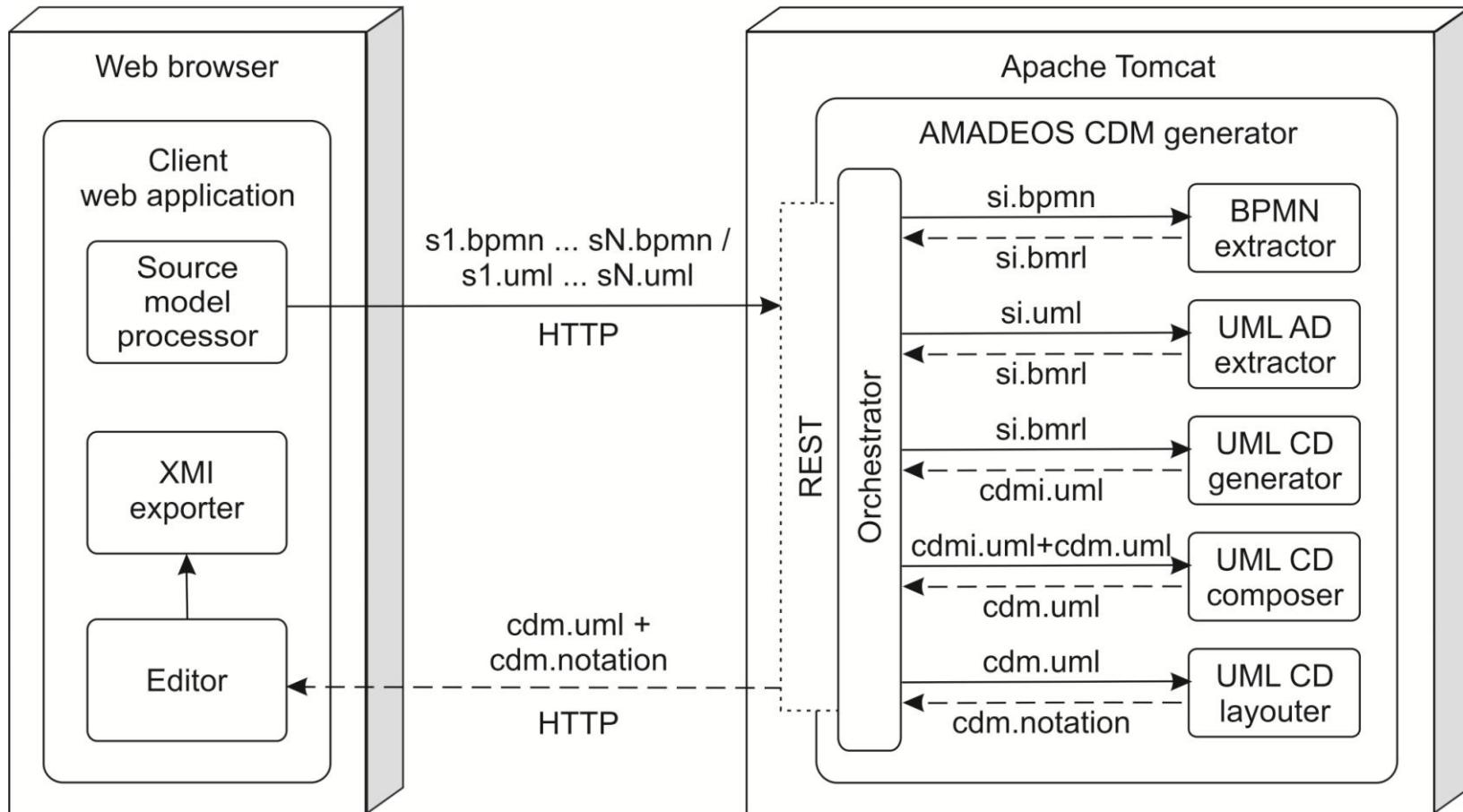


# 7.2. Distribuirana kontrola toka

Primjer servisno-orientisane arhitekture – AMADEOS

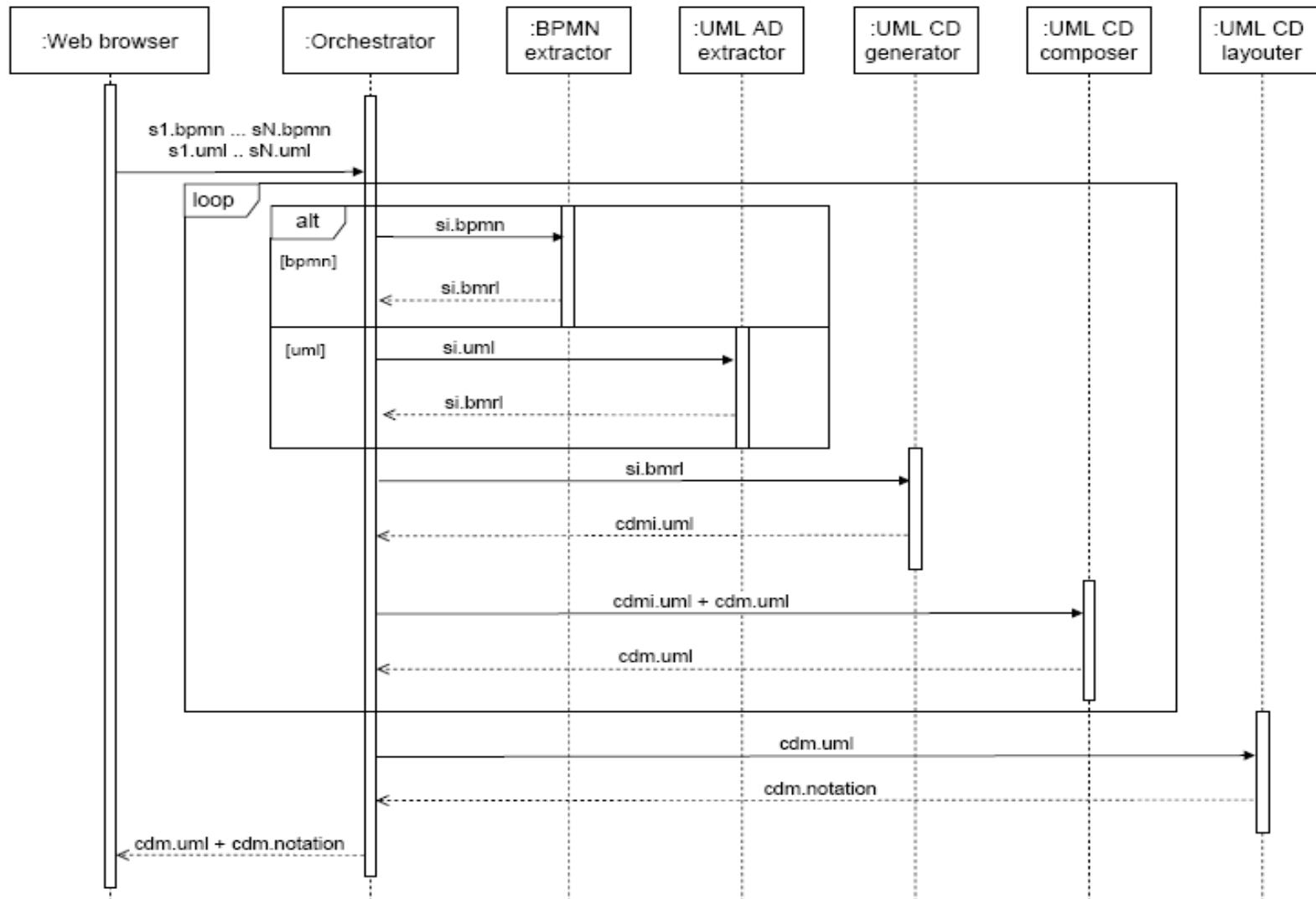
Online sistem za automatsko modelom-voden projektovanje baza podataka

<http://m-lab.etf.unibl.org:8080/amadeos>



# 7.2. Distribuirana kontrola toka

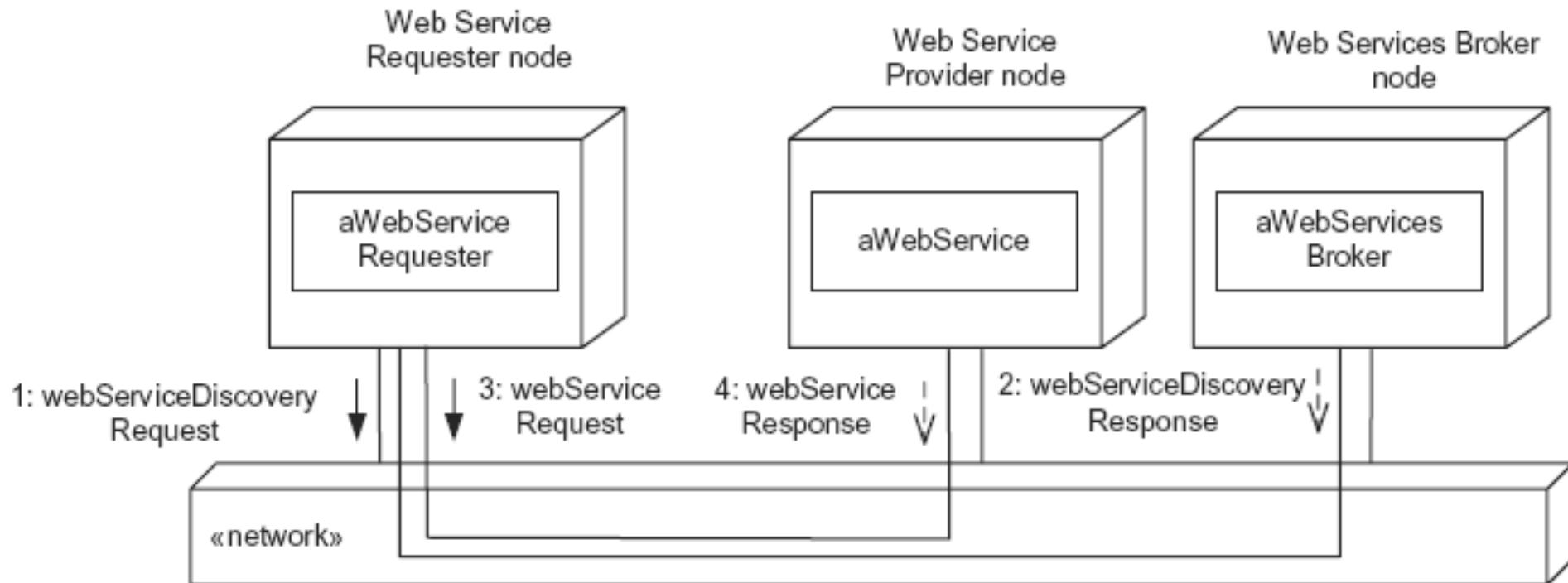
Primjer servisno-orientisane arhitekture – AMADEOS



# 7.2. Distribuirana kontrola toka

## Web servisi

- web servisi koriste web za komunikaciju i omogućavaju realizaciju SOA aplikacija
- iz softverske perspektive: web servisi su API koji omogućavaju komunikaciju između različitih komponenata na standardizovan način (HTTP)
- iz aplikativne perspektive: web servisi su funkcionalne cjeline koje obezbjeđuju neku funkcionalnost drugim aplikacijama
- broker je posrednik između servisa (broker održava registar servisa i zna sve o servisima)

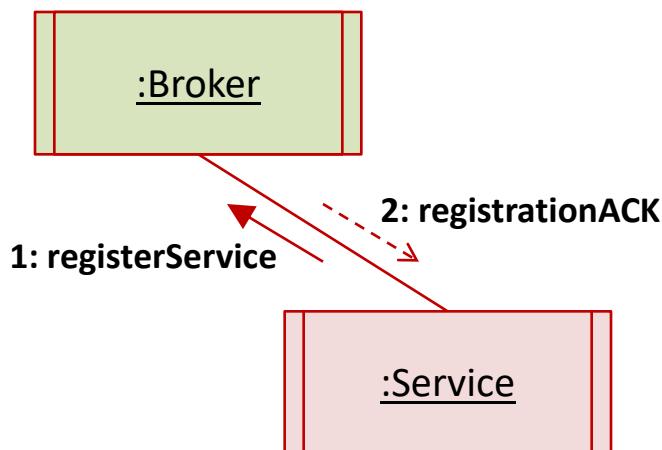


# 7.2. Distribuirana kontrola toka

## Brokerski komunikacioni obrasci

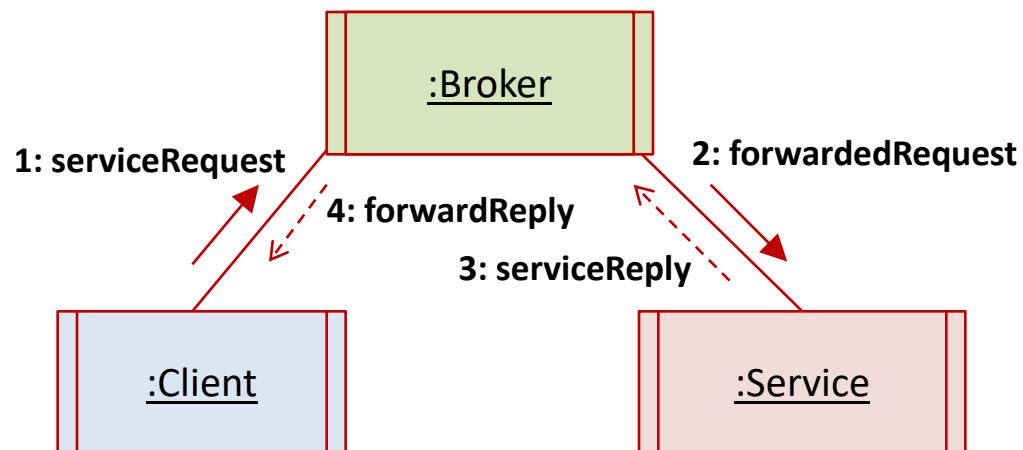
### Registracija servisa kod brokera

- broker je posrednik između servisa
- broker održava registar servisa i zna sve o servisima (gdje se nalaze, kako se pozivaju, ...)
- servisi se registruju kod brokera



### *“Broker forwarding” obrazac*

- klijent upućuje zahtjev brokeru
- broker proslijeđuje zahtjev odgovarajućem servisu
- servis obrađuje zahtjev i vraća rezultat
- broker proslijeđuje rezultat klijentu



# 7.2. Distribuirana kontrola toka

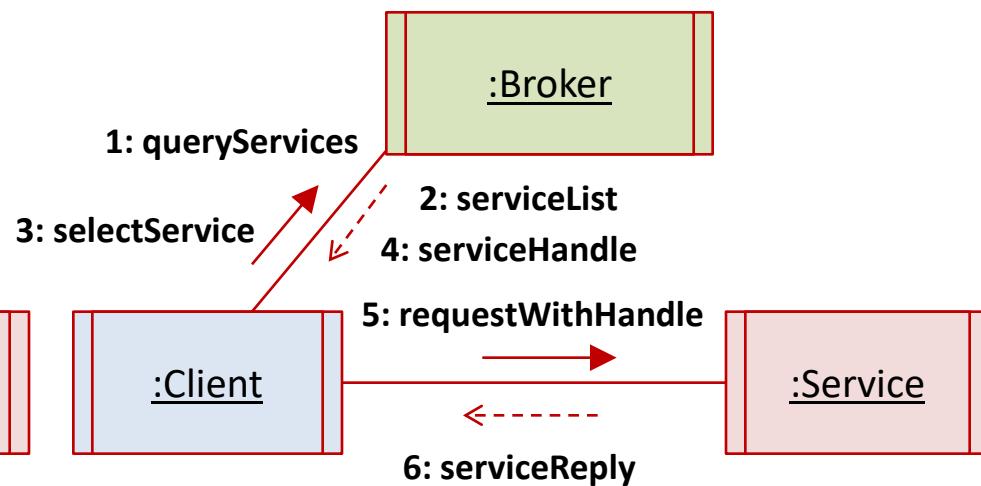
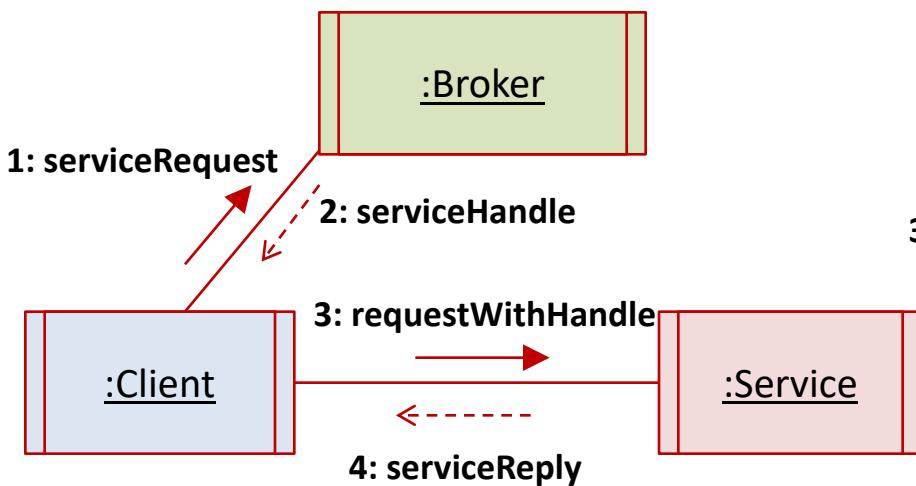
## Brokerski komunikacioni obrasci

### *“Broker handle” (white pages)*

- klijent upućuje zahtjev brokeru
- broker klijentu vraća *handle*
- klijent na osnovu primljenog *handle-a* poziva odgovarajući servis
- servis obrađuje zahtjev i vraća rezultat

### *“Service Discovery” (yellow pages)*

- klijent upućuje zahtjev brokeru za listu pogodnih servisa
- broker vraća listu servisa
- klijent traži handle za izabrani servis
- broker vraća handle za izabrani servis
- klijent na osnovu primljenog *handle-a* poziva odgovarajući servis
- servis obrađuje zahtjev i vraća rezultat



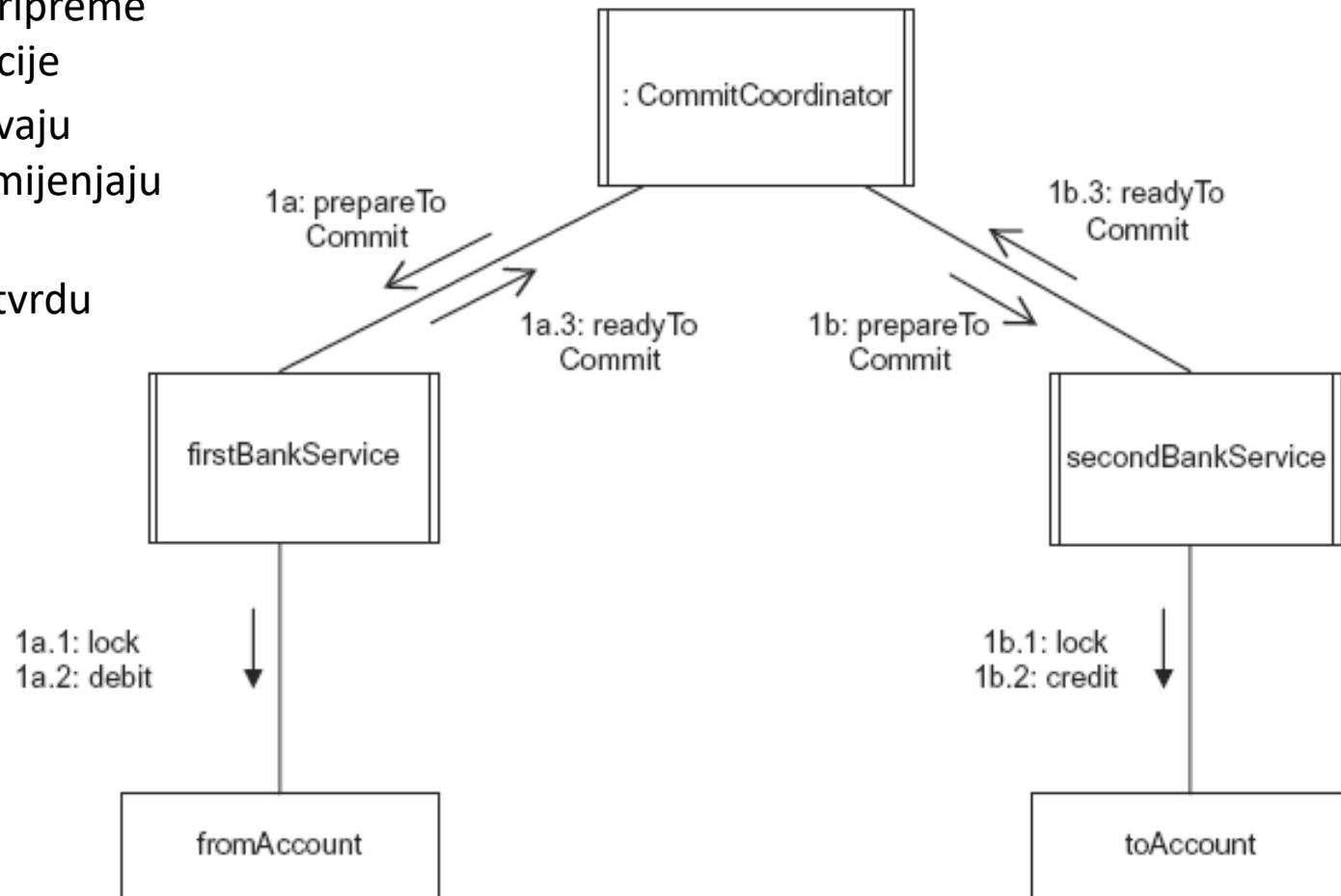
# 7.2. Distribuirana kontrola toka

## Transakcioni protokoli – “*Two phase commit*”

### 1. faza: priprema

- Koordinator traži od uključenih komponenata da se pripreme za izvršavanje transakcije
- Komponente zaključavaju objekte od interesa i mijenjaju stanje
- Komponente šalju potvrdu koordinatoru

Transakcija je operacija koja mora da se izvrši u cijelosti.



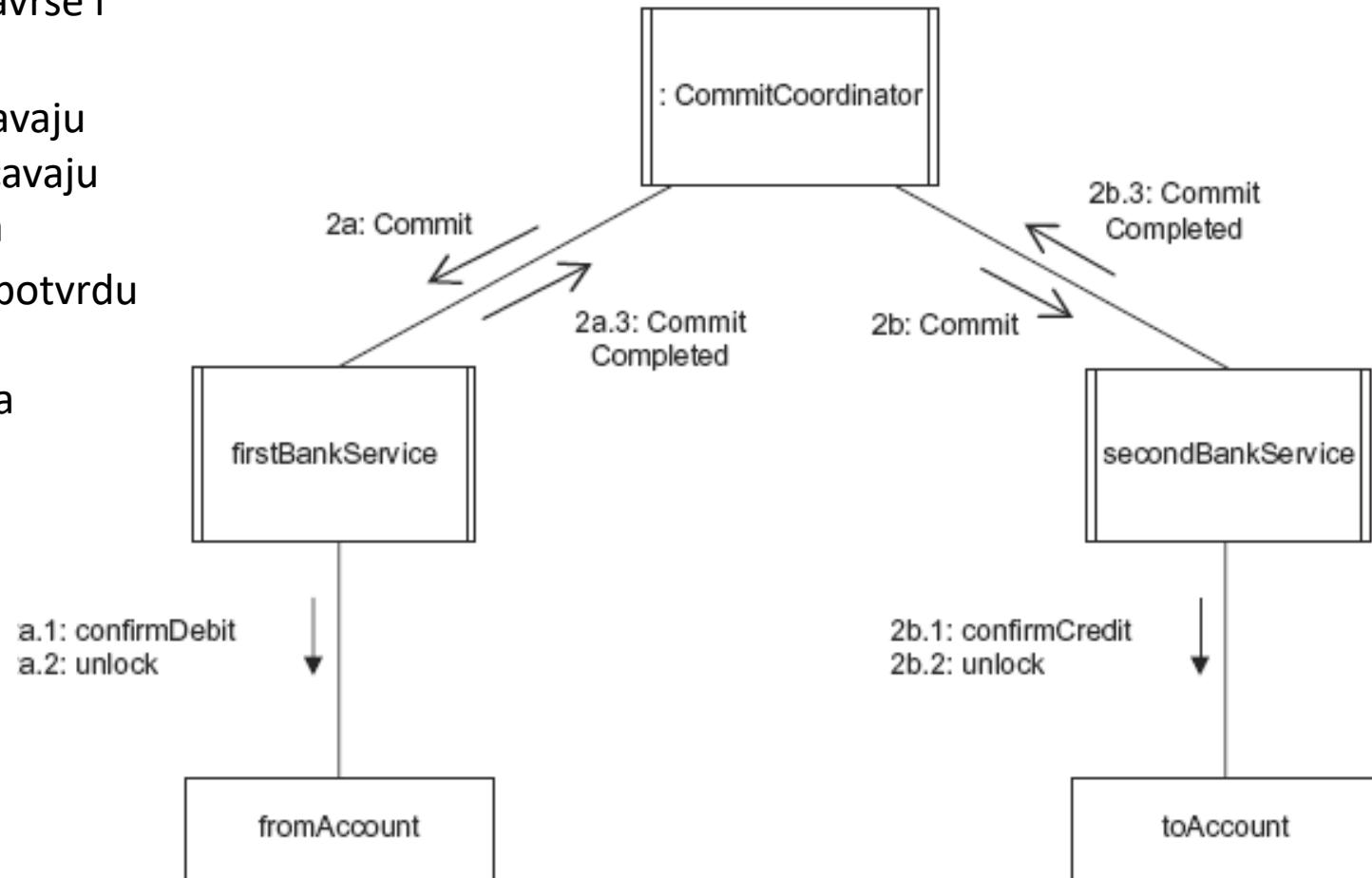
## 7.2. Distribuirana kontrola toka

### Transakcioni protokoli – “*Two phase commit*”

#### 2. faza: potvrda

- Koordinator traži od uključenih komponenata da završe i potvrde transakciju
- Komponente završavaju transakciju i otključavaju objekte od interesa
- Komponente šalju potvrdu koordinatoru da je transakcija završena

Ako transakcija nije u mogućnost da se izvrši u cijelosti, parcijalni efekti moraju biti poništeni (*roll back*)



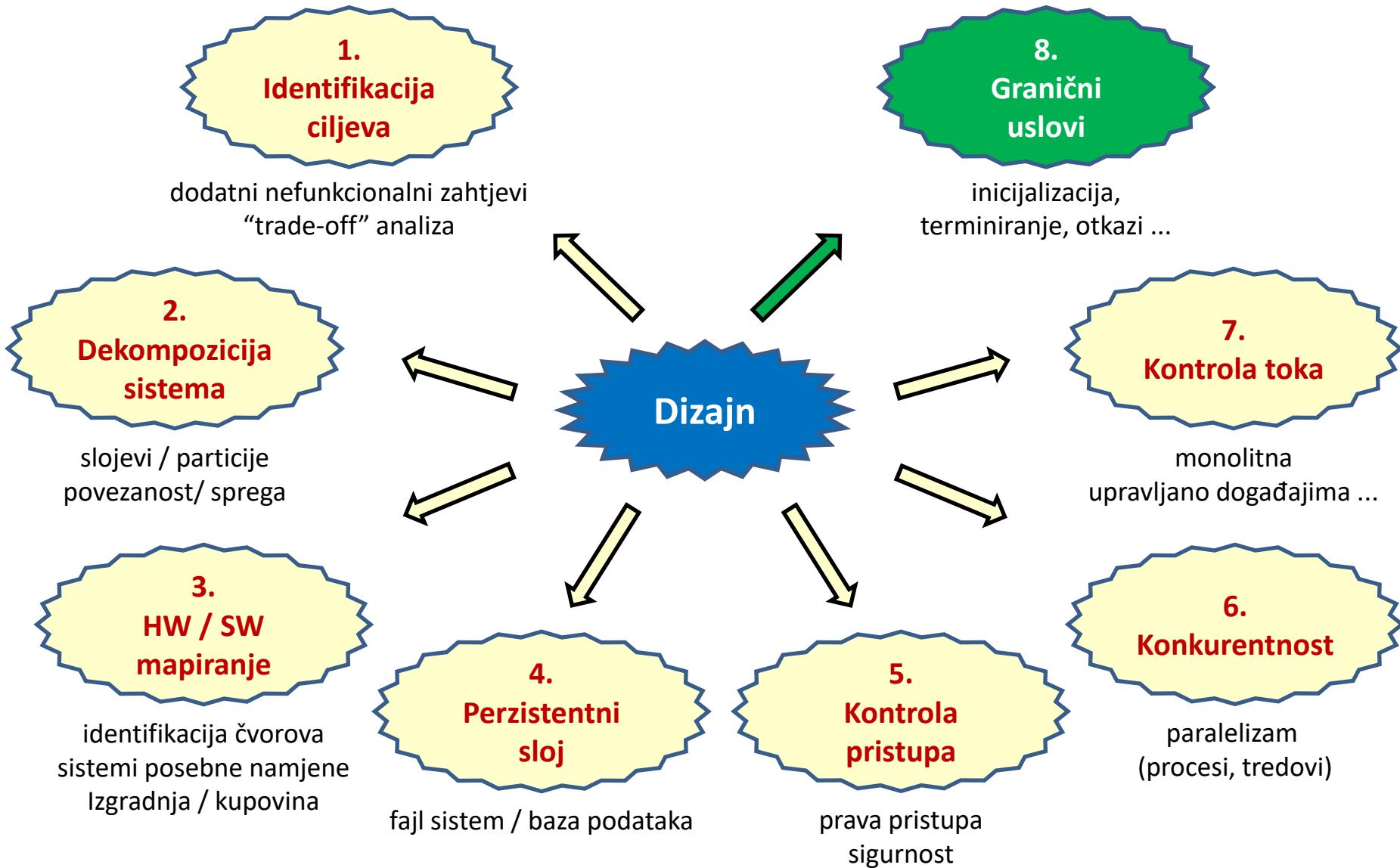
**UNIVERZITET U BANJOJ LUCI  
ELEKTROTEHNIČKI FAKULTET**

**Prof. dr Dražen Brđanin**

**PROJEKTOVANJE SOFTVERA  
/granični slučajevi upotrebe/**

**Banja Luka  
2024.**

# 8 bitnih aktivnosti u projektovanju



# 8. Granični uslovi

## Granični uslovi ili granična stanja sistema

- Vrste graničnih stanja:
  - Inicijalizacija / start sistema
    - početak rada, prelaz iz neinicijalizovanog u ustaljeno stanje
    - tipična pitanja:
      - Kojim podacima se pristupa za vrijeme startovanja sistema?
      - Kako se ponaša korisnički interfejs za vrijeme startovanja sistema?
      - Koje procedure/servisi treba da se izvrše?
  - Terminiranje / završetak rada
    - dealokacija resursa, notifikacija drugih sistema, ...
    - tipična pitanja:
      - Mogu li podsistemi pojedinačno da završavaju rad?
      - Da li druge podsisteme treba obavijestiti o terminiranju rada podsistema?
      - Kako se promjene podataka proslijeduju/upisuju u bazu?
  - Otkazi
    - greške, bagovi, spoljni uzroci (nestanak napajanja, hardverski otkazi)
    - tipična pitanja:
      - Kako se dati podsistem ponaša u slučaju otkaza ili pada nekog čvora / linka?
      - Kako se sistem oporavlja u slučaju otkaza?
    - dobro projektovan sistem ima sposobnost “predviđanja” otkaza i ima mehanizme za manipulaciju i oporavak od otkaza

# 8. Granični uslovi

## Granični slučajevi upotrebe

- Granična stanja sistema modeluju se **graničnim slučajevima upotrebe** (*boundary use cases*)
- Alternativno se koristi i termin **administrativni slučajevi upotrebe**, jer je korespondentni učesnik najčešće sistem administrator
- Granični slučajevi upotrebe **tipično se ne identifikuju tokom analize** sistema, jer su mnogi granični slučajevi posljedica dizajna (npr. administrativne funkcije, kontrola pristupa, ...)
- Granični slučajevi upotrebe **tipično se identifikuju tokom dizajna**, kroz analizu podsistema i perzistentnih objekata:
  - **konfiguracija:**
    - za svaki perzistentni objekat se analiziraju slučajevi upotrebe u kojima se perzistentni objekti kreiraju, uništavaju, arhiviraju, ...
    - ako ne postoji odgovarajući slučaj upotrebe u kojem se dešavaju granična stanja perzistentnih objekata, treba dodati novi granični slučaj upotrebe
  - **start-up/shut down:**
    - za svaku komponentu treba dodati granične slučajeve upotrebe za start, shut down i konfiguraciju
    - nekad jedan slučaj upotrebe može da kontroliše istu aktivnost nad više komponenata
  - **manipulacija izuzecima:**
    - za svaku komponentu treba analizirati moguće otkaze i načine na koje komponenta treba da se ponaša u slučaju izuzetaka
    - Može da se modeluje dodatnim SU koji proširuje funkcionalnosti osnovnog SU

**UNIVERZITET U BANJOJ LUCI  
ELEKTROTEHNIČKI FAKULTET**

**Prof. dr Dražen Brđanin**

**PROJEKTOVANJE SOFTVERA  
/dokumentovanje sistemskog dizajna/**

**Banja Luka  
2024.**

# Dokumentovanje sistemskog dizajna

**Dokument kojim rezultuje sistemska dizajn  
(System Design Document):**

- **Uvodni dio**
  - kratak pregled softverske arhitekture i projektnih ciljeva,
  - Veza sa drugim dokumentima (specifikacija zahtjeva, tehnička specifikacija)
- **Postojeće stanje**
  - Arhitektura postojećeg sistema (ako se radi reinženjering), ili pregled postojećih relevantnih arhitektura za dati sistem (ako se realizuje novi sistem)
- **Projektovano stanje**
  - Kratak pregled predložene arhitekture iz ptičje perspektive i funkcionalnosti podistema
  - Specifikacija po pojedinim tačkama iz prezentacije

**Šablon za dokumentovanje  
sistemskega dizajna**

- 1. Uvod**
  - 1.1. Namjena sistema
  - 1.2. Projektni ciljevi
  - 1.3. Definicije, skraćenice
  - 1.4. Referentni dokumenti
  - 1.5. Kratak pregled dokumenta i predložene arhitekture
- 2. Arhitektura postojećeg sistema**
- 3. Predložena arhitektura**
  - 3.1. Kratak pregled arhitekture i funkcionalnosti podistema
  - 3.2. Dekompozicija sistema
  - 3.3. HW/SW mapiranje
  - 3.4. Perzistentni sloj
  - 3.5. Kontrola prava pristupa i sigurnost
  - 3.6. Kontrola toka
  - 3.7. Granična stanja sistema

**UNIVERZITET U BANJOJ LUCI**  
**ELEKTROTEHNIČKI FAKULTET**

**Prof. dr Dražen Brđanin**

**PROJEKTOVANJE SOFTVERA**  
***/ reuse /***

**Banja Luka**  
**2024.**

# ***Reuse***

**Skup aktivnosti i tehnika kojima se postojeće komponente ili obrasci prilagođavaju za novu primjenu.**

## **Glavni ciljevi:**

- Ponovna upotreba znanja stečenih u prethodnim projektima
- Ponovna upotreba postojećih funkcionalnosti & artefakata

## **Vidovi ponovne upotrebe: *Black-box reuse / White-box reuse***

### ***Black-box reuse: KOMPOZICIJA***

- Nova funkcionalnost realizuje se agregacijom/kompozicijom
- Novi objekat sa više funkcionalnosti realizuje se kao kompozicija koja sadrži postojeći objekat

### ***White-box reuse: NASLJEĐIVANJE***

- Nova funkcionalnost realizuje se nasljeđivanjem/specijalizacijom
- Novi objekat sa više funkcionalnosti je specijalizovani postojeći objekat

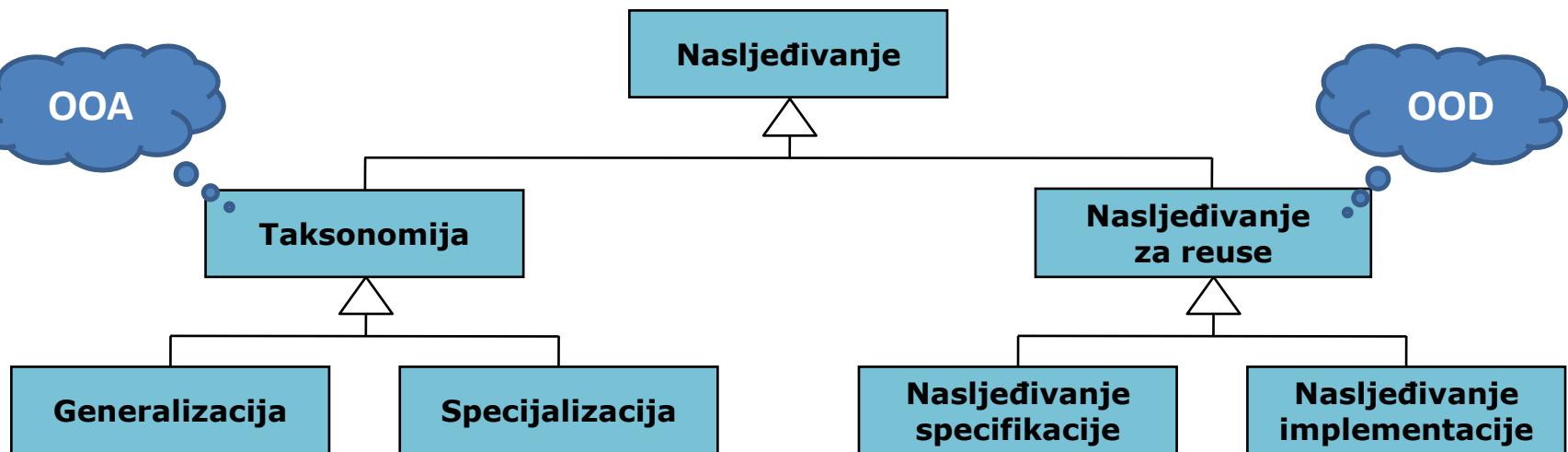
# *Reuse*

## NASLJEĐIVANJE u kontekstu ponovne upotrebe

- Koncept nasljeđivanja koristi se na četiri različita načina:
  - **generalizacija** (*generalization*)
  - **specijalizacija** (*specialization*)
  - **nsljeđivanje specifikacije** (*specification inheritance*)
  - **nsljeđivanje implementacije** (*implementation inheritance*)

} OOA  
} OOD

## Meta-model nasljeđivanja



# *Reuse*

## “Otkrivanje” veza nasljeđivanja u objektnom modelu

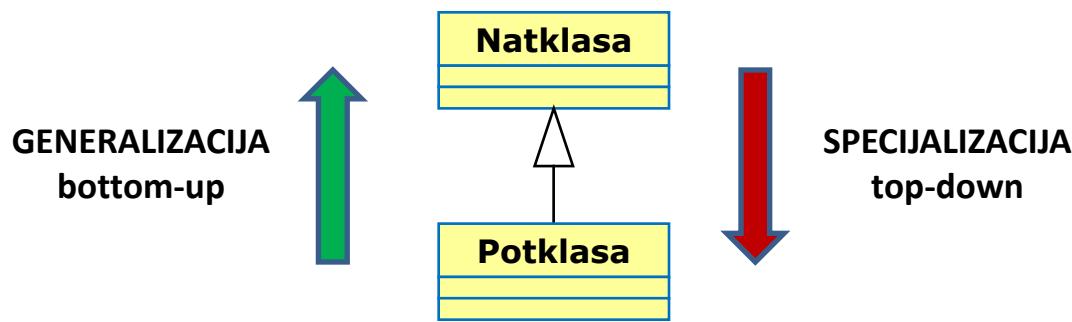
- nasljeđivanje se “otkriva” postupkom **generalizacije** ili **specijalizacije**

### generalizacija

- “**bottom-up**” princip identifikacije nasljeđivanja
- na osnovu postojeće klase (potklasa) vrši se generalizacija (uopštavanje) i identificuje natkласа

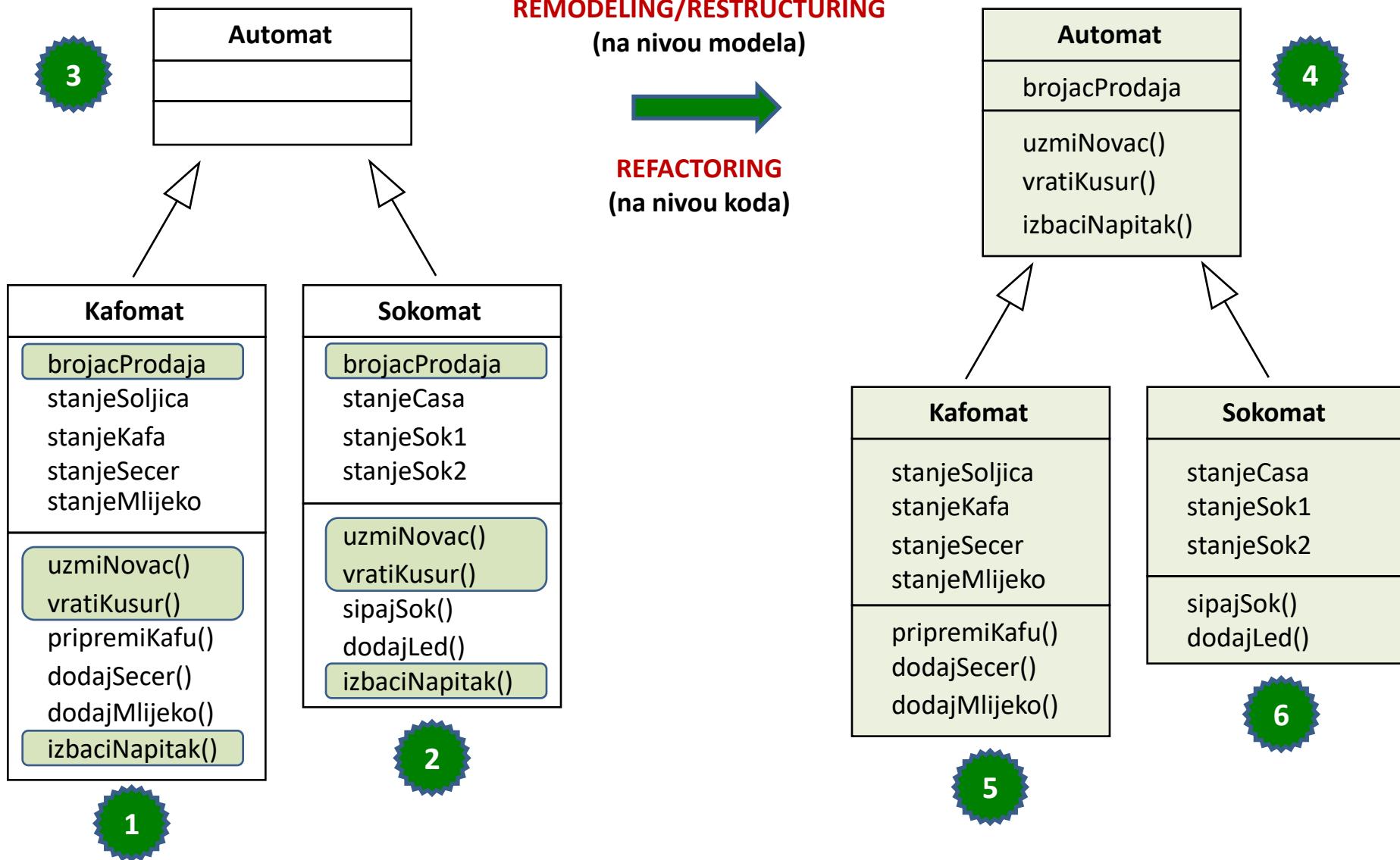
### specijalizacija

- “**top-down**” princip identifikacije nasljeđivanja
- na osnovu postojeće klase (natklasa) vrši se specijalizacija i identificuje potklasa



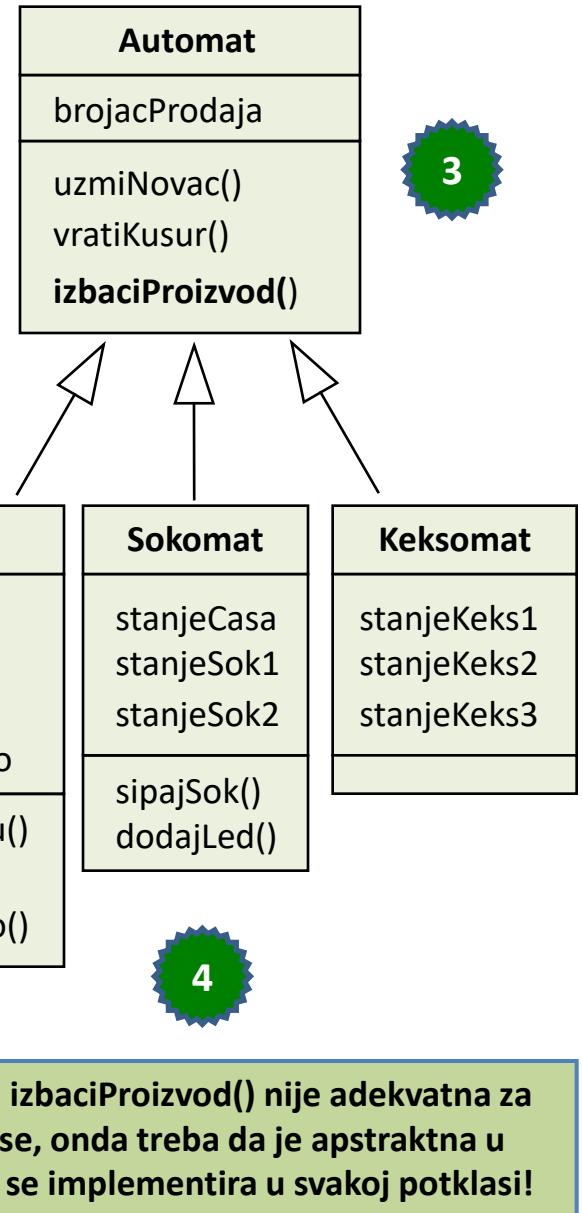
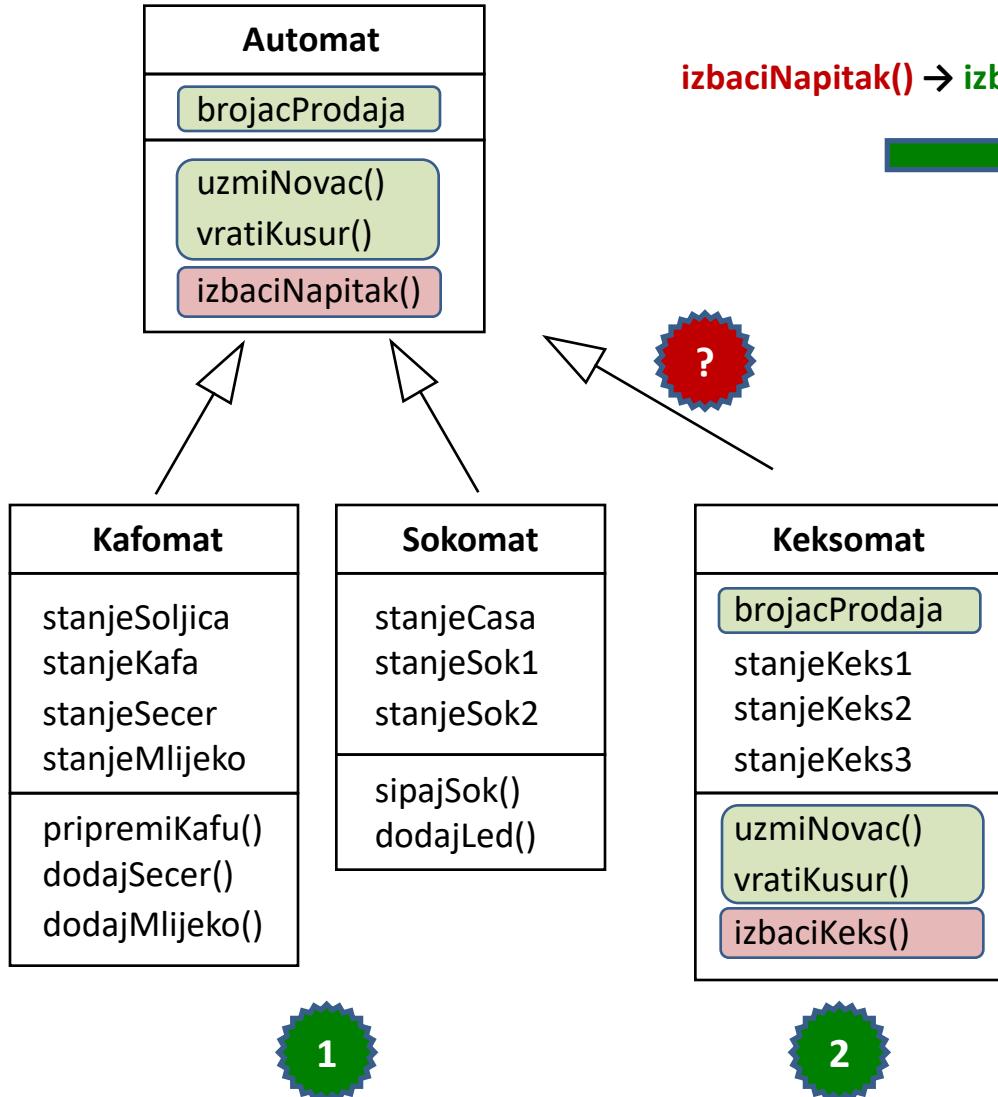
# *Reuse*

Primjer GENERALIZACIJE:



# Reuse

Primjer SPECIJALIZACIJE:



# *Reuse*

## Nasljeđivanje SPECIFIKACIJE / Nasljeđivanje IMPLEMENTACIJE

- Koristi se u OOD u cilju smanjenja redundantnosti i omogućavanja proširljivosti dizajna

### **nasljeđivanje specifikacije**

(*specification inheritance*)

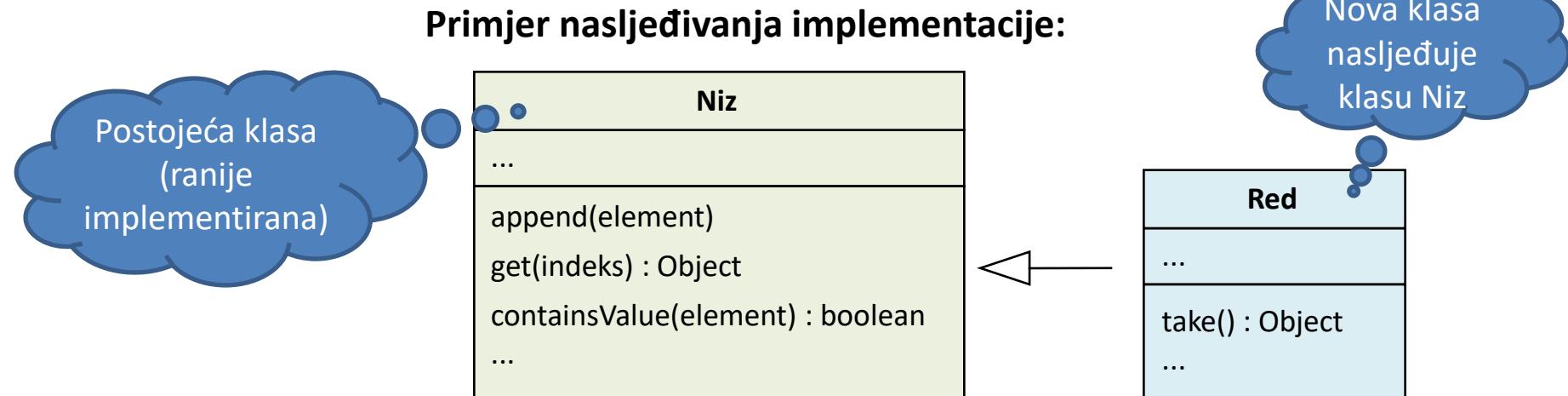
- Specifikacija je apstraktna klasa (klasa sa operacijama bez implementacije)
- Često se naziva i **subtyping** (**specijalizacija tipa/interfejsa**)
- Koristi se za klasifikaciju tipova (**hijerarhija tipova**)

### **nasljeđivanje implementacije**

(*implementation inheritance*)

- Često se naziva i **nasljeđivanje klase** = **nasljeđivanje konkretnе klase** (implementirane operacije)
- **Cilj: proširivanje skupa funkcionalnosti korišćenjem funkcionalnosti neke postojeće klase**

### Primjer nasljeđivanja implementacije:

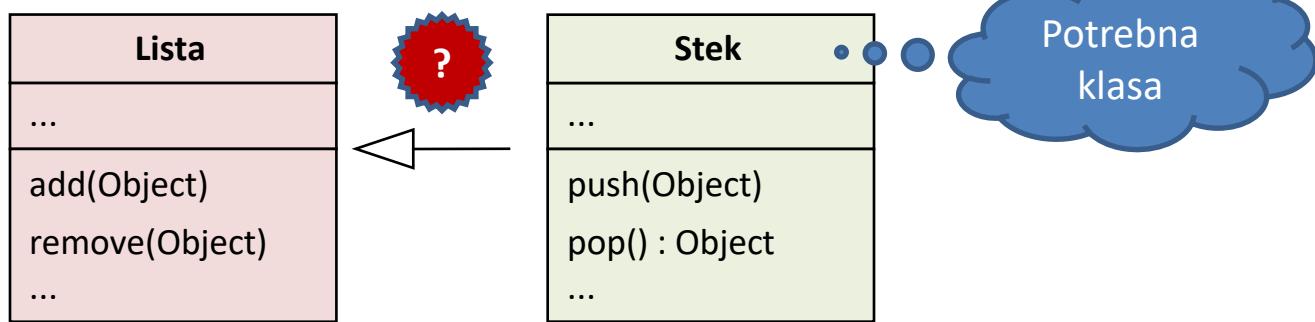


# *Reuse*

## Problemi kod nasljeđivanja IMPLEMENTACIJE

- Često postojeća klasa nije u potpunosti pogodna za dobijanje željene potklase
  - Npr. postojeća klasa ima suvišne operacije ili operacije koje ne daju poželjan rezultat

Primjer neprilagođene postojeće osnovne klase:



## Prevazilaženje problema kod nasljeđivanja IMPLEMENTACIJE

- Problem suvišne operacije ili operacije koja ne daje poželjan rezultat, može da se riješi:
  - Nasljeđivanjem
    - Dodavanjem nove operacije u osnovnu klasu (ako je moguće)
    - Redefinisanjem operacije u izvedenoj klasi
  - Delegacijom

# *Reuse*

## Nasljeđivanje SPECIFIKACIJE

- Često se naziva i **subtyping**
- **Koristi se za klasifikaciju tipova (hijerarhija tipova)**

### **Princip SUPSTITUCIJE (Liskov, 1988)**

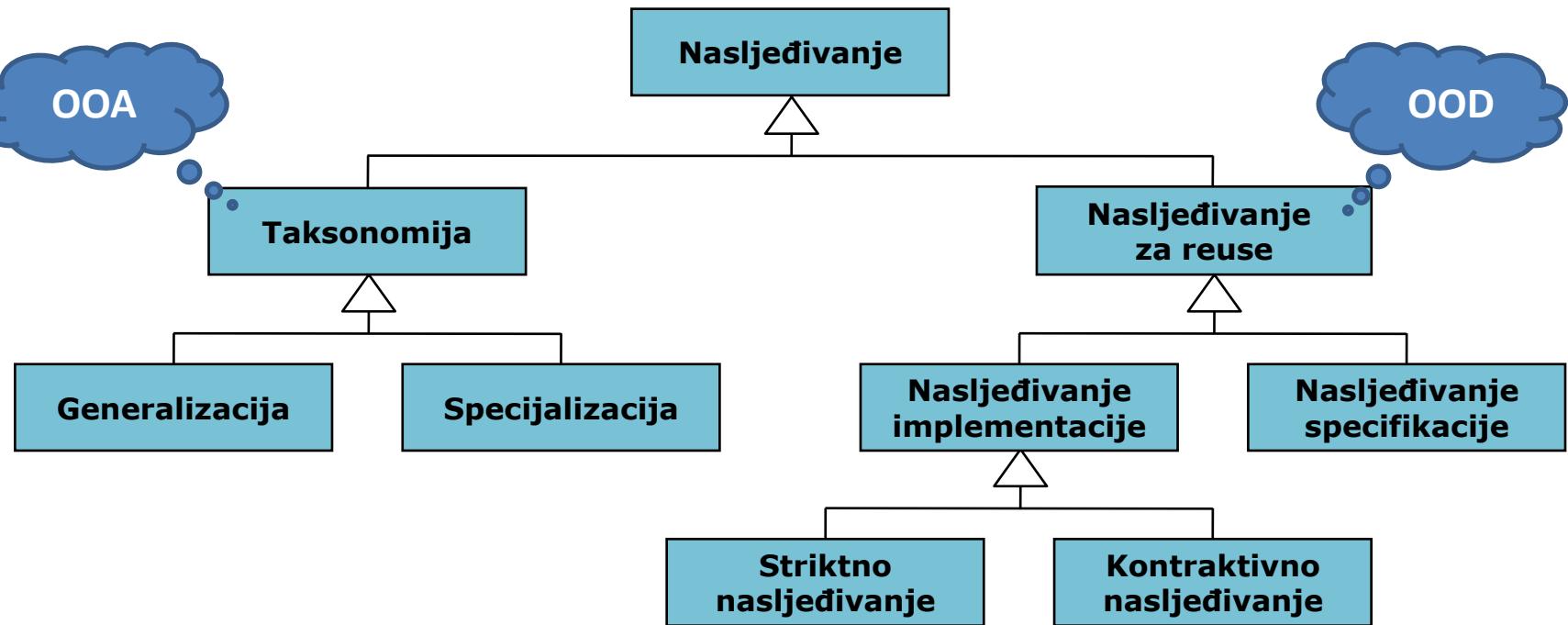
- Formalna definicija nasljeđivanja specifikacije:

**Ako objekat tipa S može bilo gdje da zamijeni očekivani objekat tipa T, tada je S podtip od T.**

- Interpretacija:
  - Za sve klase koje su podtipovi neke superklase vrijedi da **sve veze nasljeđivanja predstavljaju nasljeđivanje specifikacije**, odnosno da neka metoda kako je definisana u superklasi mora biti dostupna i u bilo kojoj potklasi, i klijent može da je koristi kao da pristupa superklasi.
  - Postojećoj hijerarhiji klasa (tipova) **može da se dodaje nova potkласа bez modifikacije метода iz superklase**, čime se omogućava proširljivost.
- **Nasljeđivanje u skladu sa principom supstitucije naziva se striktno nasljeđivanje, a ako nije u skladu sa principom supstitucije naziva se kontraktivno nasljeđivanje.**

# *Reuse*

## Revidirani meta-model nasljeđivanja



# *Reuse*

## Delegacija

Delegacija je tehnika kojom klasa implementira operaciju prosljeđujući poruku drugoj klasi.

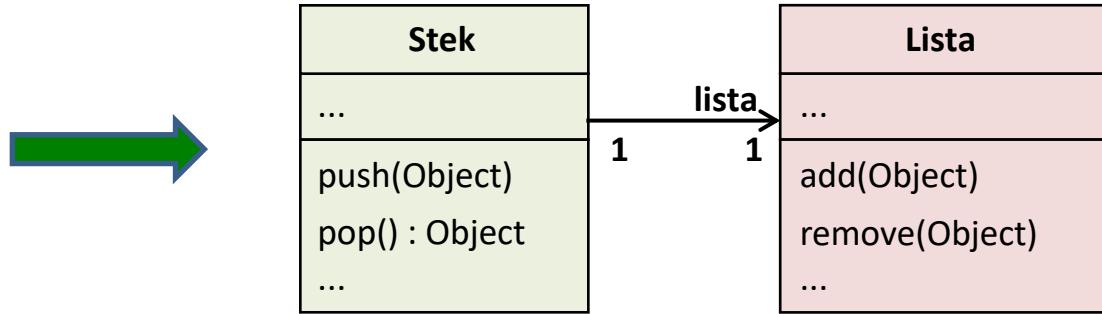
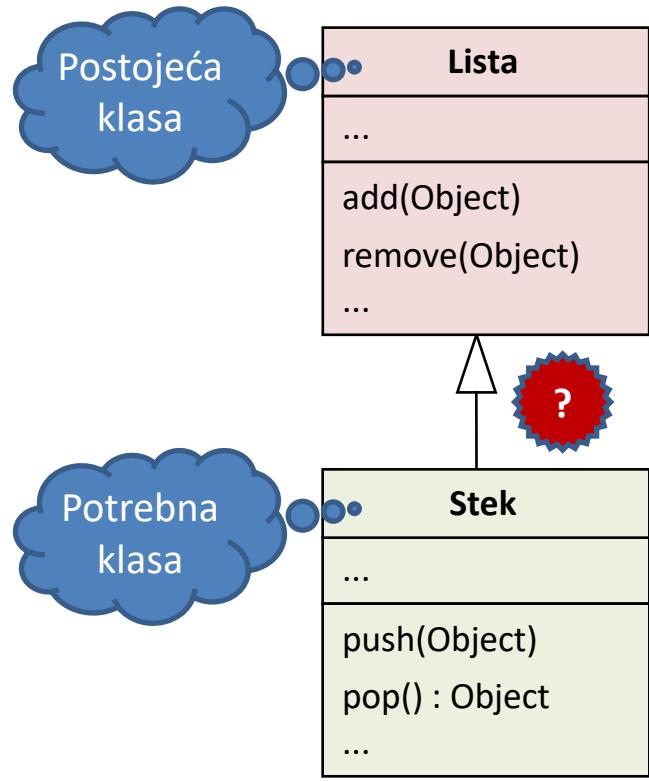
- U rješavanju zahtjeva nekog klijenta učestvuju dva objekta:
  - delegator (proxy) prima zahtjev klijenta i prosljeđuje ga delegatu
  - delegator provjerava/štiti delegata od “zloupotrebe” od strane klijenta



- Primjena delegacije za reuse postojećih klasa:
  - Delegacija je ekvivalentna kompoziciji
  - Delegacija predstavlja bolji mehanizam za reuse postojeće klase:
    - delegacija obezbjeđuje novu klasu od neželjenog ponašanja iz postojeće klase
    - nova klasa ne nasljeđuje ništa iz postojeće klase, pa se ne ponaša kao postojeća klasa i nema straha da će doći do pogrešne upotrebe nove klase umjesto postojeće
    - delegacija ne zahtijeva nikakve izmjene na postojećoj klasi

# Reuse

Primjer DELEGACIJE:



Rezultantni kod:

```
// Implementacija Steka delegacijom preko Liste
class Stek
{
    private Lista lista;
    Stek() { lista = new Lista(); }
    void push(Object element)
    {
        ...
        lista.add(element);
        ...
    }
    ...
}
```

# *Reuse*

## Najvažnije prednosti višestruke upotrebe dizajna i implementacije

- efikasnost razvoja (*efficiency*)
  - ponovna upotreba postojećeg dizajna i implementacije **značajno skraćuje vrijeme i ubrzava razvoj i smanjuje cijenu razvoja** ciljnog sistema
- robusnost i pouzdanost (*robustness / reliability*)
  - provjerene komponente i obrasci rezultat su dugogodišnjeg rada velikog broja domenskih eksperata, pa njihova primjena **smanjuje rizik od pogrešnog dizajna i povećava pouzdanost** ciljnog sistema
- proširljivost i prilagodljivost (*extensibility / adaptability*)
  - AF omogućava **proširivanje postojeće funkcionalnosti i prilagođavanje za konkretnu primjenu**
- modularnost (*modularity*)
  - AF poboljšavaju **modularnost**, jer AF uvode komponente sa stabilnim implementacijama i dobro definisanim interfejsima
- olakšana komunikacija
  - projektni obrasci imaju jedinstvene nazine, što **uniformiše korišćenu terminologiju i olakšava komunikaciju** između pojedinaca / projektnih tim(ov)a

# ***Reuse – Aplikativni radni okviri***

## ***Application framework – AF***

- Parcijalna aplikacija koja se koristi kao osnov za produkciju prilagođene i specijalizovane aplikacije za konkretan domen (Johnson & Foote, 1988)
- Prije pojave AF, u razvoju softvera *reuse* funkcionalnosti ostvarivao se pomoću:
  - biblioteka funkcija (razvoj softvera primjenom proceduralnih prog. jezika)
  - biblioteka klase (razvoj softvera primjenom O-O prog. jezika), npr. STL
- AF su u softversko inženjerstvo donijeli **reuse dizajna i reuse koda**
- AF omogućavaju **visok nivo višestruke upotrebe koda (large scale reuse)**
- AF definišu dizajn i implementaciju specifičnih softverskih sistema/komponenata
- AF definiše:
  - dizajn ciljnog softverskog sistema
  - osnovne klase, odgovornosti i interakcije objekata
  - kontrolu toka (izvršavanje uvijek započinje unutar AF koda)
- **Mehanizmi za proširenje** AF funkcionalnosti (*extension points*)
  - nasljeđivanje (*hook* metode) – “white box”
  - delegacija – “black box”

# *Reuse – AF*

## Klasifikacija AF prema poziciji u softverskom procesu

### Infrastrukturni AF

- namijenjeni za pojednostavljenje softverskog procesa i brži razvoj pojedinih softverskih podsistema (arhitektura, GUI, perzistencija, web aplikacije, ...)
- infrastrukturni AF koriste se u okviru softverskog projekta i tipično se ne isporučuju klijentu
- **tipični primjeri infrastrukturnih AF:**
  - **za razvoj GUI** – AF koji realizuju GUI
    - npr. Swing, JavaFX, ...
  - **za perzistenciju** – AF koji realizuju ORM
    - npr. Hibernate, Django, ...
  - **za razvoj softverskih alata vezanih za modelovanje**
    - npr. EMF (Eclipse Modeling Framework)
  - **web AF** – AF za razvoj web aplikacija (arhitektura, web servisi, API)
    - npr. Angular, Django, ASP.NET, Meteor, Spring, ...



# ***Reuse – AF***

## **Klasifikacija AF prema poziciji u softverskom procesu**

### ***Middleware AF***

- koriste se za integraciju distribuiranih aplikacija i komponenata
- tipični primjeri: RMI, JMS, CORBA, DCOM, ...

mAF su ključni za  
brz i kvalitetan razvoj  
softvera!

### ***Enterprise AF***

- aplikativno specifični i fokusirani na konkretnе domene
- neki primjeri:
  - **ERP (*Enterprise Resource Planning*)** – poslovni informacioni sistemi (upravljanje dokumentima, magacinsko poslovanje, kupci-dobavljači, ...)
    - MiTOS, Apache OFBiz, ...
  - **eZdravstvo (*eHealth*)**
    - OpenEHR, ...
  - **javne nabavke (*public procurement*)**
    - EllectraWEB

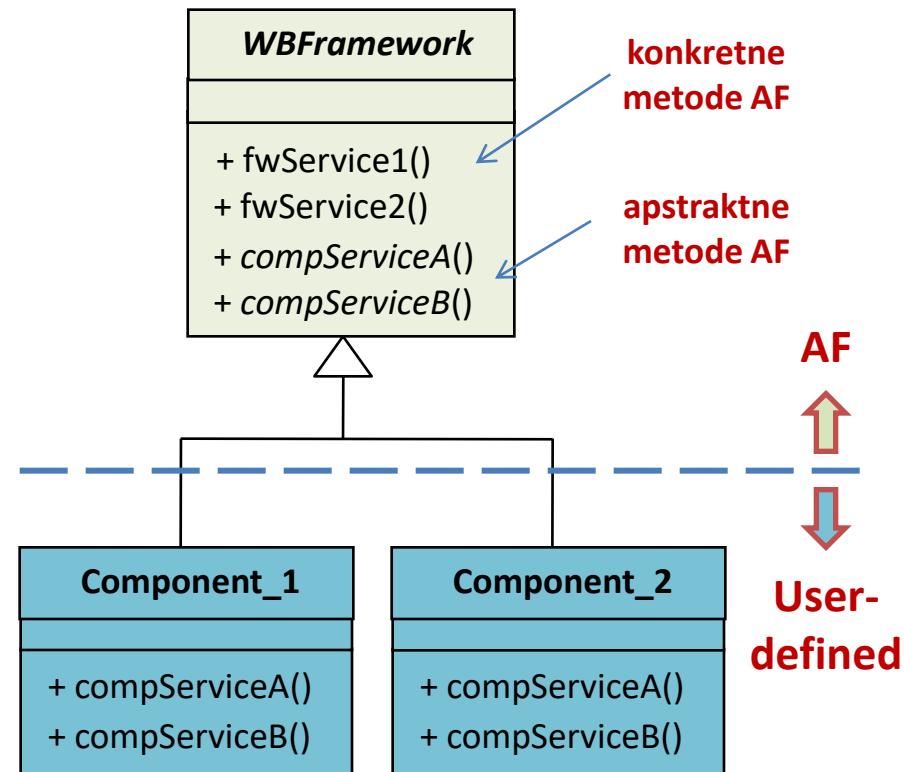
eAF pogodni za razvoj  
end-user aplikacija!

# *Reuse – AF*

## Klasifikacija AF prema načinu proširivanja funkcionalnosti

### *“white box” AF*

- mehanizmi za proširivanje funkcionalnosti:  
**nasljeđivanje i dinamičko vezivanje**  
(*dynamic binding*),
- postojeće AF funkcionalnosti proširuju se  
**nasljeđivanjem osnovnih klasa iz AF i redefinisanjem *hook* metoda primjenom odgovarajućih obrazaca** (npr. *template method*)
- da bi se proširila postojeća funkcionalnost,  
**neophodno je poznavanje interne strukture AF**
- zbog velike sprege sa hijerarhijom klasa u AF, izvedeni softverski sistemi moraju da se rekompajliraju u slučaju izmjena u AF
- primjenjuje se princip “**inverzija kontrole**” (*inversion of control*) – metoda iz osnovne klase iz AF poziva redefinisanu metodu proširenja



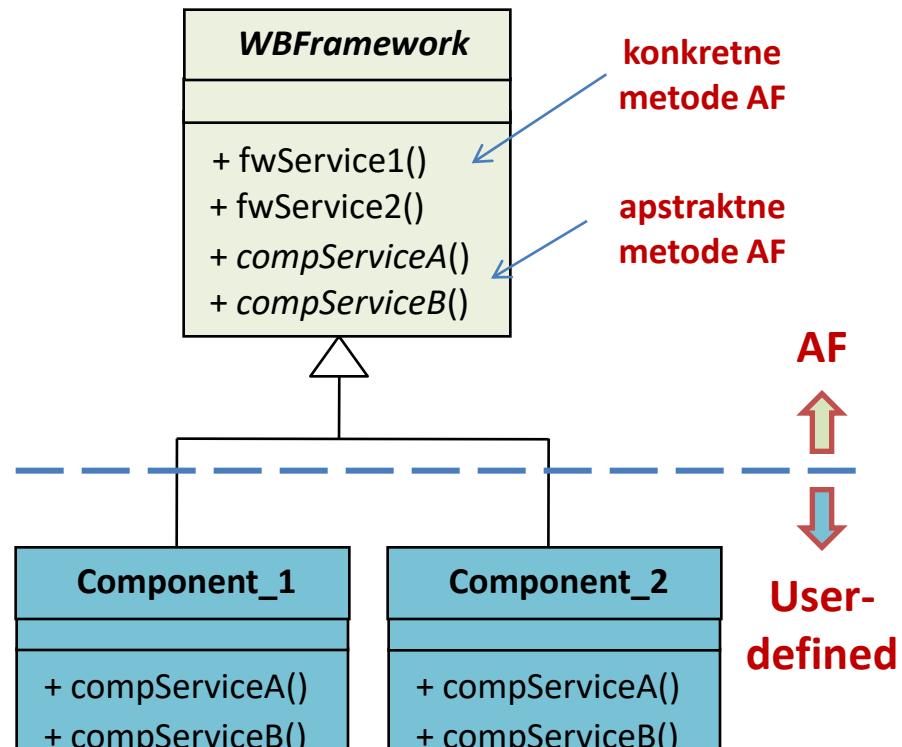
# ***Reuse – AF***

## ***“white box” AF***

```
package whiteboxAF;  
public abstract class KlasaAF  
{  
    public void f() { metoda(); }  
    protected abstract void metoda();  
}
```

```
package myAppWhitebox;  
import whiteboxAF.KlasaAF;  
public class MyApp extends KlasaAF  
{  
    @Override  
    protected void metoda() { // spec.impl }  
}
```

```
package myAppWhitebox;  
public class App  
{  
    public static void main(String args[])  
    {  
        MyApp m = new MyApp();  
        m.f();  
    }  
}
```

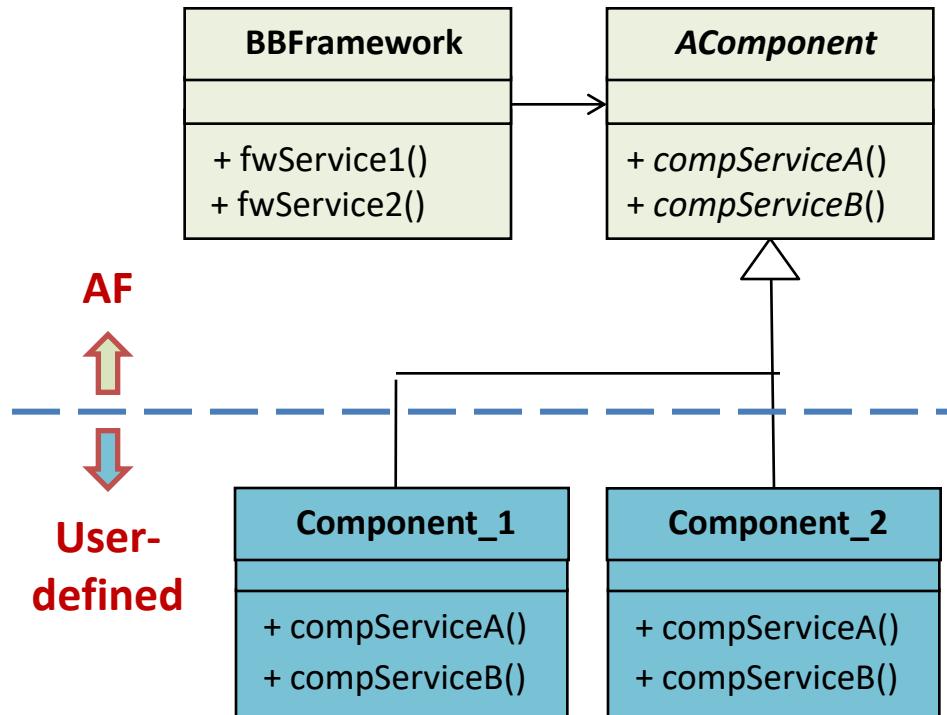


# *Reuse – AF*

## Klasifikacija AF prema načinu proširivanja funkcionalnosti

### *“black box” AF*

- proširivanje funkcionalnosti zasniva se na definisanju odgovarajuće sprege sa AF komponentama
- postojeće funkcionalnosti se višestruko koriste definisanjem komponenata u skladu sa specifičnim interfejsom i integracijom tih komponenata sa AF **primjenom delegacije**
- korišćenje *black box* AF je **jednostavnije** od *white box* AF, jer se zasniva na primjeni delegacije, a ne na nasljeđivanju
- razvoj *black box* AF je **složeniji** nego razvoj *white box* AF, jer neophodno definisanje interfejsa i *hook* metoda za širok spektar slučajeva upotrebe



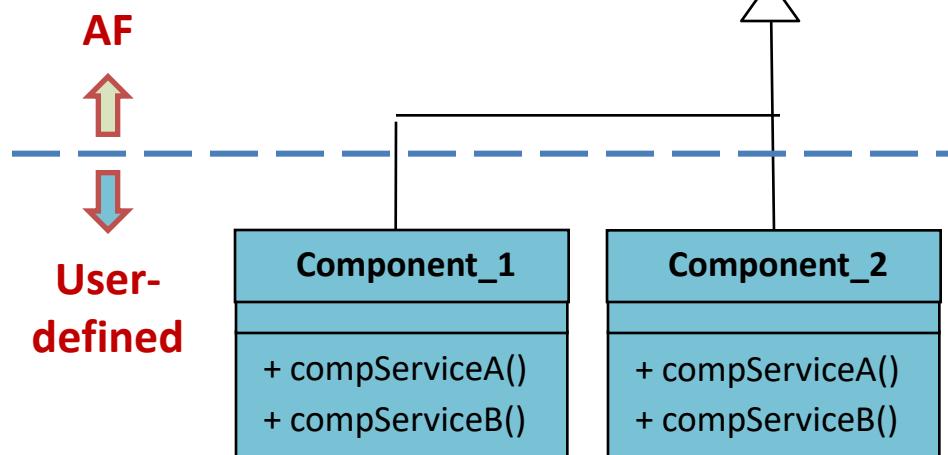
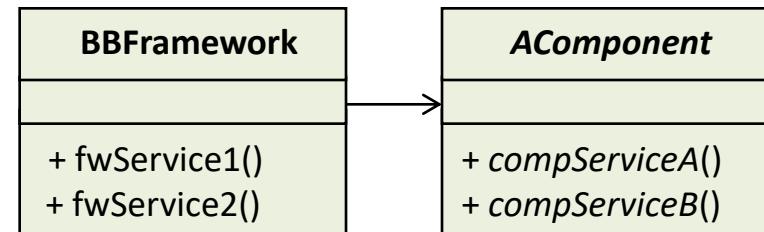
# *Reuse – AF*

**“black box” AF**

```
package blackboxAF;  
public final class KlasaAF {  
    public void fun(Funktionalnost f)  
    { f.metoda(); }  
}
```

```
package blackboxAF;  
public interface Funktionalnost {  
    public void metoda();  
}
```

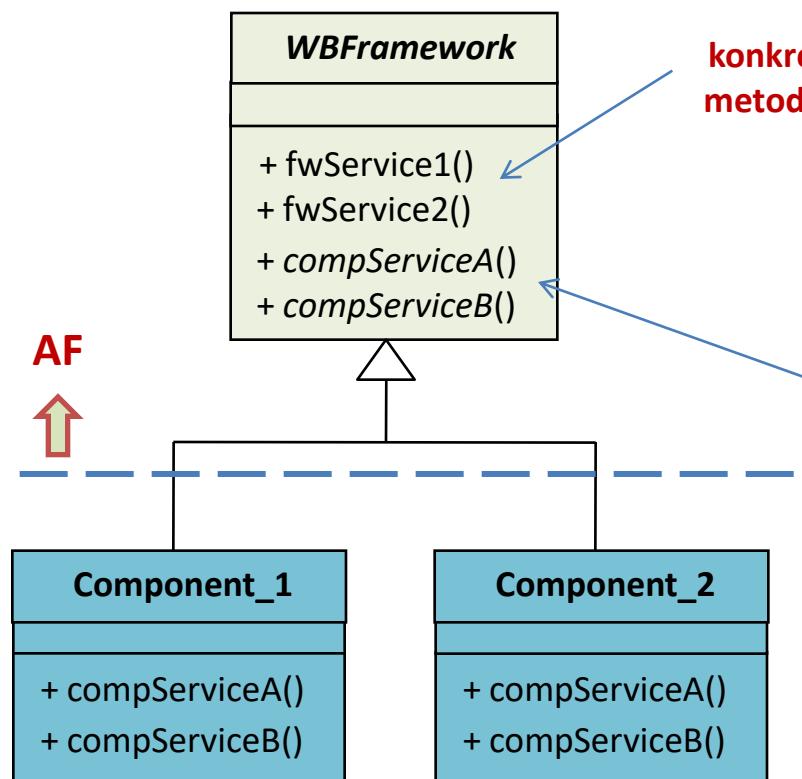
```
package myAppBlackbox;  
public class myF implements Funktionalnost  
{  
    public void metoda() { // impl. }  
}  
  
public class App  
{  
    public static void main(String args[])  
    {  
        myK mk = new KlasaAF();  
        mk.fun(new myF());  
    }  
}
```



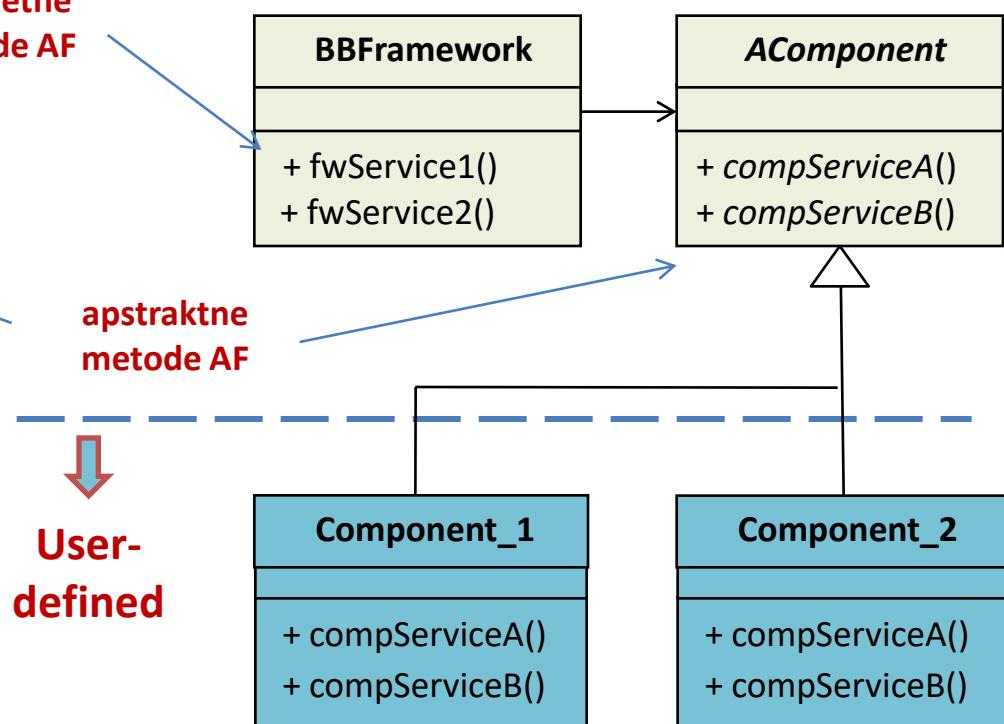
# *Reuse – AF*

## Klasifikacija AF prema načinu proširivanja funkcionalnosti

### *“white box” AF*



### *“black box” AF*



# *Reuse* – AF

## AF ↔ biblioteke klasa

- Klase u **AF** su u međusobnim vezama koje obezbjeđuju višestruko upotrebljiv **arhitekturni skeleton** za familiju sličnih aplikacija.
- **Biblioteke klasa** (npr. kontejnerske klase) su manje domenski specifične i omogućavaju *reuse* manjeg obima, ali **u različitim domenima**.
- **Biblioteke klasa su tipično pasivne** – ne implementiraju niti ograničavaju kontrolu toka.
- **AF su aktivni** – kontrolišu tok u aplikaciji.
  - **princip inverzije kontrole** (metoda iz osnovne klase iz AF poziva redefinisanu metodu proširenja)
- U praksi se **AF i biblioteke klasa zajedno** koriste u realizaciji sistema – AF tipično koriste biblioteke funkcija za razvoj i proširivanje funkcionalnosti AF, implementacija specifičnog aplikativnog koda takođe se zasniva na bibliotekama klasa (obrada stringova, manipulacija fajlovima, ...)

# *Reuse – AF*

## AF ↔ projektni obrasci

- AF je parcijalna aplikacija koja se koristi kao osnov za produkciju prilagođene i specijalizovane aplikacije za konkretan domen.
- AF se fokusira na **višestruku upotrebu konkretnog dizajna**, algoritama i **implementacije** u konkretnom programskom jeziku i u konkretnom aplikativnom domenu.
- Projektni obrasci fokusiraju se na **apstraktan dizajn** i male kolekcije povezanih klasa za primjene u različitim domenima.
- Projektni obrazac je uopšteno (šablonsko) rješenje za tipične projektne probleme (strukturni obrasci / obrasci ponašanja / kreacioni obrasci), koje omogućava višestruku upotrebu u rješavanju konkretnih projektnih situacija u različitim domenima.
- Projektni obrasci nisu fokusirani na implementaciju (nego na projektovanje), a AF predstavljaju parcijalno implementiranu aplikaciju.

# ***Reuse – AF***

## **Ključne koristi / prednosti primjene AF**

- **višestrukost upotrebe (reusability):**
  - veliko domensko znanje i iskustvo projektanata ugrađeno u razvoj AF značajno skraćuje vrijeme implementacije
  - nema potrebe za ponovnim razvojem i validacijom
- **proširljivost (extensibility):**
  - AF tipično raspolaže **hook metodama**, koje se redefinišu u aplikaciji, čime se AF proširuje i prilagođava za konkretnu primjenu
- **modularnost (modularity):**
  - AF poboljšavaju modularnost, jer AF uvode komponente sa stabilnim implementacijama i dobro definisanim interfejsima
- **efikasnost razvoja (efficiency):**
  - primjena AF značajno skrajuće vrijeme i ubrzava razvoj ciljnog sistema

# Projektni obrasci (*Design patterns*)

- Projektni obrazac predstavlja **uopšteno (šablonsko) rješenje za tipične projektne probleme**, koje omogućava **višestruku upotrebu u rješavanju konkretnih projektnih situacija**.
- Projektni obrazac predstavlja **jezgro za rješavanje konkretnih problema iste/slične prirode – svaki put isto jezgro uz prilagođavanje konkretnoj situaciji**.
- Mnogi projektni obrasci su **sistematično dokumentovani** i mogu slobodno da se **koriste** u projektovanju.
- Projektni obrasci su dobar mehanizam za **učenje projektovanja na osnovu tuđih iskustava** (višegodišnje, višestruko rafinirano iskustvo u rješavanju klasa sličnih/srodnih problema)!
- Projektni obrasci imaju dugu istoriju:
  - 1960-70: **Christopher Alexander** uveo pojам projektnih obrazaca u arhitekturi i urbanizmu,
  - 1980-90: **OOPSLA** (*Object-Oriented Programming, Systems, Languages and Applications*)
    - projektni obrasci u O-O softverskom inženjerstvu
    - **Gang of Four (GoF)** obrasci:  
*E. Gamma, R. Helm, R. Johnson, J. Vlissides:*  
**“Design Patterns: Elements of Reusable Object-Oriented Software”**, Addison-Wesley, 1995.

# GoF projektni obrasci

- GoF (*Gang of Four*) projektni obrasci – katalog sa 23 fundamentalna obrasca
- Karakteristike svakog projektnog obrasca:
  - **naziv**: jedinstveno ime (kratko, 1-2 riječi), koje dobro reprezentuje namjenu obrasca
  - **klasifikacija** :
    - **kreacioni obrazac** (ako je fokusiran na kreiranje objekata)
    - **strukturni obrazac** (ako je fokusiran na strukturu)
    - **obrazac ponašanja** (ako je fokusiran na kolaboraciju objekata)
  - **namjena**: kratak opis (jedna-dvije rečenice) o namjeni obrasca  
npr. "*Prilagođenje interfejsa jedne klase u interfejs kakav očekuje druga klasa*"
  - **aliasi**: alternativni nazivi koji se ponekad koriste
  - **motivacija**: opis projektne situacije/problema koji se rješava primjenom datog obrasca
  - **primjenljivost**: oblasti u kojima se može primijeniti dati obrazac i kako ih prepoznati
  - **struktura**: skup povezanih klasa i interfejsa za rješavanje projektnog problema
  - **učesnici**: kratak opis uključenih objekata i njihovih uloga
  - **kolaboracije**: opis kolaboracija između učesnika (tekstualno i/ili grafički)
  - **posljedice**: prednosti i nedostaci datog obrasca, uključujući preporuke za nedostatke
  - **implementacija**: preporuke za implementaciju obrasca, uključujući korisne "trikove"

# Klasifikacija GoF projektnih obrazaca

strukturni obrasci	kreacioni obrasci	obrasci ponašanja
<b>Adapter</b> (Pregovarač, Prilagođenje)	<b>Abstract Factory</b> (Apstraktna fabrika)	<b>Command</b> (Komanda)
<b>Bridge</b> (Most)	<b>Builder</b> (Graditelj)	<b>Chain of Responsibility</b> (Komandni lanac)
<b>Composite</b> (Kompozicija)	<b>Factory Method</b> (Fabrički metod)	<b>Interpreter</b> (Tumač)
<b>Decorator</b> (Dekorater)	<b>Prototype</b> (Prototip)	<b>Memento</b> (Podsjetnik, Podsjećanje)
<b>Facade</b> (Fasada)	<b>Singleton</b> (Usamljenik)	<b>Iterator</b> (Brojač)
<b>Flyweight</b> (Superlaki)		<b>Mediator</b> (Posrednik)
<b>Proxy</b> (Poslanik)		<b>State</b> (Stanje)
		<b>Observer</b> (Posmatrač, Nadzornik)
		<b>Visitor</b> (Posjetilac)
		<b>Strategy</b> (Strategija)
		<b>Template Method</b> (Šablonski metod)

**UNIVERZITET U BANJOJ LUCI  
ELEKTROTEHNIČKI FAKULTET**

**Prof. dr Dražen Brđanin**

**OBJEKTNNO-ORIJENTISANI DIZAJN  
/kreacioni projektni obrasci/**

**Banja Luka  
2024.**

# Kreacioni projektni obrasci

## Uloga kreacionih projektnih obrazaca

- Kreacioni obrasci pomažu u implementaciji sistema tako što smanjuju zavisnost od načina kreiranja, kompozicije i reprezentacije objekata
- Kreacioni obrasci apstrahuju proces kreiranja (instanciranja) objekata:
  - *class creational patterns*
    - nasljeđivanje omogućava različite načine instanciranja klase
    - predstavnici: *Factory method*
  - *object creational patterns*
    - delegacija instanciranja drugom objektu
    - predstavnici: *Abstract Factory, Prototype, Singleton, Builder*

# Kreacioni projektni obrasci

## Motivacija za kreacione projektne obrasce

- Kreacioni obrasci dobijaju na značaju **kad je kompozicija objekata dominantnija u odnosu na nasljeđivanje, i kad se kreiranje objekata ne svodi samo na puko instanciranje klasa**

## Ključne karakteristike kreacionih projektnih obrazaca

- kreacioni obrasci **inkapsuliraju znanje o konkretnim klasama** koje se koriste u sistemu
- kreacioni obrasci **skrivaju način** na koji se kreiraju i komponuju instance tih klasa (sve što se na višem nivou zna o tim objektima, zna se kroz njihove interfejse/apstraktne klase),
- kreacioni obrasci **omogućavaju veliku fleksibilnost** u smislu: **šta** se kreira, **ko** kreira i **kako**
- kreacioni obrasci **omogućavaju različite konfiguracije** (u smislu strukture i funkcionalnosti) sistema sa tzv. “*product*” objektima, koje mogu biti **statičke** (*compile-time*) ili **dinamičke** (*run-time*)

# Kreacioni projektni obrasci

## naziv

## kratak opis

**Factory Method**  
(Fabrički metod)

**Kreira instance različitih izvedenih klasa**

Definiše interfejs za kreiranje objekata, ali dozvoljava da izvedena klasa odluči koju će klasu da instancira

**Abstract Factory**  
(Apstraktna fabrika)

**Kreira instance različitih familija klasa**

Obezbeđuje interfejs za kreiranje familije povezanih i međusobno zavisnih objekata bez specificiranja njihovih konkretnih klasa

**Builder**  
(Graditelj)

**Odvaja kreiranje objekta od njegove reprezentacije**

Razdvaja kreiranje složenih objekata od njihove reprezentacije tako da se za isti proces konstruisanja mogu kreirati različite reprezentacije

**Prototype**  
(Prototip)

**Potpuno inicijalizovana instanca koja može biti kopirana/klonirana**  
Služi za specifikaciju vrste objekata koji će biti kreirani pomoću tzv. prototipske instance, kao i za kreiranje novih objekata kopiranjem prototipa

**Singleton**  
(Unikat, Usamljenik)

**Klasa koja može imati samo jedan živi objekat**

Obezbeđuje da određena klasa ima maksimalno jedan aktivni objekat, i obezbjeđuje način za pristup tom objektu

**Kreacioni obrasci su usko povezani i često su "konkurenti" za primjenu, npr:**

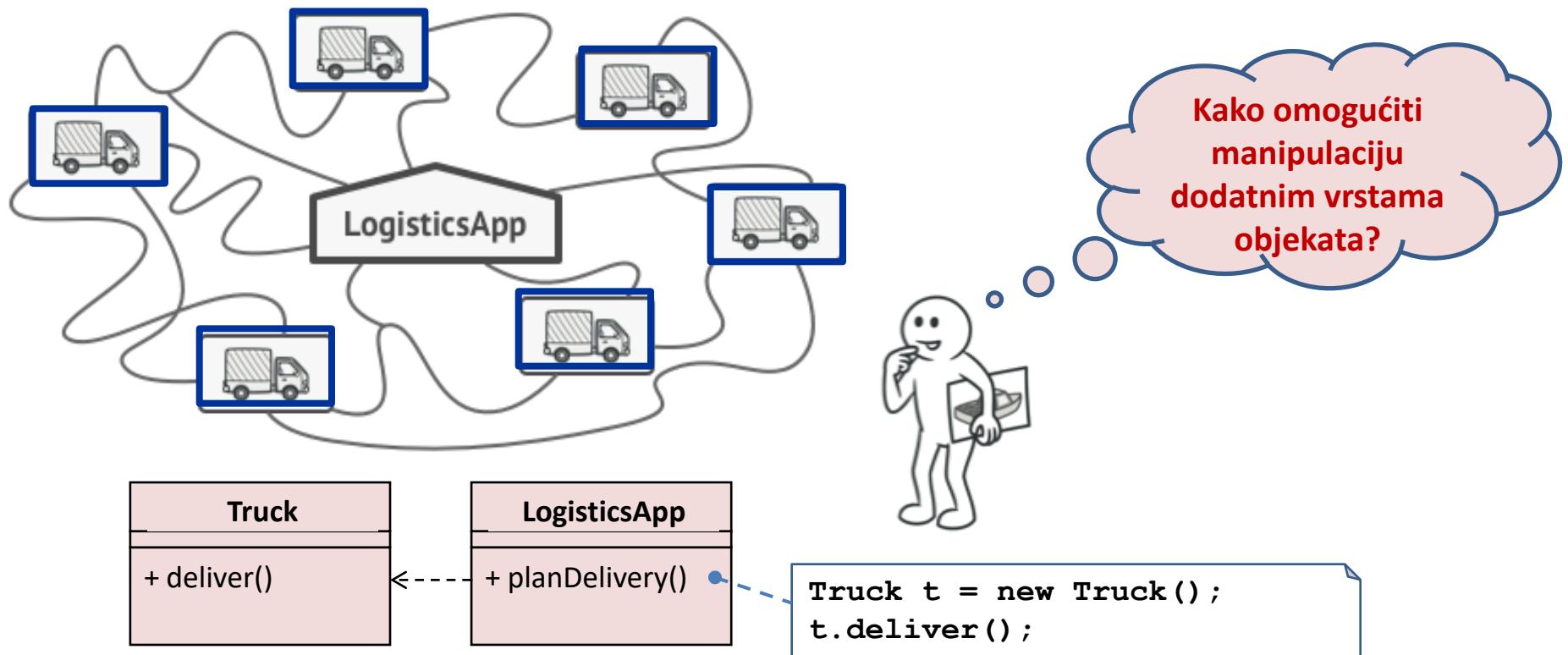
- prototip ili apstraktna fabrika,
- graditelj često koristi druge obrasce za implementaciju komponenata, ...

# Kreacioni obrasci – Fabrički metod

## Motivacija za uvođenje fabričkog metoda

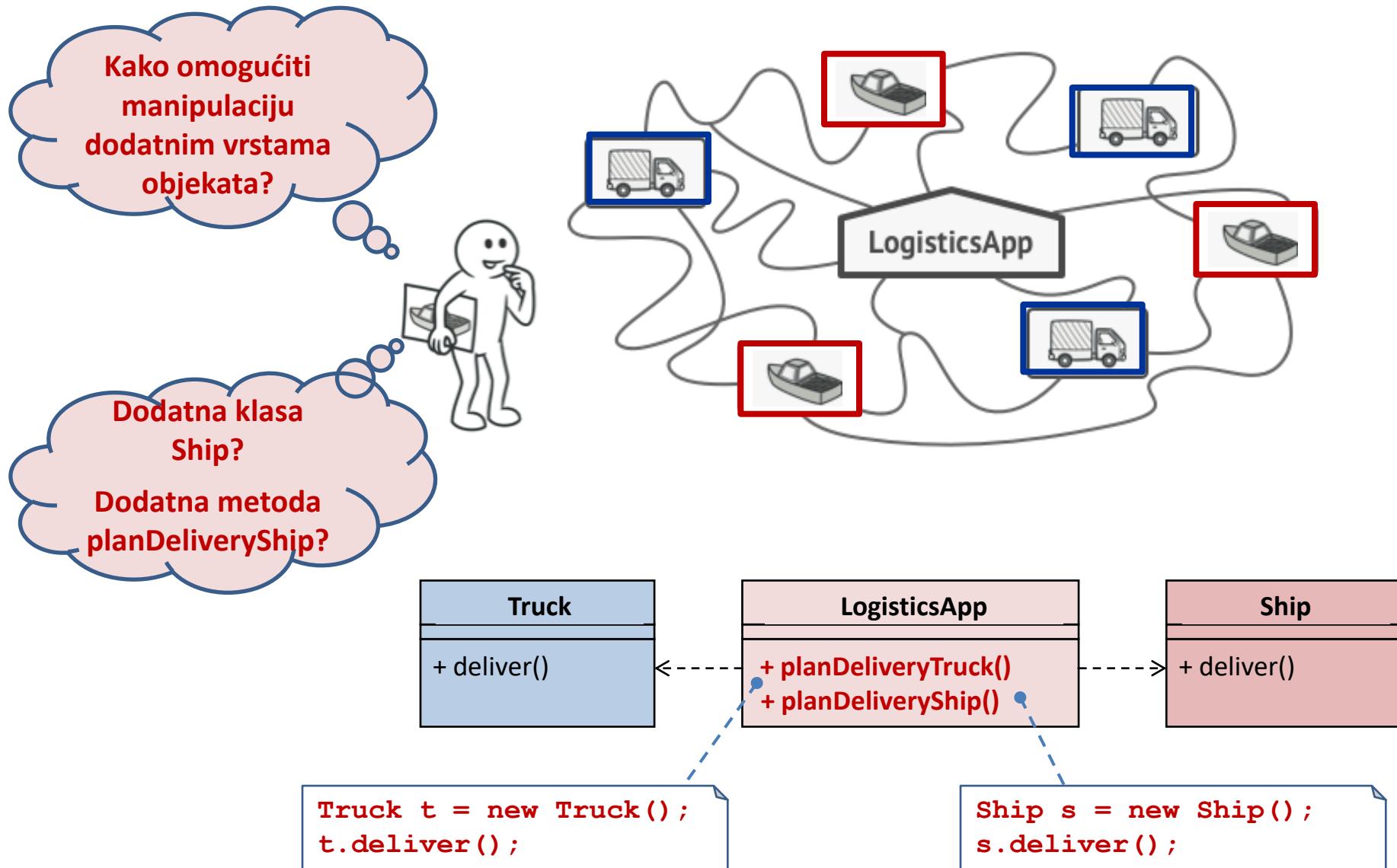
Prepostavimo da aplikacija ima mogućnost manipulacije jednom vrstom objekata (npr. aplikacija za organizaciju transporta, koja ima mogućnost organizacije prevoza robe kamionom)

Prepostavimo da treba omogućiti manipulaciju dodatnim vrstama objekata (npr. aplikacija treba da omogući i organizaciju prevoza robe brodom)



# Kreacioni obrasci – Fabrički metod

## Motivacija za uvođenje fabričkog metoda

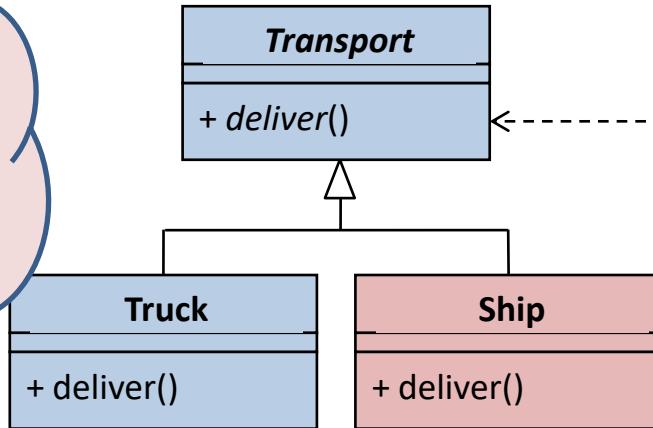


# Kreacioni obrasci – Fabrički metod

## Motivacija za uvođenje fabričkog metoda

LogisticsApp dosta zavisi od interfejsa Transport  
I dalje ima dosta manuelnog programiranja zbog dosta različitih new

...

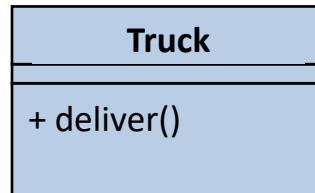


Transport tt = new Truck();  
tt.deliver();



Transport tt = new Ship();  
tt.deliver();

PROGRAMIRANJE PREMA INTERFEJSU

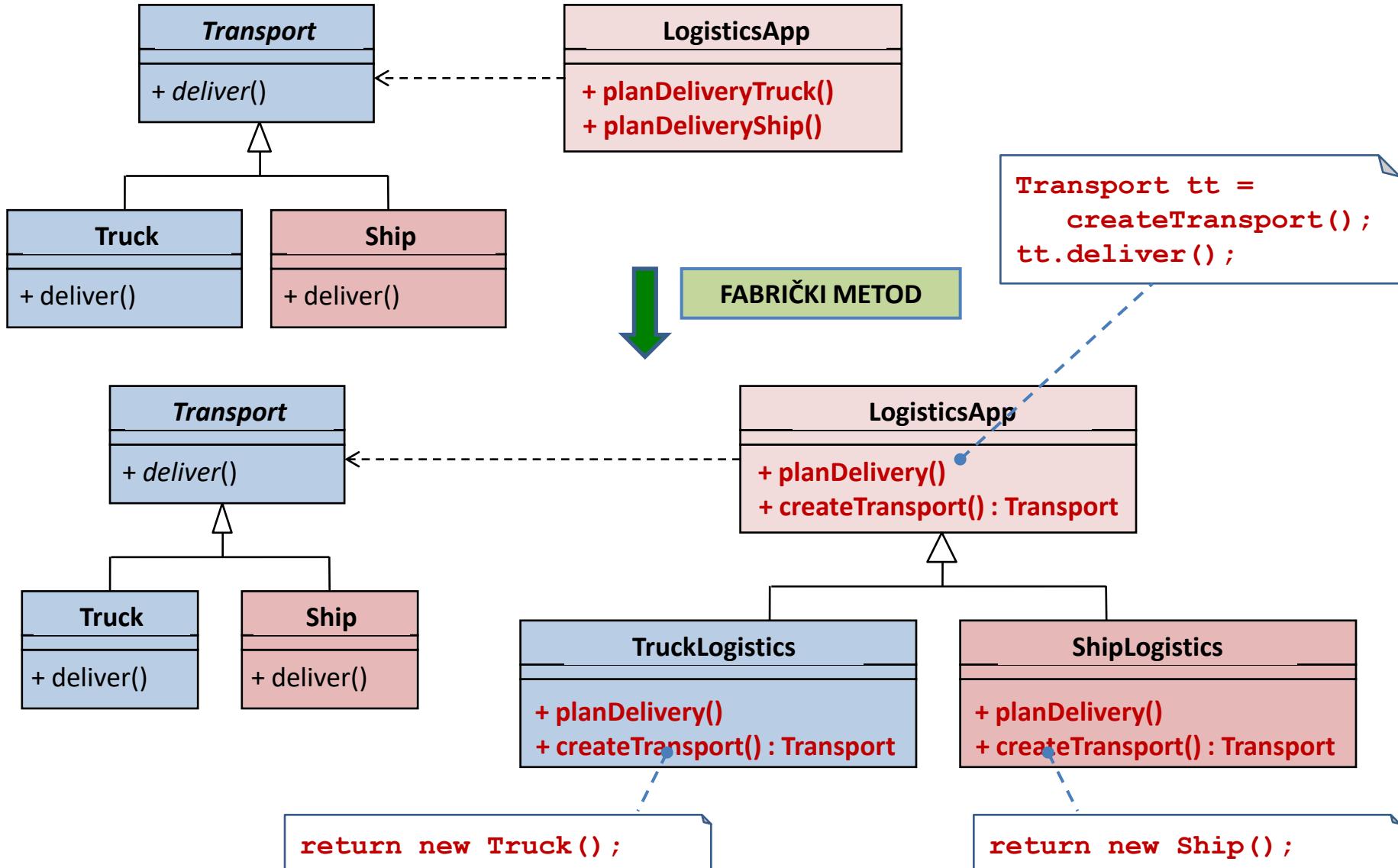


Truck t = new Truck();  
t.deliver();

Ship s = new Ship();  
s.deliver();

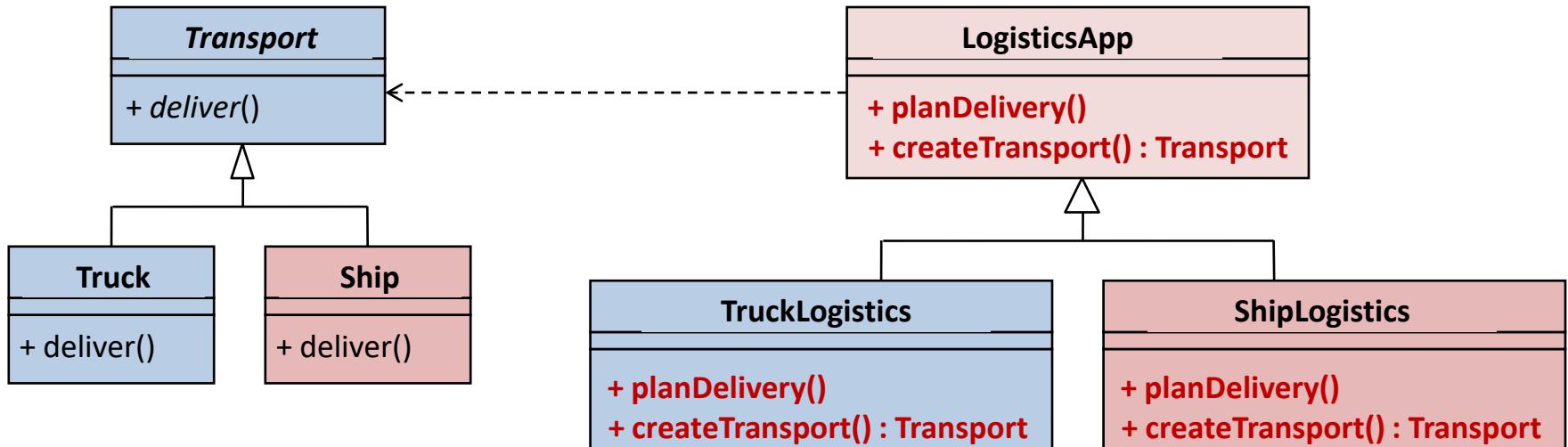
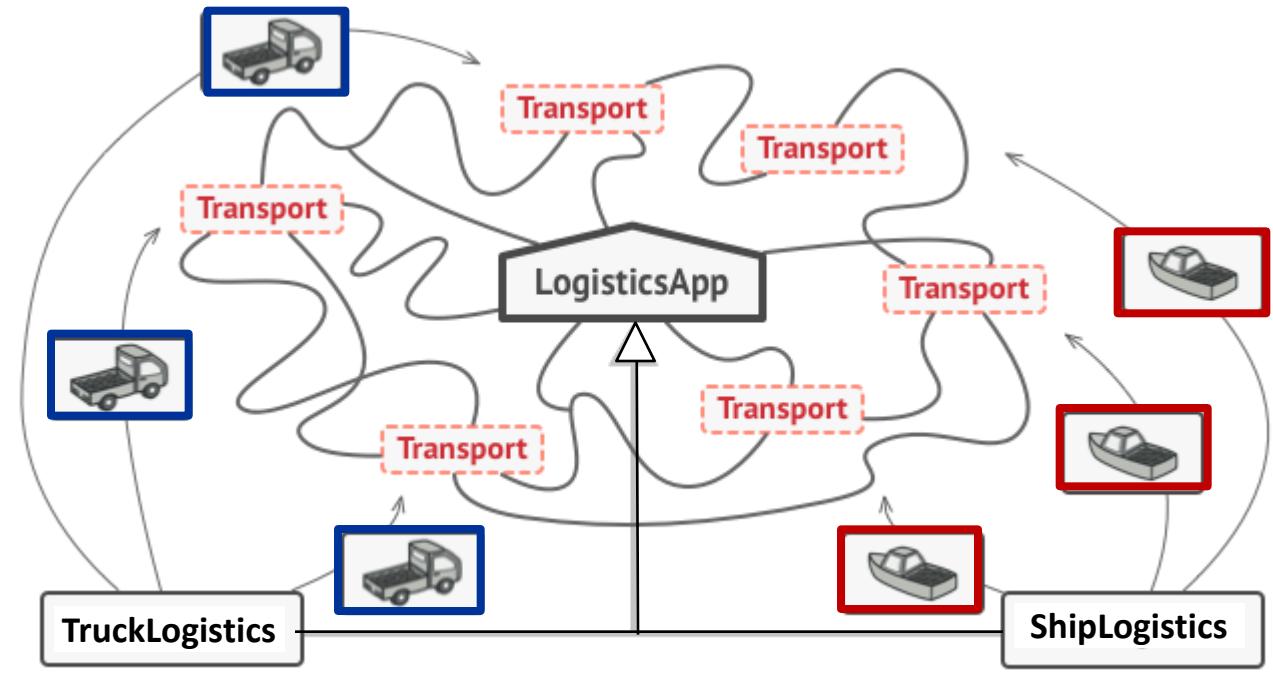
# Kreacioni obrasci – Fabrički metod

## Motivacija za uvođenje fabričkog metoda



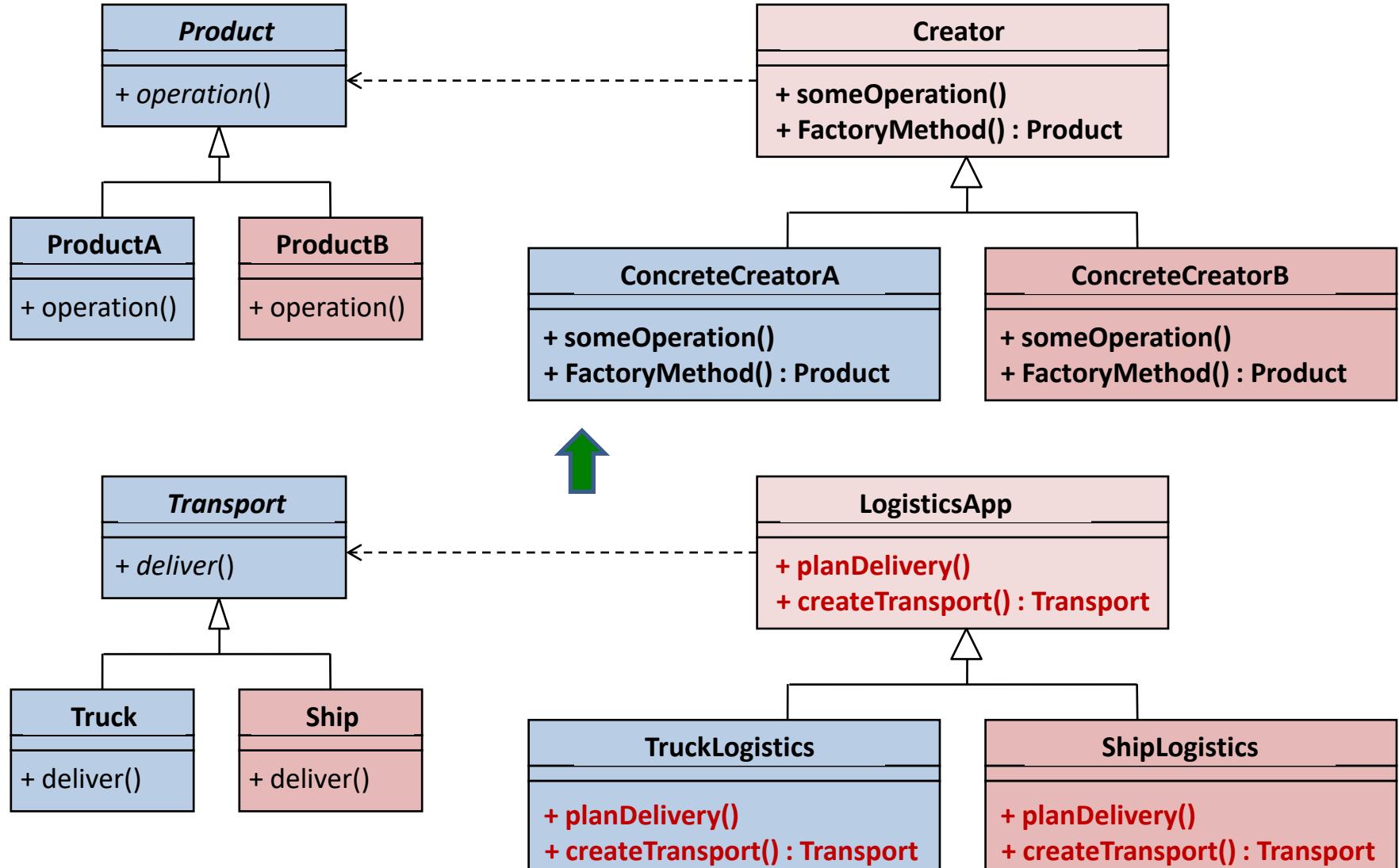
# Kreacioni obrasci – Fabrički metod

Motivacija za uvođenje  
fabričkog metoda



# Kreacioni obrasci – Fabrički metod

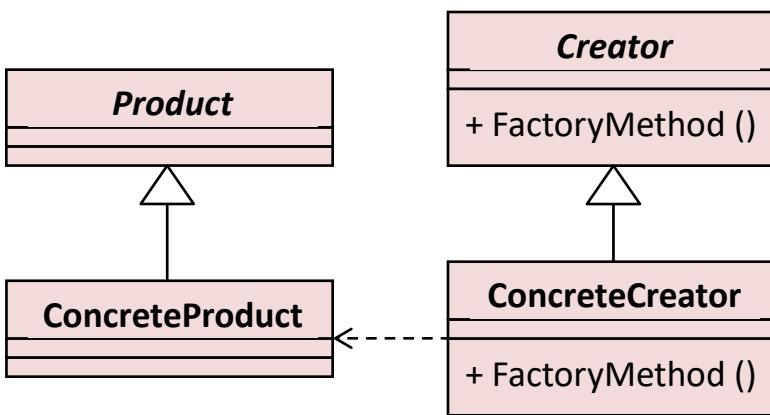
## Motivacija za uvođenje fabričkog metoda



# Kreacioni obrasci – Fabrički metod

## Factory Method (Fabrički metod)

- Kreira instance različitih izvedenih klasa.
- Definiše interfejs za kreiranje objekata, ali dozvoljava da izvedena klasa odluči koju će klasu da instancira.



Ovaj projektni obrazac omogućava da se definiše poseban metod zainstanciranje objekata, pri čemu potklase imaju mogućnost da ga redefinišu i specifikuju koji izvedeni proizvod će biti instanciran.

### Product

- deklariše interfejs za objekat koji će biti kreiran

### ConcreteProduct

- implementira Product interfejs i reprezentuje objekat koji će biti kreiran

### Creator

- deklariše **FactoryMethod** koji treba da vrati objekat tipa **Product**. Klasa **Creator** može i da definiše osnovnu implementaciju ovog metoda koji će vratiti najjednostavniji (onaj koji se kreira podrazumijevanim konstruktorom) objekat klase **ConcreteProduct**.

```
Product p = FactoryMethod();
```

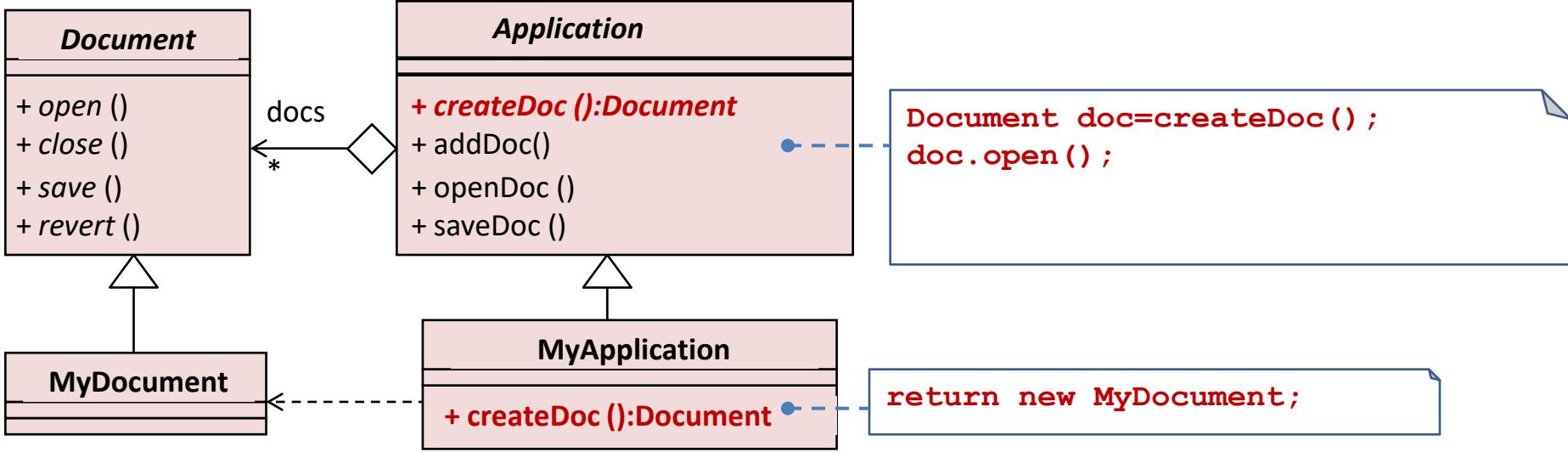
### ConcreteCreator

- redefiniše **FactoryMethod** kako bi vraćao instancu klase **ConcreteProduct**
- ```
return new ConcreteProduct();
```

# Kreacioni obrasci – Fabrički metod

Primjer:

Fabrički metod se veoma često koristi u AFs  
(Application Frameworks) za kreiranje objekata.



Prepostavimo da se AF koristi za aplikacije koje omogućavaju manipulaciju različitim dokumentima:

- klasa **Application**:
  - reprezentuje različite vrste aplikacija koje mogu da se implementiraju u AF,
  - omogućava manipulaciju različitim dokumentima (otvara, kreira novi, snima, ...)
- klasa **Document**:
  - reprezentuje različite vrste dokumenata koji mogu da se koriste u aplikacijama,
  - potklase reprezentuju konkretnе dokumente
- **Application** ne može da predvidi koji će se dokument instancirati (to zavisi od konkretnе aplikacije)
- AF mora da instancira klasu, ali zna samo apstraktnu klasu koju ne može da instancira

# Kreacioni obrasci – Fabrički metod

Uobičajeni alternativni naziv za FABRIČKI METOD: **VIRTUELNI KONSTRUKTOR**

## Tipične primjene FACTORY METHOD obrasca

- Klasa ne može da procijeni klasu čije objekte treba da instancira
- Klasa „želi“ da potklase specifikuju objekte koji seinstanciraju
- Klasa prepušta odgovornost zainstanciranje jednoj od pomoćnih potklasa

## Prednosti rješenja zasnovanog na FACTORY METHOD obrascu

- Funkcionalne metode u ApplicationClass raspregnute od različitih konkretnih implementacija Product objekata
- Kreiranje objekata u fabričkom metodu je fleksibilnije nego direktno u kodu, jer se potklasama omogućavaju različite mogućnosti instanciranja objekata
- U konkretnim kreatorima nema potrebe za ponavljanjem funkcionalnih metoda (one se nasljeđuju), samo se implementira koncretan fabrički metod

## Nedostaci rješenja zasnovanog na FACTORY METHOD obrascu

- definišu se potklase samo da bi se redefinisao fabrički metod

# Kreacioni obrasci – Fabrički metod

## Varijante fabričkog metoda

### Kreator je apstraktna klasa

- ne implementira fabrički metod
- potklase su neophodne

### Kreator je konkretna klasa

- obezbeđuje podrazumijevanu implementaciju fabričkog metoda
- fabrički metod koristi se radi fleksibilnosti
- kreiranje objekata u posebnoj metodi koja može da se redefiniše u potklasama

## Parametrizacija fabričkog metoda

- Varijacija obrasca koja omogućava da fabrički metod instancira različite konkretne proizvode
- Instanciranje konkretnog proizvoda specifikuje se parametrom

```
// PRIMJER PARAMETRIZACIJE

class Creator
{
    public Product create(int id)
    {
        if (id==1)
            return new ProductA();
        if (id==2)
            return new ProductB();
    }
}

class MyCreator extends Creator
{
    @Override
    public Product create(int id)
    {
        if (id==3)
            return new ProductC();
        return super.create(id);
    }
}
```

# Kreacioni obrasci – Apstraktna fabrika

## Motivacija za uvođenje apstraktne fabrike

Pretpostavimo da aplikacija treba da ima mogućnost manipulacije familijama različitih objekata

Npr. aplikacija za prodaju namještaja

- istoj familiji pripadaju: stolica, sofa, stolić
- može da postoji više verzija familije



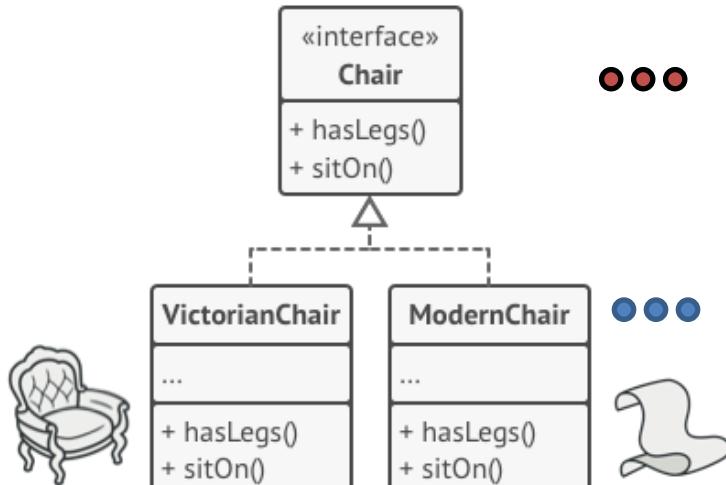
Aplikacija treba da omogući instanciranje pojedinačnih objekata koji pripadaju odgovarajućoj familiji!

Arhitektura treba da je stabilna i da se ne mijenja dodavanjem novog proizvoda ili nove familije!

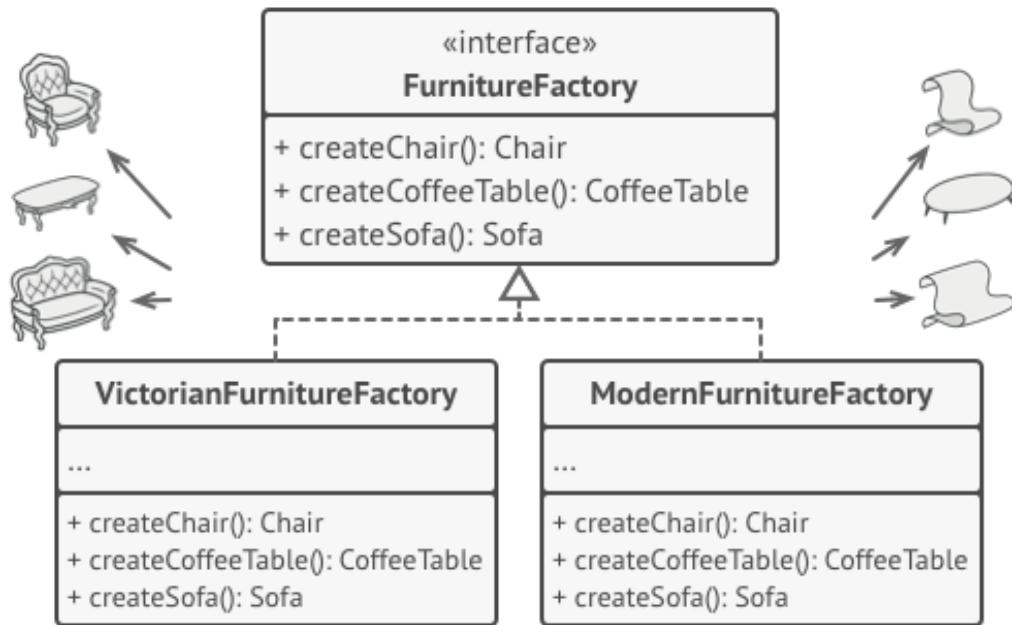
# Kreacioni obrasci – Apstraktna fabrika

## Motivacija za uvođenje apstraktne fabrike

Definisati zajednički interfejs za svaku vrstu proizvoda iz familije  
(Chair, Sofa, CoffeeTable, ...)



Definisati zajednički interfejs za fabrike familija  
(Apstraktna fabrika)

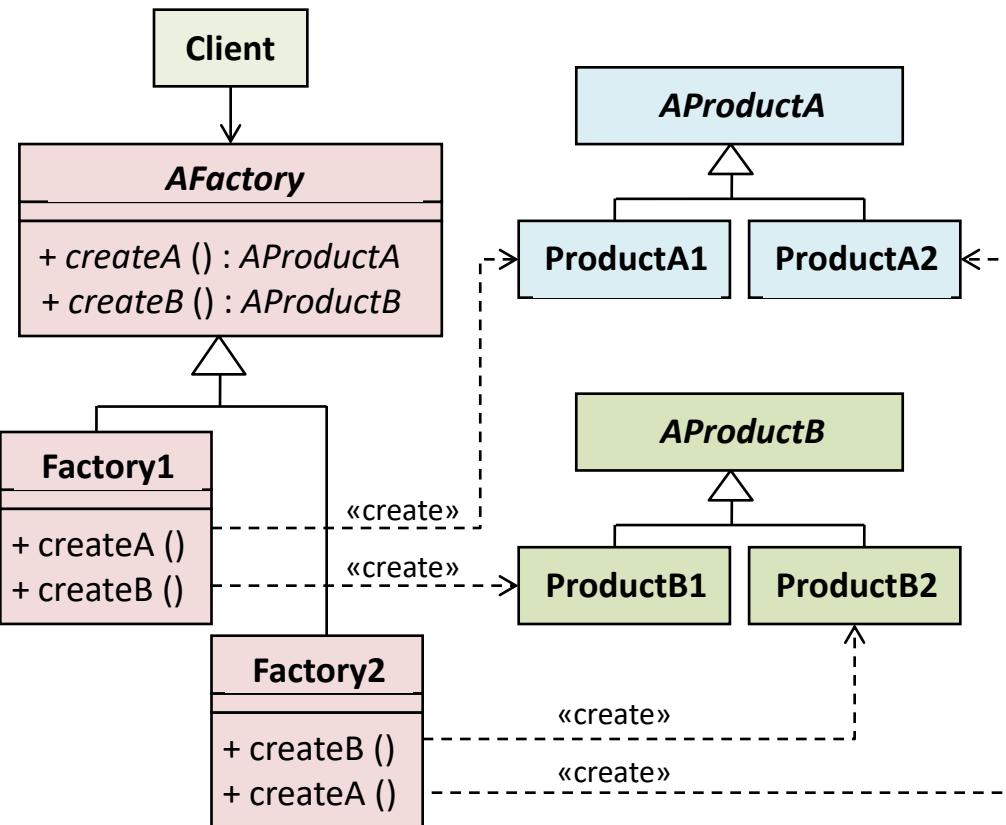


Definisati klase koje reprezentuju konkretnе proizvode za svaki tip proizvoda  
(npr. za Chair: VictorianChair, ModernChair, ...)  
(npr. za Sofa: VictorianSofa, ModernSofa, ...)

Definisati klase koje reprezentuju konkretnе fabrike za svaku familiju proizvoda  
(npr. VictorianFactory, ModernFactory, ... )

# Kreacioni obrasci – Apstraktna fabrika

## Abstract Factory (Apstraktna fabrika)



- Kreira instance nekoliko familija klasa.
- Obezbeđuje interfejs za kreiranje familije povezanih i međusobno zavisnih objekata bez specificiranja njihovih konkretnih klasa.

### AbstractFactory (AFactory)

- deklariše interfejs za konkretne fabrike (klase koje kreiraju objekte)

### ConcreteFactory (Factory1, Factory2)

- implementira operacije za kreiranje određene vrste konkretnih objekata

### AbstractProduct (AProductA, AProductB)

- deklariše interfejs za određenu vrstu objekata

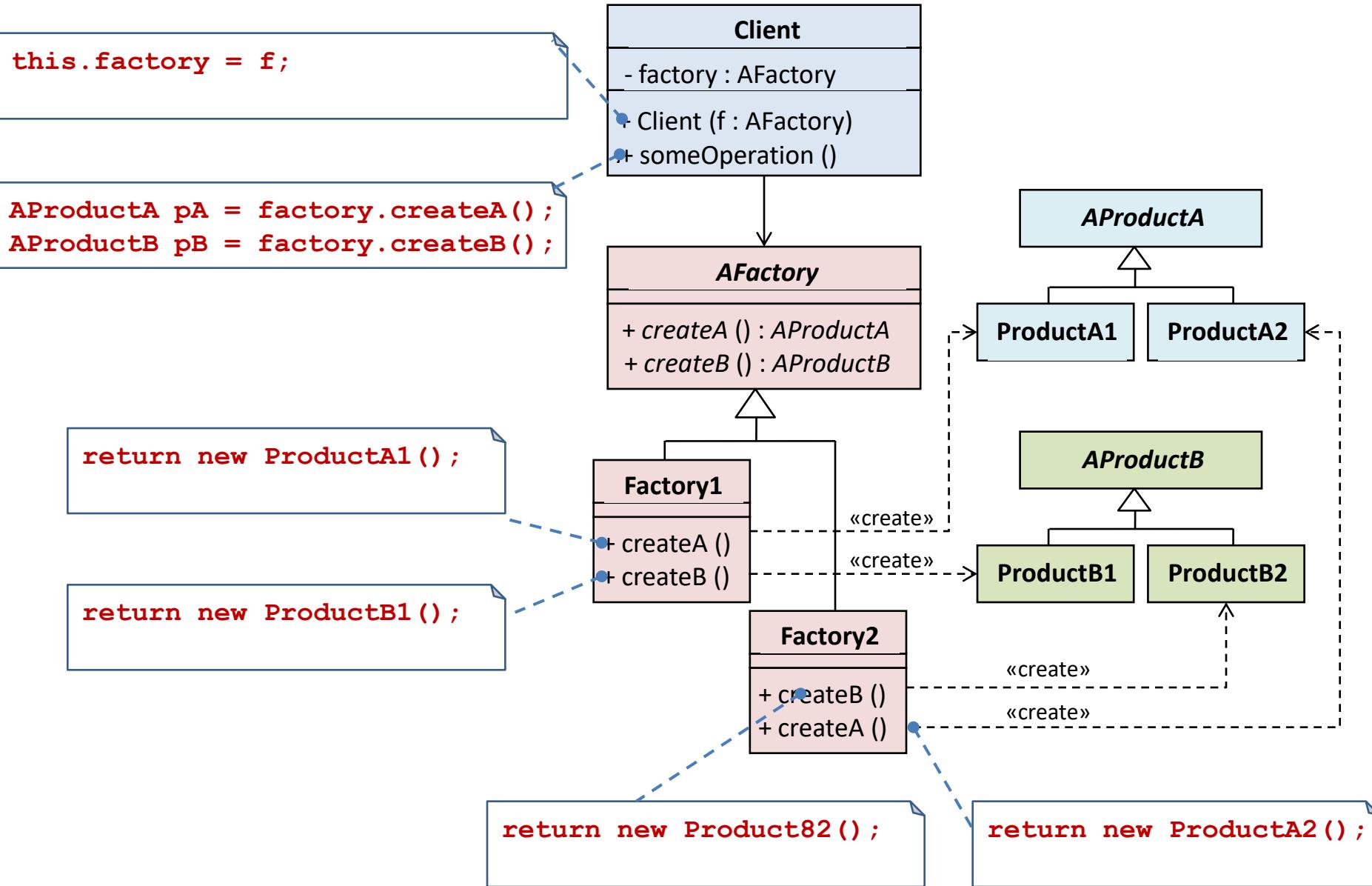
### ConcreteProduct (ProductA1, ProductA2)

- implementira interfejs koji definiše klasu **AProduct**
- definiše objekat koji će biti kreiran odgovarajućom **ConcreteFactory** klasom

#### Motivacija za apstraktnu fabriku:

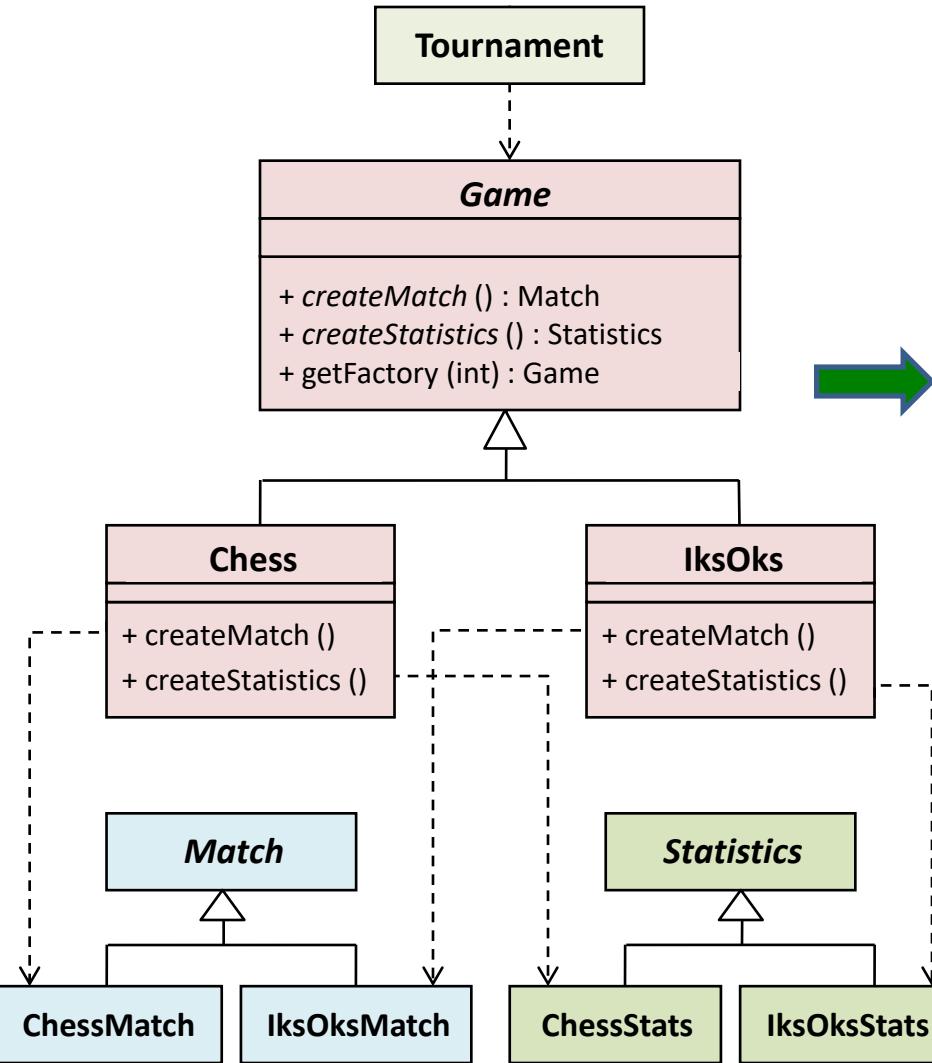
Npr. GUI koji treba da je portabilan na različite OS.  
Npr. upravljački sistem nezavisan od ugrađene opreme.

# Kreacioni obrasci – Apstraktna fabrika



# Kreacioni obrasci – Apstraktna fabrika

Primjer primjene ABSTRACT FACTORY obrasca:



```
abstract class Match { // ... }
class ChessMatch extends Match { // ... }
class IksOksMatch extends Match { // ... }

abstract class Game
{
    static final Game getFactory(int g)
    {
        if (g==1) return new Chess();
        else return new IksOks();
    }
    public abstract Match createMatch();
    // ...
}

class Chess extends Game
{
    public ChessMatch createMatch()
    { return new ChessMatch(); }
    // ...
}

public class Tournament
{
    public void gaming()
    {
        Game game = Game.getFactory(1);
        Match match = game.createMatch();
    }
    // ...
}
```

# Kreacioni obrasci – Apstraktna fabrika

## Tipične primjene ABSTRACT FACTORY obrasca

### – nezavisnost od inicijalizacije ili reprezentacije

- sistem treba da bude nezavisan od načina na koji se proizvodi kreiraju, komponuju ili reprezentuju

### – nezavisnost od proizvođača

- treba da postoji mogućnost konfigurisanja sistema sa jednom familijom proizvoda, pri čemu postoji mogućnost izbora više različitih familija
  - (npr. projektovanje elektronskog sklopa sa nekoliko potencijalnih familija čipova)

### – ograničenja uslovljena vezanim proizvodima

- familija povezanih proizvoda je projektovana tako da moraju zajedno da se koriste

### – kompatibilnost sa budućim (prethodnim) familijama

- projektovanje sa trenutnom familijom proizvoda, ali se ostavlja mogućnost zamjene drugom (budućom ili nekom prošlom) familijom

**Tipični nefunkcionalni sistemske zahtjevi koji su osnov za ABSTRACT FACTORY:**

“nezavisnost od proizvođača”

“tehnološka nezavisnost”

“mora da podrži manipulaciju familijom objekata”

...

# Kreacioni obrasci – Apstraktna fabrika

## Implementacioni detalji

### – konkretna fabrika kao singleton

- Aplikacija tipično treba jednu instancu neke konkretne fabrike za jednu familiju proizvoda, pa se **uobičajeno konkretna fabrika implementira kao singleton.**

### – kreiranje proizvoda

- Apstraktna fabrika deklariše samo interfejs za kreiranje proizvoda, a instanciranje konkretnih proizvoda je najčešće odgovornost fabrika – **uobičajeno se za instanciranje proizvoda koriste fabričke metode u hijerarhiji klasa koje reprezentuju fabrike.**

- **U slučaju većeg broja familija proizvoda, konkretna fabrika može da se implementira primjenom prototip obrasca** – konkretna fabrika se inicijalizuje prototipskom instancom za svaki proizvod u familiji, a nakon toga se svaki proizvod dalje kreira kloniranjem prototipa. Prototipski pristup eliminiše potrebu za novom konkretnom fabrikom u slučaju nove familije.

### – „proširljive“ fabrike

- Apstraktna fabrika obično definiše različite operacije za svaku vrstu mogućih proizvoda – vrsta proizvoda se specifikuje u deklaraciji metoda. **Dodavanje nove vrste proizvoda zahtijeva izmjenu apstraktne fabrike i njenih potklasa.** Fleksibilniji način jeste **da se u apstraktnoj fabrici uvede samo jedna Make/Create operacija sa odgovarajućim parametrom** koji ukazuje na vrstu proizvoda koji treba da se instancira.

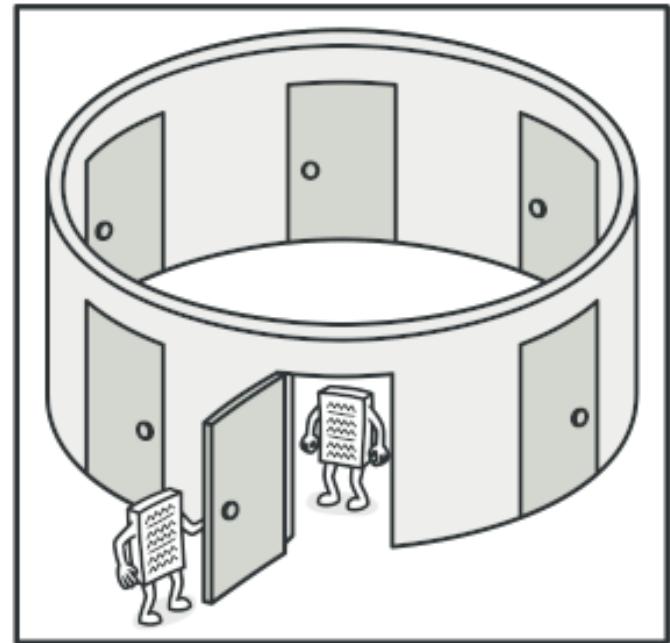
# Kreacioni obrasci – Singleton

## Motivacija za uvođenje singltona

Pretpostavimo da aplikacija treba da obezbijedi da u isto vrijeme može da postoji samo jedan jedini objekat neke klase

Npr. kontroler pristupa domenskom objektu ili kontroler pristupa štampaču

Takvo ponašanje ne može biti omogućeno uobičajenom implementacijom klasa, jer se (podrazumijevano) svakim izvršavanjem konstruktora instancira po jedan objekat



## Moguće rješenje problema

Konstruktor nije public, nego private (ili protected), tako da izraz new Singleton() predstavlja *compile-time* grešku

Dobijanje jedinstvene instance može se realizovati pozivom statičke metode koja vraća jedinstvenu instancu

# Kreacioni obrasci – Singleton

## Singleton

(Unikat, Usamljenik)

- Klasa koja može da ima samo jedan živi objekat (ili kontrolisan broj živih objekata)
- Obezbeđuje odgovarajući pristup jedinstvenoj instanci klase.

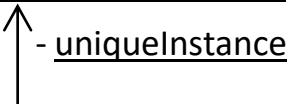
### Singleton

```
- uniqueInstance : Singleton  
- Singleton ()  
+ getInstance() : Singleton
```



### Singleton

```
- Singleton ()  
+ getInstance() : Singleton
```



### Singleton

- definiše statičku operaciju getInstance() koja obezbeđuje korisniku pristup jedinom kreiranom objektu
- definiše privatni (ili zaštićeni) konstruktor koji je odgovoran za kreiranje i održavanje svog jedinog objekta
- jedinstvena instanca mora biti statički atribut

```
static public Singleton getInstance()  
{  
    if(uniqueInstance == null)  
        uniqueInstance = new Singleton();  
    return uniqueInstance;  
}
```

```
// pristup singltonu iz klijentskog koda  
Singleton s = Singleton.getInstance();  
  
// s.doSomething();
```

# Kreacioni obrasci – Singleton

## Implementacioni detalji

### – obezbjeđivanje jedinstvene instance

- Singleton klasa treba da se implementira tako da je omogućeno instanciranje samo jednog objekta.
- Tipično se operacija za kreiranje objekta „skriva“ (uobičajeno **protected konstruktor**) iza operacije koja omogućava pristup tom objektu (uobičajeno **zajednička/static metoda**).
- **Zaštićeni konstruktor** onemogućava instanciranje objekata (*compile-time error*), a omogućava specijalizaciju singletona.

```
public class Singleton
{
    protected Singleton()
    {
        // ...
    }

    static private Singleton uniqueInstance = null;

    static public Singleton getInstance()
    {
        if(uniqueInstance == null)
            uniqueInstance = new Singleton();
        return uniqueInstance;
    }
}
```

```
// pristup singletonu iz klijentskog koda
Singleton s = Singleton.getInstance();

// s.doSomething();
```

# Kreacioni obrasci – Singleton

## Specijalizacija singletona

- Klijent treba da ima pristup jedinstvenoj instanci potklase singletona.
- Tehnike za specijalizaciju:  
**parametrizacija / registar singltona**

### parametrizacija

- implementacija u odgovornosti singleton klase
- getInstance() instancira i omogućava pristup željenom specijalizovanom singletonu na osnovu parametra

### singleton registar

- implementacija u odgovornosti potklasa
- potklase registruju svoje instance (razlikuju se po imenu) u registru
- getInstance() konsultuje registar (na osnovu imena) i pronalazi instancu ili instancira potreban singleton

```
class Singleton
{
    static private HashMap<String,Singleton>
        rs = new HashMap<String,Singleton>();
    static protected void reg(String i, Singleton s)
    { rs.put(i,s); }
    static public Singleton getInstance(String s)
    {
        if (rs.get(s)==null)
        {
            Class as = Class.forName(s);
            Class[] args = new Class[0];
            Method mr = as.getMethod("register",args);
            mr.invoke(null,args);
        }
        return rs.get(s);
    }
    static protected Singleton() { // ... }
    static protected void register()
    { reg("Singleton", new Singleton()); }
}

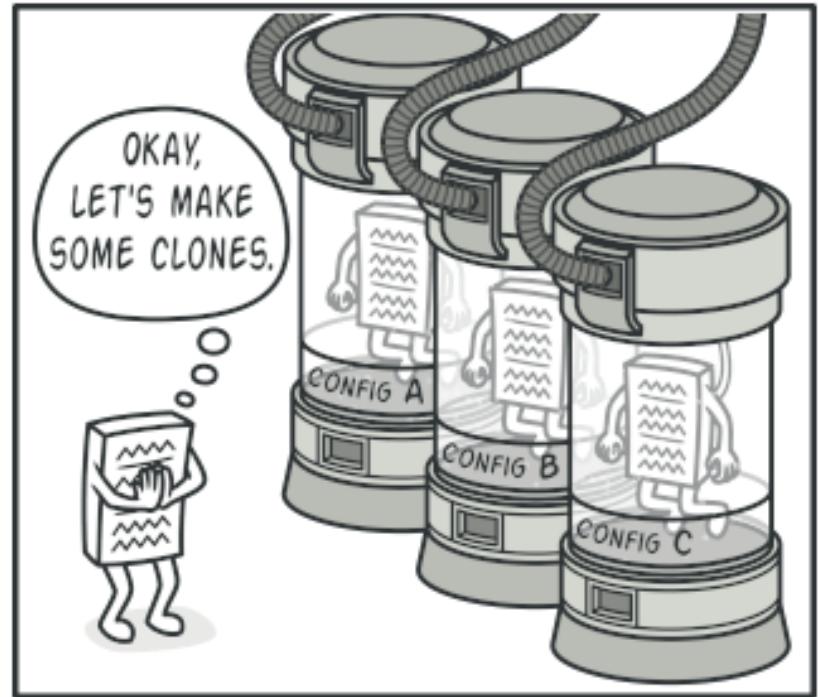
class MySingl extends Singleton
{
    protected MySingl() { // ... }
    static protected void register()
    { Singleton.reg("MySingl", new MySingl()); }
}
```

# Kreacioni obrasci – Prototip

## Motivacija za uvođenje prototipa

Ponekad je proces instanciranja objekata i njihove potpune inicijalizacije veoma spor i kompleksan pa je zgodnije imati jedan ili više prototipa koje možemo klonirati (kopirati) i po potrebi prilagođavati

Npr. treba nam kopija nekog složenog (Composite obrazac) objekta, npr. neka teritorijalna jedinica sa svim komponentama – mnogo jednostavnije je kopirati takav objekat, nego ga instancirati izvršavanjem konstruktora odgovarajuće hijerarhije klasa



## Moguće rješenje problema

Klase čije objekte želimo da kloniramo, treba da ima odgovarajuću **metodu za kloniranje**, koja vraća kopiju datog objekta

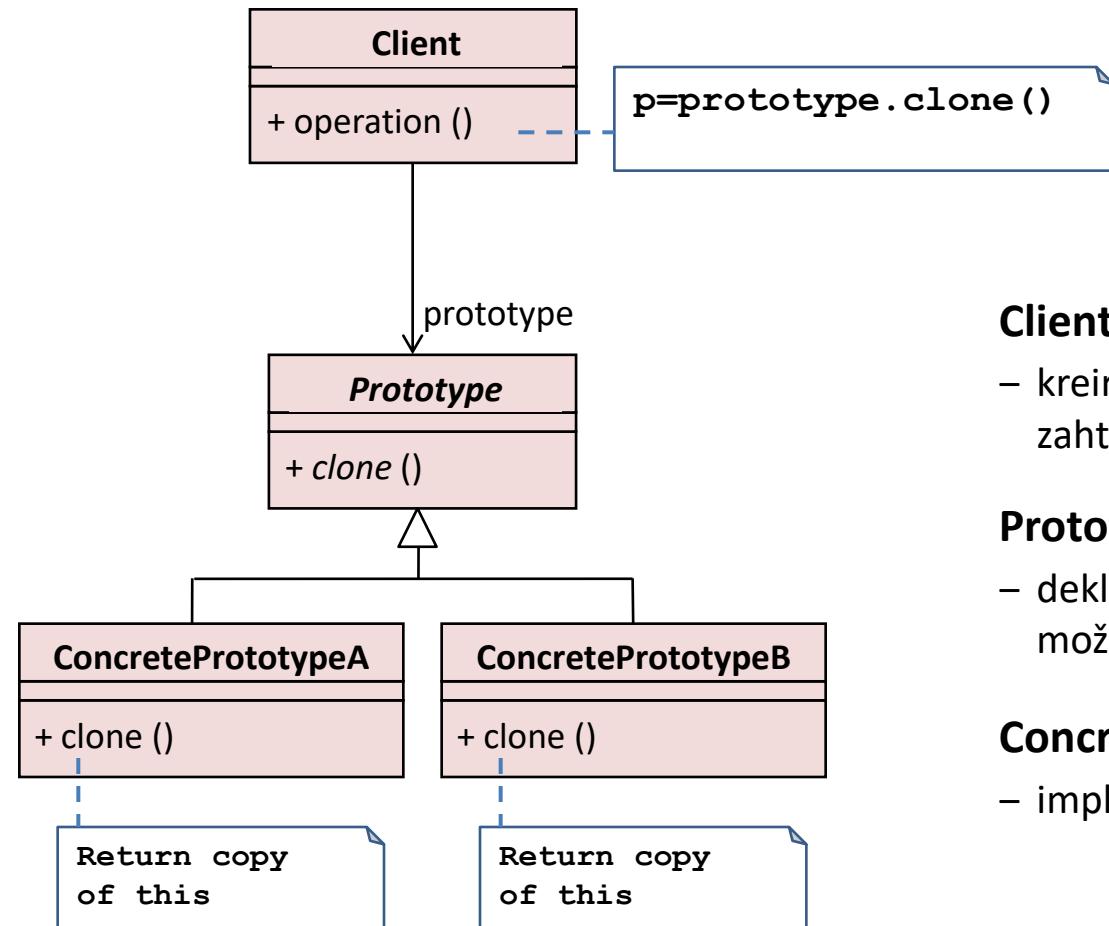
Objekat koji može da se klonira, naziva se **prototip**

Kad nam je neophodan novi objekat, samo kloniramo neki prototip (bez potrebe da konstruišemo novi objekat)

# Kreacioni obrasci – Prototip

## Prototype (Prototip)

- Potpuno inicijalizovana instanca koja može biti kopirana ili klonirana.
- Služi za specificiranje vrste objekata koji će biti kreirani pomoću tzv. **prototipske instance**, kao i za kreiranje novih objekata kopiranjem prototipa.



### Client

- kreira novi objekat tako što od prototipa zahtijeva da klonira samog sebe

### Prototype

- deklariše interfejs pomoću kojeg objekat može da klonira samog sebe

### ConcretePrototype

- implementira operaciju kloniranja samog sebe

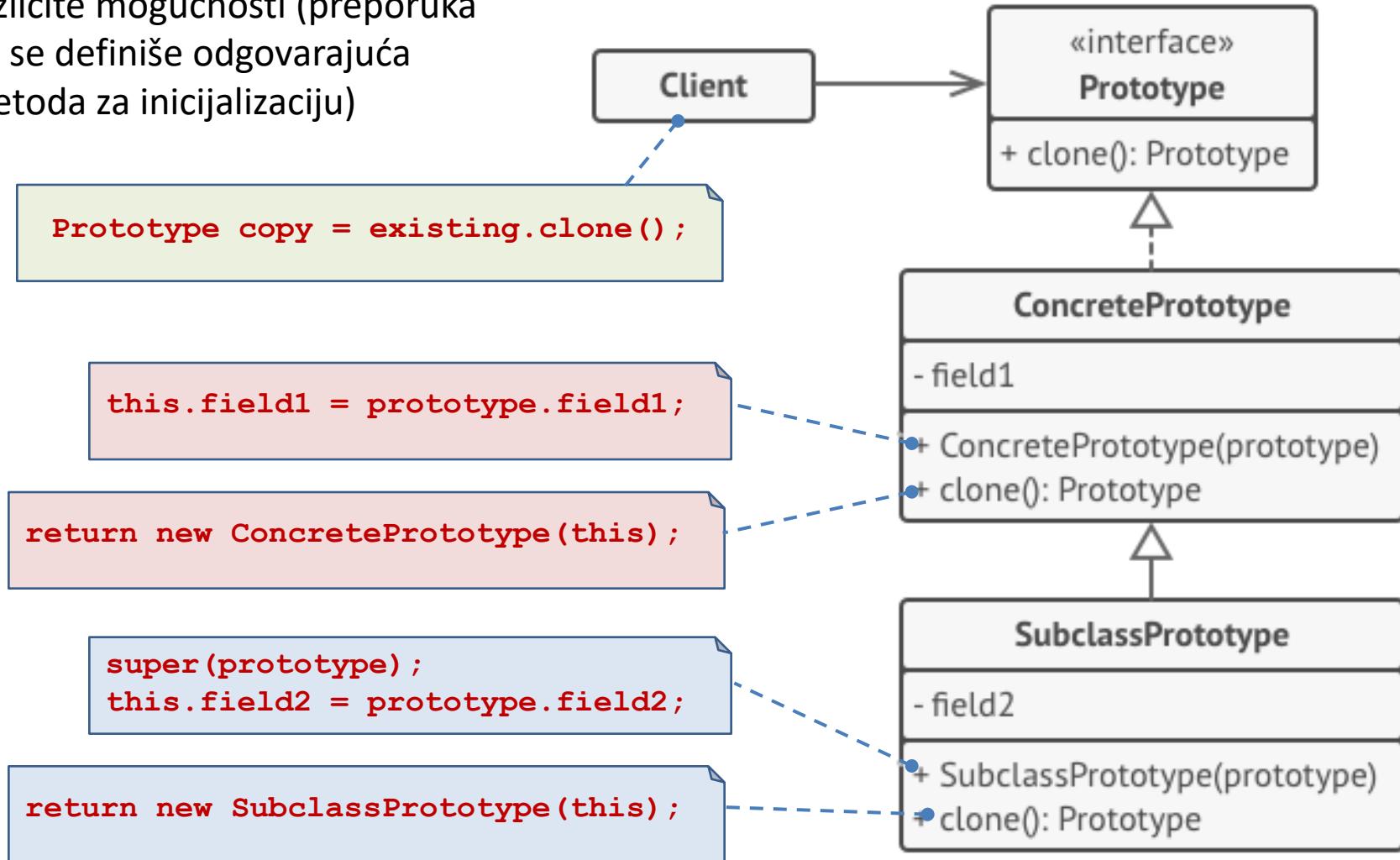
# Kreacioni obrasci – Prototip

## Implementacioni detalji

### – Inicijalizacija klonova

- različite mogućnosti (preporuka da se definiše odgovarajuća metoda za inicijalizaciju)

U najjednostavnijem slučaju, kloniranje može da se realizuje pomoću konstruktora kopije

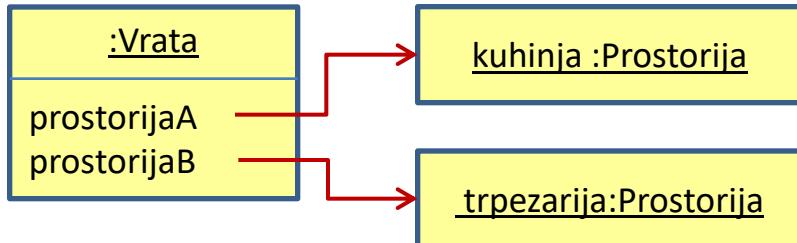
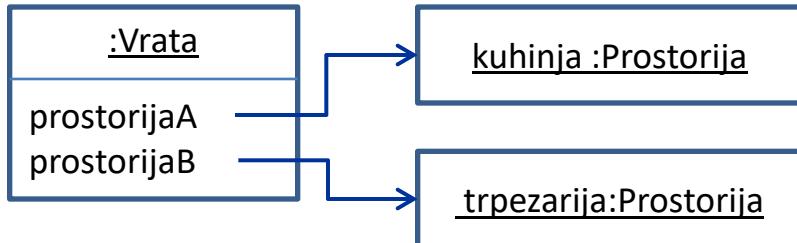


# Kreacioni obrasci – Prototip

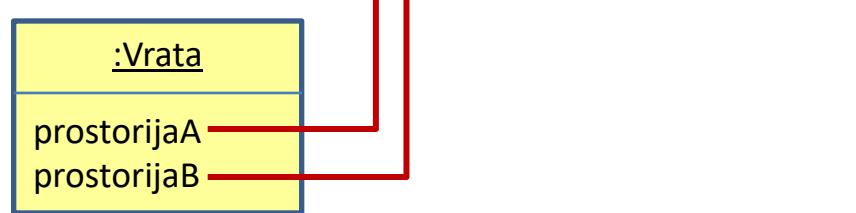
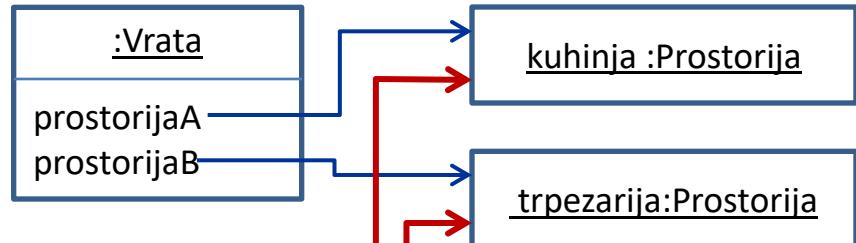
## Implementacioni detalji

- Implementacija metode `clone()`:

***deep copy***  
(duboka kopija)



***shallow copy***  
(plitka kopija)



Ugrađeni mehanizmi za kloniranje tipično prave plitke kopije (*memberwise copy*)!

# Kreacioni obrasci – Prototip

## Kloniranje u JAVI – **Object clone()**

```
public Object clone() throws CloneNotSupportedException
```

kreira klon objekta:

- alocira novu instancu, i
- smješta bitski klon tekućeg objekta u novi objekat

```
class UserObject implements Cloneable
{
    // ...

    public Object clone() throws CloneNotSupportedException
    {
        return (UserObject) super.clone();
    }
}
```

Implementacija interfejsa **Cloneable**

Redefinicija metode **clone()**

# Kreacioni obrasci – Prototip

## Kloniranje u JAVI – **Object clone()**

**public Object clone() throws CloneNotSupportedException**

```
public class Department implements Cloneable
{
    private String dname;
    public Department(String dname) { this.dname = dname; }
    public String getName() { return dname; }
    @Override
    public Object clone() throws CloneNotSupportedException
        { return (Department) super.clone(); }

    public static void main(String[] args)
    {
        Department obj1 = new Department("Biblioteka");
        try
        {
            Department obj2 = (Department) obj1.clone();
            System.out.println(obj2.getName());
        }
        catch (CloneNotSupportedException e)
            { e.printStackTrace(); }
    }
}
```

Biblioteka

# Kreacioni obrasci – Prototip

```
public class Department
{
    private String naziv;
    public Department(String naziv) { this.naziv = naziv; }
    public String getNaziv() { return naziv; }
    public void setNaziv(String naziv) { this.naziv = naziv; }
}

public class Radnik implements Cloneable
{
    private String ime;
    private Department dep;
    public Radnik(String ime, Department dep) { this.ime = ime; this.dep = dep; }
    public Department getDepartment() { return dep; }

    @Override
    public Object clone() throws CloneNotSupportedException
    { return (Radnik) super.clone(); }
}

public class Test
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Department racun = new Department("Racunovodstvo");
        Radnik original = new Radnik("Marko", racun);
        Radnik klon = original.clone();
        klon.getDepartment().setNaziv("Biblioteka");
        System.out.println(klon.getDepartment().getNaziv());
        System.out.println(original.getDepartment().getNaziv());
    }
}
```

Primjer (*Shallow clone*):

Biblioteka  
Biblioteka

```
public Department implements Cloneable
{
    // ...
    @Override
    public Object clone() throws CloneNotSupportedException
    { return (Department) super.clone(); }
}

public class Radnik implements Cloneable
{
    // ...
    @Override
    public Object clone() throws CloneNotSupportedException
    {
        Radnik klon = (Radnik) super.clone();
        klon.setDepartment((Department) klon.getDepartment().clone());
        return klon;
    }
}

public class Test
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Department racun = new Department("Racunovodstvo");
        Radnik original = new Radnik("Marko", racun);
        Radnik klon = (Radnik) original.clone();
        klon.getDepartment().setNaziv("Biblioteka");
        System.out.println(klon.getDepartment().getNaziv());
        System.out.println(original.getDepartment().getNaziv());
    }
}
```

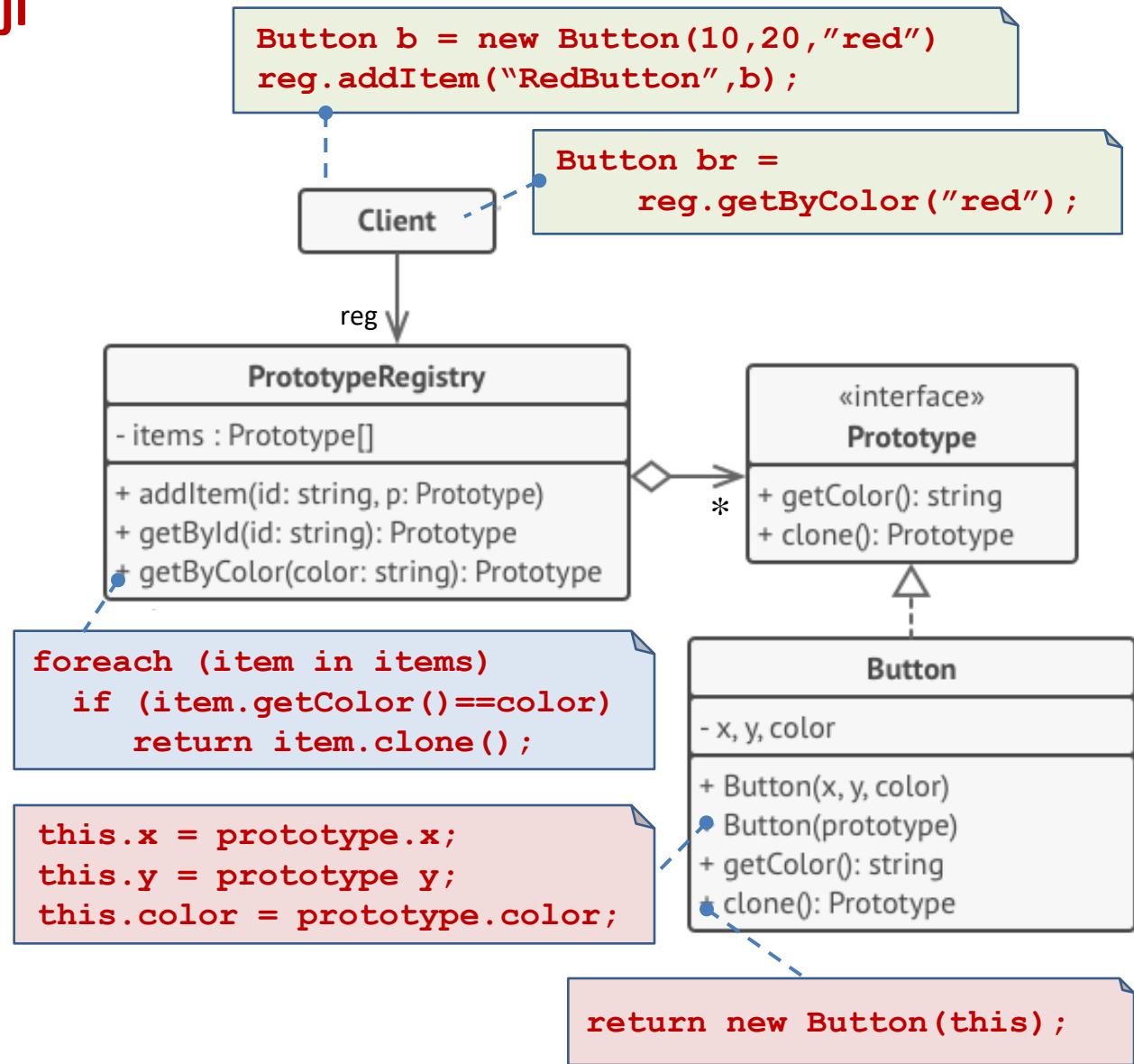
Primjer (*Deep clone*):

Biblioteka  
Racunovodstvo

# Kreacioni obrasci – Prototip

## Implementacioni detalji

- Preporuka da se koristi **prototype manager** (registrovati prototipove):
  - registrovati prototipova  
sadrži sve raspoložive  
prototipove
  - **asocijativna memorija**  
(ključ↔prototip) koja  
vraća odgovarajući  
prototip za dati ključ
  - **klijent ima mogućnost**  
registrovanja novog /  
izbacivanja postojećeg  
prototipa



# Kreacioni obrasci – Prototip

## Dobre strane korišćenja prototipa

- Omogućeno run-time dodavanje/uklanjanje objekata
  - klijent može dinamički da dodaje (registruje) nove prototipove
- Kloniranje je nekad mnogo efikasnije nego kreiranje objekata
  - npr. brzina izvršavanja
- Specifikacija novih (vrsta) objekata variranjem vrijednosti klonova
  - na osnovu iste klase može da se kreira više različitih prototipova sa različitim vrijednostima pojedinih atributa, čime mogu da se reprezentuju različite vrste objekata (nije nužno da postoji potklasa da bi se reprezentovala posebna vrsta objekata)
- Specifikacija novih (vrsta) objekata variranjem strukture klonova
  - na osnovu iste klase, koja reprezentuje objekte sa složenom strukturuom, može da se instancira više različitih prototipa sa različitim strukturama (manje ili više složene)
- Redukcija hijerarhije klasa
  - prototip omogućava implementaciju sistema bez hijerarhije kreatora – ne poziva se fabrički metod da bi se instancirao konkretni proizvod

## Nedostaci

- Svaka potklasa prototipa mora da implementira operaciju kloniranja, što nije jednostavno:
  - u slučaju postojećih klasa, ili ako
  - interni objekti ne omogućavaju kloniranje ili sadrže cirkularne reference

# Kreacioni obrasci – Builder

## Motivacija za uvođenje buildera

Prepostavimo da treba kreirati složene objekte, što zahtijeva napornu, korak po korak, inicijalizaciju mnogih atributa i podobjekata.

Takva inicijalizacija obično se izvodi unutar ogromnog konstruktora ili je "razbacana" u klijentskom kodu.

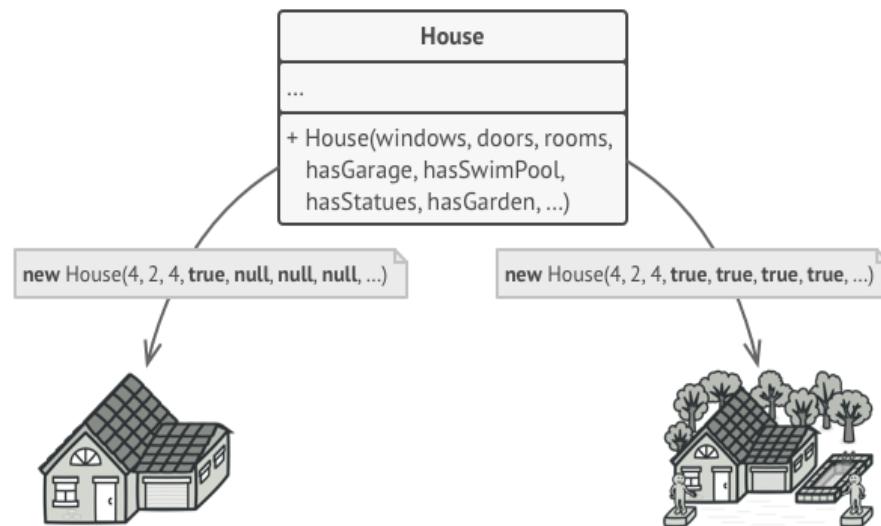
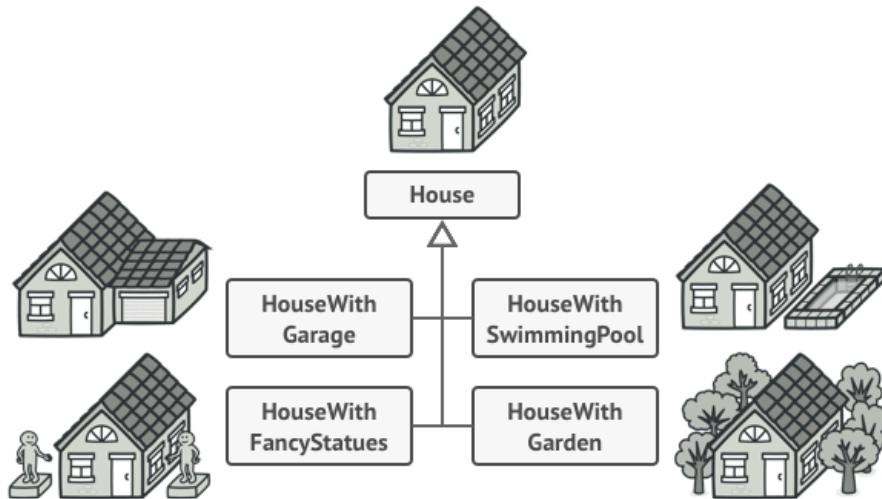
Npr. formiranje različitih konfiguracija proizvoda (računar, automobil, ...)

Prepostavimo objekat House (osnovni model + garaža + bazen + ventilacija + grijanje + ...)

Hijerarhija klasa za sve moguće konfiguracije?

Nova potklasa za novu konfiguraciju?

Jedna osnovna klasa sa огромним konstruktorom i brojnim argumentima?



# Kreacioni obrasci – Builder

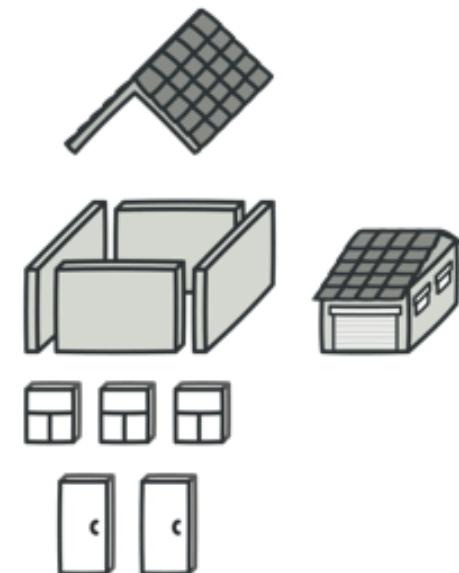
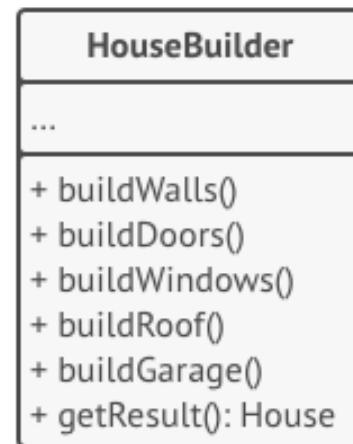
## Motivacija za uvođenje buildera

Dobro projektno rješenje za izgradnju složenih objekata jeste **izdvajanje aplikativne logike za proces izgradnje** objekata iz klase koja reprezentuje objekt **u posebnu klasu** koja se zove **builder**

Izgradnja složenog objekta izvodi se korak po korak – dio po dio, svaki dio odgovarajućom metodom

Za kreiranje odgovarajuće konfiguracije, pozivamo samo neophodne metode

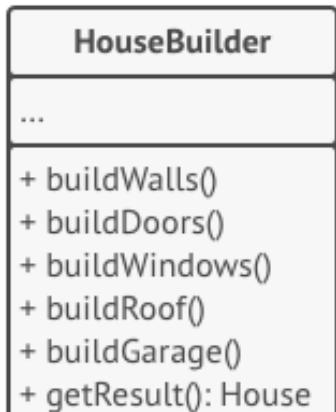
Builder je kreacioni obrazac koji omogućava formiranje složenih objekata korak po korak.



# Kreacioni obrasci – Builder

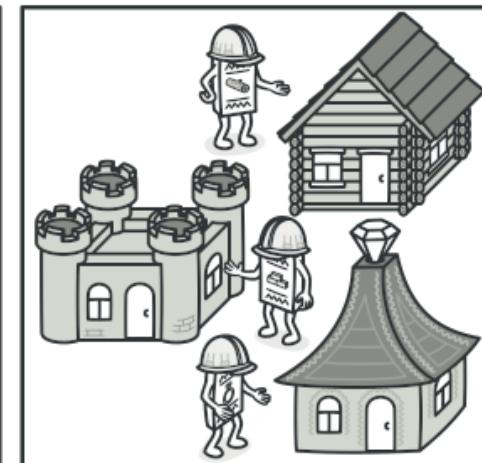
## Motivacija za uvođenje buildera

Specijalizacijom buildera možemo postići različite načine izgradnje složenog objekta



Builder je kreacioni obrazac koji omogućava formiranje složenih objekata korak po korak.

Builder omogućava proizvodnju različitih tipova objekata na različite načine.



# Kreacioni obrasci – Builder

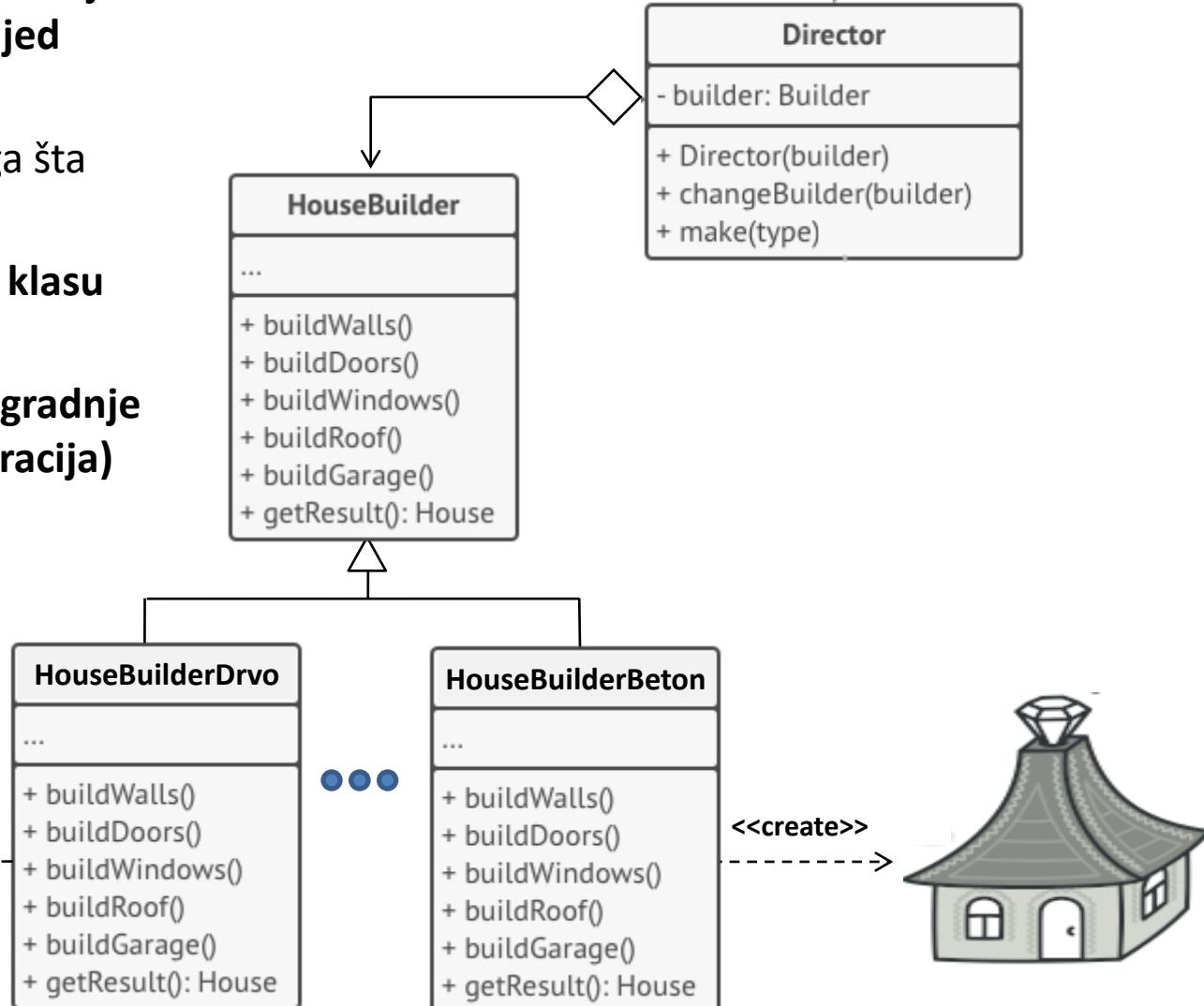
## Motivacija za uvođenje buildera

Još je bitno riješiti proces izgradnje složenog objekta (redoslijed operacija)

Proces izgradnje zavisi od toga šta želimo da izgradimo

Proces izgradnje izdvaja se u klasu koja se naziva **direktor**

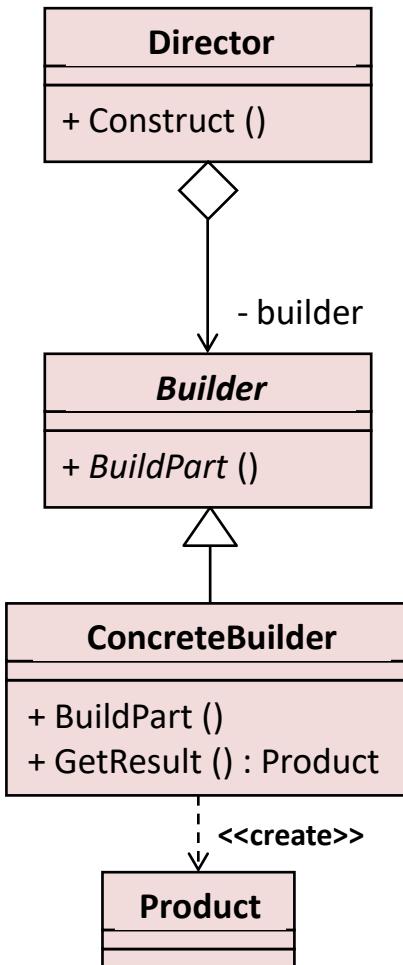
Direktor organizuje proces izgradnje (određuje redoslijed operacija)



# Kreacioni obrasci – Builder

**Builder**  
(Graditelj)

- Odvaja kreiranje objekta od njegove reprezentacije.
- Razdvaja kreiranje objekata od njihove reprezentacije tako da se za isti proces konstruisanja mogu kreirati različite reprezentacije.



## Builder

- Specifikuje interfejs za kreiranje objekata klase Product, dio po dio
- Builder je apstraktna klasa ili interfejs, kako bi se klijentu obezbijedio odgovarajući interfejs za različite konkretne buildere.

## ConcreteBuilder

- Vodi računa o objektima koje kreira
- Kreira i sklapa dijelove složenog objekta
- Obezbeđuje interfejs za pribavljanje objekata klase Product

## Director

- Konstruiše objekte na osnovu interfejsa koji obezbeđuje klasa Builder
- Konstruktor klase Director prima od klijenta kao argument objekat tipa Builder i odgovoran je za pozivanje odgovarajućih metoda u klasi Builder

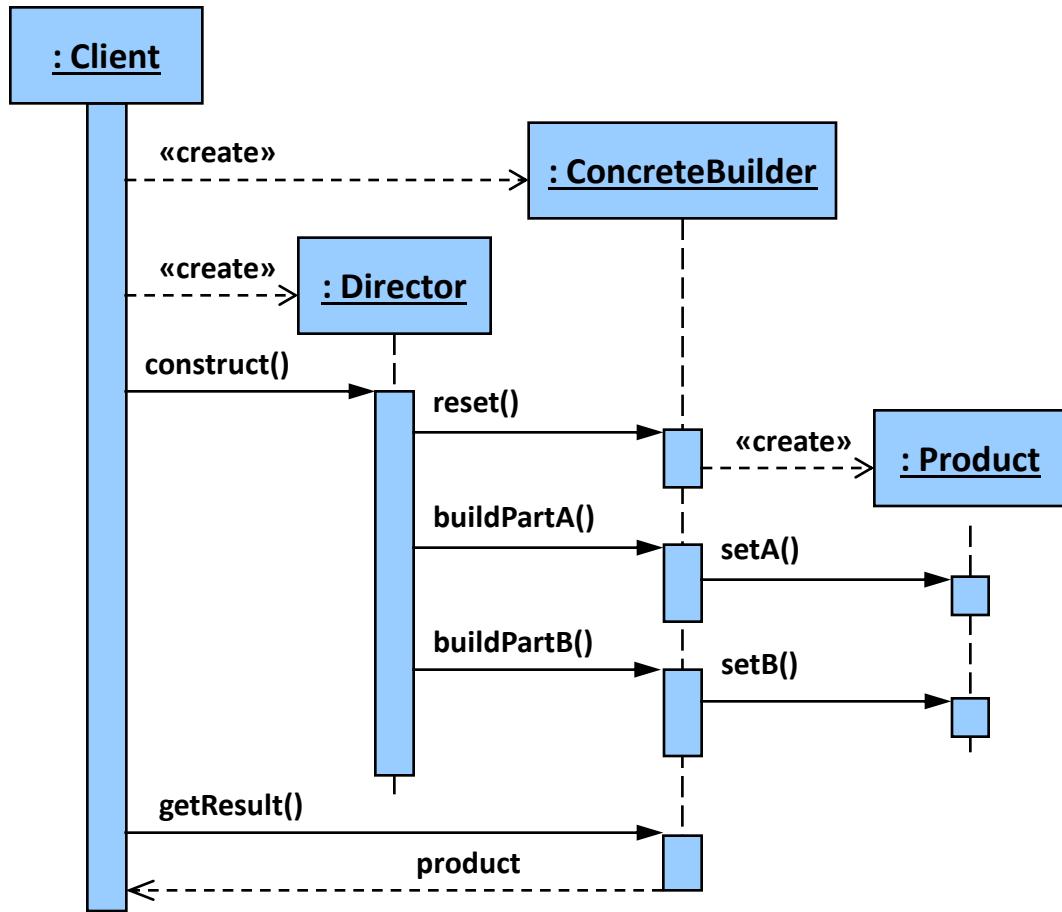
## Product

- Predstavlja složeni objekat koji treba kreirati.
- Uključuje klase koje definišu dijelove objekta koji treba kreirati, uključujući i interfejs za sklapanje njegovih dijelova u finalni objekat

# Kreacioni obrasci – Builder

## Implementacioni detalji

### Kolaboracija objekata



### Proces konstrukcije

- proizvodi se kreiraju „step-by-step“ (često suksesivno dodavanje novog dijela)
- **BUILDER obrazac se uobičajeno koristi za kreiranje kompozicija** (COMPOSITE obrazac)

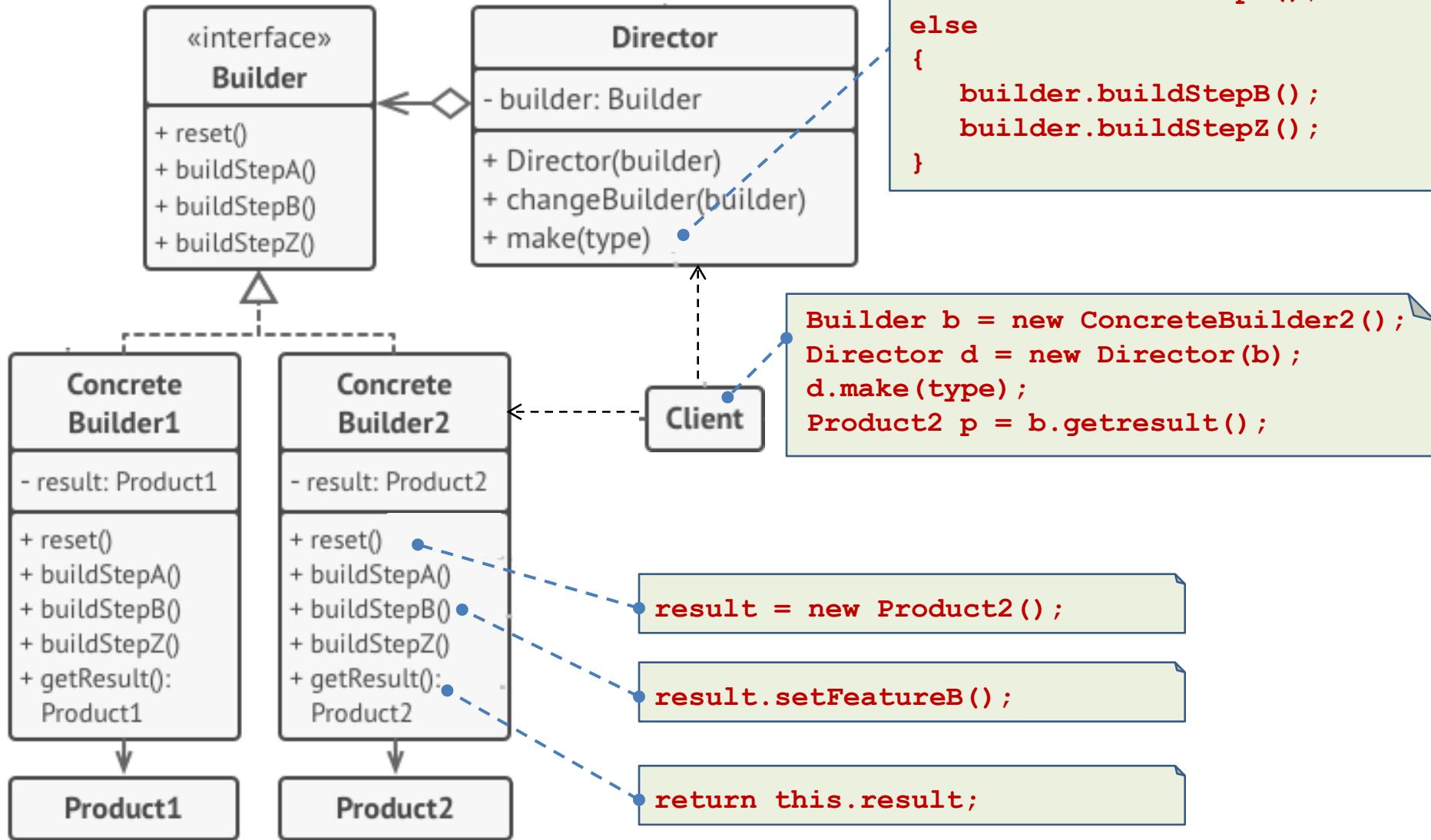
### Apstraktna klasa za proizvode?

- može da se uvede apstraktna klasa za proizvode, u cilju formiranja odgovarajuće hijerarhije klasa
- često builderi generišu veoma različite proizvode, pa nekad apstraktni proizvodi nemaju smisla

Klijent je jedini koji zna sve o tipovima objekata koji se prave.

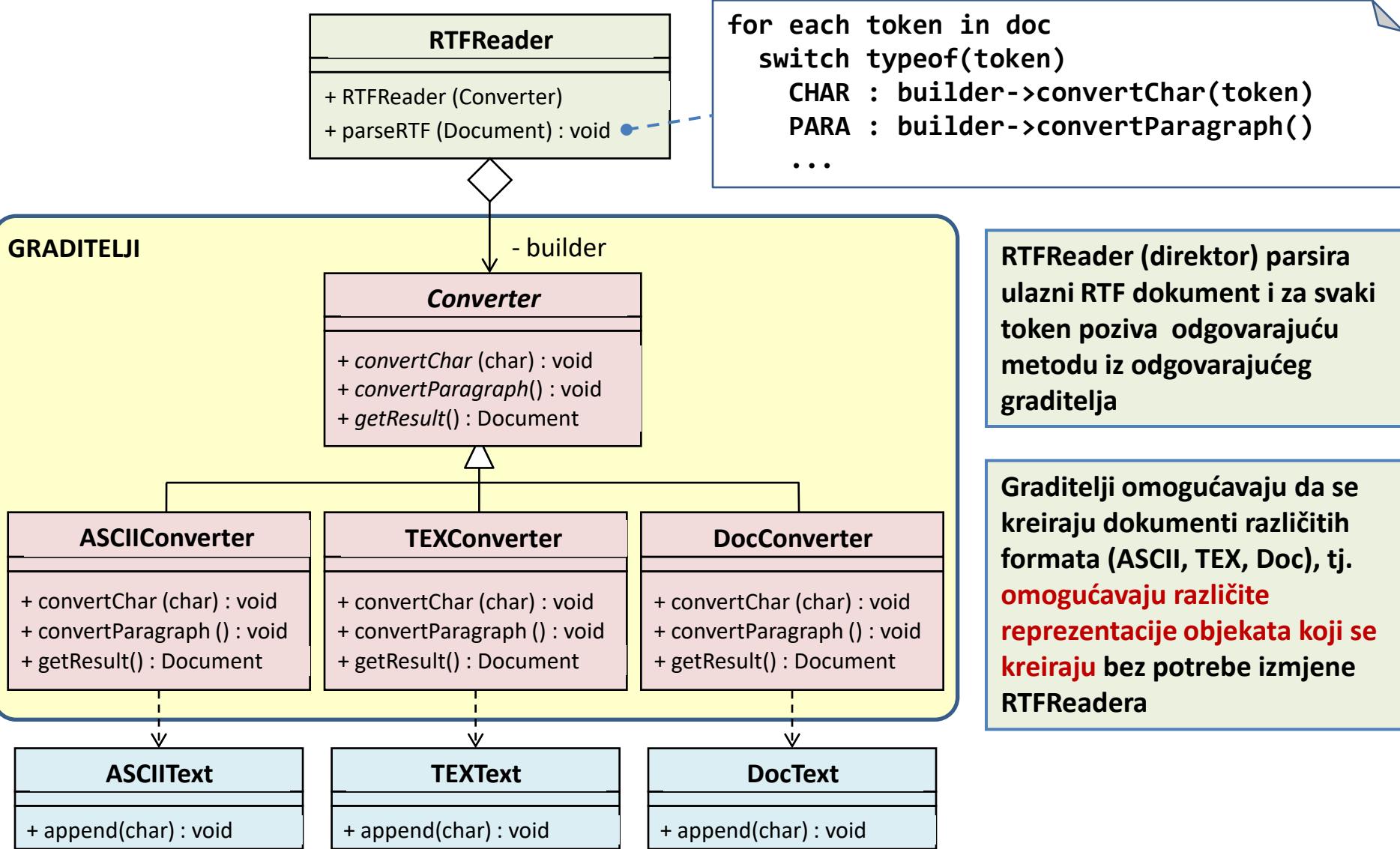
# Kreacioni obrasci – Builder

## Implementacioni detalji – kolaboracija objekata



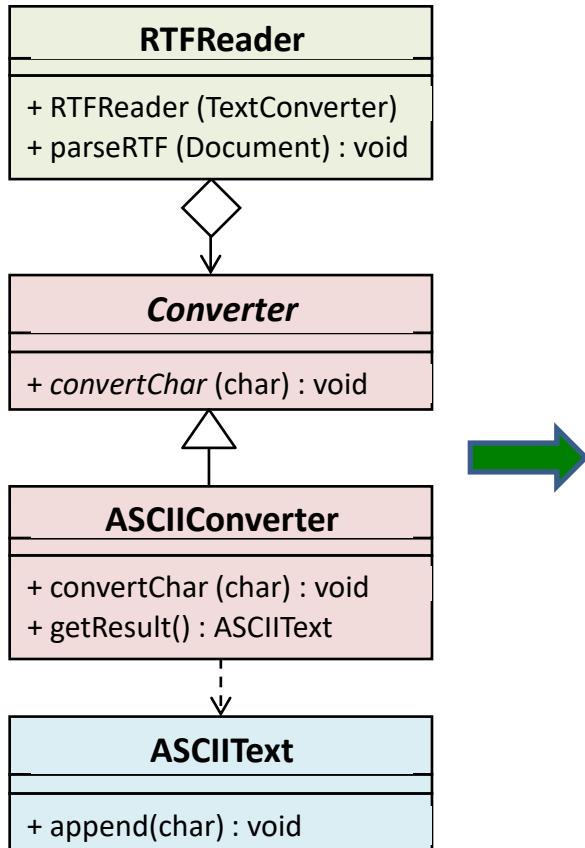
# Kreacioni obrasci – Builder

Primjer primjene BUILDER obrasca:



# Kreacioni obrasci – Builder

Primjer primjene:



```
abstract class Converter // AbstractBuilder
{
    abstract public void convertChar(char c);
}

class ASCIIText // Product
{
    public void append(char c) { /* implementacija */ }
}

class ASCIIConverter extends Converter // ConcreteBuilder
{
    private ASCIIText asciiText;
    public void convertChar(char c) { asciiText.append(asciiChar); }
    public ASCIIText getResult() { return asciiText; }
}

class RTFReader // Director
{
    private Converter builder;
    public RTFReader(Converter obj){ builder=obj; }
    public void parseRTF(Document doc)
    {
        // za svaki znak iz dokumenta doc
        builder.convertChar(znak);
    }
}

...
ASCIIConverter asciiBuilder = new ASCIIConverter();
RTFReader rtfReader = new RTFReader(asciiBuilder);
rtfReader.parseRTF(doc);
ASCIIText asciiText = asciiBuilder.getResult();
```

Klijent je jedini koji zna sve o tipovima objekata koji se prave.

**UNIVERZITET U BANJOJ LUCI  
ELEKTROTEHNIČKI FAKULTET**

**Prof. dr Dražen Brđanin**

**OBJEKTNOPRIJENTISANI DIZAJN  
/obrasci ponašanja/**

**Banja Luka  
2024.**

# Obrasci ponašanja

## Mjesto i uloga obrazaca ponašanja

- Obrasci ponašanja blisko su vezani sa algoritmima, raspodjelom uloga između objekata te njihovom međusobnom komunikacijom.
- Klasifikacija obrazaca ponašanja:
  - **class behavioral patterns** (raspodjela ponašanja između klasa pomoću **nasljeđivanja**)
    - *Template method, Interpreter*
  - **object behavioral patterns** (uglavnom zasnovani na **delegaciji**)
    - kooperacija grupe *peer* objekata u izvršavanju zadataka koje ne može da realizuje samo jedan objekat
      - *Mediator, Chain of Responsibility, Opserver*
    - inkapsulacija ponašanja u jedan objekat i delegiranje poziva tom objektu
      - *Strategy, Command, State, Visitor, Iterator, Memento*
- Obrasci ponašanja tipično su **uzajmno komplementarni** i forsiraju **uzajamnu primjenu**, npr:
  - “komanda” često uključuje “memento”
  - “komandni lanac” tipično uključuje primjenu “šablonskog metoda” i “komandi”
  - “interpreter” tipično uključuje “state” obrazac
- Obrasci ponašanja često se koriste u kombinaciji sa drugim vrstama obrazaca (strukturni)

# Obrasci ponašanja

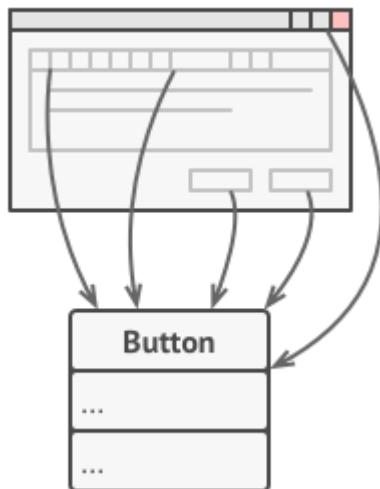
| naziv                                       | kratak opis                                                       |
|---------------------------------------------|-------------------------------------------------------------------|
| Command<br>(Komanda)                        | Inkapsulacija komande u objekat                                   |
| Memento<br>(Podsjetnik, Podsjećanje)        | Pamćenje i restauriranje stanja objekta                           |
| Iterator<br>(Brojač)                        | Sekvencijalni pristup elementima kolekcije                        |
| State<br>(Stanje)                           | Promjena ponašanja objekta u zavisnosti od promjene stanja        |
| Observer<br>(Posmatrač, Nadzornik)          | Prosljeđivanje informacije o nekoj promjeni većem broju objekata  |
| Strategy<br>(Strategija)                    | Inkapsulacija algoritma u klasu                                   |
| Chain of Responsibility<br>(Komandni lanac) | Prosljeđivanje zahtjeva kroz lanac objekata                       |
| Interpreter<br>(Tumač)                      | Uključivanje elemenata jezika u program                           |
| Mediator<br>(Posrednik)                     | Definisanje uprošćene komunikacije među klasama                   |
| Visitor<br>(Posjetilac)                     | Definisanje nove operacije u klasi bez njene eksplicitne promjene |
| Template Method<br>(Šablonski metod)        | Definisanje algoritamskih koraka koji se razlikuju u potklasama   |

# Obrasci ponašanja - Komanda

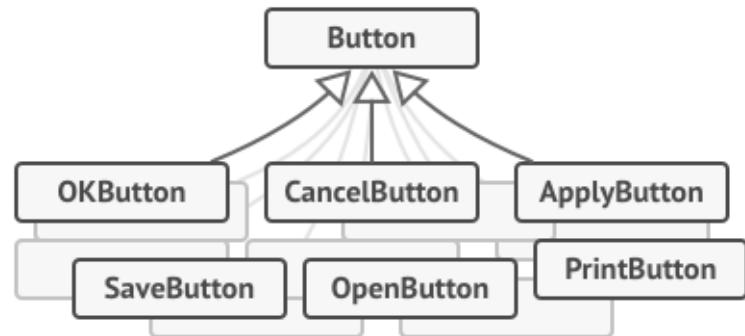
## Motivacija za uvođenje projektnog obrasca Komanda

Pretpostavimo da projektujemo GUI

(npr. klasa Button je osnovna klasa koja predstavlja apstrakciju svih dugmadi koja će se koristiti u aplikaciji)



U aplikaciji ima veći broj različitih dugmadi za izvršavanje različitih akcija pa je logično da svako različito dugme reprezentujemo odgovarajućom potklasom



### Ogroman broj potkласa

- + potklase koje imaju (gotovo) istu implementaciju
- + (gotovo) ista funkcionalnost i u drugim dijelovima aplikativnog koda, npr. "save" opcija u meniju, "Ctrl+S" prečica na tastaturi

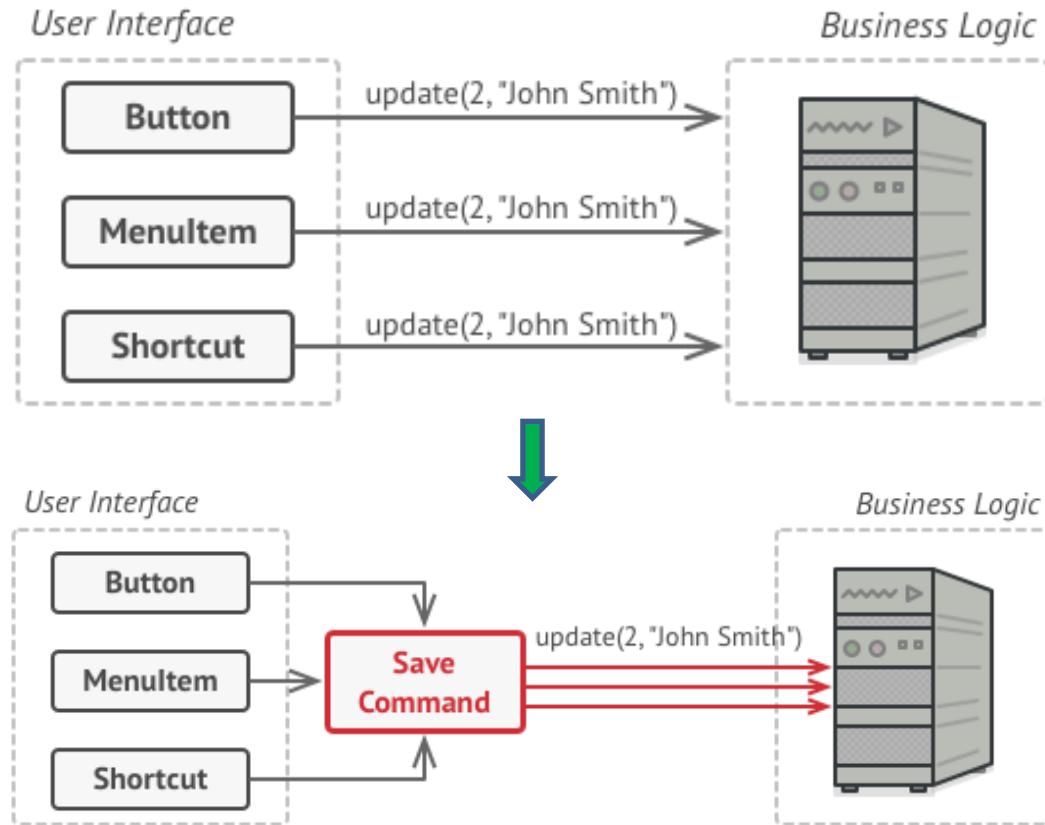
# Obrasci ponašanja - Komanda

## Motivacija za uvođenje projektnog obrasca Komanda

Dobro projektno rješenje bazira se na principu **“Separation of concerns”** – razdvajanje elemenata korisničkog interfejsa i aplikativne logike:

- GUI odgovoran za interakciju sa korisnikom
- aplikativni sloj odgovoran za implementaciju funkcionalnosti

Objekti iz korisničkog interfejsa šalju zahtjeve (*requests*) objektima u sloju aplikativne logike



Korisnici često žele da ponište efekte jedne ili više posljednjih izvršenih akcija (undo)  
Akcije se mogu izvršavati nad različitim objektima (npr. promjena boje nekog objekta)  
**Zato je poželjno da se takve akcije (komande) izdvoje u zasebne objekte**

# Obrasci ponašanja - Komanda

## Motivacija za uvođenje projektnog obrasca Komanda

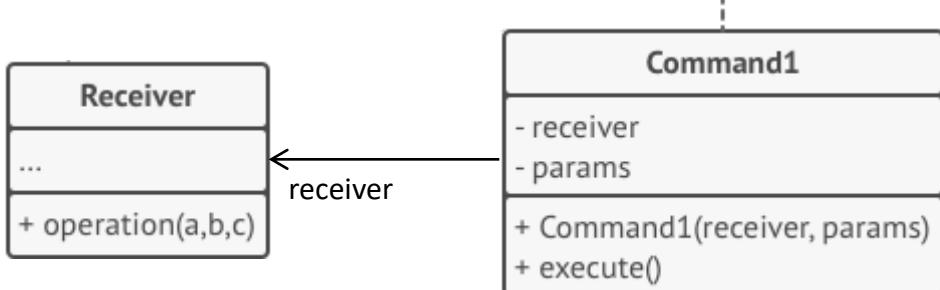
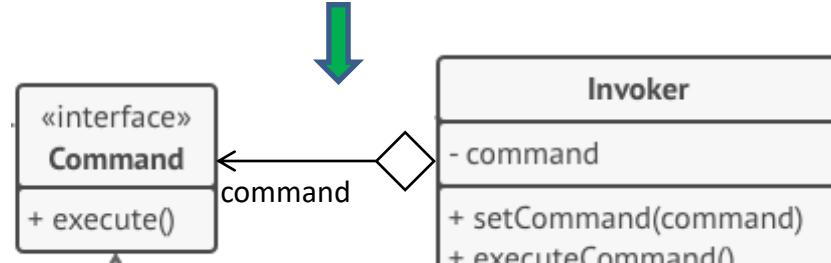
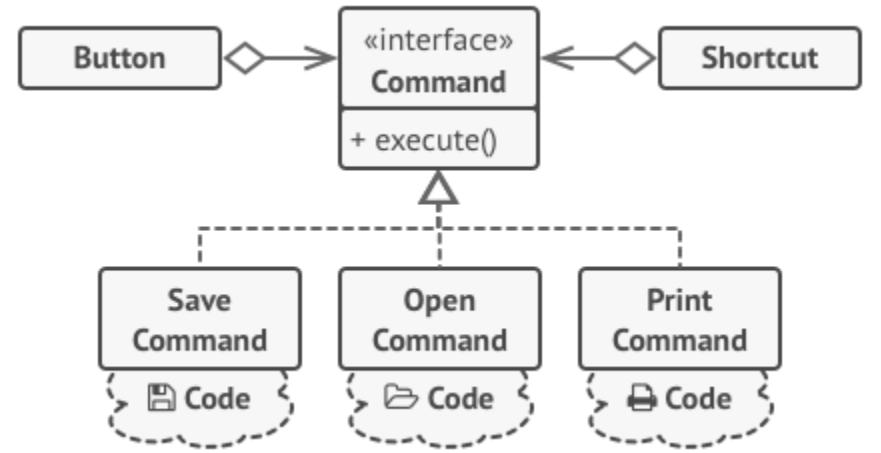
Dobro projektno rješenje:

- **zajednički interfejs za sve komande**  
(obično samo jedna metoda execute)
- **konkretne klase za svaku komandu**  
(implementiraju metodu execute)

Kako se postiže različitost komandi,  
ako imaju samo execute metodu?

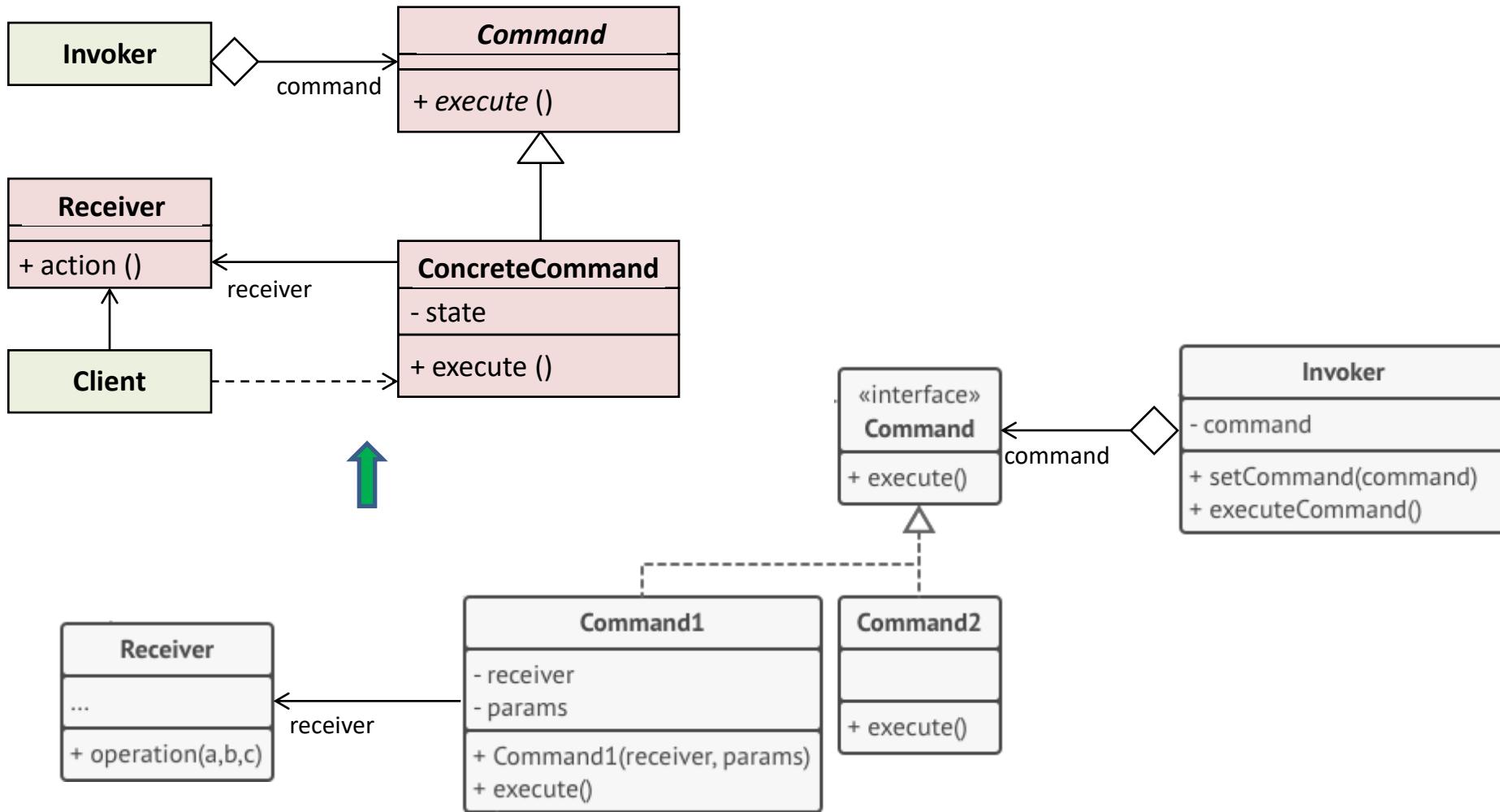
Komande se prilikom instanciranja inicializuju  
odgovarajućim parametrima:

- receiver,
- parametri operacije



# Obrasci ponašanja - Komanda

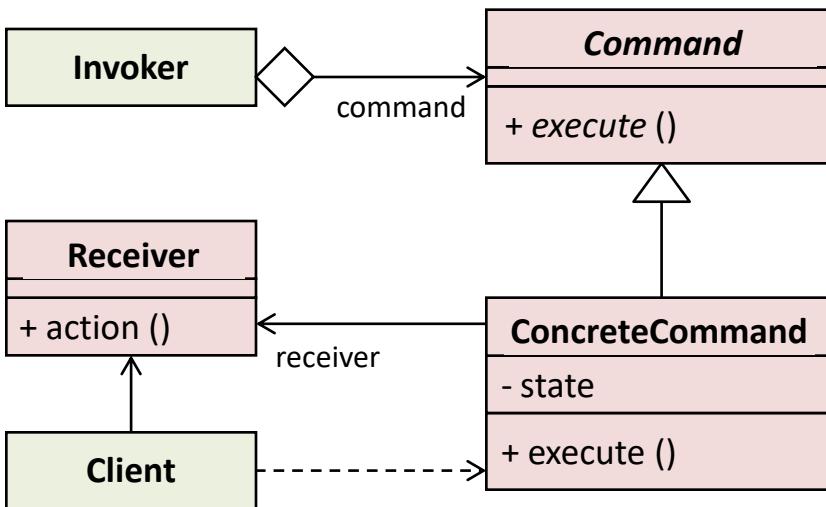
## Motivacija za uvođenje projektnog obrasca Komanda



# Obrasci ponašanja - Komanda

## Command (Komanda)

- Inkapsulira zahtjev (komandu) u objekat što omogućava parametrizaciju klijenta različitim zahtjevima, nizovima poruka i omogućava realizaciju operacija nad kojima je moguće izvršiti „undo“ operaciju.



**Uobičajeni alternativni nazivi za KOMANDU:**  
**AKCIJA, TRANSAKCIJA**

Tipične primjene COMMAND obrasca:  
GUI komandni elementi (menuitem, button)

**COMMAND** obrazac tipično služi za  
„rasprezanje“ **BOUNDARY ↔ CONTROL**

## Command

- deklariše interfejs za izvršavanje operacije

## ConcreteCommand

- definiše vezu između objekta klase **Receiver** i akcije koja mu se upućuje
- implementira **execute()** metodu pozivajući odgovarajuće operacije u klasi **Receiver**  
`receiver.action()`

## Client

- kreira objekat klase **ConcreteCommand** i postavlja objekat klase **Receiver** koji će prihvati njegove pozive

## Invoker

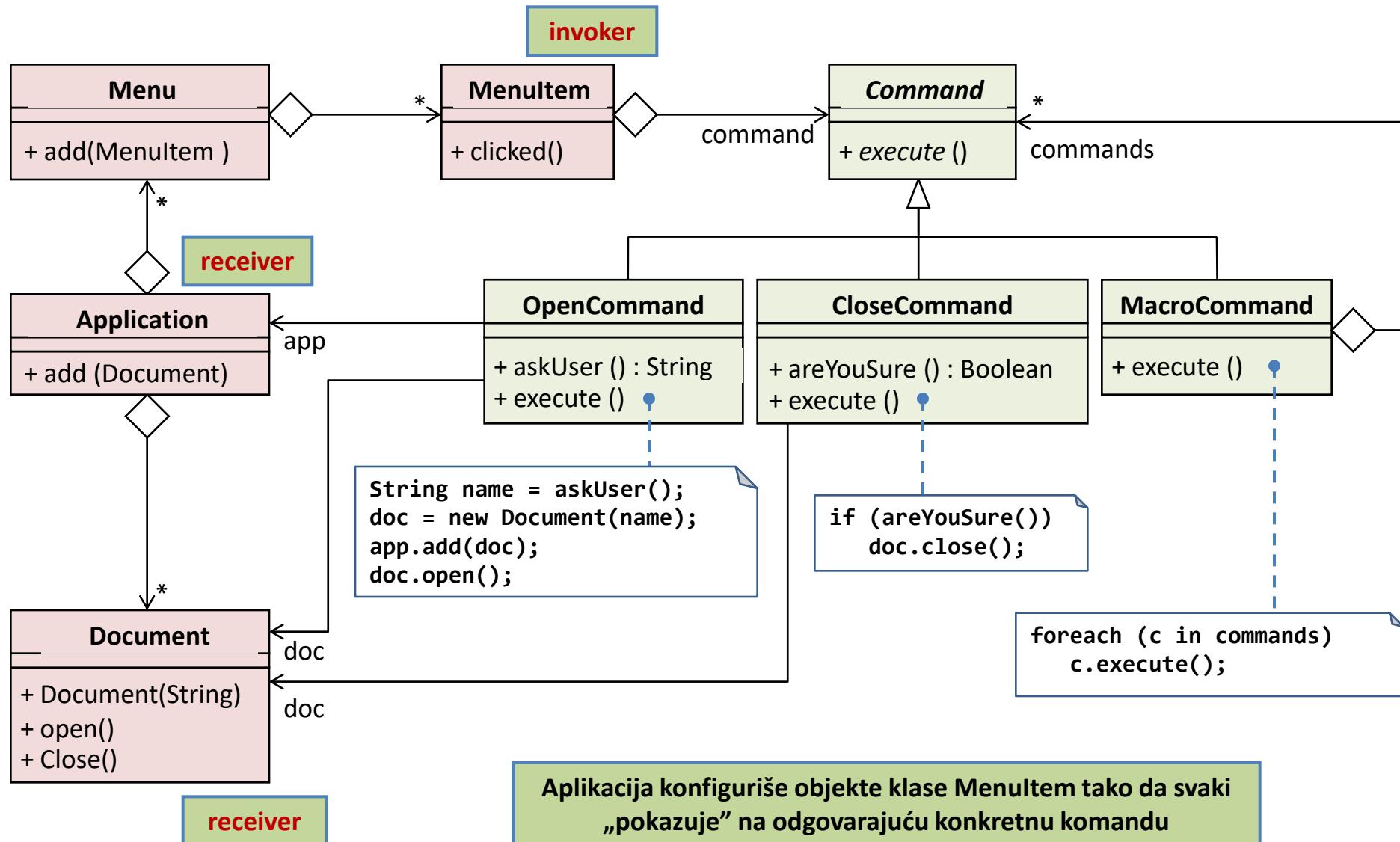
- traži da mu komanda obradi zahtjev

## Receiver

- izvršava operaciju na zahtjev komande

# Obrasci ponašanja - Komanda

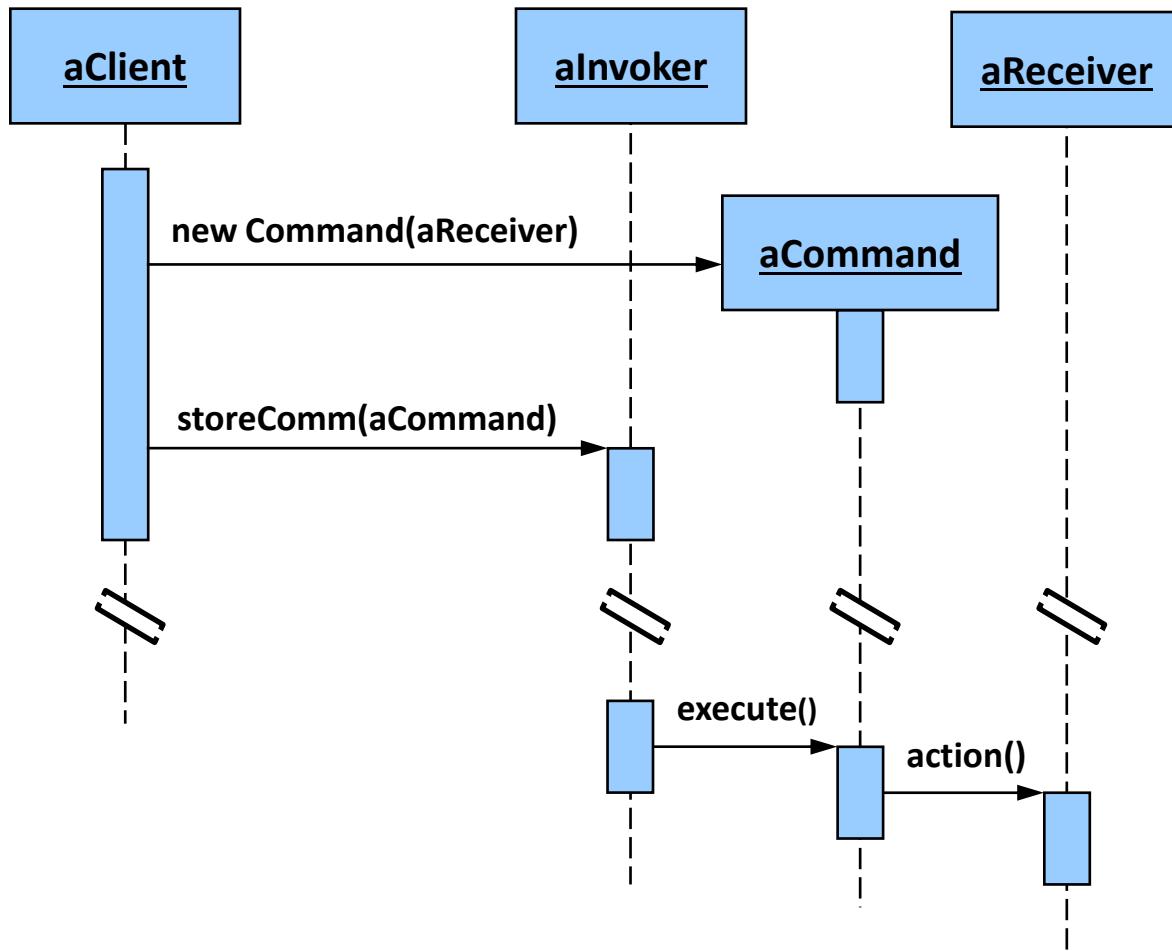
Primjer (Realizacija menija zasnovana na COMMAND obrascu):



# Obrasci ponašanja - Komanda

## Implementacioni detalji

– kolaboracija objekata



*Klijent kreira komandu i specifičuje njen receiver.*

*Klijent konfiguriše invoker kreiranim komandom.*

*Invoker šalje zahtjev komandi (Execute). Ako je “undoable”, komanda pamti stanje.*

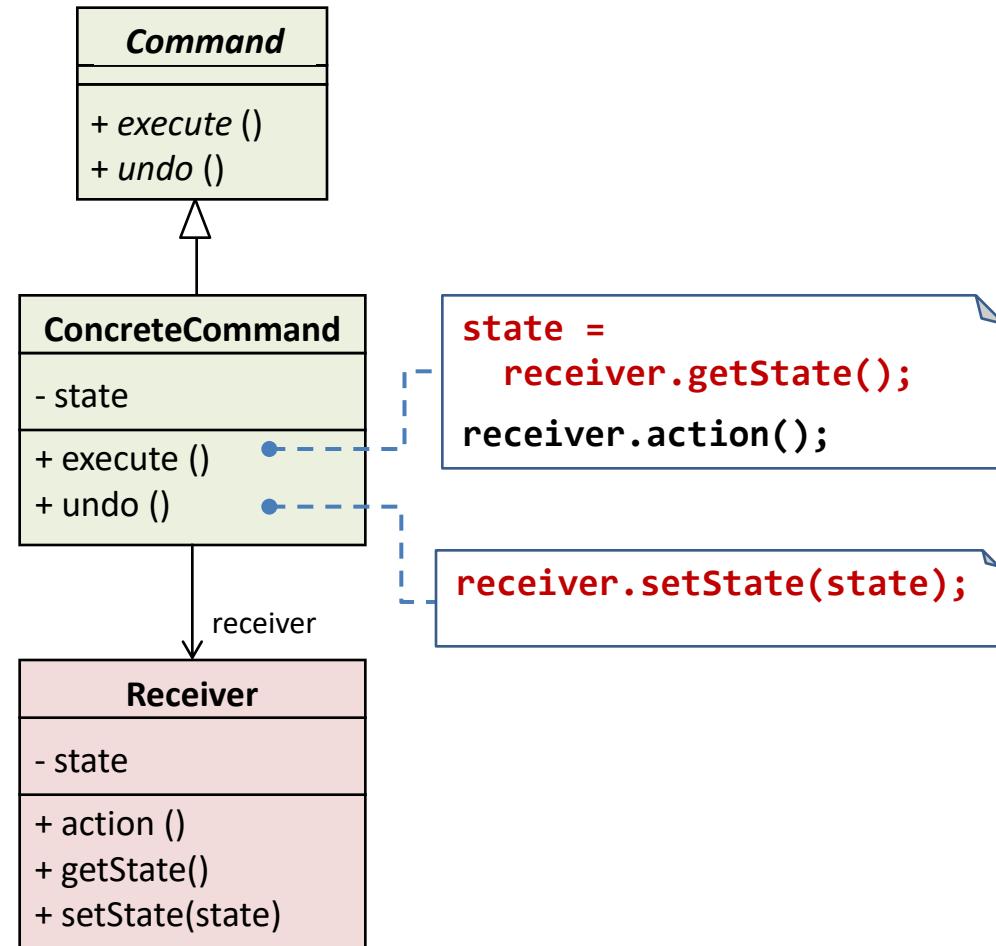
*Komanda inicira izvršavanje odgovarajuće operacije na receiveru.*

# Obrasci ponašanja - Komanda

## Implementacioni detalji

### Podrška za UNDO/REDO

- Komanda treba da ima operaciju **undo()** ili **unExecute()**
- Neophodno je pamćenje **stanja** (komanda pamti stanje), što uključuje:
  - prethodno stanje receivera,
  - receiver mora da ima operaciju za vraćanje u prethodno stanje
- **Za jedan UNDO nivo:** aplikacija treba da pamti samo posljednju komandu
- U slučaju složenih objekata, pamćenje stanja može da se realizuje pomoću Memento obrasca

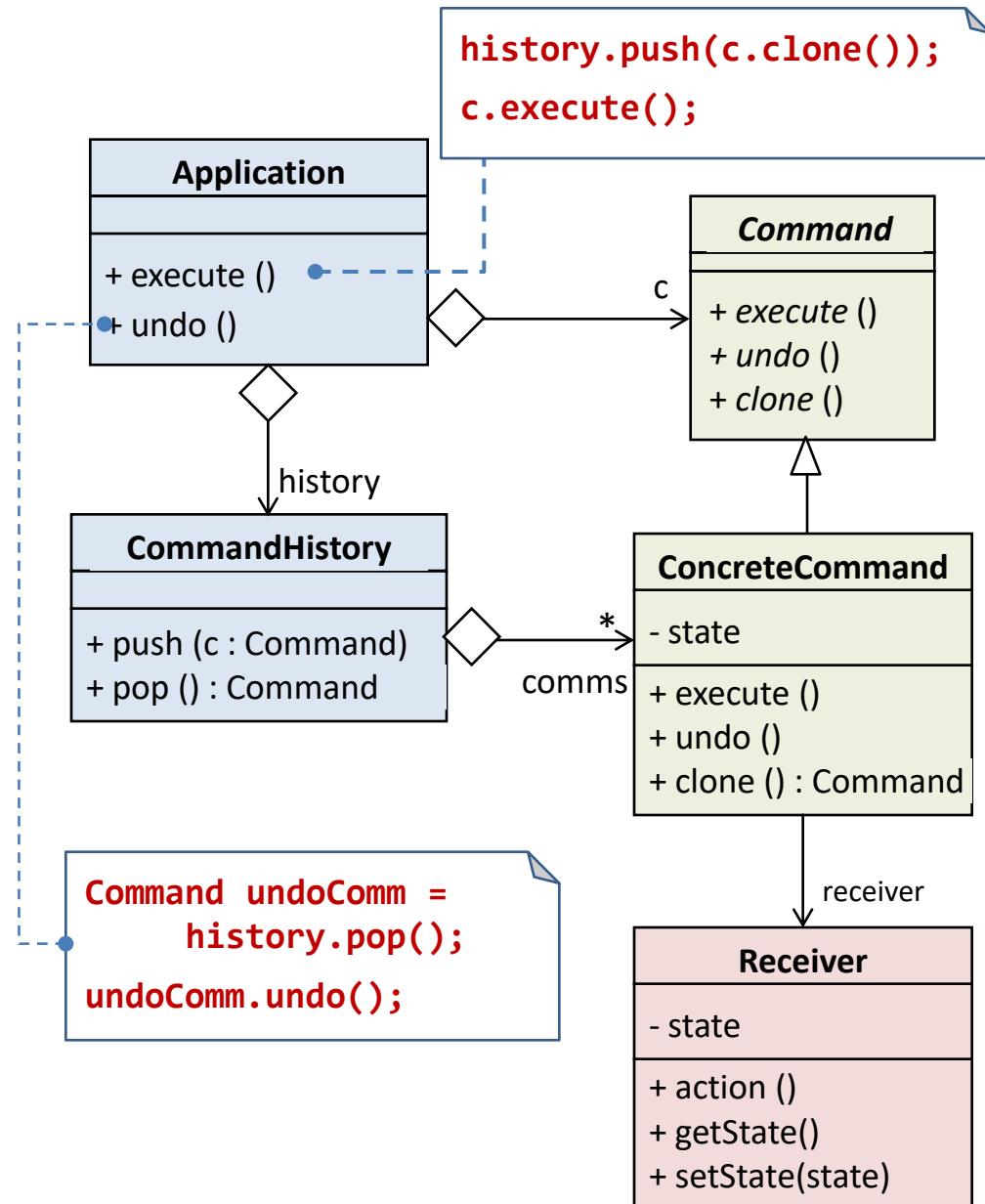


# Obrasci ponašanja - Komanda

## Implementacioni detalji

### Podrška za višestruki UNDO/REDO

- aplikacija treba da ima **history list** sa svim izvršenim komandama
  - UNDO: iteriranje unazad
  - REDO: iteriranje unaprijed
- stavljanje komande u history list tipično zahtijeva kopiranje komande (kopija se stavlja u listu, a original može kasnije da se koristi za izvršavanje nove operacije)
  - npr. delete komanda koja briše selektovane objekte, svaki put kad se izvršava mora da se konfiguriše različitim *receiver* objektima na koje se primjenjuje, pravi se kopija koja se dodaje u *history list*, a zatim se izvršava akcija
- za kopiranje komande tipično se koristi PROTOTIP obrazac

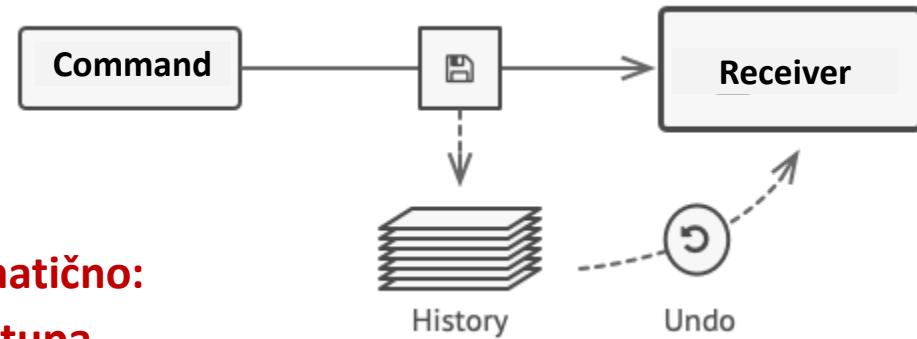


# Obrasci ponašanja - Memento

## Motivacija za uvođenje projektnog obrasca Memento

Pretpostavimo da aplikacija treba da omogući UNDO operacije nad objektima, zbog čega treba da se pamti stanje objekata

(npr. kod obrasca "komanda", prije nego što se komanda izvrši treba da se zapamti stanje datog objekta)



Memorisanje stanja objekta može biti problematično:

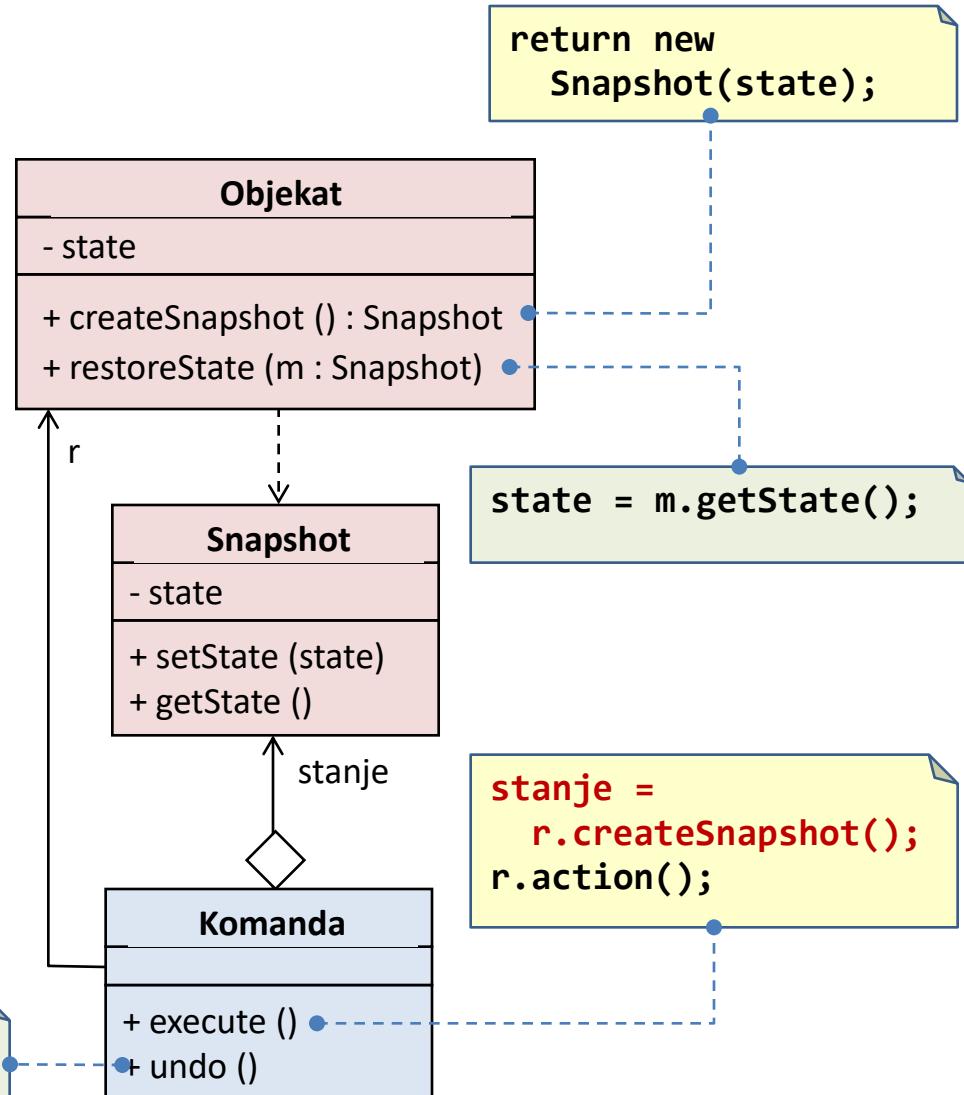
- Možda objekat ne omogućava da neko pristupa njegovim atributima i čita stanje!
- Koje atrIBUTE treba memorisati i kako da to realizuju različite komande, pogotovo ako komande mogu da se primjenjuju nad različitim objektima? (svaka komanda mora da memoriše stanje receivera neposredno prije izvršenja)

# Obrasci ponašanja - Memento

## Motivacija za uvođenje projektnog obrasca Memento

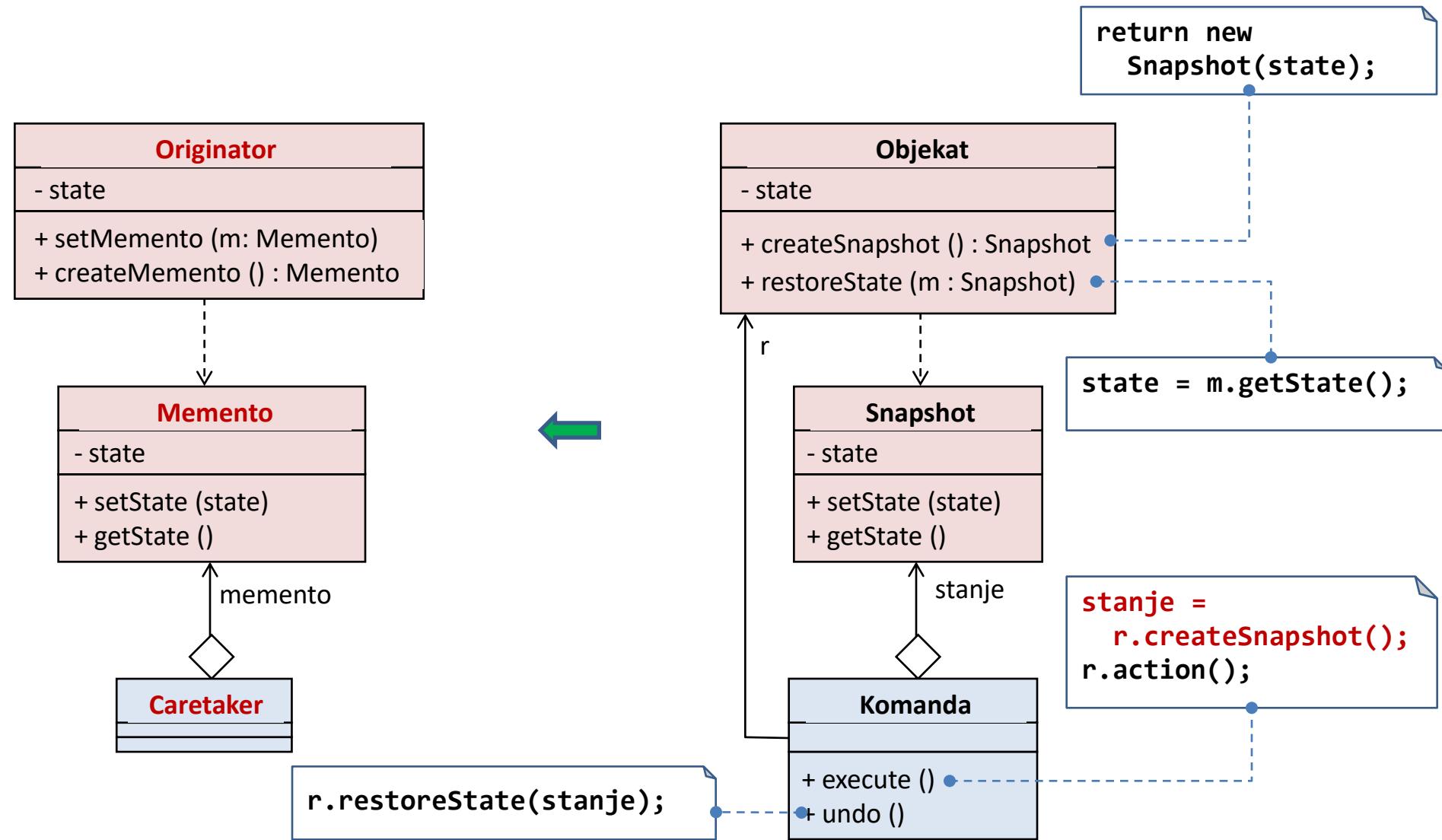
Dobro projektno rješenje:

- Objekat od interesa (receiver) treba da ima mogućnost da po potrebi kreira snapshot (stanje u datom trenutku)
- Objekat od interesa (receiver) treba da ima mogućnost restauriranja stanja na osnovu nekog snapshota
- Subjekat (komanda) koji je zainteresovan za pamćenje stanja objekta od interesa, po potrebi može da traži od objekta da kreira snapshot, odnosno da restaurira stanje na osnovu nekog snapshota



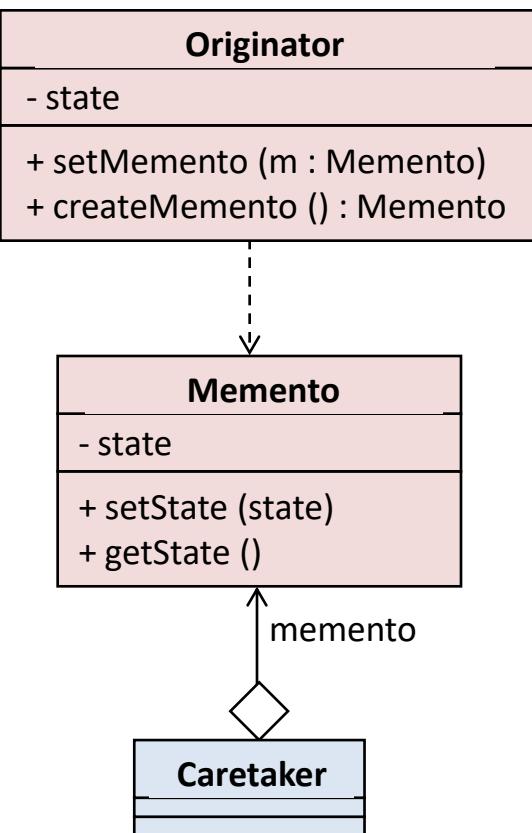
# Obrasci ponašanja - Memento

## Motivacija za uvođenje projektnog obrasca Memento



# Obrasci ponašanja - Memento

Memento  
(Podsjećanje)



- Bez narušavanja inkapsulacije, hvata i čuva interno stanje objekta kako bi on kasnije mogao biti vraćen u sačuvano stanje

## Memento

- Čuva interno stanje objekta klase Originator - treba da sačuva dovoljno podataka kako bi omogućio njegovu kasniju restauraciju.
- Čuva objekte klase Originator od neželjene izmjene.
- Memento ima dva interfejsa:
  - **Caretaker vidi “uski” interfejs** prema objektu klase Memento (može samo da proslijedi jedan Memento objekat drugim objektima).
  - **Originator vidi “širok” interfejs** (da postavi vrijednosti svih neophodnih polja koja će mu omogućiti kasniju restauraciju).
- Samo Originator koji je kreirao Memento može da pristupi mementovom unutrašnjem stanju (idealno!).

## Originator

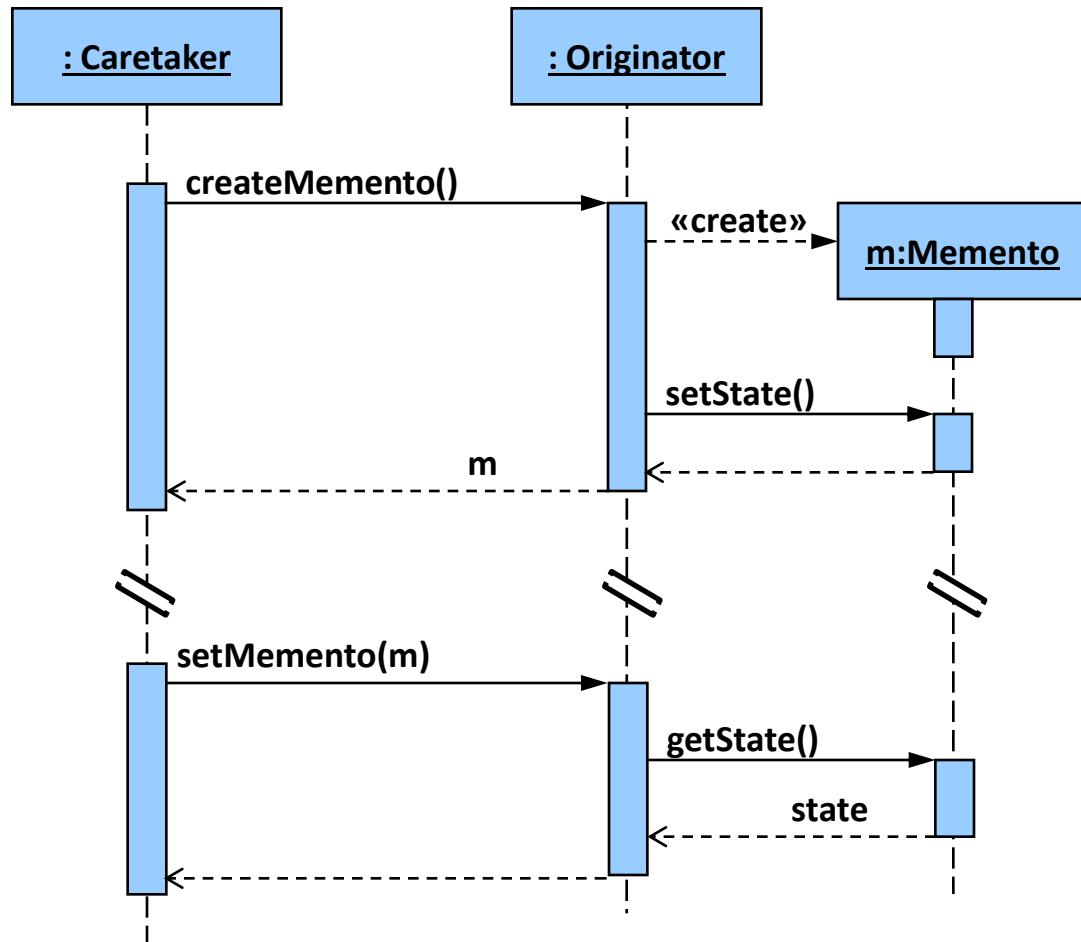
- Kreira Memento (snimak njegovog trenutnog stanja).
- Koristi Memento da resturira svoje unutrašnje stanje.

## Caretaker

- Odgovoran za čuvanje Mamenta.
- Nikada ne ispituje niti otvara sadržaj Mamenta.

# Obrasci ponašanja - Memento

## Tipična kolaboracija



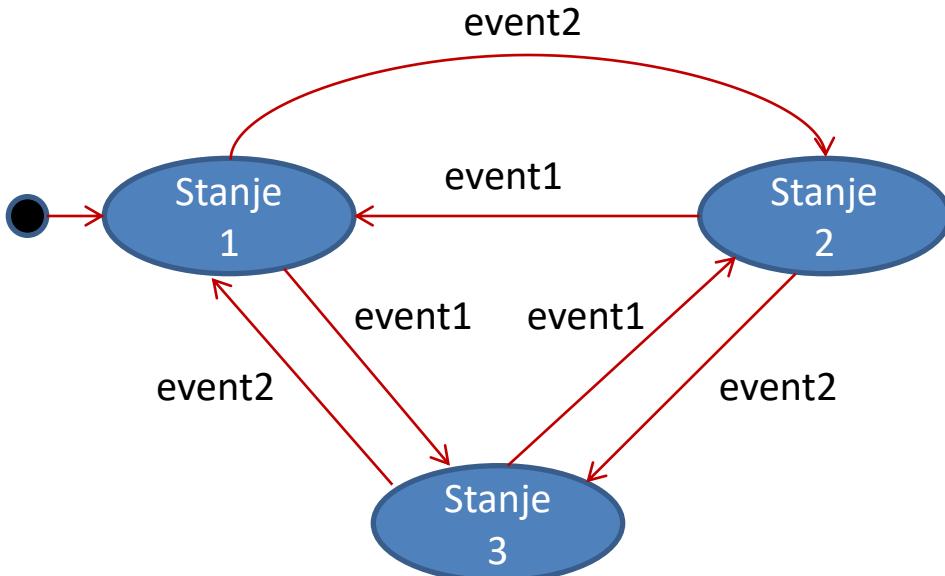
- Caretaker zahtijeva da originator kreira memento, kako bi se kasnije moglo vratiti staro stanje.
- Kad zatreba restauracija starog stanja, caretaker zahtijeva od originatora da restaurira stanje na osnovu magenta.
- Memento je pasivan – samo originator može da ga kreira i restaurira svoje stanje na osnovu magenta.

# Obrasci ponašanja - State

## Motivacija za uvođenje projektnog obrasca State

Pretpostavimo da aplikacija treba da implementira mašinu stanja koja reprezentuje životni vijek nekog objekta (ponašanje objekta zavisi od stanja u kojem se nalazi)

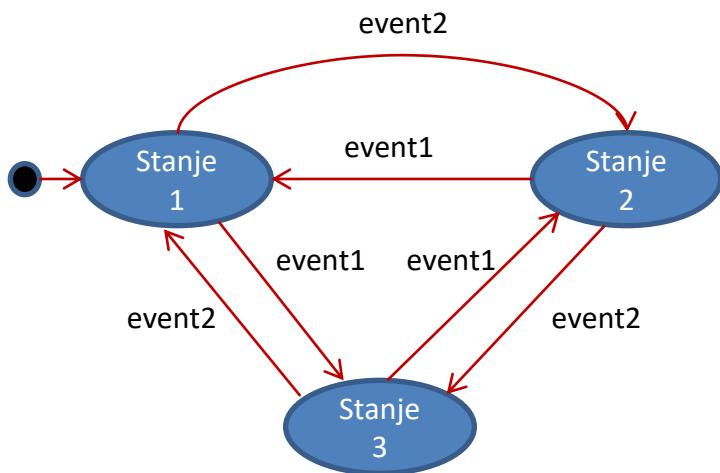
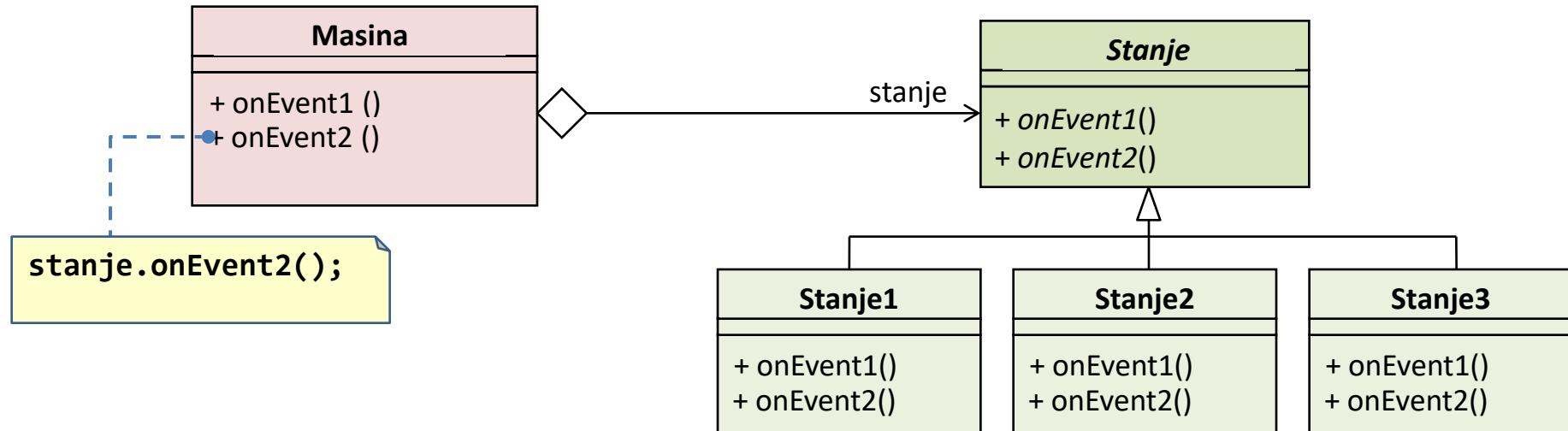
Ponašanje objekta na neki događaj zavisi od stanja u kojem se nalazi, što može da rezultuje metodama čija implementacija sadrži dosta grananja i provjera većeg broja uslova:



```
public class Masina
{
    private int stanje;
    public void onEvent1()
    {
        if (stanje==1) { stanje=3; // ... }
        if (stanje==2) { stanje=1; // ... }
        if (stanje==3) { stanje=2; // ... }
    }
    // ...
}
```

# Obrasci ponašanja - State

## Motivacija za uvođenje projektnog obrasca State

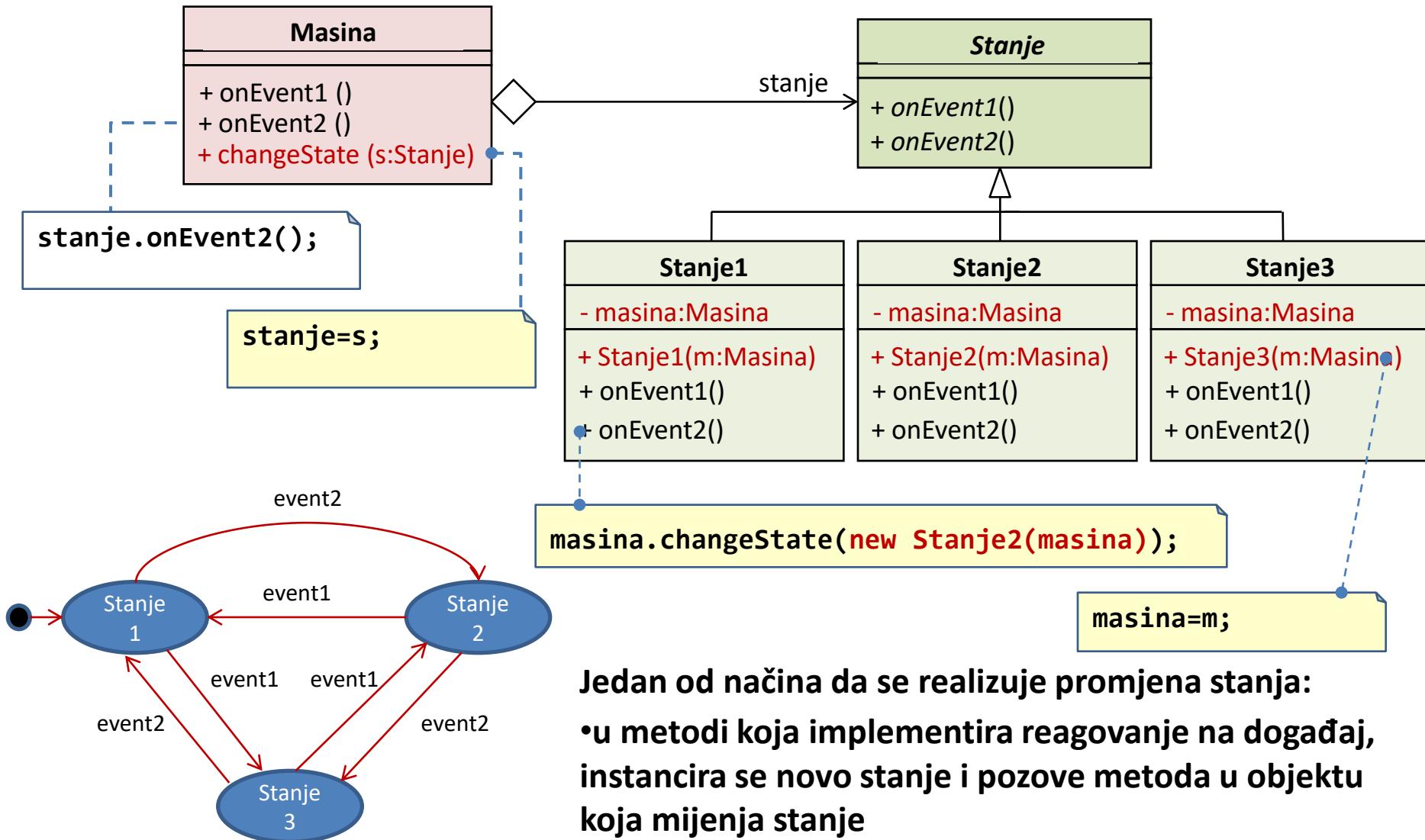


Dobro projektno rješenje jeste izvlačenje aplikativne logike iz klase koja reprezentuje objekat u posebne klase – za svako stanje po jedna klasa koja implementira ponašanje karakteristično za dato stanje.

Objekat ima referencu na trenutno stanje i reagovanje na događaj svodi se na delegiranje poziva objektu koji reprezentuje trenutno stanje  
`stanje.onEvent();`

# Obrasci ponašanja - State

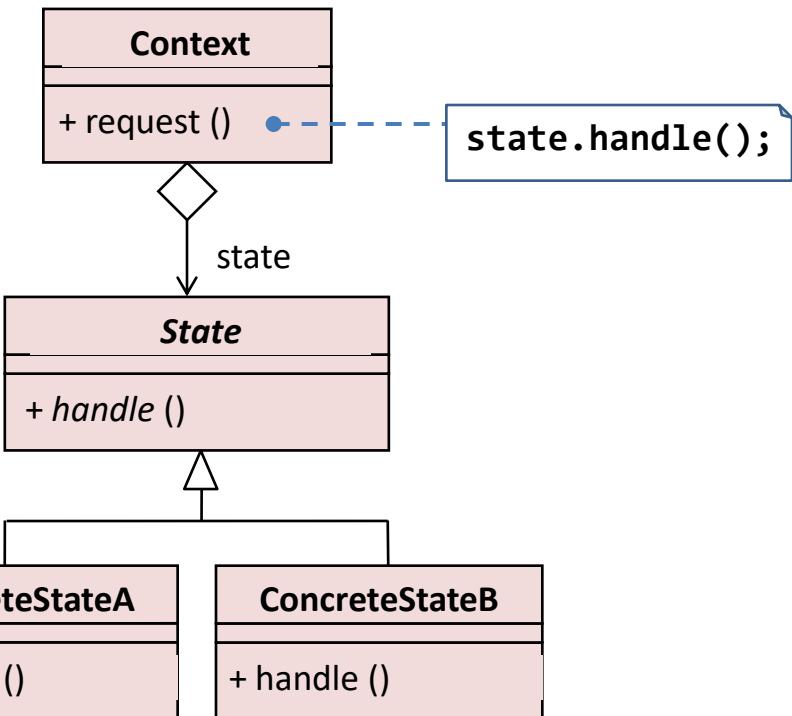
## Motivacija za uvođenje projektnog obrasca State



# Obrasci ponašanja - State

## State (Stanje)

- Dozvoljava da objekat promijeni ponašanje kada se njegovo interno stanje promijeni – liči na to da objekat postaje instanca druge klase



## Context

- definiše interfejs koji je potreban klijentu
- vodi računa o instanci klase **ConcreteState** koja definiše trenutno stanje

## State

- deklariše interfejs za inkapsulaciju ponašanja koje je povezano sa određenim stanjem klase Context

## ConcreteState

- implementira ponašanje vezano za odgovarajuće stanje klase Context
- za svako stanje po jedna **ConcreteState** klasa

## Kad se koristi STATE obrazac?

- Kad ponašanje objekta zavisi od stanja u kojem se nalazi i mora da se promijeni u toku izvršavanja zavisno od promjene stanja.
- Kad je implementacija operacija uveliko zavisna od stanja objekta (mnogo uslovnih iskaza, ...).

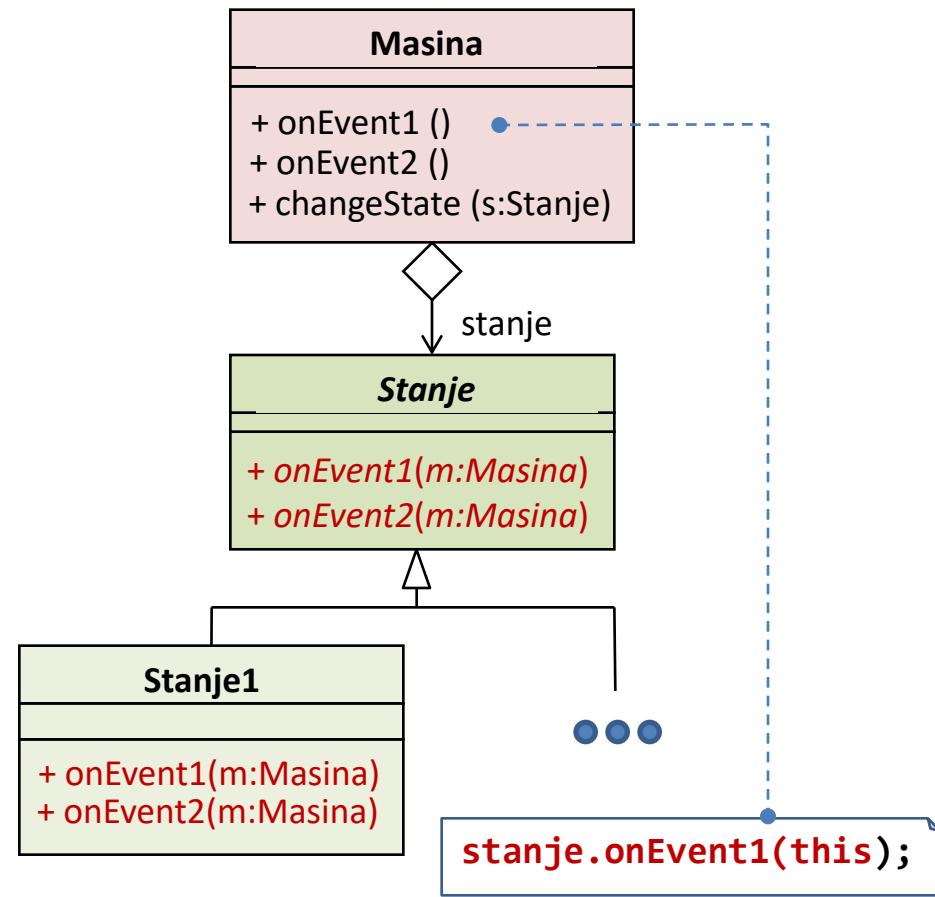
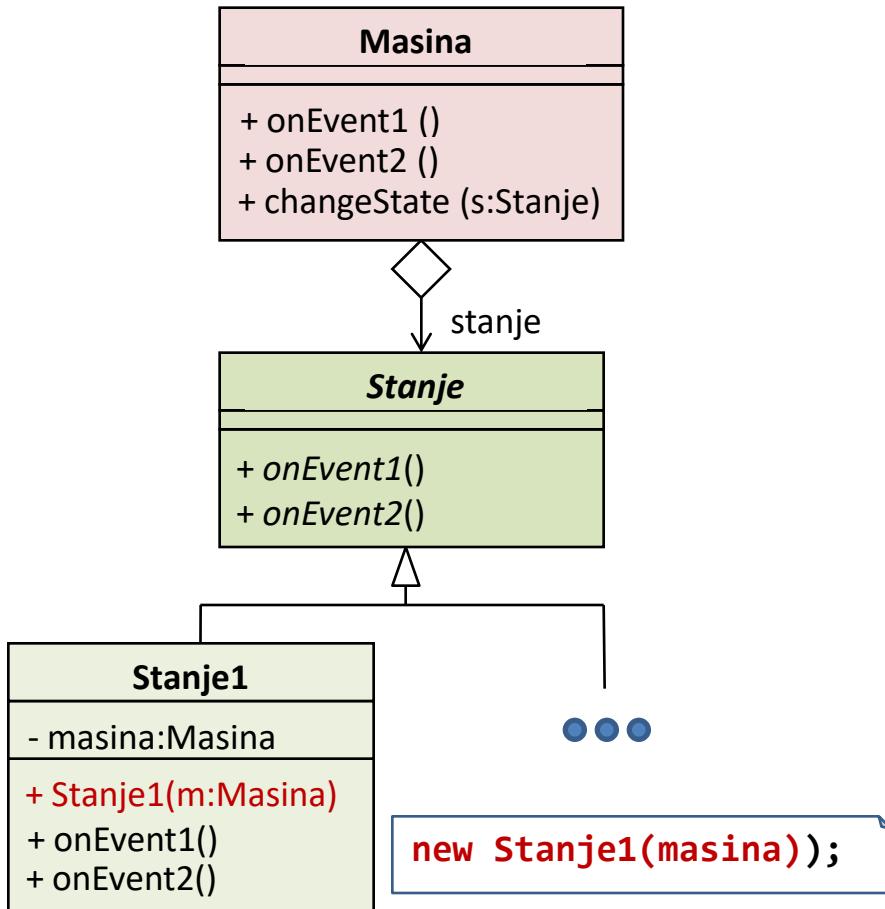
# Obrasci ponašanja - State

# Implementacioni detalji – referenca na Context

- Stanje mora da ima odgovarajuću referencu na Masinu (Context)

stanje može da se inicijalizuje odgovarajućom mašinom prilikom instanciranja stanja

stanje može da dobije referencu na mašinu kao parametar u poruci za izvršavanje operacije



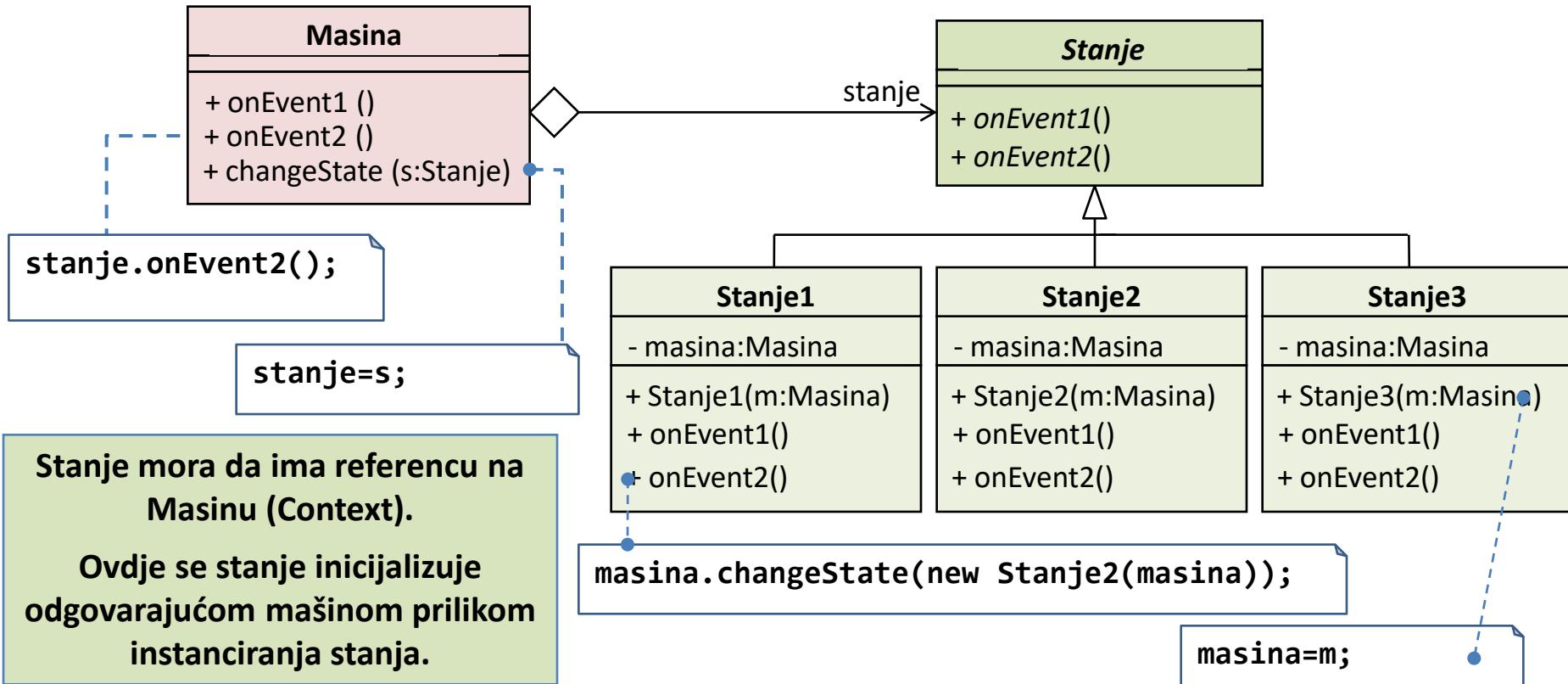
# Obrasci ponašanja - State

## Implementacioni detalji – Promjene stanja

- STATE obrazac ne specifikuje odgovornost za promjenu stanja objekta
- Implementacija kriterijuma za promjenu stanja: **decentralizovano / centralizovano**

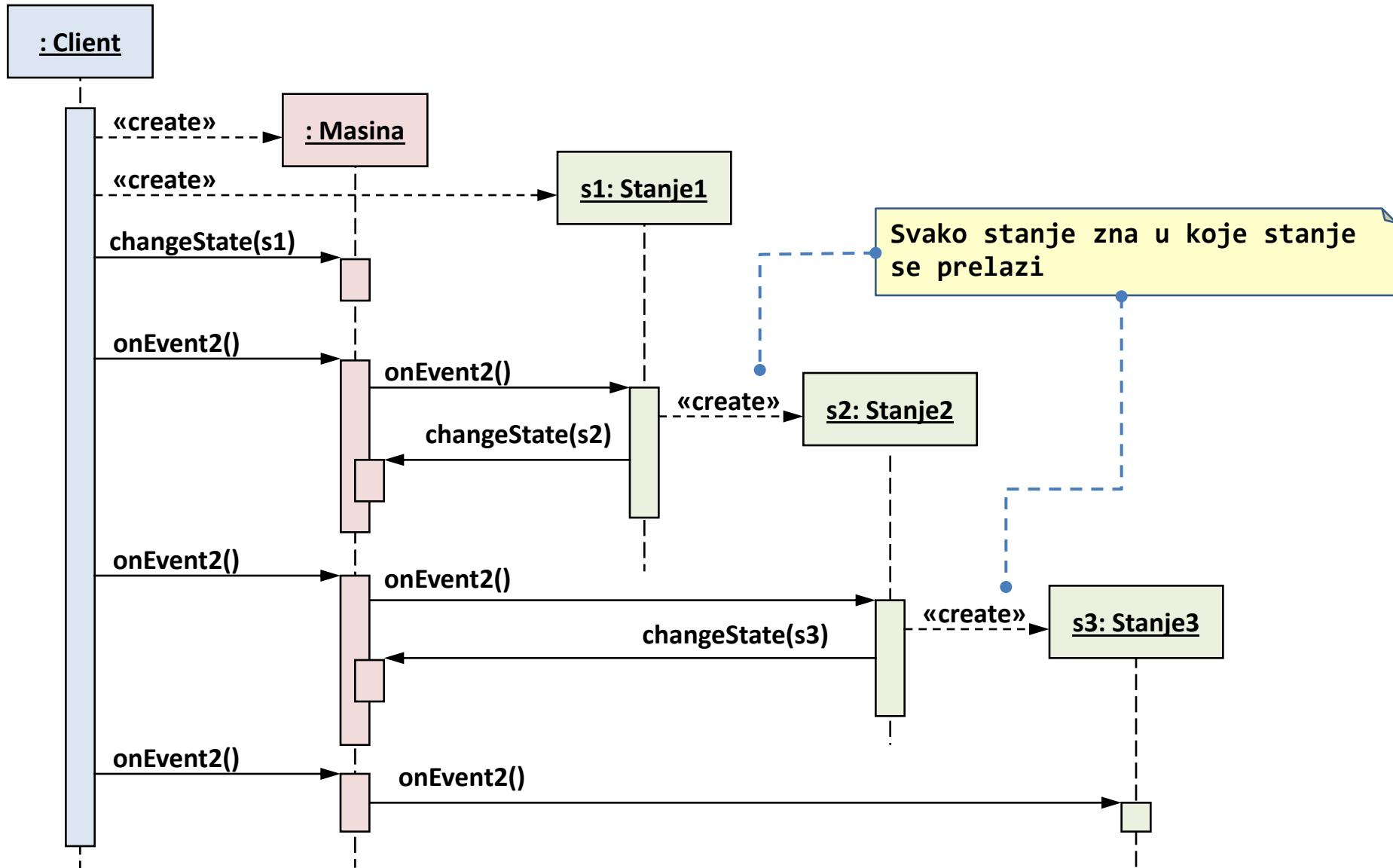
### Decentralizovana kontrola promjene stanja

- svako konkretno stanje definiše kriterijume za tranziciju u sljedeće stanje



# Obrasci ponašanja - State

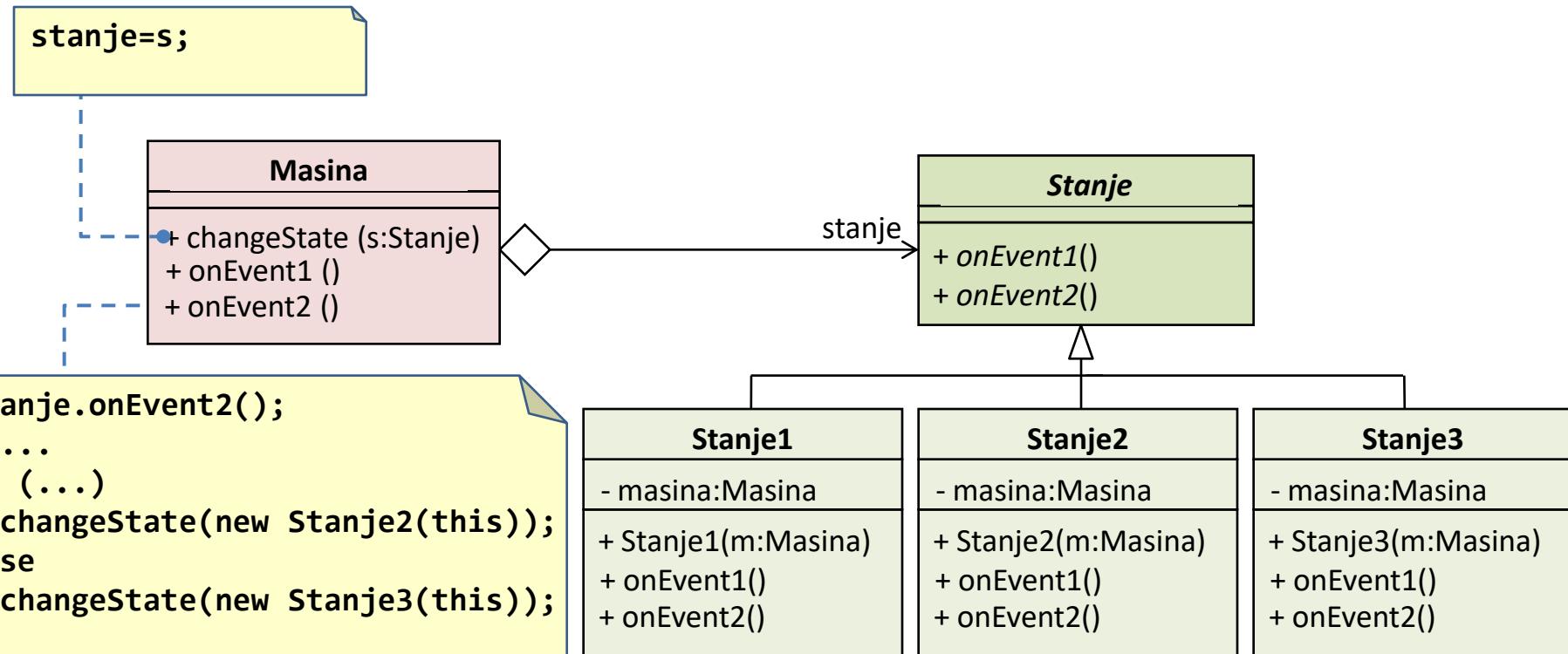
Interakcija objekata u decentralizovanoj kontroli promjene stanja



# Obrasci ponašanja - State

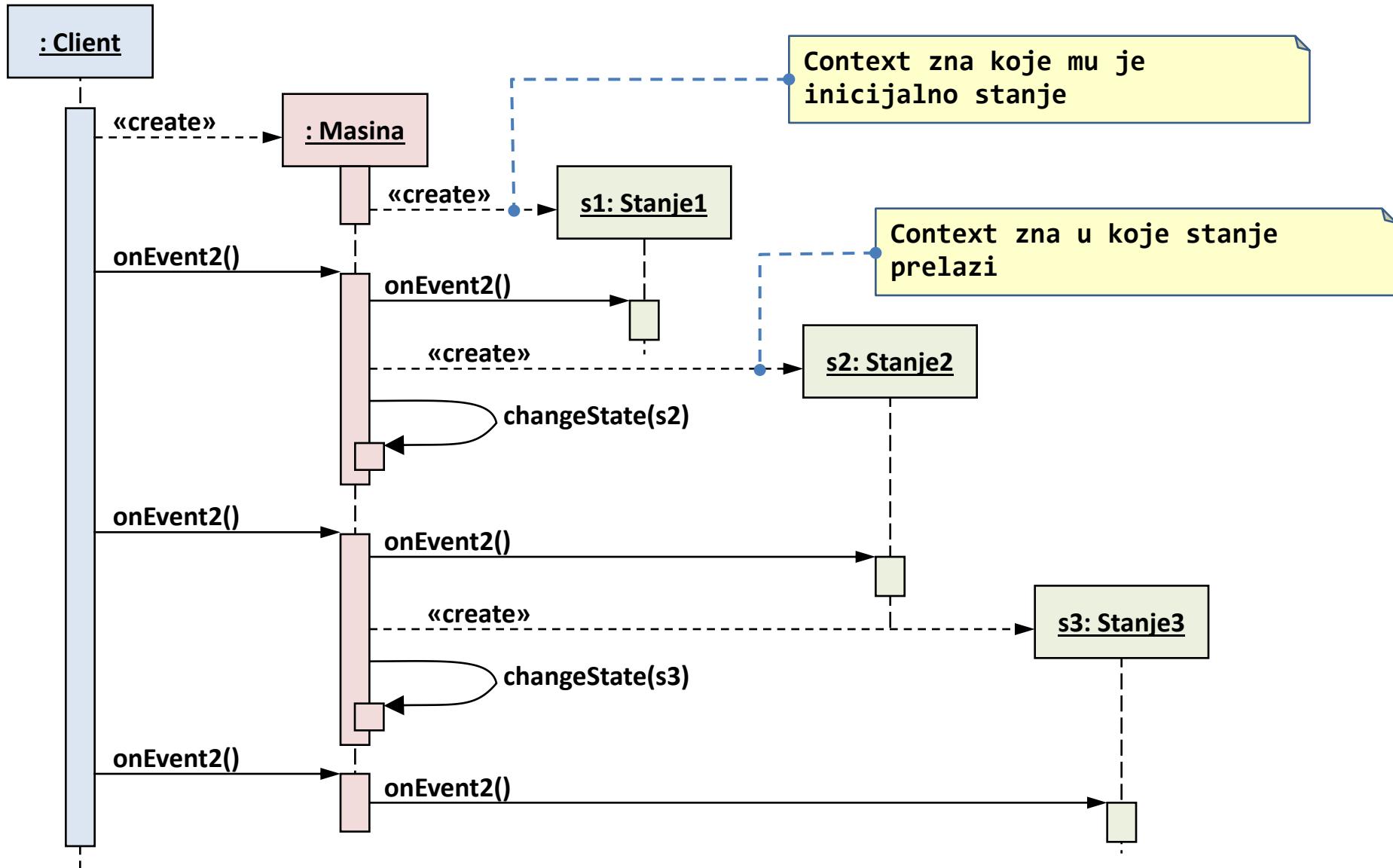
## Centralizovana kontrola promjene stanja

- kontrola promjene stanja realizuje se unutar Context klase  
(kontekst instancira odgovarajuće stanje i sam vrši promjenu stanja)
- dobro rješenje za tranzicije: **look-up tabela (polazno stanje → mogući ulazi → ciljno stanje)**



# Obrasci ponašanja - State

Interakcija objekata u centralizovanoj kontroli promjene stanja



# Obrasci ponašanja - Strategy

## Motivacija za uvođenje projektnog obrasca Strategy

Pretpostavimo da aplikacija za **rad sa kolekcijom podataka** treba da omogući **sortiranje**, pri čemu **korisnik može da bira strategiju za sortiranje** (u zavisnosti od polazne kolekcije, neke strategije daju bolje rezultate, itd.)

Zadatak može da se riješi implementacijom odgovarajućih metoda (za svaku strategiju) u klasi koja reprezentuje kolekciju (pod uslovom da možemo modifikovati polaznu klasu).

### Problemi:

- **ako nemamo mogućnost modifikovanja klase i dodavanje odgovarajućih metoda;**
- **sav kod je u jednoj klasi (i reprezentacija kolekcije i aplikativna manipulacija);**
- **dodavanjem novih strategija klasa raste pa je teže održavanje; ...**

Nije teško zamisliti neke druge domene u kojima se klasa značajno uvećava dodavanjem različitih strategija, npr. rutiranje u aplikacijama za navigaciju

```
public class Kolekcija
{
    // ...
    public void sort1() { // ... }
    public void sort2() { // ... }
    // ...
    public void sortN() { // ... }
    public void sort(int strategija)
    {
        switch (strategija)
        {
            case 1: sort1(); break;
            case 2: sort2(); break;
            // ...
        }
    }
}
```

# Obrasci ponašanja - Strategy

## Motivacija za uvođenje projektnog obrasca Strategy

### Kolekcija

```
- list : ArrayList  
+ add (Object)  
+ sort (strategija : int)  
+ sort1 ()  
+ sort2 ()  
...  
+ sortN ()
```

### Problemi:

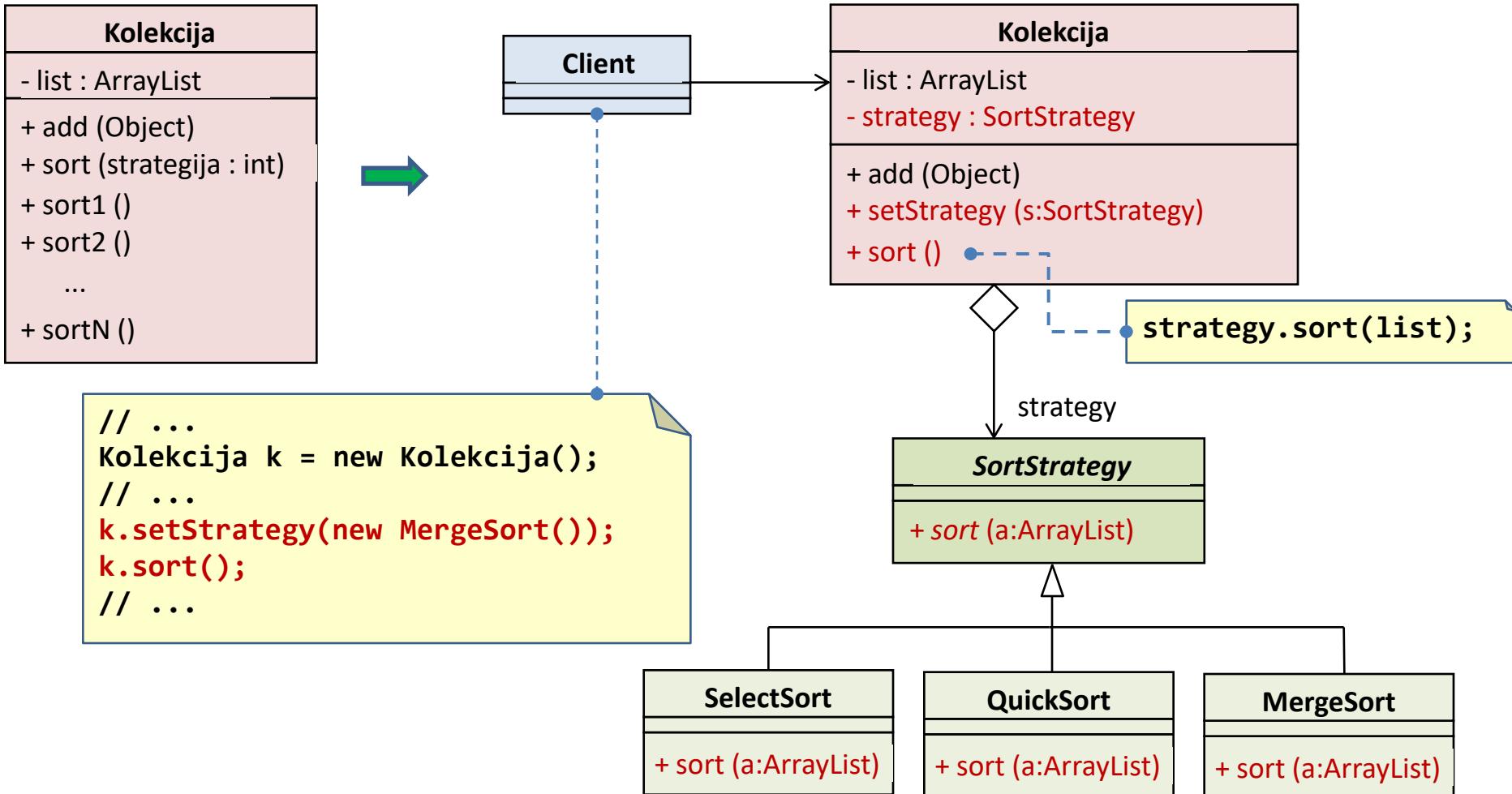
- ako nemamo mogućnost modifikovanja klase i dodavanje odgovarajućih metoda;
- sav kod je u jednoj klasi (i reprezentacija kolekcije i aplikativna manipulacija);
- dodavanjem novih strategija klasa raste pa je teže održavanje; ...

```
public class Kolekcija  
{  
    // ...  
    public void sort1() { // ... }  
    public void sort2() { // ... }  
    // ...  
    public void sortN() { // ... }  
    public void sort(int strategija)  
    {  
        switch (strategija)  
        {  
            case 1: sort1(); break;  
            case 2: sort2(); break;  
            // ...  
        }  
    }  
}
```

Dobro projektno rješenje: izdvajanje aplikativnog koda za svaku strategiju u posebnu klasu

# Obrasci ponašanja - Strategy

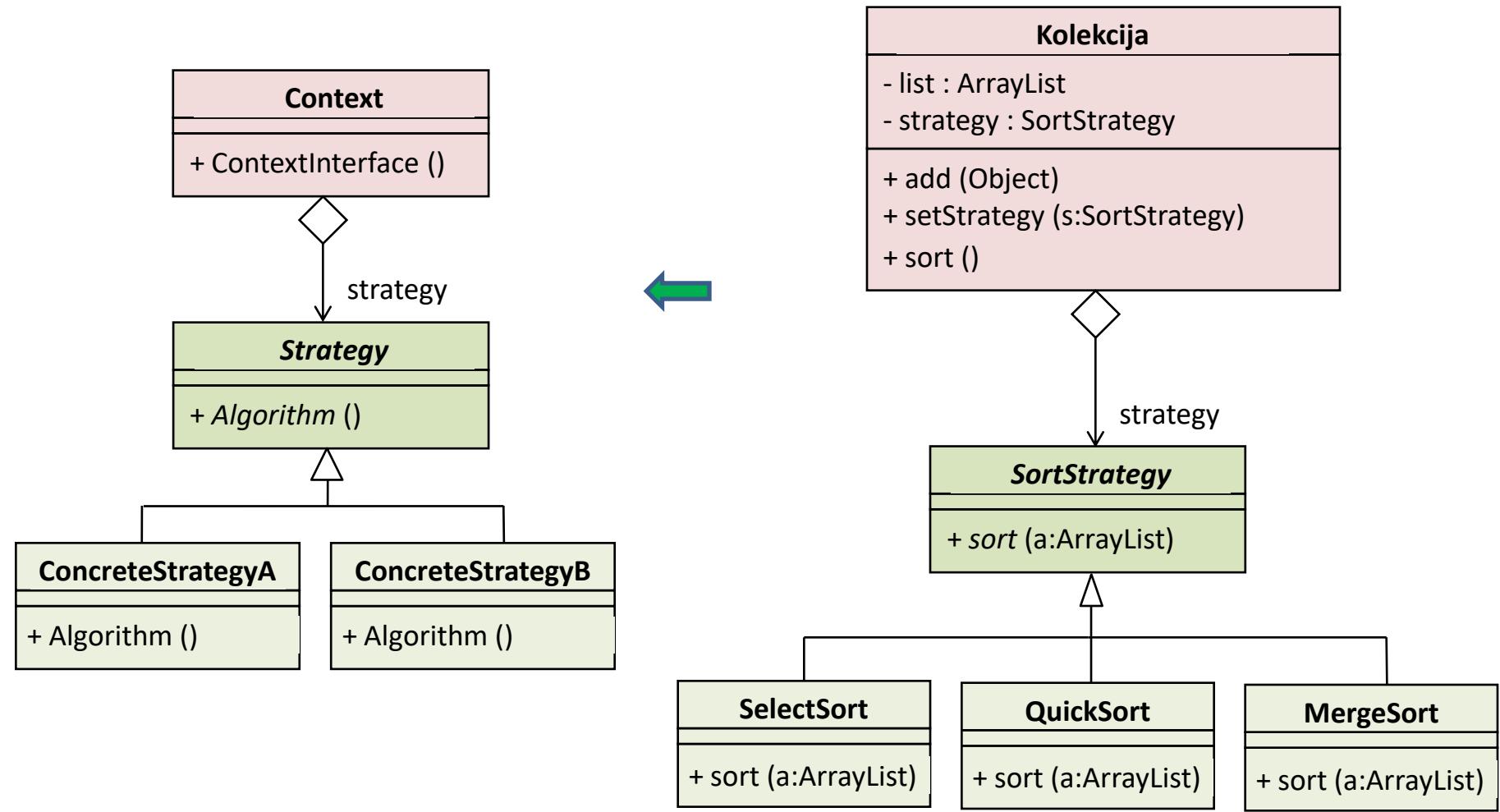
## Motivacija za uvođenje projektnog obrasca Strategy



Dobro projektno rješenje: izdvajanje aplikativnog koda za svaku strategiju u posebnu klasu

# Obrasci ponašanja - Strategy

## Motivacija za uvođenje projektnog obrasca Strategy

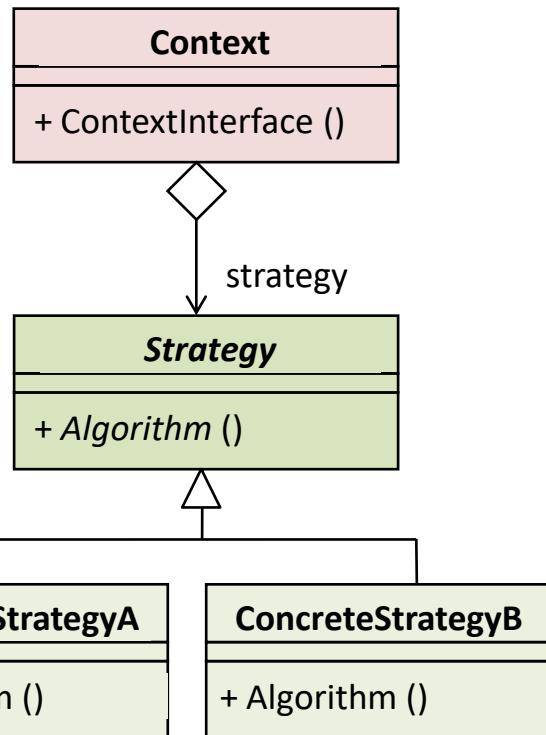


Dobro projektno rješenje: izdvajanje aplikativnog koda za svaku strategiju u posebnu klasi

# Obrasci ponašanja - Strategy

## Strategy (Strategija)

- Definiše familiju algoritama, inkapsulira svaki od njih u odgovarajući objekat, i čini ih međusobno dostupnim. Strategija dozvoljava algoritmu da bude promjenljiv nezavisno od klijenata koji ga koriste.



### Strategy

- deklariše zajednički interfejs za sve podržane algoritme
- objekti klase **Context** koriste ovaj interfejs da pozovu algoritme definisane **ConcreteStrategy** familijom klasa

### ConcreteStrategy

- implementira algoritam koristeći interfejs **Strategy**

### Context

- konfiguriše se pomoću **ConcreteStrategy** objekata
- čuva referencu na objekat tipa **Strategy**
- može da definiše interfejs koji objektu klase **Strategy** dozvoljava pristup podacima

## Kad se koristi STRATEGY obrazac?

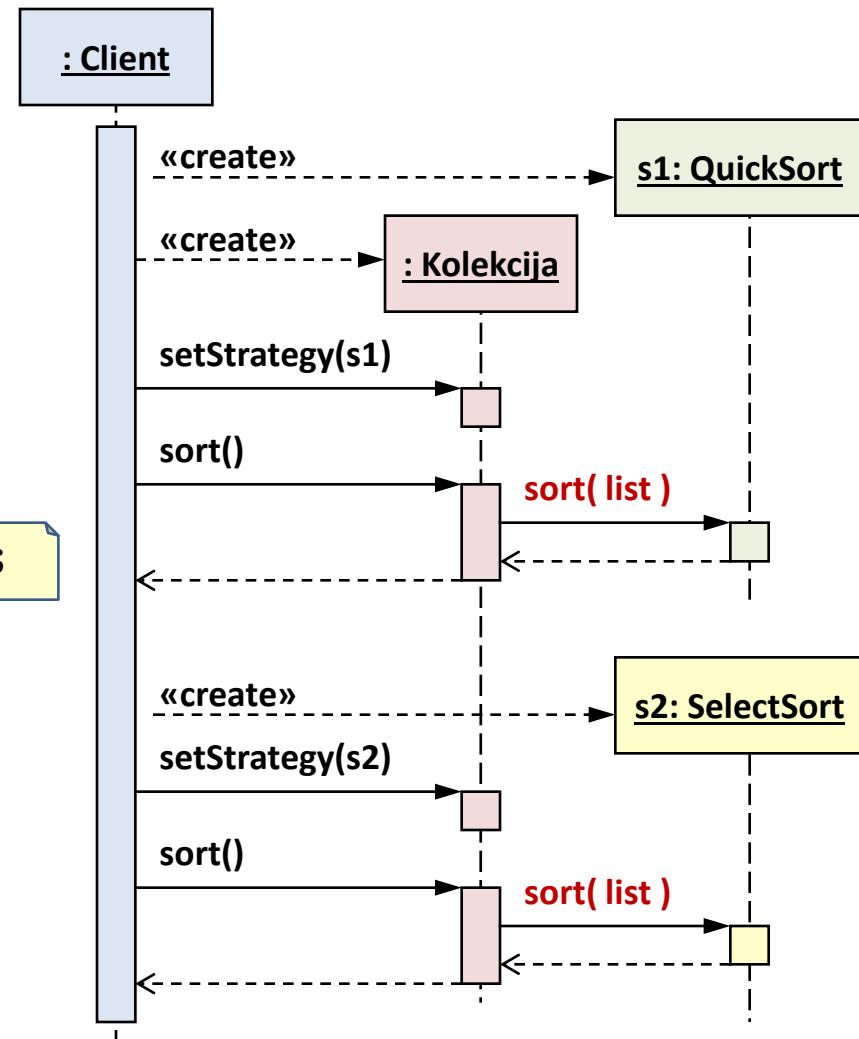
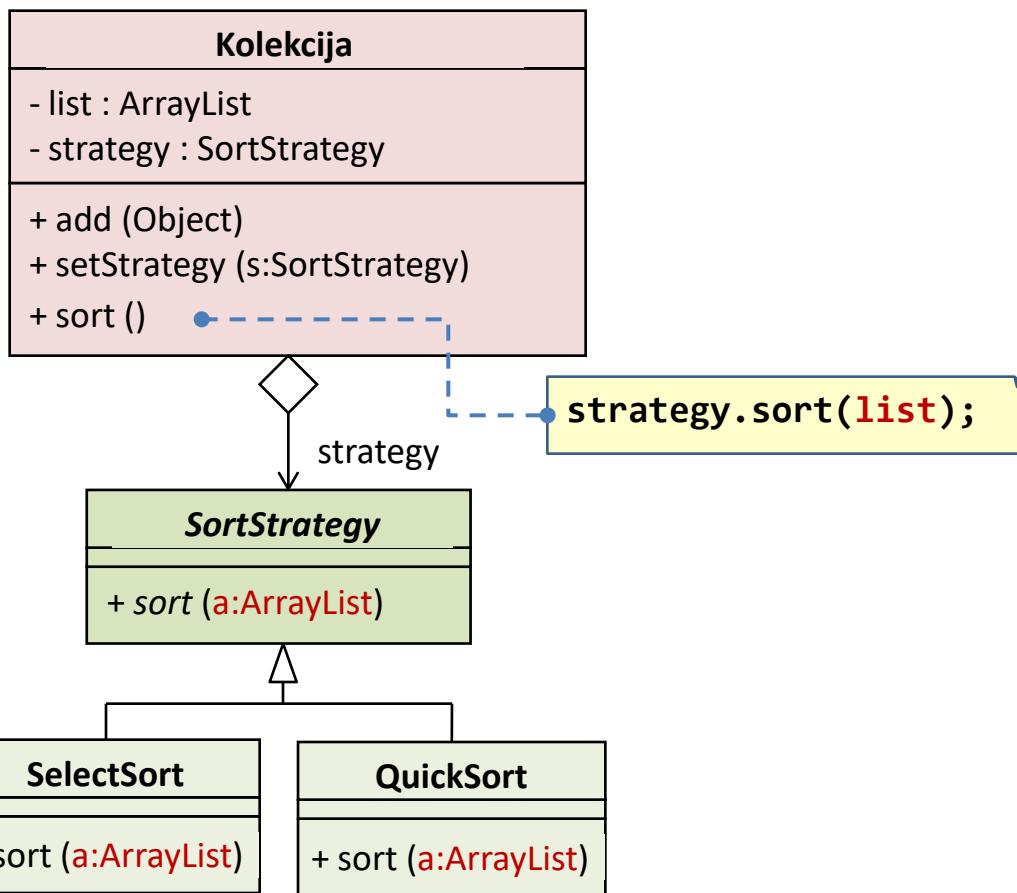
- Kad se više klasa razlikuje samo u ponašanju. Strategija omogućava konfiguriranje objekta jednim od više mogućih ponašanja.
- Kad postoji više varijacija nekog algoritma (s obzirom na to kakve su željene performanse).

# Obrasci ponašanja - Strategy

## Implementacioni detalji – prosljeđivanje podataka u strategiju

- Projektni obrazac Strategy ne propisuje način prosljeđivanja podataka u strategiju

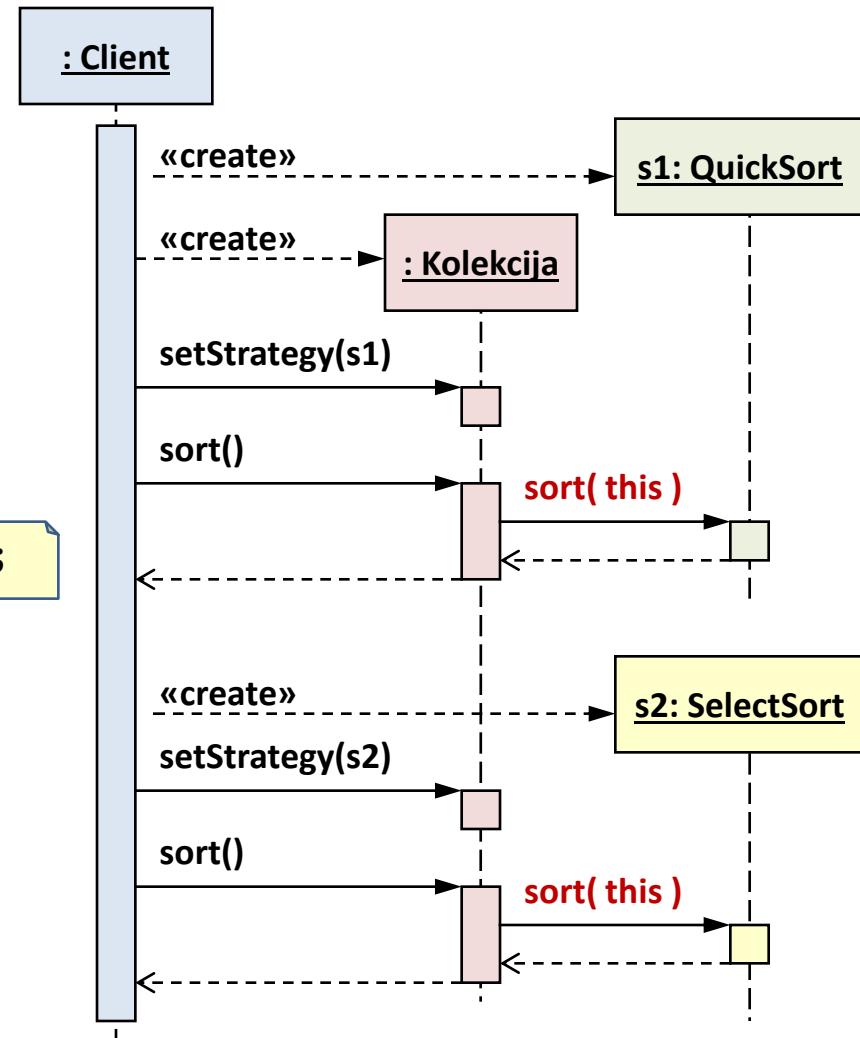
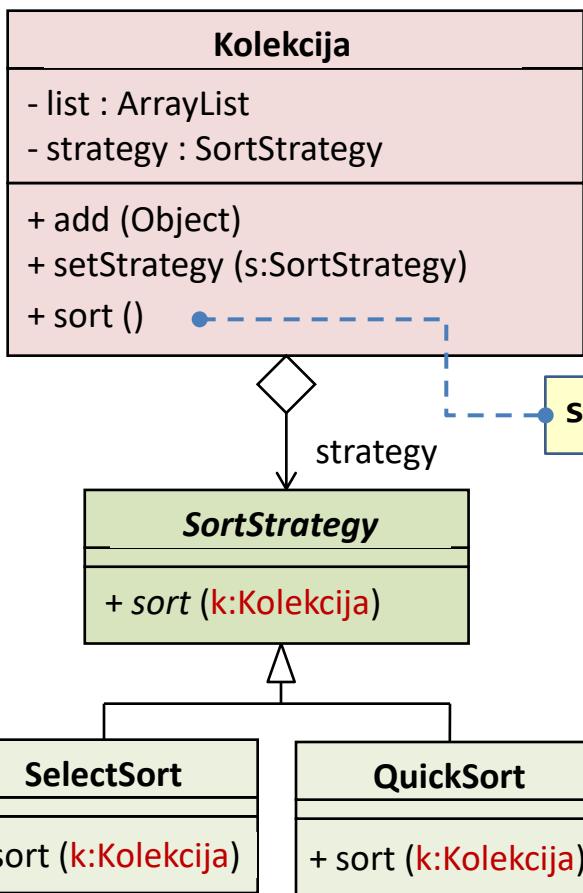
Kontekst može da proslijedi samo potrebne podatke u strategiju kao argument metode



# Obrasci ponašanja - Strategy

## Implementacioni detalji – prosljeđivanje podataka u strategiju

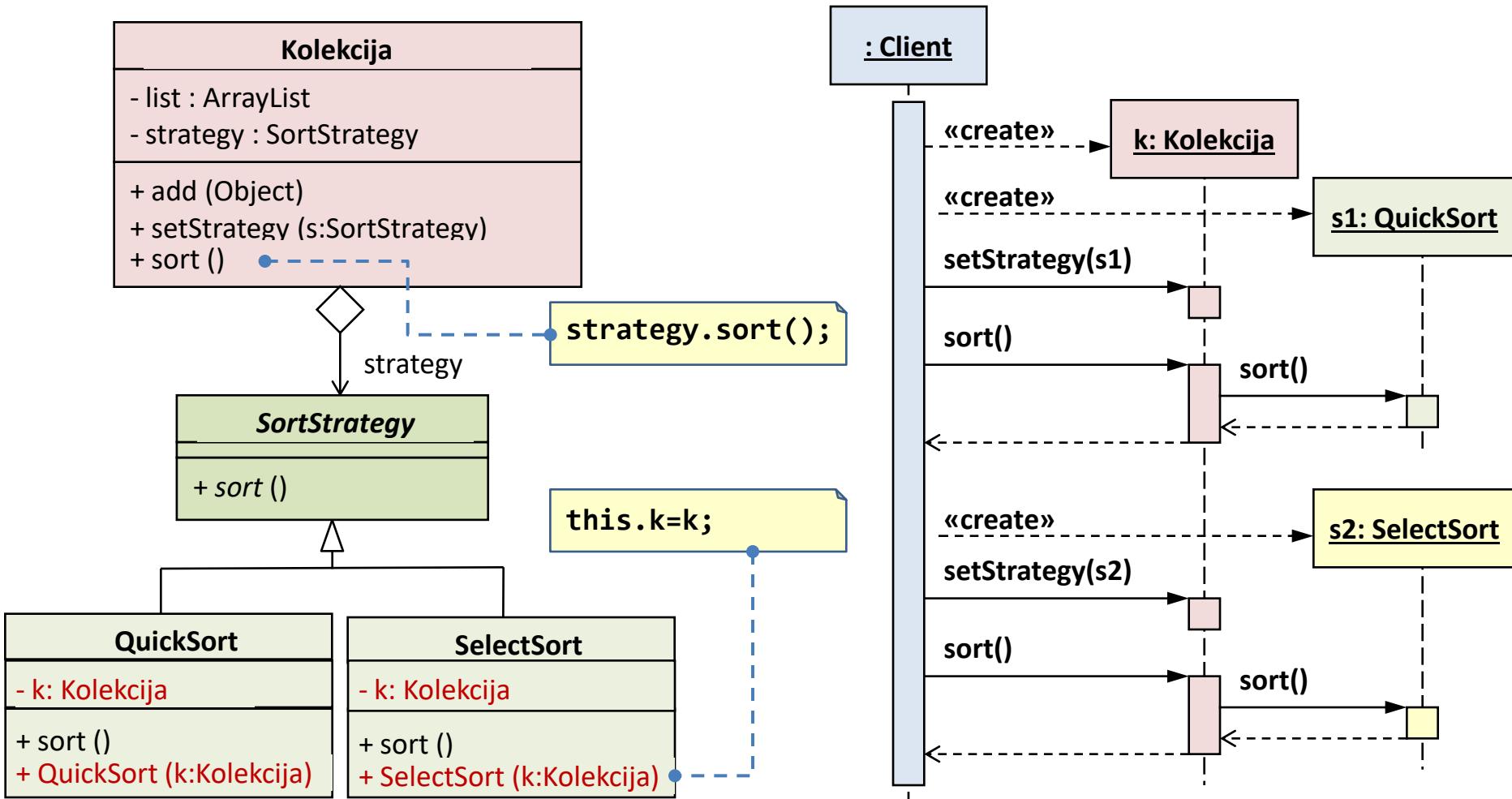
Kontekst može da proslijedi sebe (this) u strategiju kao argument metode



# Obrasci ponašanja - Strategy

## Implementacioni detalji – prosljeđivanje podataka u strategiju

Strategija može da ima referencu na kontekst – eliminiše se potreba za prenosom podataka u strategiju, ali se povećava sprega između konteksta i strategije



# Obrasci ponašanja - Opserver

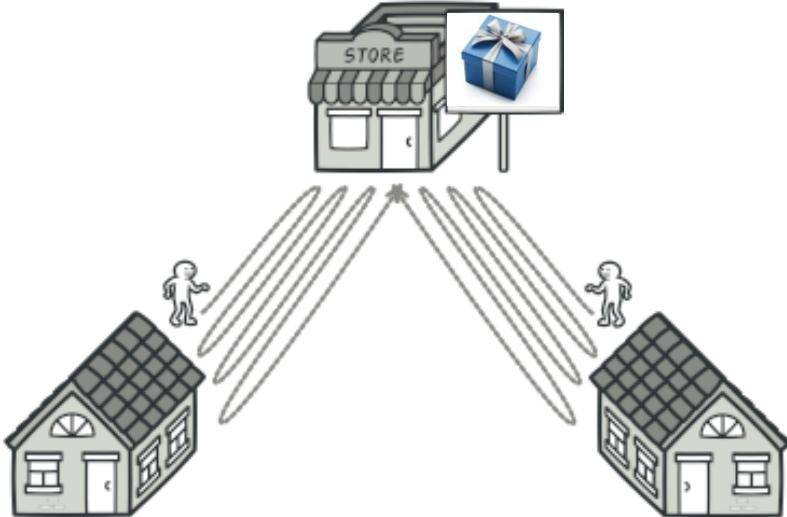
## Motivacija za uvođenje projektnog obrasca Opserver

Pretpostavimo da imamo objekte (posmatrači) koji su zainteresovani za stanje nekog subjekta (npr. kupci zainteresovani za kupovinu proizvoda)



Kako posmatrači mogu da dobiju informaciju o promjeni stanja subjekta:

- 1) posmatrači redovno provjeravaju da li se subjekat promijenio (beskonačna petlja)
- 2) subjekat obavještava sve zainteresovane posmatrače da je došlo do promjene (poruka se šalje kad dođe do promjene)



Problematično projektno rješenje

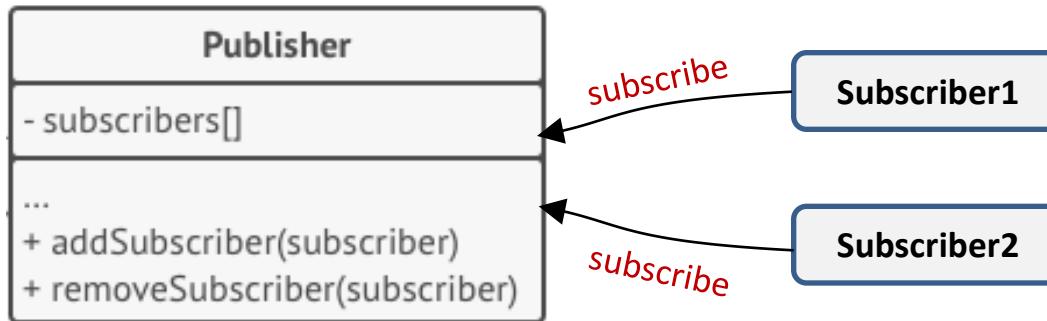


DOBRO PROJEKTNO RJEŠENJE

# Obrasci ponašanja - Opserver

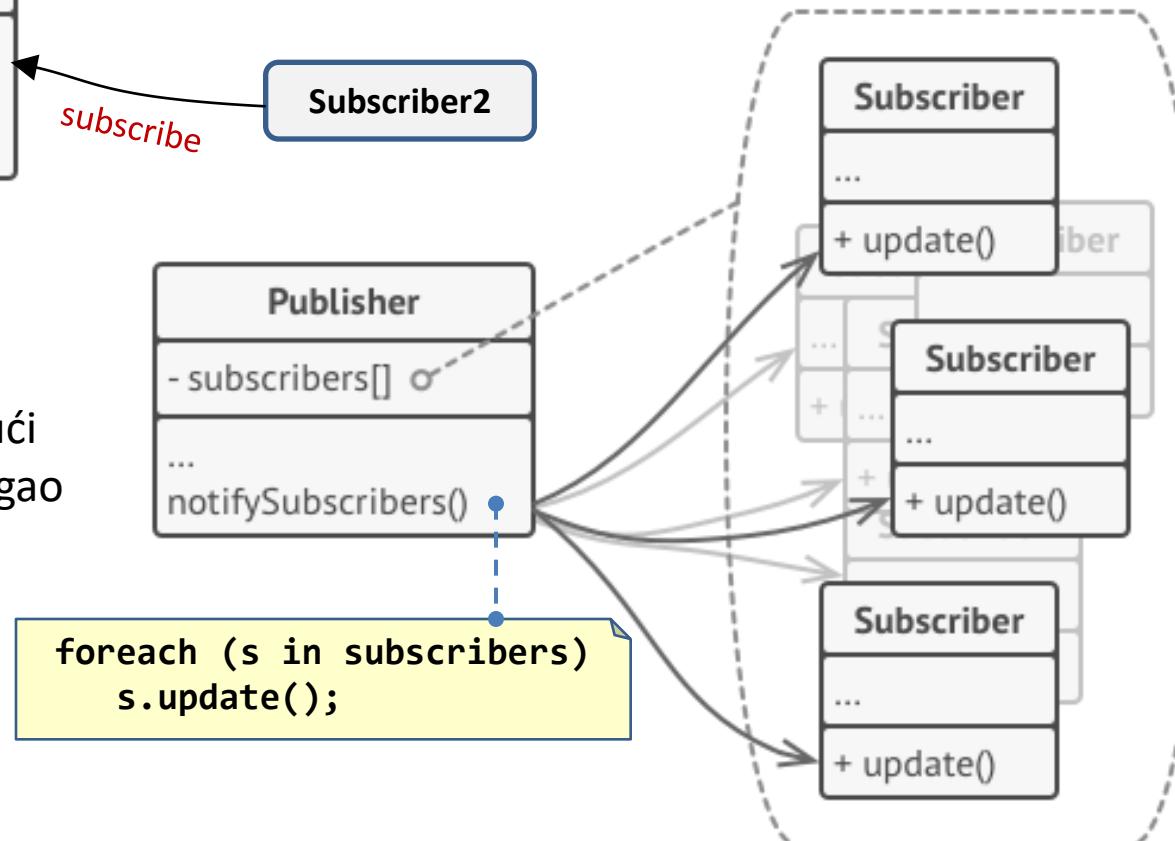
## Motivacija za uvođenje projektnog obrasca Opserver

- 1 Da bi posmatrači mogli da dobiju informaciju o promjeni stanja subjekta prvo moraju da se registruju



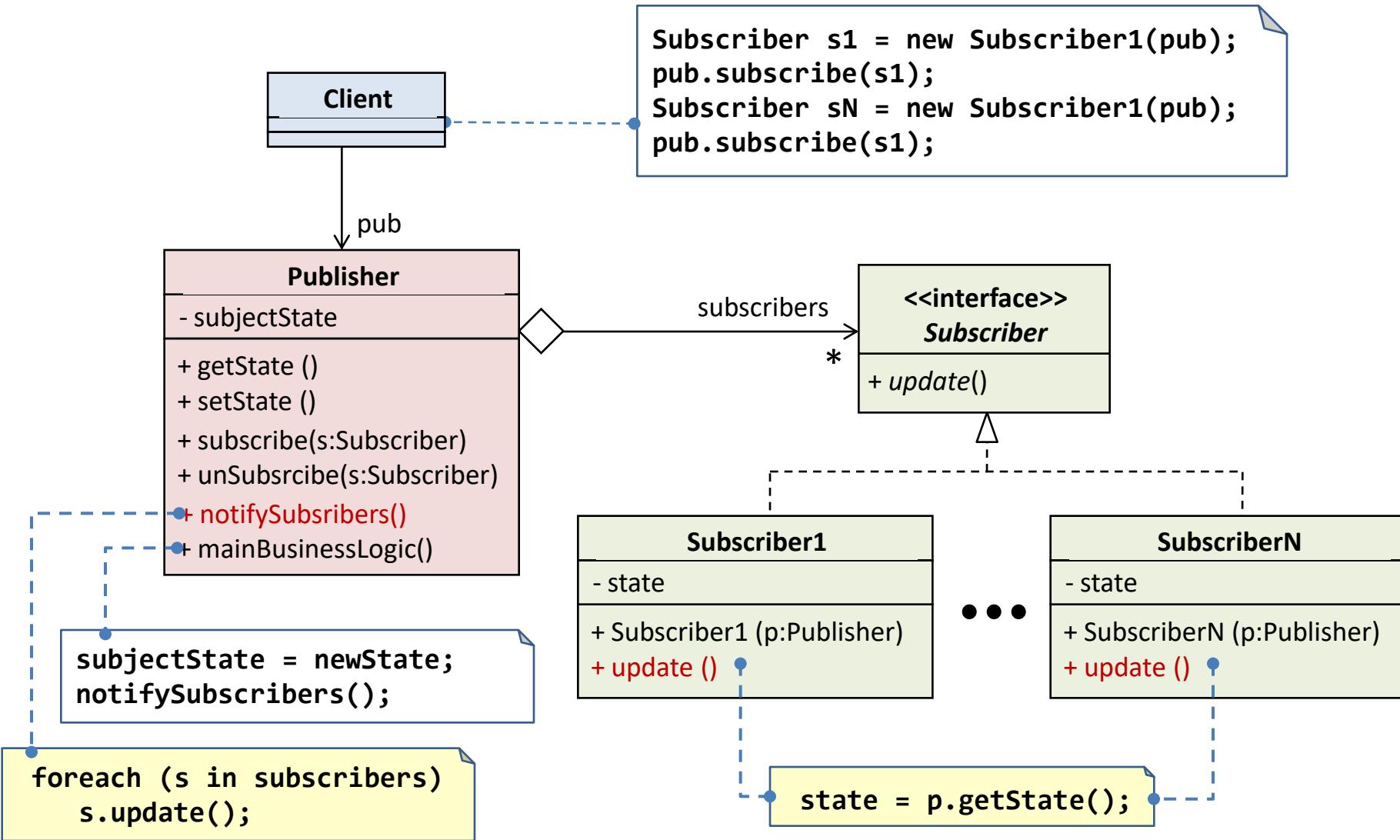
Subjekat održava registar prijavljenih posmatrača:  
**niz referenci + jednostavan interfejs za prijavljivanje/odjavljivanje**

- 2 Posmatrači moraju da implementiraju odgovarajući interfejs da bi subjekat mogao da ih obavijesti o promjeni



# Obrasci ponašanja - Opserver

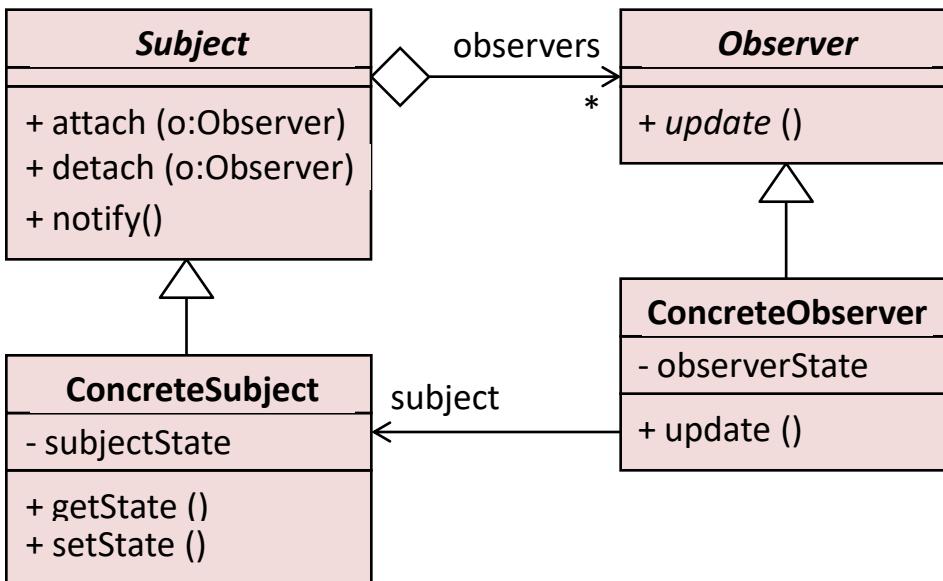
## Motivacija za uvođenje projektnog obrasca Opserver



# Obrasci ponašanja - Opserver

**Observer**  
(Nadzornik, Posmatrač)

- Definiše zavisnosti tipa jedan:više među različitim objektima i obezbjeđuje da se promjena stanja u jednom objektu automatski reflektuje u svim zavisnim objektima.



## Subject

- čuva reference prema opserverima
- obezbjeđuje interfejs za dodavanje i uklanjanje opservera

## ConcreteSubject

- čuva stanje od interesa za konkretnе opservere
- šalje notifikaciju opserverima kad promijeni stanje

## Observer

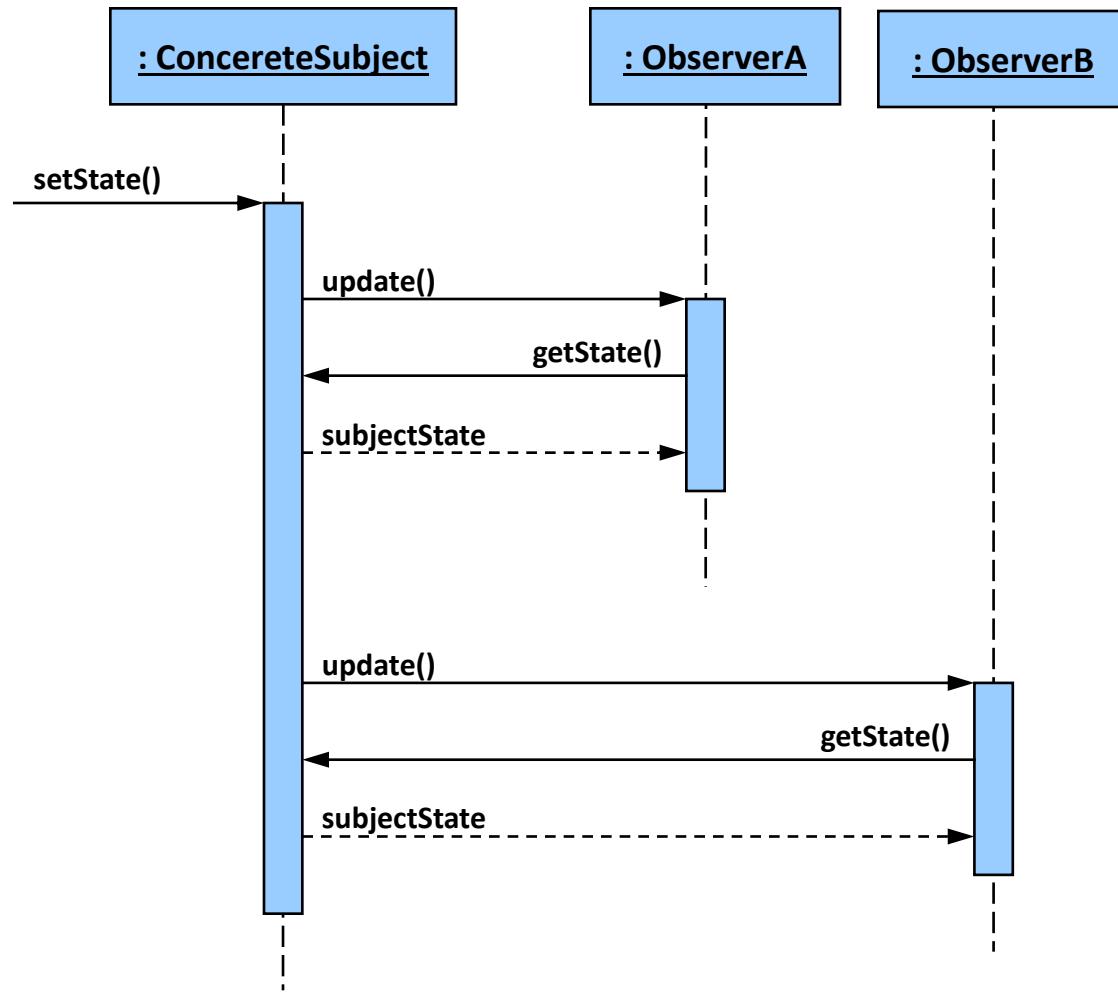
- definiše interfejs za ažuriranje opservera kad se subject promijeni

## ConcreteObserver

- čuva referencu na **ConcreteSubject** objekte
- čuva stanje koje treba da je konzistentno sa stanjem konkretnog subjekta
- implementira interfejs za ažuriranje objekata koji je definisan u klasi **Observer**

# Obrasci ponašanja - Opserver

## Tipična kolaboracija



- Konkretan subjekat na svaku promjenu stanja šalje notifikaciju konkretnim opserverima
- Notifikacija podrazumijeva da se svakom opserveru pošalje informacija da je stanje subjekta promijenjeno.
- Notifikacija može da uključi informaciju o novom stanju (u slučaju jednostavnih subjekata) – tada nema potrebe da opserveri očitavaju stanje
- U slučaju složenih subjekata, opserveri nakon notifikacije očitavaju stanje subjekta i ažuriraju svoje stanje na osnovu stanja subjekta

# Obrasci ponašanja - Opserver

## Implementacioni detalji

### – Mapiranje subjekat → opserveri

- Najjednostavniji način:
  - čuvanje referenci u subjektu prema svim njegovim opserverima
  - „skupo” rješenje u slučaju velikog broja subjekata i malog broja opservera
  - bolje rješenje: asocijativna *look-up* tabela (*hash table*) sa parovima `<subjekat, opserver>`

### – Jeden opserver za više subjekata

- Treba da postoji mehanizam na osnovu kojeg opserver zna koji subjekat je promijenio stanje (npr. subjekat prosljeđuje sam sebe):

### – Trigerovanje usklađivanja stanja opservera sa stanjem subjekta

#### – operacije setovanja stanja subjekta trigeruju `notify()`:

- na svaku promjenu stanja subjekta poziva se `notify()`, pa opserveri ažuriraju stanje
- „skupo” rješenje u slučaju velikog broja promjena stanja subjekta, jer promjene stanja subjekta mogu biti „prebrze” za opservere

#### – klijent odgovoran za trigerovanje:

- klijent bira trenutak trigerovanja `notify()`
- dobro rješenje u slučaju velikog broja promjena stanja subjekta, jer se opserveri ažuriraju tek nakon što se završi prelazni proces
- loša strana: odgovornost je na klijentu

# Obrasci ponašanja - Opserver

## Implementacioni detalji

### Protokol ažuriranja stanja opservera

#### – „push“ model:

- subjekat šalje sve informacije opserveru o promjeni stanja
- subjekat „zna“ šta treba opserveru

#### – „pull“ model:

- subjekat samo obavještava operver da je došlo do promjene stanja, a opserver sve potrebne informacije o stanju traži od subjekta
- može biti neefikasno ako opserver treba da provjeri mnogo atributa koji reprezentuju stanje subjekta

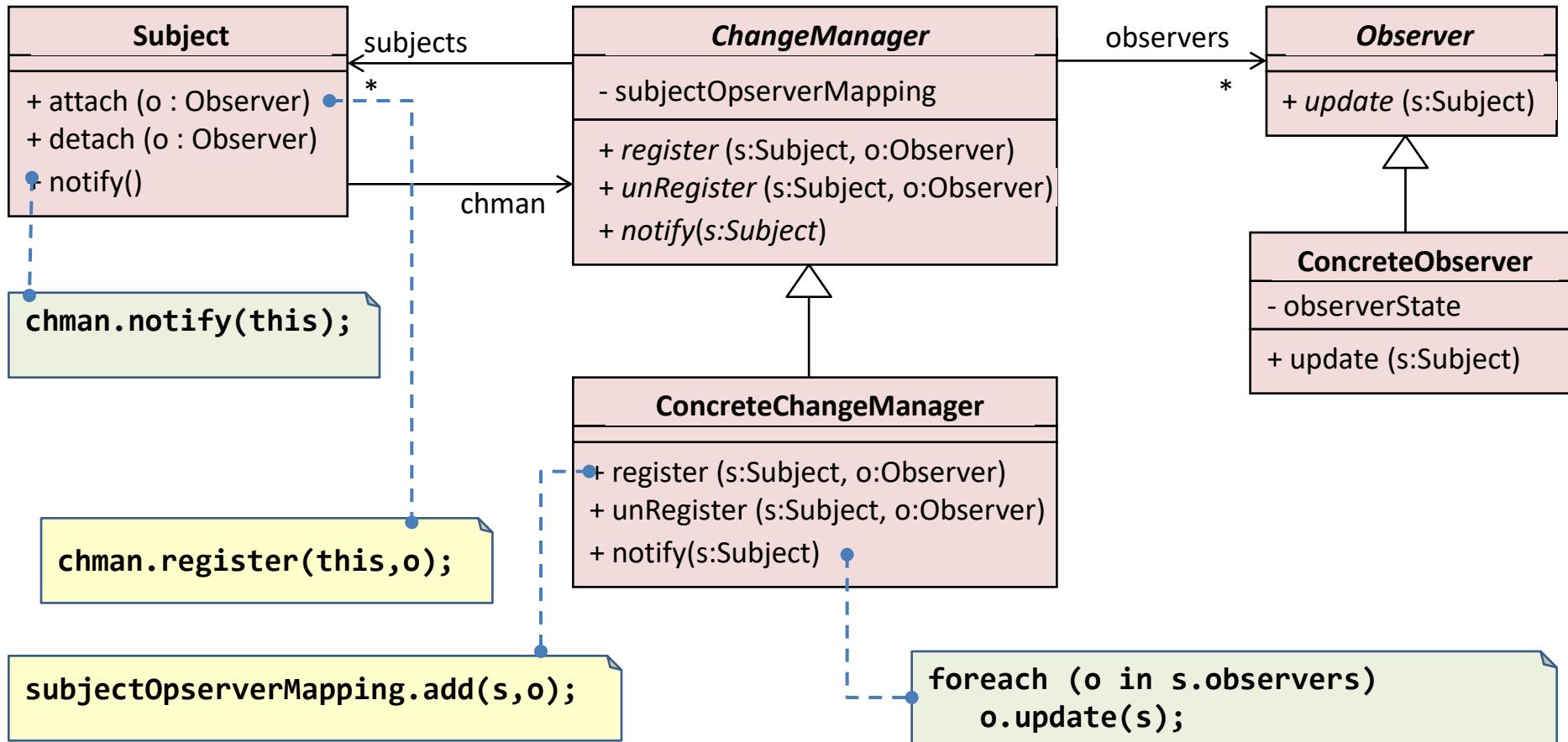
#### – hibridni model: između push i pull

### ChangeManager

- U slučaju kompleksnih veza subjekti $\leftrightarrow$ opserveri
- ChangeManager je poseban objekat koji upravlja optimalnim ažuriranjem opservera
- Uloge:
  - mapiranje subjekat $\rightarrow$ opserveri, čime se subjekat rasterećuje od referenci prema opserverima
  - definiše i implementira strategiju ažuriranja opservera
  - obavještava sve potrebne opservere na zahtjev subjekta

# Obrasci ponašanja - Opserver

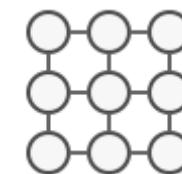
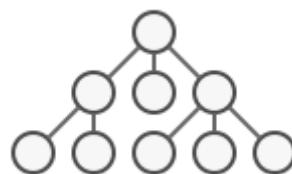
Primjer (primjena ChangeManagera):



# Obrasci ponašanja - Iterator

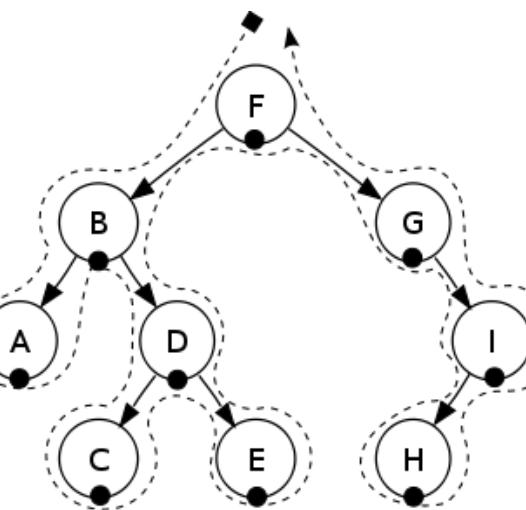
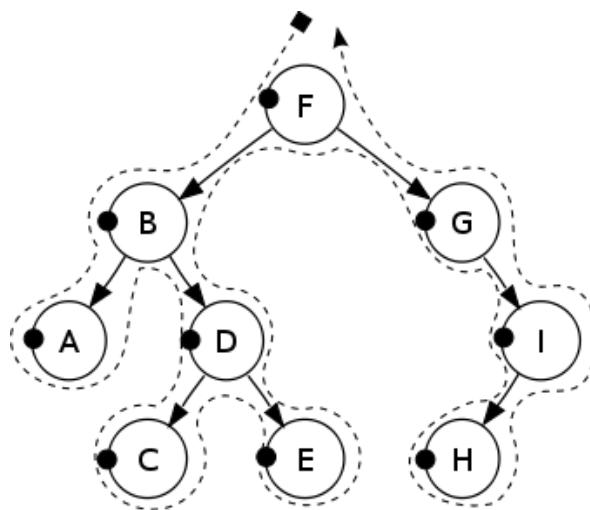
## Motivacija za uvođenje projektnog obrasca Iterator

Aplikacije često manipulišu različitim kolekcijama podataka (liste, stabla, grafovi, ...)



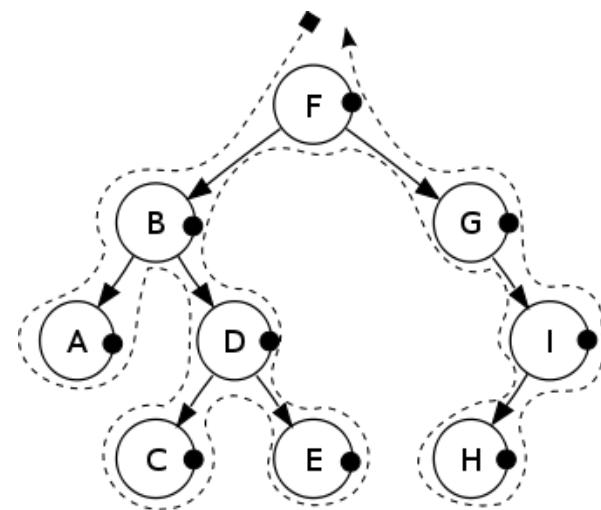
Često su potrebni različiti načini obilaska kolekcije

(npr. za binarno stablo – obilazak po dubini: pre-order, in-order, post-order)



Pre-order: F, B, A, D, C, E, G, I, H

In-order: A, B, C, D, E, F, G, H, I



Post-order: A, C, E, D, B, H, I, G, F

**DOBRO PROJEKTNO RJEŠENJE**

Izdvajanje aplikativne logike za obilazak kolekcije iz klase koja reprezentuje kolekciju u posebnu klasu

# Obrasci ponašanja - Iterator

## Motivacija za uvođenje projektnog obrasca Iterator

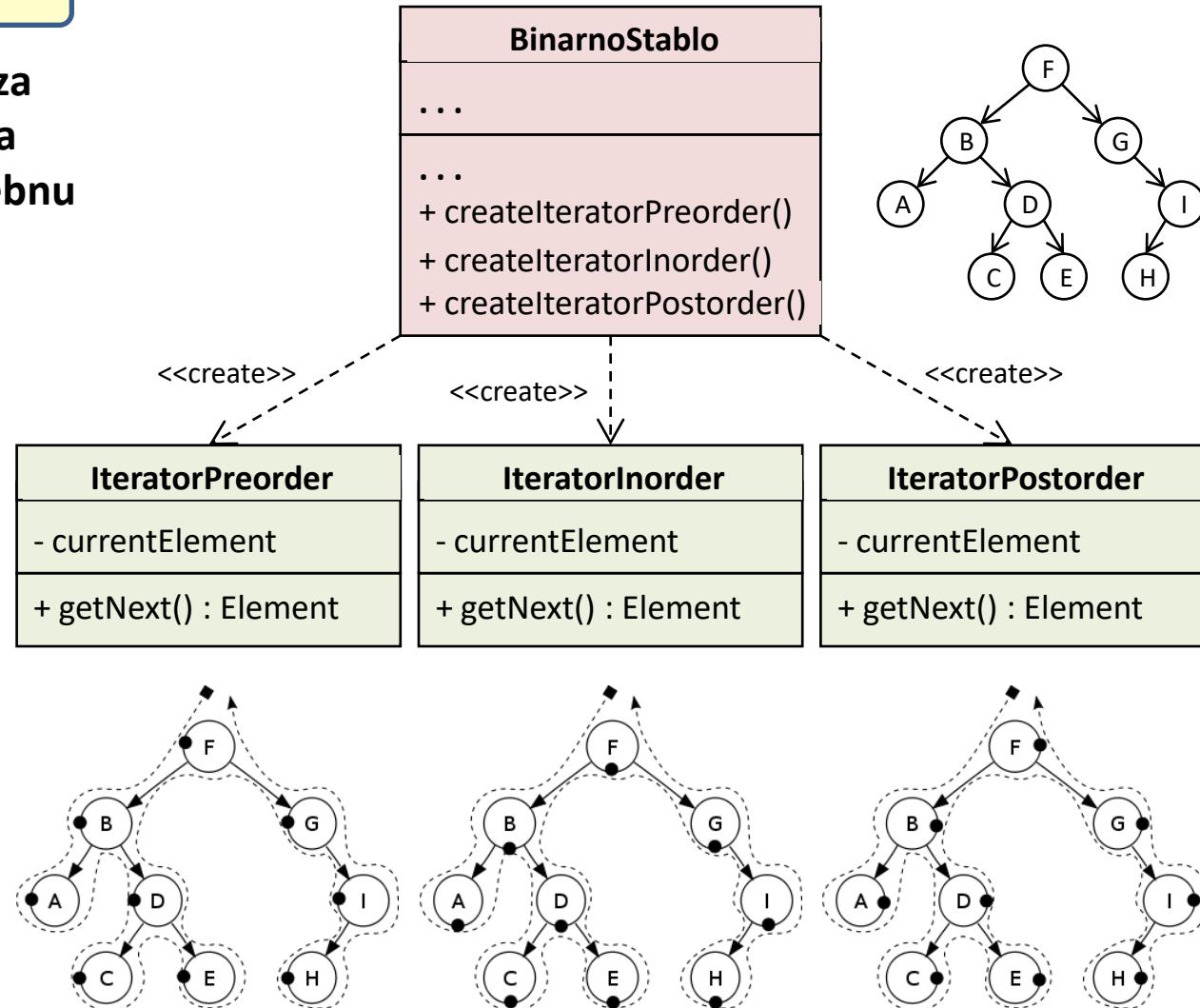
### DOBRO PROJEKTNO RJEŠENJE

Izdvajanje aplikativne logike za obilazak kolekcije iz klase koja reprezentuje kolekciju u posebnu klasu koja se naziva **Iterator**

Klasa koja reprezentuje kolekciju (BinarnoStablo) zadužena je samo za reprezentaciju kolekcije

Iterator implementira svu logiku za obilazak kolekcije

Za istu kolekciju istovremeno možemo da imamo veći broj različitih iteratora za istu ili različite strategije obilaska



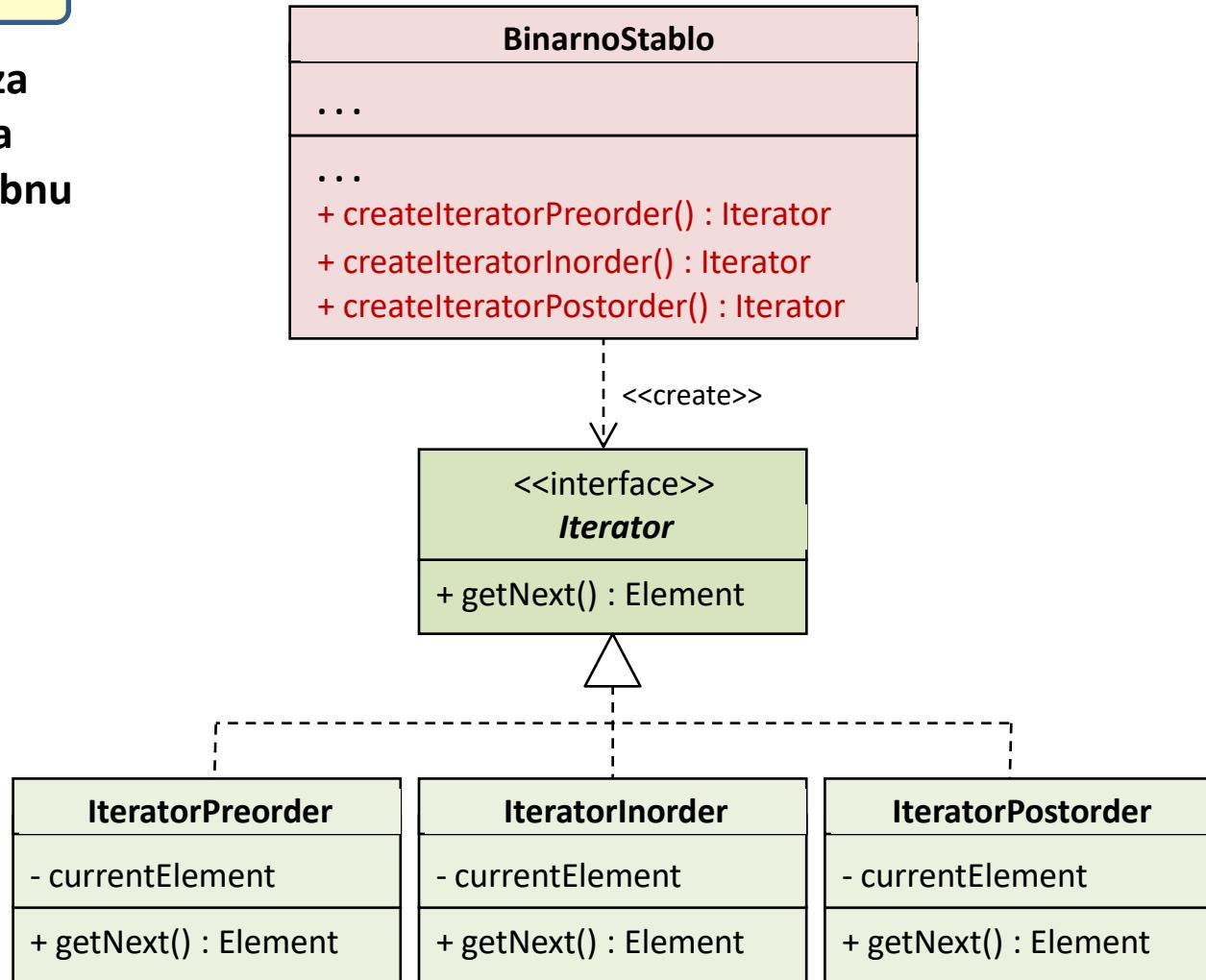
# Obrasci ponašanja - Iterator

## Motivacija za uvođenje projektnog obrasca Iterator

### DOBRO PROJEKTNO RJEŠENJE

Izdvajanje aplikativne logike za obilazak kolekcije iz klase koja reprezentuje kolekciju u posebnu klasu koja se naziva **Iterator**

Svi iteratori treba da implementiraju isti interfejs



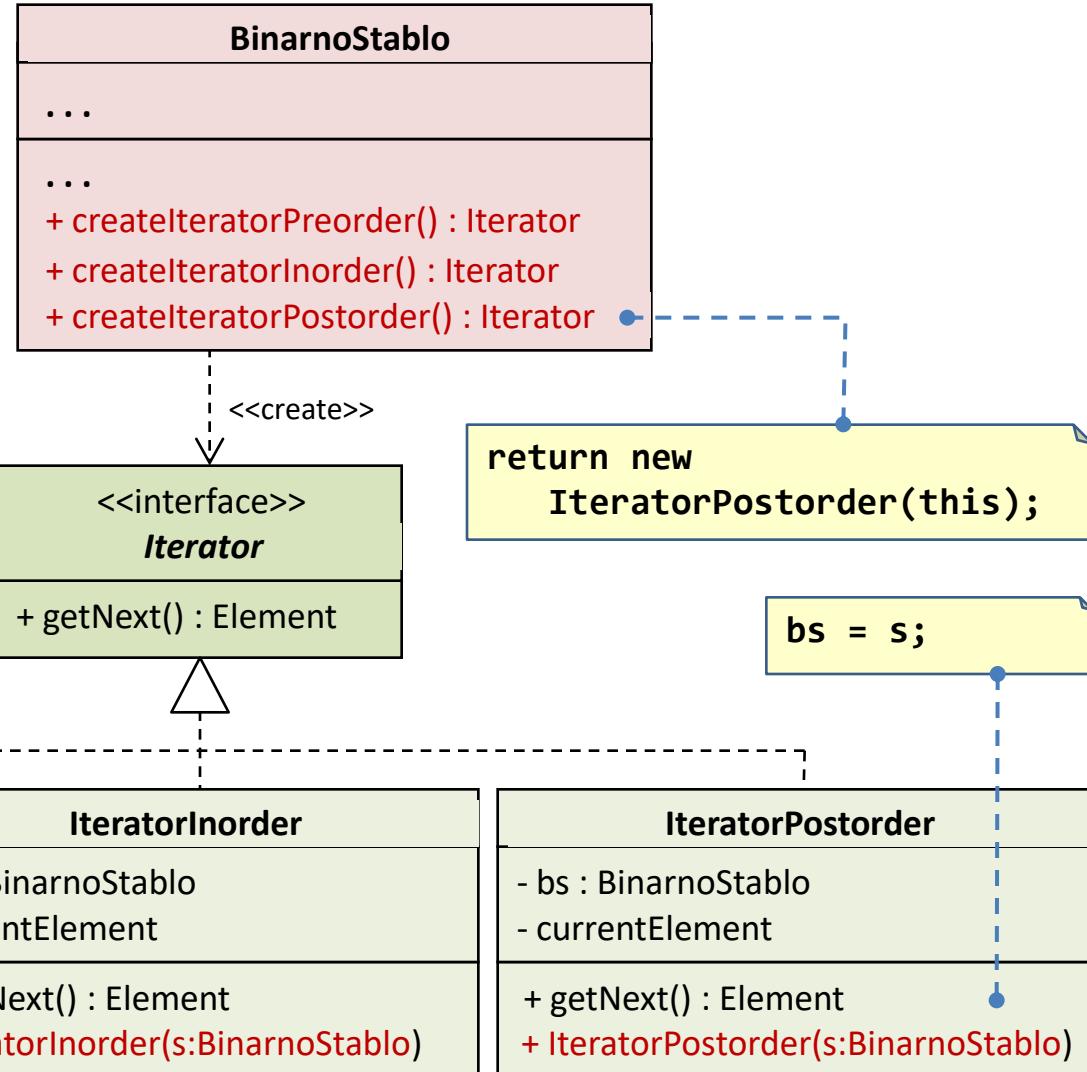
# Obrasci ponašanja - Iterator

## Motivacija za uvođenje projektnog obrasca Iterator

### DOBRO PROJEKTNO RJEŠENJE

Kolekcija implementira fabričke metode koje kreiraju i vraćaju odgovarajući iterator

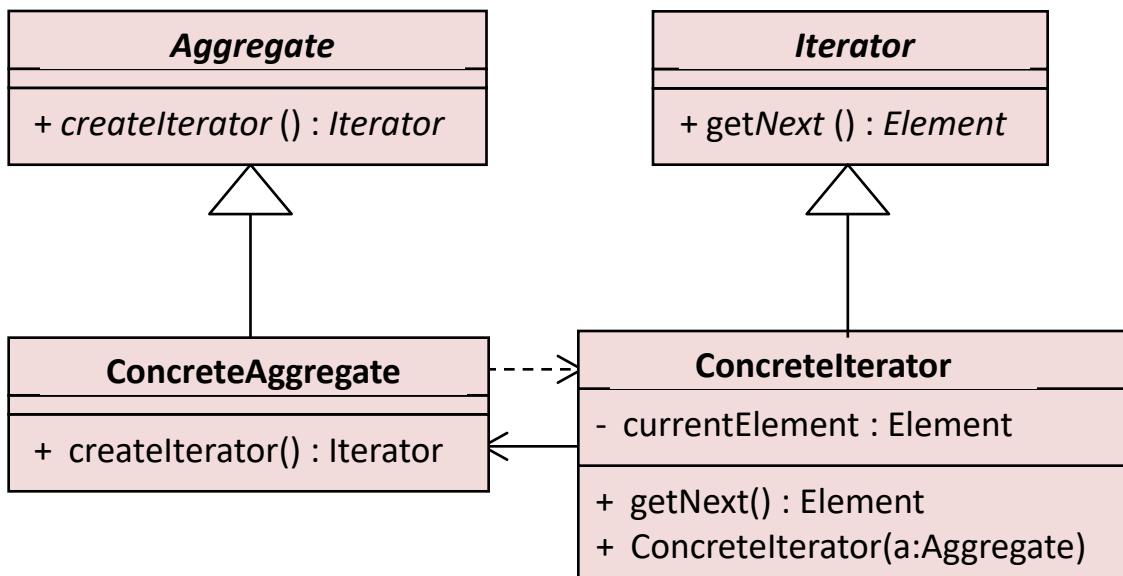
Prilikom instaciranja iterator se inicijalizuje odgovarajućom kolekcijom



# Obrasci ponašanja - Iterator

## Iterator (Brojač)

- Obezbeđuje sekvenčijalni pristup elementima objekta nastalog agregacijom bez potrebe za pristupanjem njegovoj stvarnoj reprezentaciji.



## Iterator

- Deklarije interfejs za pristupanje elementima i kretanje od jednog do drugog

## Concreteliterator

- Implementira interfejs Iterator
- Vodi računa o poziciji tekućeg elementa u agregaciji (kolekciji)

## Aggregate

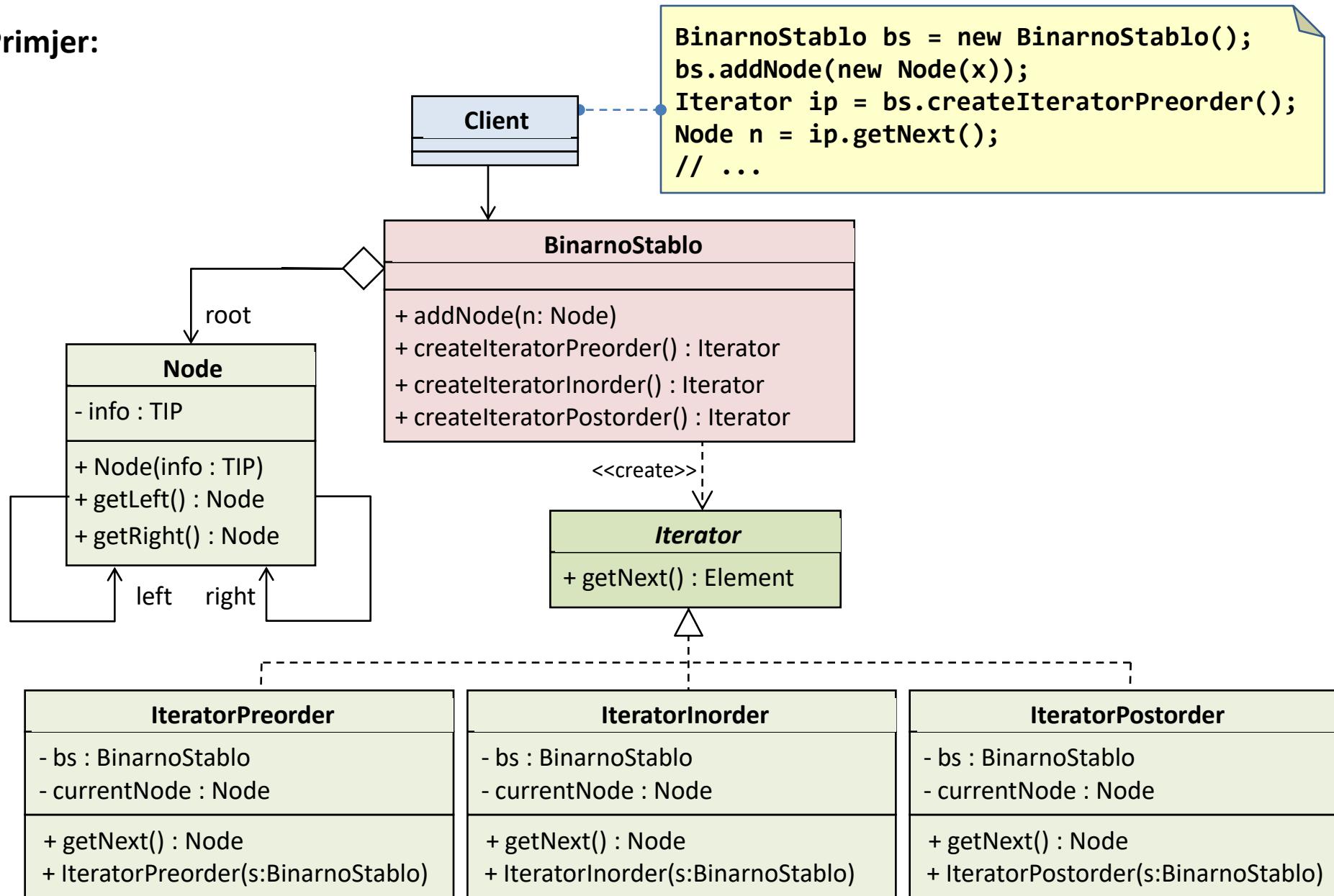
- Deklarije interfejs za kreiranje objekata tipa Iterator

## ConcreteAggregate

- Implementira interfejs za kreiranje Iterator objekata i vraća reference na odgovarajuće Concreteliterator objekte

# Obrasci ponašanja - Iterator

Primjer:



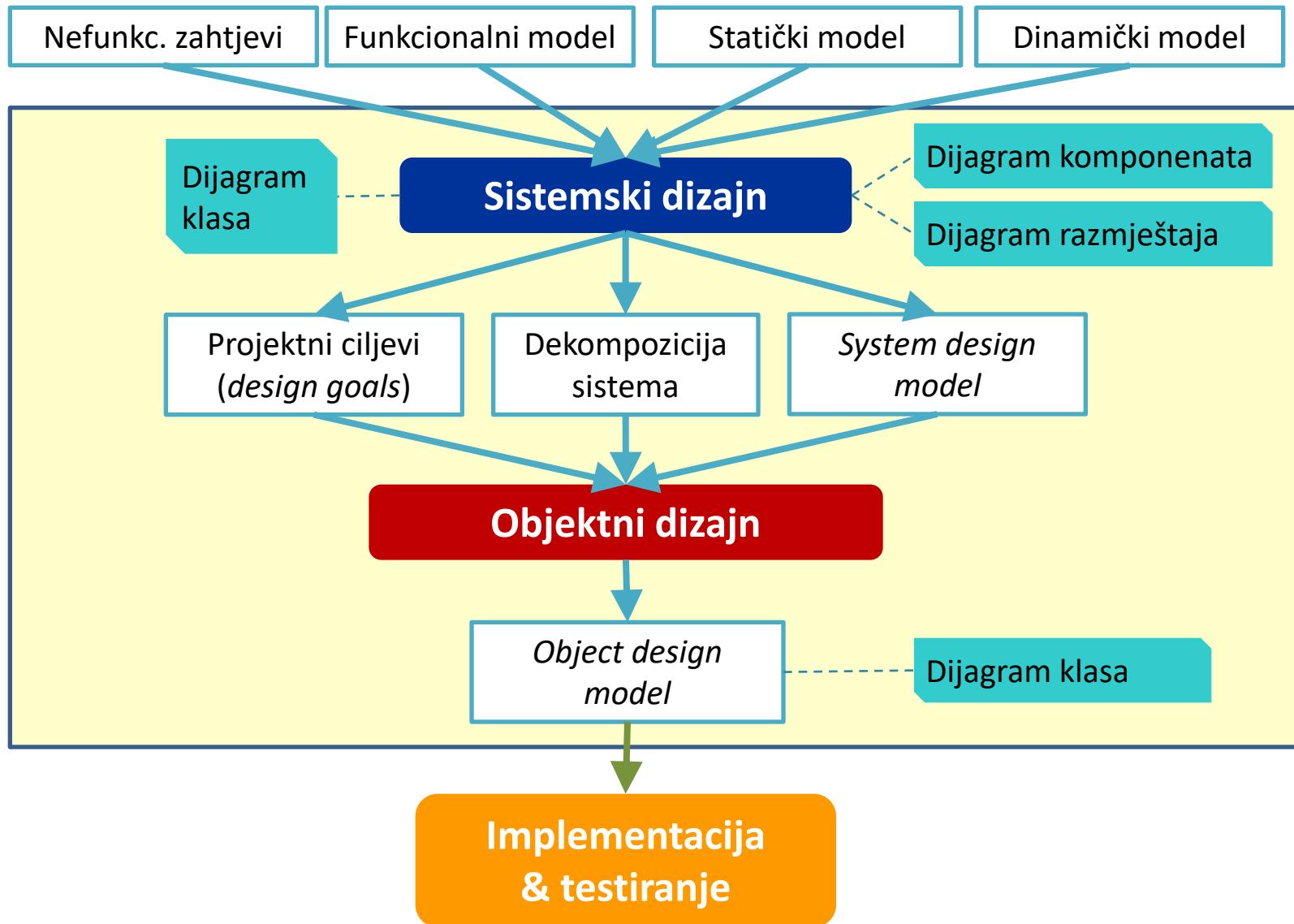
**UNIVERZITET U BANJOJ LUCI  
ELEKTROTEHNIČKI FAKULTET**

**Prof. dr Dražen Brđanin**

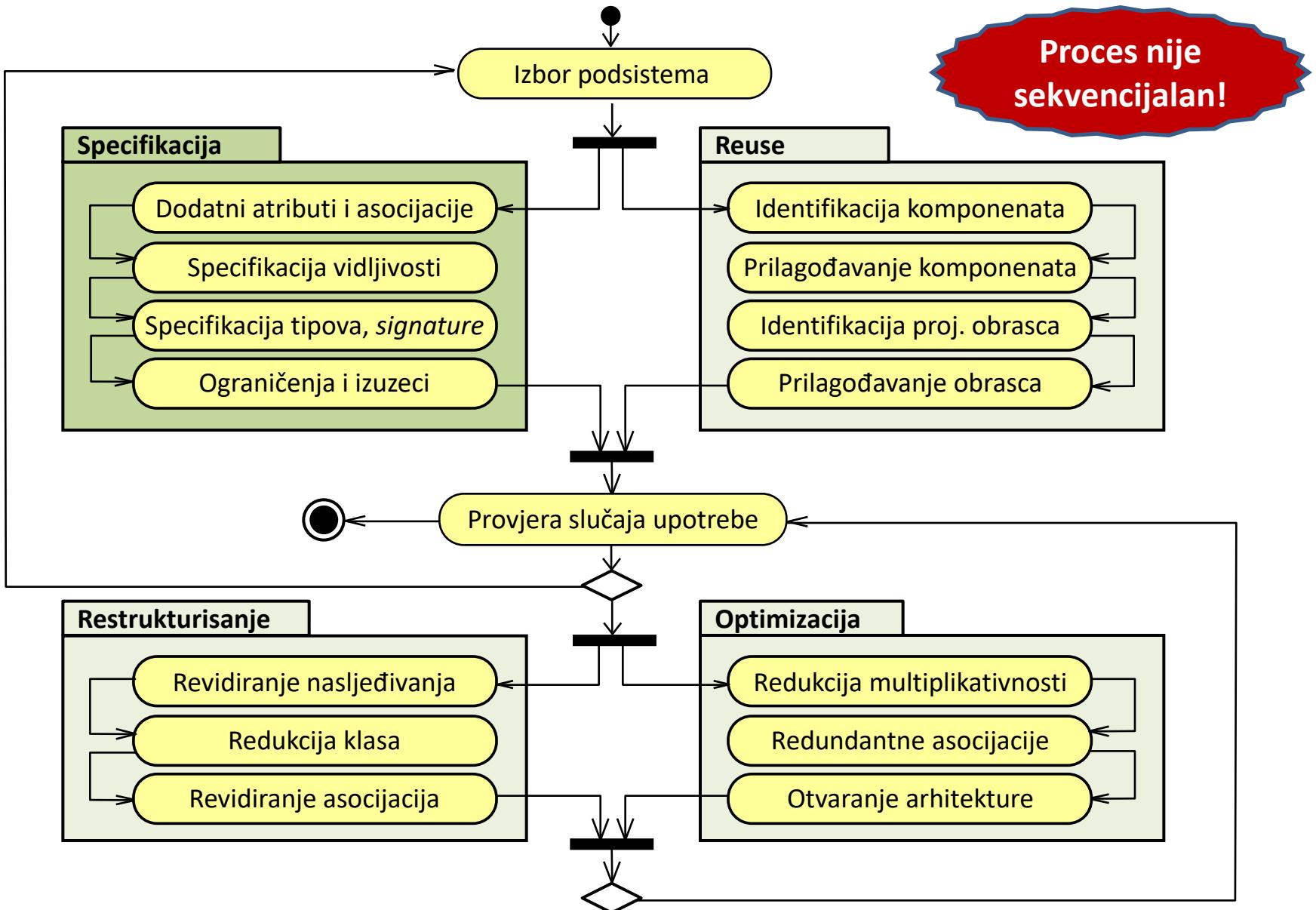
**OBJEKTNOPROGRAMSKI DIZAJN  
/specifikacija interfejsa/**

**Banja Luka  
2024.**

# Objektno-orientisano projektovanje



# Aktivnosti u objektnom dizajnu



# Specifikacija interfejsa

## Specifikacija interfejsa

- Precizne specifikacije interfejsa za svaku komponentu (**component API**) i za svaku klasu
- **Osnovni ciljevi:**
  - smanjenje stepena sprege između podsistema/klasa,
  - specifikacija što jednostavnijeg interfejsa koji je razumljiv za svakog pojedinačnog programera

## OOA:

- Identifikacija atributa i operacija bez specifikacije tipova i argumenata

## OOD:

- Specifikacija tipova i argumenata (*signature*)
- Specifikacija vidljivosti (*public, protected, private*)
- Specifikacija ograničenja (*contracts*)
  - uslovi koji moraju da budu ispunjeni:
    - za svaku instancu klase,
    - prije pozivanja metode,
    - poslije izvršavanja metode
  - specifikacija rezultata koji metoda vraća kao rezultat izvršavanja

# Specifikacija tipova/argumenata (*signature*)

OOA:

Identifikacija atributa i operacija bez specifikacije tipova i argumenata

|               |
|---------------|
| Hashtable     |
| numOfElements |
| put()         |
| get()         |
| remove()      |
| containsKey() |
| size()        |



Inicijalna  
specifikacija

OOD:

Specifikacija tipova i argumenata

|                                 |
|---------------------------------|
| Hashtable                       |
| numOfElements:int               |
| put(key:String,entry:String)    |
| get(key:String):String          |
| remove(key:String)              |
| containsKey(key:String):boolean |
| size():int                      |



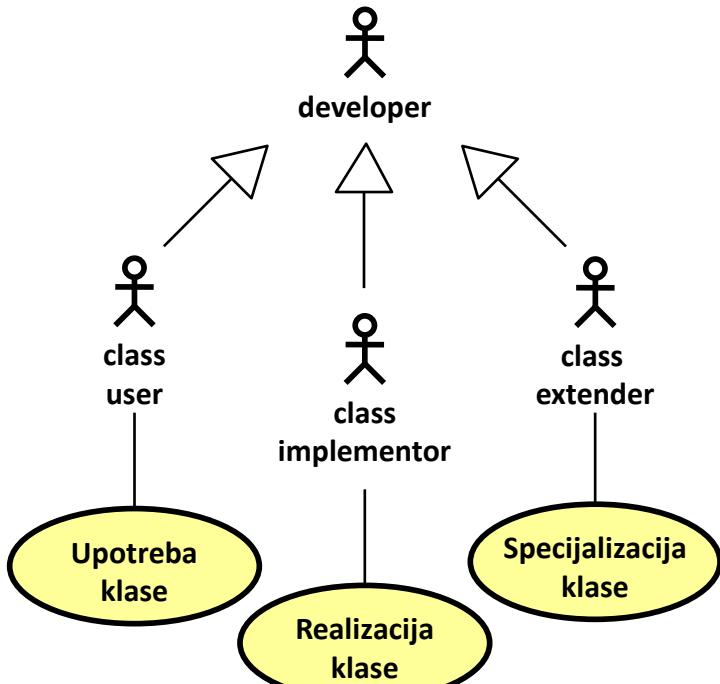
Naknadna specifikacija  
uključuje uopštavanja  
(generalizacije)

|                                 |
|---------------------------------|
| Hashtable                       |
| numOfElements:int               |
| put(key:Object,entry:Object)    |
| get(key:Object):Object          |
| remove(key:Object)              |
| containsKey(key:Object):boolean |
| size():int                      |

# Specifikacija vidljivosti

## Specifikacija interfejsa iz perspektive developera

Uloge *developera* u odnosu na klasu:



### *class implementor*

- odgovoran za realizaciju (implementaciju) klase
- specifikacija interfejsa za njega predstavlja radni zadatak (implementira operacije)

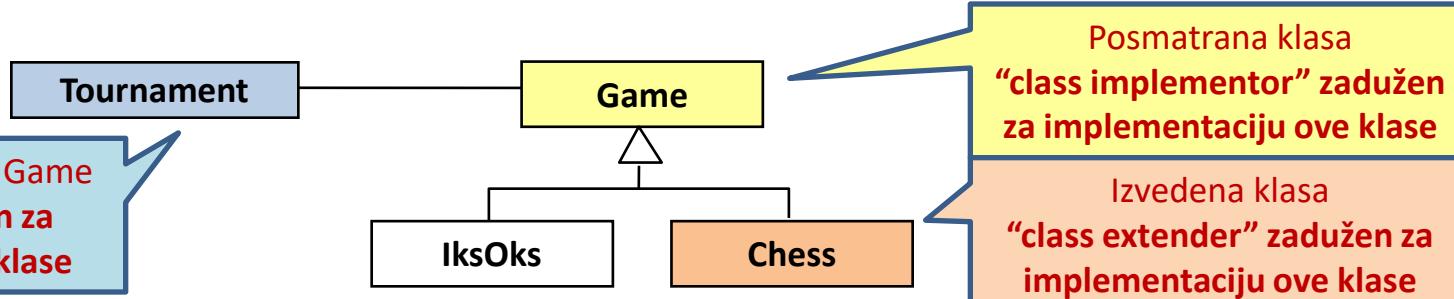
### *class user*

- koristi klasu za realizaciju aplikativnog koda ili za realizaciju druge klase koja agregira objekte date klase ili poziva operacije date klase
- specifikacija interfejsa za njega predstavlja specifikaciju onog što mu je dostupno iz klase, onog što mora da ima u vidu kad koristi datu klasu

### *class extender*

- specijalizuje posmatranu klasu
- fokusira se na postojeću specifikaciju interfejsa posmatrane klase, jer je ona osnov za specifikaciju interfejsa specijalizovane klase

Primjer:



# Specifikacija vidljivosti

## Nivoi vidljivosti članova klase

### Private (-)

- Privatni atributi su **dostupni samo iz unutrašnjosti klase** kojoj pripadaju (može da im se pristupi samo iz operacija koje pripadaju istoj klasi).
- Privatne operacije su **dostupne samo iz unutrašnjosti klase** kojoj pripadaju (mogu da ih pozivaju samo operacije koje su definisane u istoj klasi).
- Privatni atributi i operacije **nisu dostupni izvan klase** kojoj pripadaju i ne može da im se pristupi iz drugih klasa.

### Protected (#)

- Zaštićeni atributi i operacije **dostupni su iz unutrašnjosti klase** kojoj pripadaju (može da im se pristupi iz drugih operacija koje pripadaju istoj klasi), ali **i iz klasa koje su izvedene iz date klase**.

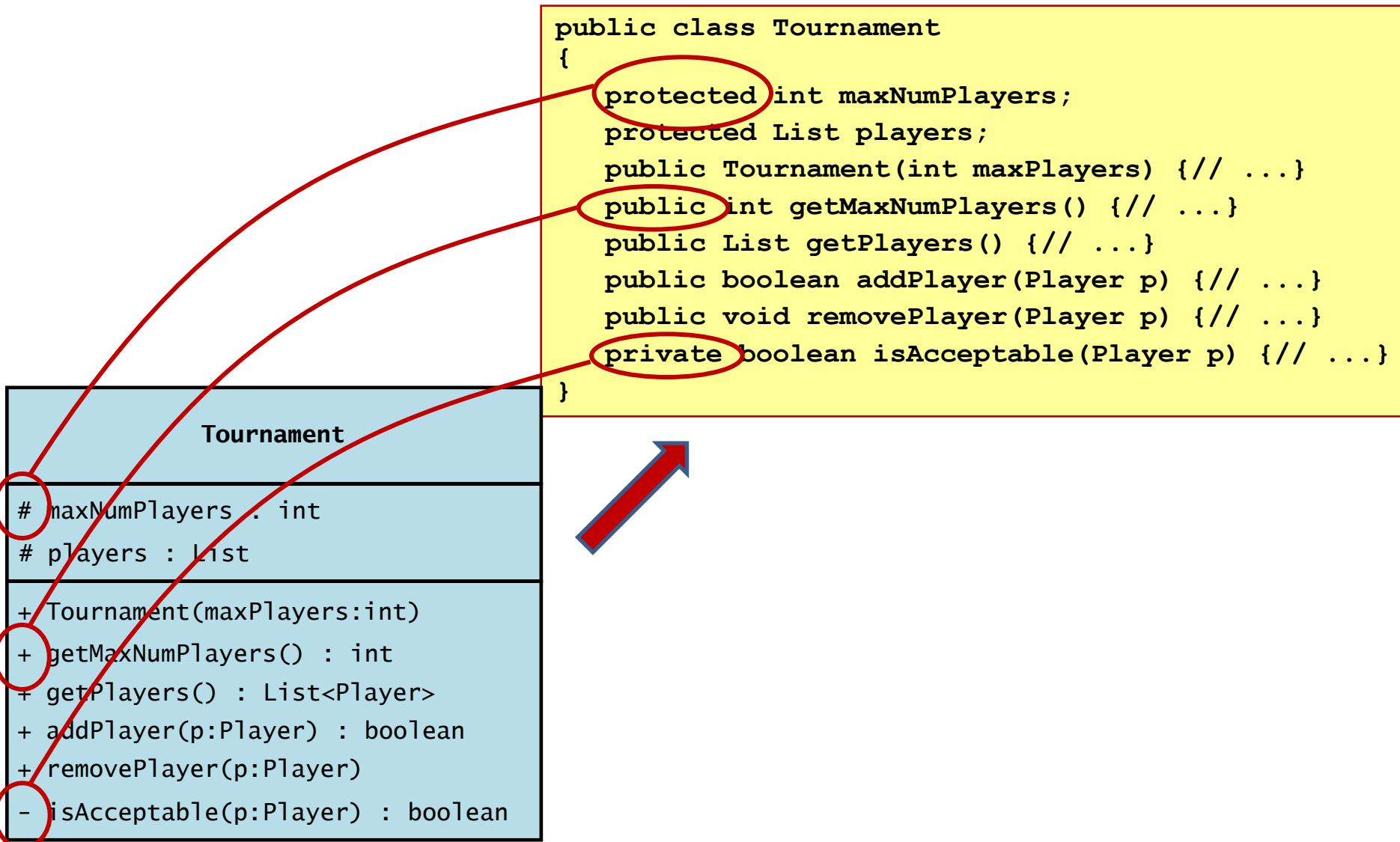
### Public (+)

- Za javne članove klase **ne postoje ograničenja u dostupnosti** (dostupni su i spolja, iz unutrašnjosti klase, kao i svih izvedenih klasa).



# Specifikacija vidljivosti

Primjer (Implementacija vidljivosti):



# Specifikacija vidljivosti

## Heuristike za skrivanje informacija

- Skrivanje informacija nije predmet razmatranja tokom analize, ali je **veoma bitan zahtjev u projektovanju!**
- Interfejse treba definisati veoma oprezno (**što manji broj argumenata**), bilo da je riječ o klasama ili podsistemima (treba nastojati da se primjenjuje **FASADA/INTERFEJSNI OBJEKAT**)
- Primjena principa “**TREBA DA ZNA**”
  - član klase treba da je vidljiv samo ako neko “treba da zna”
- **Korisnik klase treba da zna što manje detalja o klasi (BLACK BOX)**
  - tada je lakše mijenjati datu klasu, jer je manja sprega sa drugim klasama
  - izmjene na datoј klasi imaće manji uticaj na spregnute klase
- “**Trade-off**”: **SKRIVANJE INFORMACIJA ↔ EFIKASNOST**
  - pristup privatnim članovima usporava rad (kritično u *real-time* sistemima, igrama, ...)

# Specifikacija ograničenja

## Contracts

- **Kontrakt** specifikuje ograničenja koja korisnik klase mora da zadovolji da bio koristio klasu, kao i ograničenja koja *class implementor* i *class extender* moraju da obezbijede u implementaciji klase.
- Kontrakt uključuje tri tipa ograničenja:
  - **invarijanta** (eng. *invariant*)
    - invarijanta je predikat koji je istinit za sve instance date klase
    - invarijante su ograničenja vezana za klase (interfejse)
    - invarijante se koriste za specifikaciju ograničenja između atributa
  - **preduslov** (eng. *precondition*) – “**PRAVA**”
    - preduslov je predikat koji mora biti istinit prije poziva operacije
    - preduslov je vezan za konkretnu operaciju
    - preduslov se koristi za specifikaciju ograničenja koje korisnik klase mora da zadovolji prije poziva konkretnе operacije
  - **postuslov** (eng. *postcondition*) – “**OBAVEZE**”
    - postuslov je predikat koji mora biti istinit po završetku izvršenja operacije
    - postuslov je vezan za konkretnu operaciju
    - postuslov se koristi za specifikaciju ograničenja koje *class implementor/extender* mora da obezbijedi po završetku izvršenja operacije

# Specifikacija ograničenja

Ako su invarijante, preduslovi i postuslovi jednoznačni, tada se kontrakt naziva FORMALNA SPECIFIKACIJA.

## Primjeri ograničenja

### invarijanta

Maksimalan broj igrača na turniru mora biti pozitivan (ako se kreira instanca u kojoj je maxNumPlayers=0, biće narušen kontrakt)

$\forall t : \text{typeof}(t)=\text{Tournament} \Rightarrow t.\text{getMaxNumPlayers}() > 0$

### preduslov

Da bi se novi igrač mogao priključiti turniru, moraju da budu zadovoljena dva uslova:

1. dati igrač se još ne nalazi na listi uključenih igrača,
2. još nije dostignut maksimalan broj igrača

$\forall t : \text{typeof}(t)=\text{Tournament} \wedge \forall p : \text{typeof}(p)=\text{Player}$   
 $\Rightarrow \neg t.\text{isAccepted}(p) \wedge$   
 $t.\text{getNumPlayers}() < t.\text{getMaxNumPlayers}()$

### postuslov

Ako se novi igrač uspješno uključi u turnir, tada se taj igrač nalazi u listi igrača:

$t.\text{addPlayer}(p) \Rightarrow t.\text{isAccepted}(p)$

```
Tournament

# maxNumPlayers : int
# numPlayers : int
# players : List

+ Tournament(maxPlayers:int)
+ getMaxNumPlayers() : int
+ getNumPlayers() : int
+ getPlayers() : List
+ addPlayer(p:Player) : boolean
+ removePlayer(p:Player)
+ isAccepted(p:Player) : boolean
```

# Specifikacija ograničenja

## Reprezentacija ograničenja u UML modelima

Ograničenje može da se prikaže kao *note* (vezan za odnosni UML koncept) koji sadrži logički izraz.

«invariant»

`getMaxNumPlayers () >0`

«precondition»

`!isAccepted(p) and  
getNumPlayers () < getMaxNumPayers ()`

«precondition»

`isAccepted(p)`

Tournament

# maxNumPlayers:int  
# numPlayers:int  
# players>List

+ Tournament(maxPlayers:int)  
+ getMaxNumPlayers():int  
+ getNumPlayers():int  
+ getPlayers():List  
+ addPlayer(p:Player):boolean  
+ removePlayer(p:Player)  
+ isAccepted(p:Player):boolean

«postcondition»

`isAccepted(p)`

«postcondition»

`!isAccepted(p)`

ALTERNATIVA:

UML specifikacija uključuje OCL (*Object Constraint Language*) za specifikaciju ograničenja

# OCL

## OCL (Object Constraint Language)

- OCL je formalni jezik za specifikaciju ograničenja na skupu objekata i njihovim atributima
- Dio standardnog UML2 (od 2006. godine)
- Koristi se za specifikaciju ograničenja koja ne mogu da se specifikuju standardnom UML notacijom
- **Osnovne karakteristike:** deklarativni jezik (bez bočnih efekata i kontrole toka)
- **Osnovni koncepti:** **OCL izrazi**
  - vraćaju TRUE ili FALSE
  - evaluiraju se u specifikovanom kontekstu (KLSA / OPERACIJA)
  - sva ograničenja primjenjuju se na sve instance

## Opšti oblik OCL ograničenja

context <scope> inv: |pre: |post: <expr>

Ključna riječ za specifikaciju OCL ograničenja

Klasa ili operacija na koju se odnosi ograničenje

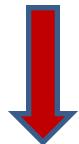
inv: ≡ «invariant»  
pre: ≡ «precondition»  
post: ≡ «postcondition»

OCL izraz

# OCL

Primjer OCL specifikacije INVARIJANTE:

«invariant»  
getMaxNumPlayers ()>0



context Tournament inv:  
self.getMaxNumPlayers ()>0

self

Ima istu semantiku kao this u OO programskim jezicima.  
Ovdje označava sve instance klase Tournament (za svaku instancu).

Tournament

```
# maxNumPlayers:int  
# numPlayers:int  
# players>List  
  
+ Tournament(maxPlayers:int)  
+ getMaxNumPlayers():int  
+ getNumPlayers():int  
+ getPlayers():List  
+ addPlayer(p:Player):boolean  
+ removePlayer(p:Player)  
+ isAccepted(p:Player):boolean
```

Interpretacija OCL pravila:

“MAKSIMALAN BROJ IGRAČA NA SVAKOM TURNIRU TREBA DA BUDE POZITIVAN”

# OCL

Primjer OCL specifikacije PREDUSLOVA:

Kontekst ograničenja (operacija)  
Tournament::removePlayer (p:Player)

**context Tournament::removePlayer (p:Player)**  
**pre:** `isAccepted(p)`



«precondition»  
`isAccepted(p)`

**Tournament**

# maxNumPlayers:int  
# numPlayers:int  
# players>List

+ Tournament(maxPlayers:int)  
+ getMaxNumPlayers():int  
+ getNumPlayers():int  
+ getPlayers():List  
+ addPlayer(p:Player):boolean  
+ removePlayer(p:Player)  
+ isAccepted(p:Player):boolean

Interpretacija OCL pravila:

“OPERACIJA Tournament::removePlayer(p:Player)  
**MOŽE BITI POZVANA SAMO AKO IGRAČ p UČESTVUJE NA TURNIRU”**

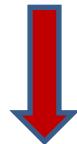
Alternativno je moglo: `self.isAccepted(p)`

Self ne mora (a može) da se koristi ako nema nedvosmislenosti, kao što je slučaj sa preduslovima i postuslovima (zna se da je kontekst operacija za objekat za koji se poziva).

# OCL

Primjer OCL specifikacije POSTUSLOVA:

«postcondition»  
!isAccepted(p)



**context** Tournament::removePlayer(p: Player)  
**post:** not isAccepted(p)

|                                                                                                                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Tournament                                                                                                                                                                                              |
| # maxNumPlayers:int<br># numPlayers:int<br># players>List                                                                                                                                               |
| + Tournament(maxPlayers:int)<br>+ getMaxNumPlayers():int<br>+ getNumPlayers():int<br>+ getPlayers():List<br>+ addPlayer(p:Player):boolean<br>+ removePlayer(p:Player)<br>+ isAccepted(p:Player):boolean |

Alternativno pravilo:

Nakon isključivanja jednog igrača, ukupan broj igrača manji je za jedan nego prije isključivanja datog igrača.

**context** Tournament::removePlayer(p: Player)  
**post:** getNumPlayers() = self.getNumPlayers@pre() - 1

self@pre označava stanje objekta (Tournament) prije izvršenja operacije Tournamnet::removePlayer()

# OCL

Primjer:



**context** Roba::povecajStanje (kolicina:Integer)

**pre:** kolicina > 0

**post:** stanje = self.stanje@pre + kolicina

**context** Roba::getStanje ()

**post:** result = stanje

result reprezentuje rezultat izvršavanja operacije

# OCL

## Višestruka ograničenja:

- Za isti kontekst mogu da se specifikuju višestruka ograničenja
- Sva ograničenja iste vrste moraju biti zadovoljena
  - npr. operacija ne može da se pozove ako nisu svi preduslovi zadovoljeni

```
context
Tournament::addPlayer(p:Player)
pre:
not isAccepted(p)
```

```
context
Tournament::addPlayer(p:Player)
pre:
getNumPlayers() < getMaxNumPlayers()
```

```
context
Tournament::addPlayer(p:Player)
post:
isAccepted(p)
```

```
context
Tournament::addPlayer(p:Player)
post:
getNumPlayers() = getNumPlayers@pre()+1
```

KONTRAKT za  
Tournament::addPlayer(p:Player)

context
Tournament::addPlayer(p:Player)
pre:
not isAccepted(p) and
getNumPlayers() < getMaxNumPlayers()

context
Tournament::addPlayer(p:Player)
post:
isAccepted(p) and
getNumPlayers()=getNumPlayers@pre()+1

# OCL

## OCL kontrakt kao JavaDoc anotacije:

- Alati za automatsko generisanje koda na osnovu UML modela imaju mogućnost generisanja JavaDoc anotacija na osnovu specifikovanih OCL kontrakta
- Specifični JavaDoc tagovi: `@invariant`, `@pre` i `@post`
- Anotacije omogućavaju automatsku validaciju predikata (može da usporava rad sistema)

```
context Tournament inv:  
self.maxNumPlayers > 0
```

```
context Tournament::addPlayer(p:Player) pre:  
not isAccepted(p)
```

```
context Tournament::addPlayer(p:Player) pre:  
getNumPlayers() < getMaxNumPlayers()
```

```
context Tournament::addPlayer(p:Player) post:  
isAccepted(p)
```

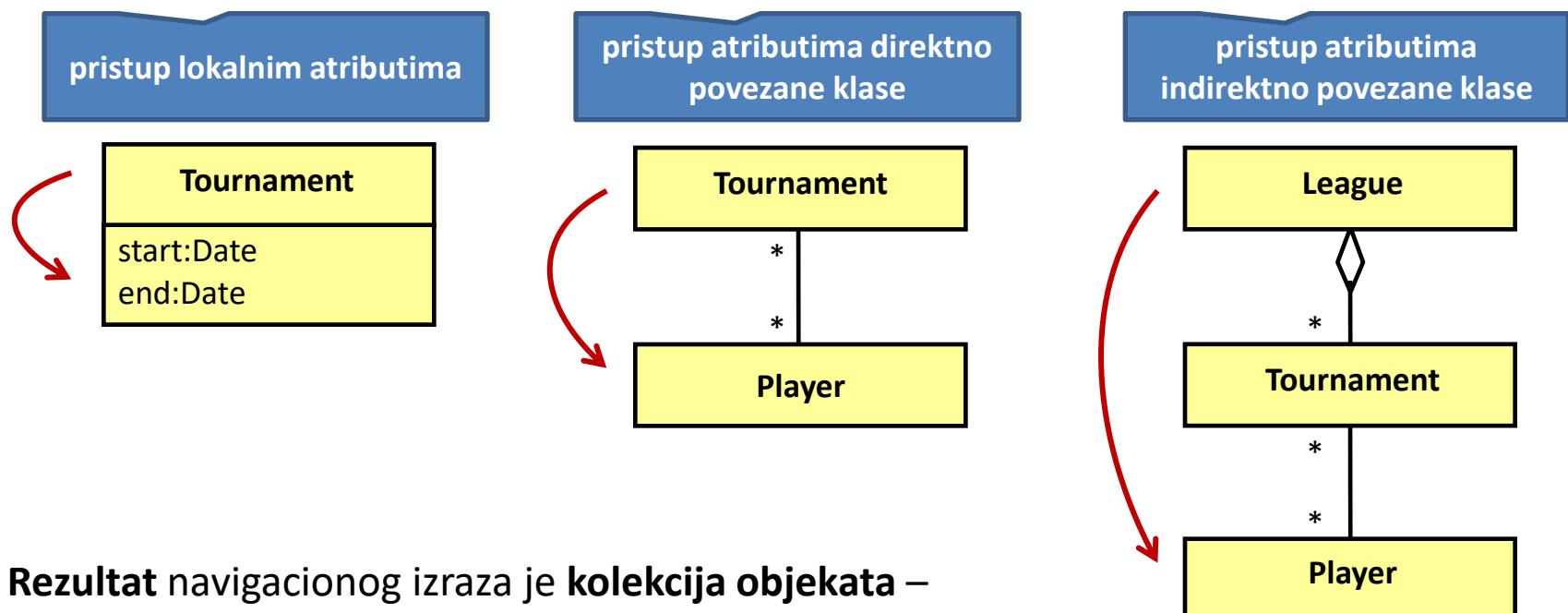
```
context Tournament::addPlayer(p:Player) post:  
getNumPlayers() = getNumPlayers@pre() + 1
```

```
/* @invariant maxNumPlayers > 0 */  
public class Tournament  
{  
    private int maxNumPlayers;  
  
    private List players;  
  
    public int getNumPlayers() {/**...}   
    public int getMaxNumPlayers() {/**...}  
  
    /**  
     * @pre !isAccepted(p)  
     * @pre getNumPlayers() < getMaxNumPlayers()  
     * @post isAccepted(p)  
     * @post getNumPlayers() =  
     *           self.getNumPlayers@pre() + 1  
     */  
    public void addPlayer (Player p) {/**...}  
  
    /* ostale metode */  
}
```

# OCL

## OCL ograničenja koja uključuju više klasa:

- Dijagram klasa predstavlja kolekciju povezanih klasa – moguća (i često potrebna) **navigacija** kroz dijagram i referenciranje drugih klasa i njihovih atributa/operacija = **navigacioni OCL izrazi**
- **tri tipa navigacije kroz model:**



- **Rezultat navigacionog izraza je kolekcija objekata** – kolekcija objekata s druge strane asocijacije (broj objekata u kolekciji zavisi od multiplikativnosti druge strane asocijacijske)

# OCL

## OCL kolekcije:

- OCL tip **Collection** je generička klasa koja reprezentuje kolekcije objekata tipa T
  - **primitivni**: Integer, String, Boolean, Real, ...
  - **korisnički definisan**: Tournament, Player, ...
- Podržane OCL kolekcije (potklase klase Collection):
  - **Set** – neuređena kolekcija bez duplikata,
  - **Bag** – neuređeni multiset (neuređena kolekcija sa mogućim duplikatima),
  - **Sequence** – uređeni multiset (uređena kolekcija sa mogućim duplikatima).
- Primjeri kolekcija:
  - Set(Integer), Bag(Player), Sequence(Integer)
- Generalizacija svih tipova podataka: **OclAny**
- OCL raspolaže velikim brojem različitih operacija koje mogu da se primjenjuju nad kolekcijama  
**kolekcija->operacija(argumenti)**

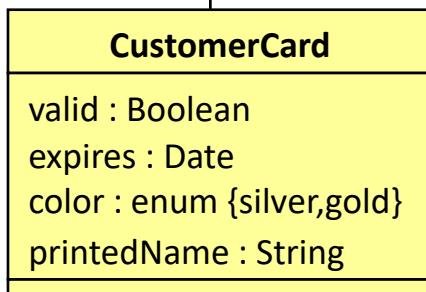
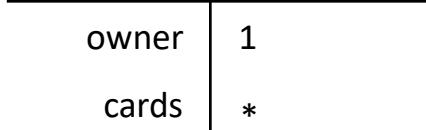
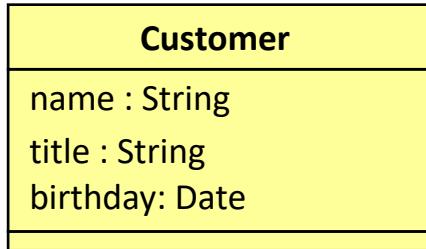
## OCL operacije nad kolekcijama:

- size() : Integer**  
ukupan broj elemenata u kolekciji
- includes(o:OclAny) : Boolean**  
TRUE, ako je objekat "o" sadržan u kolekciji
- count(o:OclAny) : Integer**  
ukupan broj pojavljivanja objekta "o" u kolekciji
- isEmpty() : Boolean**  
TRUE, ako je kolekcija prazna
- notEmpty() : Boolean**  
TRUE, ako kolekcija nije prazna
- operacije koje vraćaju kolekciju --
- union(c:Collection)**  
unija sa kolekcijom "c"
- intersection(c:Collection)**  
presjek sa kolekcijom "c"
- including(o:OclAny)**  
dodaje objekat "o" u kolekciju
- select(expr:OclExpression)**  
podskup kolekcije koji zadovoljava izraz "expr"

# OCL

## Navigacija kroz asocijaciju “1:”

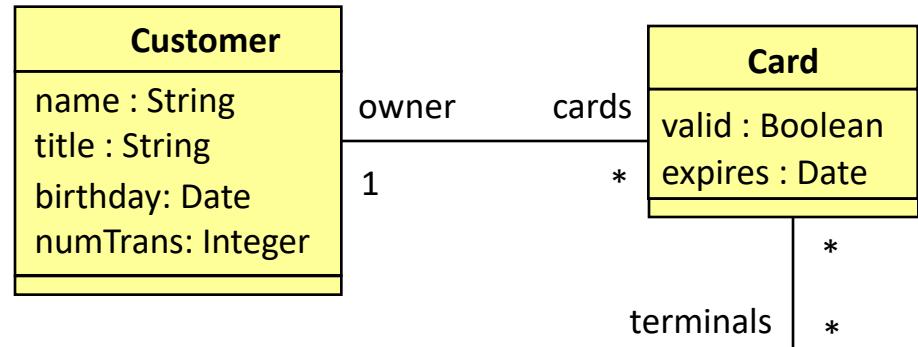
- Navigacija kroz asocijaciju “1:” kao rezultat daje **Set**
- Navigacija kroz {ordered} asocijaciju “1:” kao rezultat daje **Sequence**



```
context Customer inv:  
self.cards->size() < 4
```

## Navigacija kroz više “1:” asocijacija

- Navigacija kroz više “1:” asocijacija kao rezultat daje **Bag (multiset)**



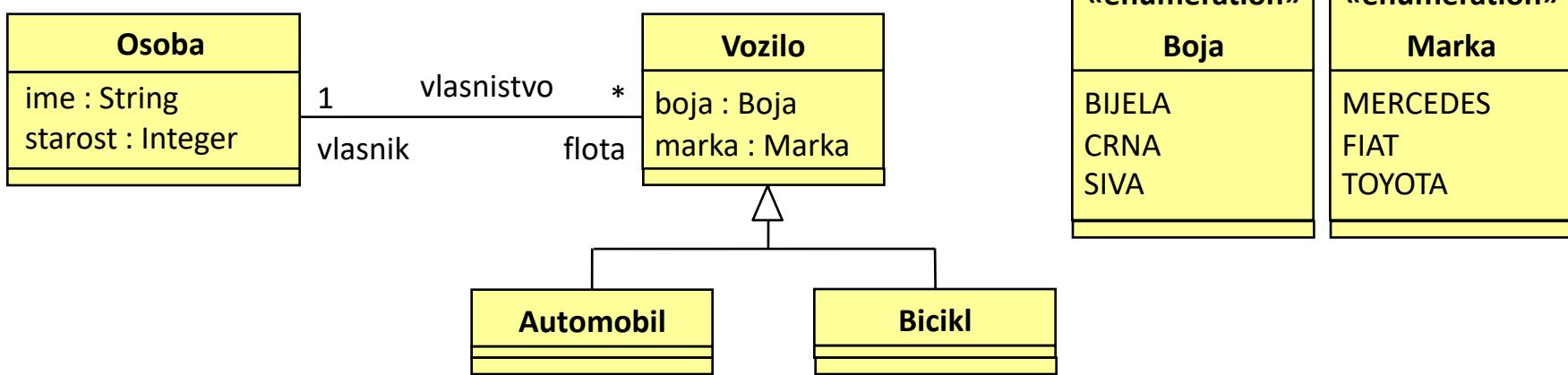
```
context Customer inv:  
self.numTrans = self.cards.terminals->size()
```

Ukupan broj transakcija koje je ostvario customer

Customer može da ima najviše tri kartice

# OCL

Primjer:

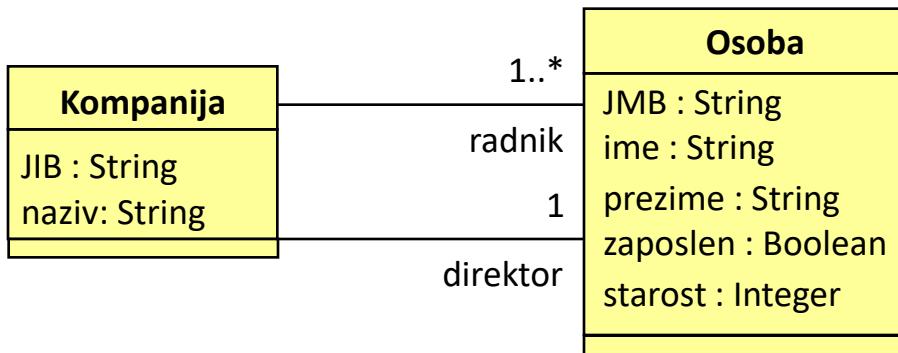


Odaberi odgovarajući upit:

- ① **context** Osoba  
**inv:** `self.starost >= 18` ✗ Sve osobe moraju da imaju bar 18 godina
- ② **context** Vozilo  
**inv:** `self.vlasnik.starost >= 18` ✗ Vlasnik svakog vozila mora da ima bar 18 godina
- ③ **context** Automobil  
**inv:** `self.vlasnik.starost >= 18` ✓ Vlasnik svakog automobila mora da ima bar 18 godina

# OCL

Primjer:



**context** Kompanija

**inv:** self.direktor.zaposlen = TRUE

Multiplikativnost asocijacije na strani Osobe je "1", pa rezultat upita jedan objekat tipa Osoba

**inv:** self.radnik->notEmpty()

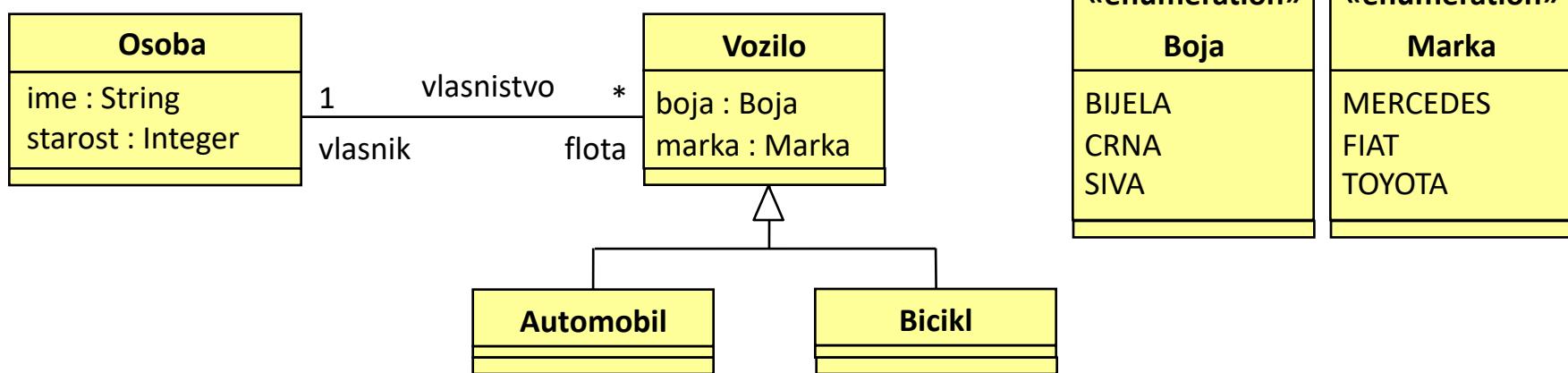
Multiplikativnost asocijacije na strani Osobe je "1..\*", pa je rezultat upita skup objekata tipa Osoba

**inv:** self.radnik->  
select(r | r.starost>=65)->isEmpty()

Svi stariji od 65 godina moraju u penziju!

# OCL

Primjer:



**context** Osoba

**inv:** self.flota->size() < 4

Svaka osoba može da ima najviše tri vozila

**context** Osoba

**inv:** self.flota->select (v | v.boja=#CRNA)->size() < 3

Svaka osoba može da ima najviše dva crna vozila

**context** Osoba

**inv:** self.flota->select (v | v.oclIsKindOf(Automobil))->size() < 3

Svaka osoba može da ima najviše dva automobila

# ocl

## Konverzije OCL kolekcija

### asSet()

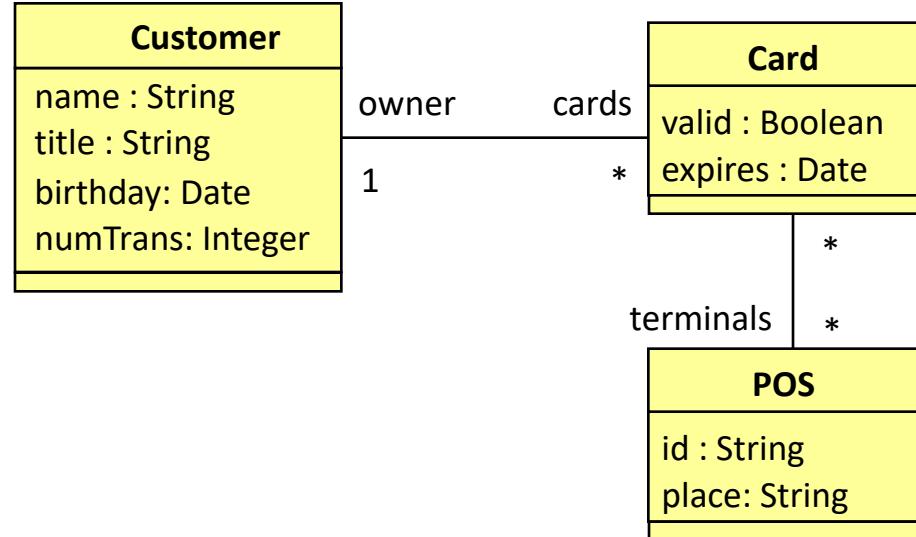
konvertuje Sequence ili Bag kolekciju u Set

### asSequence()

konvertuje Set ili Bag kolekciju u Sequence

### asBag()

Konvertuje Set ili Sequence kolekciju u Bag



Ukupan broj transakcija koje je ostvario customer

`cards . terminals->size ()`



asSet() izbacuje duplike, pa u setu imamo samo sve različite terminale na kojima su ostvarene transakcije

`cards . terminals->asSet ()->size ()`

# ocl

## OCL kvantifikatori

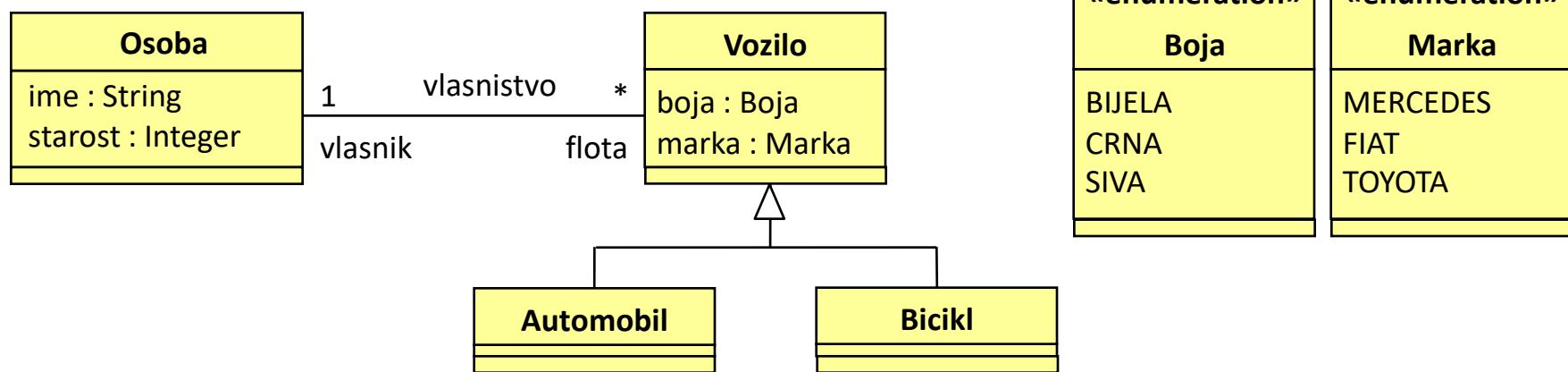
`forAll( <var> | <expr> ) : Boolean`

TRUE, ako je `<expr>` istinit za sve elemente u kolekciji

`exists( <var> | <expr> ) : Boolean`

TRUE, ako je `<expr>` istinit bar za jedan element u kolekciji

### Primjer:



`context Osoba`

`inv: self.flota->forAll(v | v.boja=#CRNA)`

Osoba mora da ima sva crna vozila

`context Osoba`

`inv: self.flota->exists(v | v.boja=#CRNA)`

Osoba mora da ima bar jedno crno vozilo

# OCL

Primjeri:

```
-- nijedan racun ne smije biti u minusu
context Racun inv:
    Racun.allInstances() ->forAll( r | r.stanje >= 0 )

-- svi radnici u kompaniji moraju biti mladji od 65 godina
context Kompanija inv:
    self.radnik->forAll( r:Osoba | r.starost < 65 )

-- ne mogu da postoje dva studenta sa istim brojem indeksa
context Student inv:
    Student.allInstances() ->
        forAll( s1,s2 | s1<>s2 implies s1.indeks<>s2.indeks )

-- mora da postoji bar jedno vozilo crne boje
context Vozilo inv:
    Vozilo.allInstances() ->exists( v | v.boja = #CRNA )

-- za svaki predmet mora da postoji bar jedan zaduzeni nastavnik
context Predmet inv:
    self.nastavnik->exists( z:Zaduzenje | z.predmet = self.naziv )
```

# Heuristike za specifikaciju ograničenja

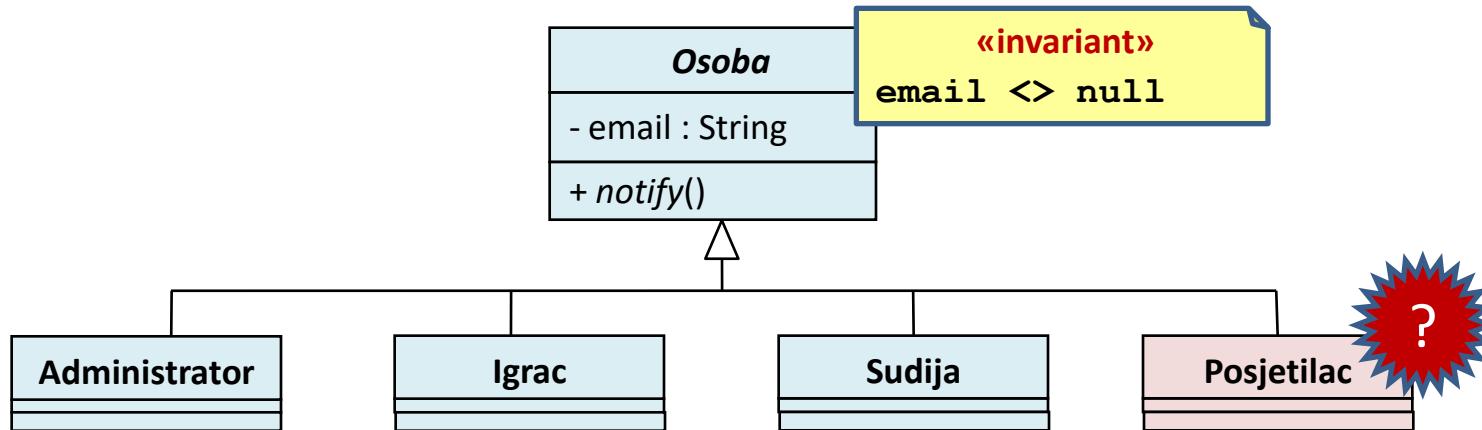
## Neke heuristike za specifikaciju ograničenja:

- fokusirati se na životni ciklus klase
  - ograničenja koja su specifična za operacije ili neka stanja objekata (ulazak u stanje/izlazak iz stanja) bolje je specifikovati kao preduslove ili postuslove
- obratiti pažnju na specifične vrijednosti svakog atributa
  - null vrijednosti, jedinstvene vrijednosti, vrijednosti koje zavise od vrijednosti drugih atributa...
- identifikovati specifične slučajeve asocijacija
  - multiplikativnosti krajeva u nekim asocijacijama ne mogu adekvatno da se specifikuju standardnom notacijom, npr. podskupovi, uslovna multiplikativnost, ...
- definisati pomoćne metode za specifična ograničenja
  - nekad je bolje uvesti novu metodu (helper) zaduženu za implementaciju specifičnih ograničenja, nego specifikovati komplikovana OCL ograničenja, npr. preklapanje događaja, ...
- minimizovati broj asocijacija uključenih u neko OCL ograničenje
  - veći broj uključenih asocijacija smanjuje čitljivost OCL ograničenja, povećava mogućnost greške, povećava broj klasa u multisetovima, ...

# Ograničenja u kontekstu nasljeđivanja

## *Contract inheritance:*

- Striktno nasljeđivanje (nasljeđivanje u skladu sa pravilom supstitucije) korisniku hijerarhije klase daje za pravo da smatra da ograničenja koja važe na superklasi, važe i na potklasama.



Operacija **notify()** služi za slanje elektronske poruke osobi.

Invarijanta za klasu **Osoba** specifikuje da svaka osoba ima e-adresu.

Striktno nasljeđivanje daje korisniku za pravo da smatra da invarijanta važi i za potklase, tj. da i administrator i igrači i sudije imaju elektronske adrese te da im je moguće poslati elektronsku poruku.

Posjetioci su specifičan slučaj: najvjerojatnije nemamo e-adrese svih posjetilaca!

RJEŠENJE: Ili klasu **Posjetilac** isključiti iz hijerarhije ili promijeniti ograničenje!

# Ograničenja u kontekstu nasljeđivanja

## Pravila za nasljeđivanje kontrakta

- **invarijante:**
  - Sve invarijante superklase moraju da važe i za potklase.
  - Za svaku potklasu **mogu da se specifikuju i dodatne invarijante.**
  - Npr. **Niz  $\Leftrightarrow$  SortiraniNiz** (sve što važi za niz, važi i za sortirani niz, uključujući i uređeni poredak).
- **preduslovi:** (*constraint weakening*)
  - Metoda u potklasi **ima pravo da “oslabi” preduslov** za metodu koju redefiniše, tj. preduslovi za metodu u potklasi su “manje strogi” nego za metodu u natklasi.
  - Npr. redefinisana metoda u potklasi može da obrađuje više različitih slučajeva u odnosu na odnosnu metodu iz superklase, pa može da oslabi preduslove iz natklase.
- **postuslovi:** (*constraint strengthening*)
  - Metoda u potklasi **mora da obezbijedi iste postuslove** kao metoda u superklasi, a **može i da ih “pooštri”**, tj. postuslovi za redefinisanu metodu u potklasi mogu da budu “više strogi” nego u natklasi.

**UNIVERZITET U BANJOJ LUCI  
ELEKTROTEHNIČKI FAKULTET**

**Prof. dr Dražen Brđanin**

**OBJEKTNNO-ORIJENTISANI DIZAJN  
/mapiranje modela/**

**Banja Luka  
2024.**

# Model-based Software Engineering

## VIZIJA (IDEALAN SLUČAJ)

Tokom OOD projektuje se objektni model koji je osnov za (automatsku) implementaciju ciljnog softverskog sistema

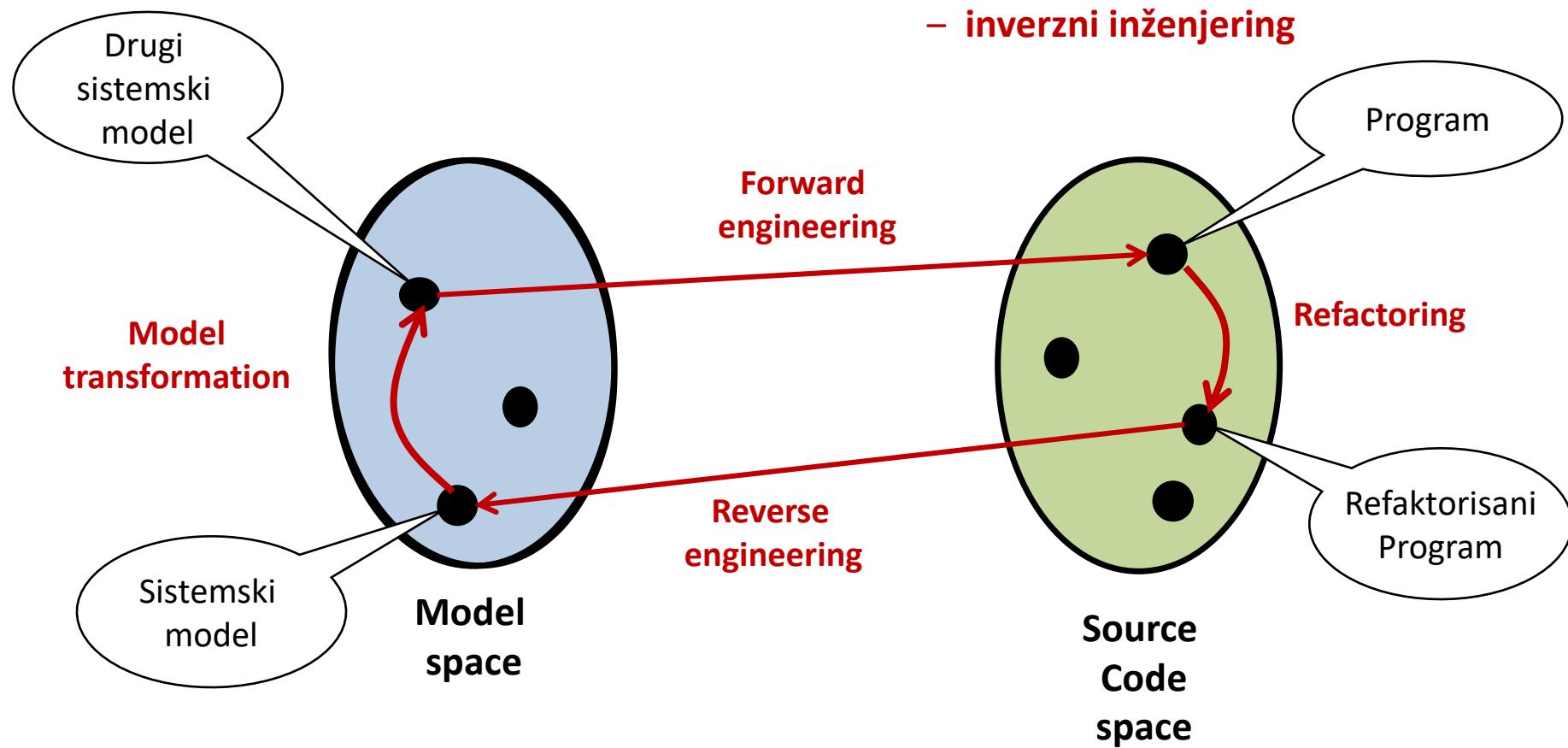
## Realnost

- Često se OO model razlikuje od ciljne implementacije:
  - izmijenjeni korisnički zahtjevi, npr. kad je projektovanje u završnoj fazi ili je već započeta implementacija, pa se izmjene rade direktno u kodu, a ne na modelu
  - nedokumentovani argumenti, npr. zbog kašnjenja u realizaciji projekta, promjene direktno u kodu, a ne na modelu (dodatni argument u metodi u kodu, ali ne i operacija u klasi u modelu)
  - neuskladenost objektnog modela i sistema za upravljanje perzistentnim slojem
- Ne postoji potpuno definisana i usaglašena pravila za automatsko mapiranje modela u kod.
- Neki programski jezici nemaju sve neophodne mehanizme, npr.
  - za implementaciju kontrakta,
  - nemaju direktnu i potpunu podršku za UML asocijacije (kolekcija referenci).
- ...

# “Model space” vs “Source Code space”

Dva prostora:

- Model space
- Source Code space



# Transformacije modela (M2M)

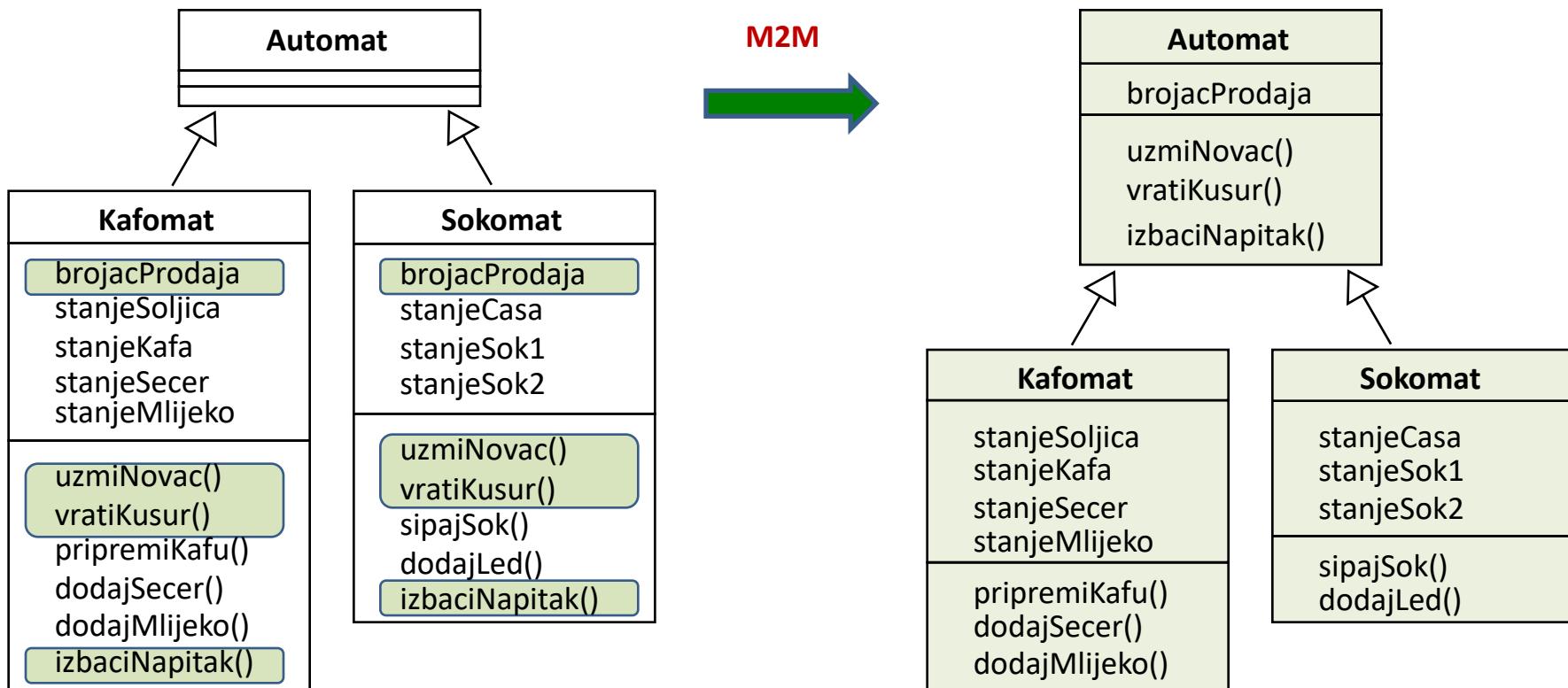
## Manuelne transformacije

- tipični primjeri:
  - generalizacija/specijalizacija
  - redukcija klasa/asocijacija
  - reuse/redundantne asocijacije

## Automatizovane transformacije

- transformacioni jezici opše namjene (XSLT)
- specijalizovani transformacioni jezici (ATL, QVT)

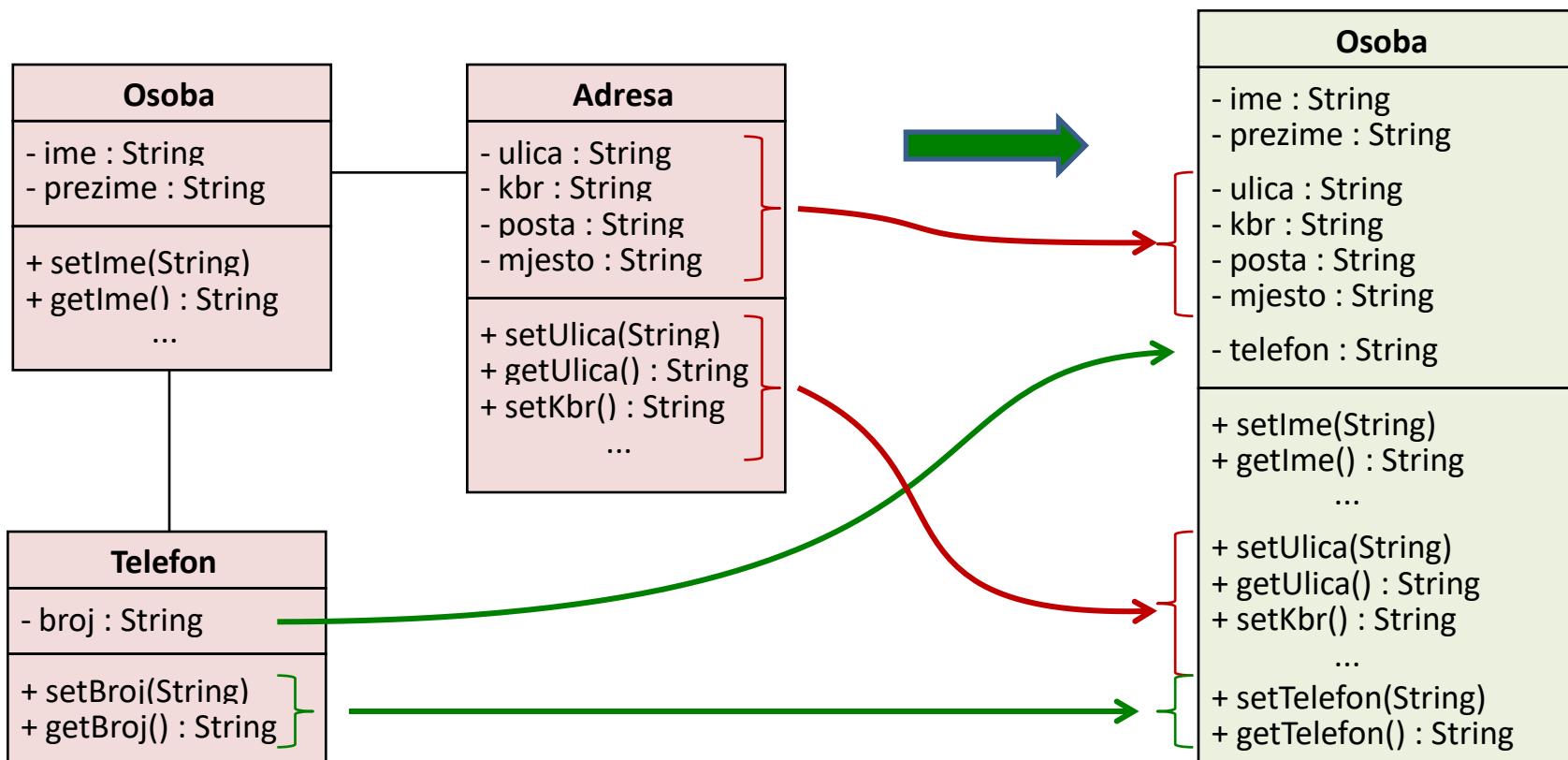
## Primjer M2M (generalizacija):



# Transformacije modela (M2M)

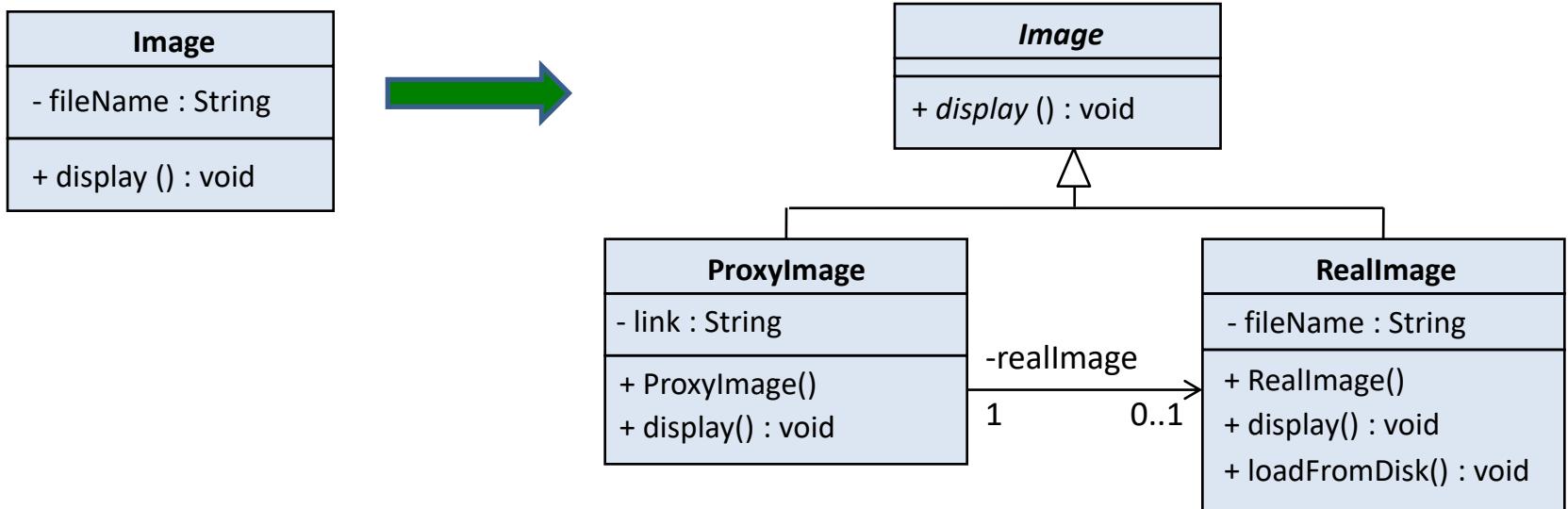
## Primjer M2M (redukcija broja klasa):

- Tipično u situacijama kad neka klasa ne inkapsulira značajno ponašanje, nego ima samo atributе i uglavnom setere/getere



# Transformacije modela (M2M)

Primjer M2M (upotreba PROXY obrasca u cilju odgađanja izvršavanja “skupe” operacije):



Cijena manipulacije realnom slikom (kreiranje, download, ...) je previsoka, pa proxy ima sopstvenu, pojednostavljenu verziju realne slike ili hiperlink, ...

# Refactoring

## REFACTORING

Transformacija izvornog koda u cilju poboljšanja razumljivosti i bolje strukturiranosti, uz zadržavanje iste funkcionalnosti (bez vidljive promjene ponašanja).

### Tipičan “refactoring” proces:

- postepene, inkrementalne (*small step-by-step*) izmjene
- uglavnom fokusirano na atributе i pojedinačne metode
- tipični primjeri:
  - **attribute pull-up** (izvlačenje atributa iz potomaka u roditelje)
  - **constructor pull-up** (izvlačenje i prilagođavanje konstruktora)

#### Primjer (**attribute pull-up**):

```
public class Player
{
    private String email;
    //...
}

public class Referee
{
    private String eMail;
    //...
}
```

Attribute  
pull-up  


```
public class User
{
    protected String email;
}

public class Player extends User
{
    //...
}

public class Referee extends User
{
    //...
}
```

# Refactoring

Primjer (**constructor pull-up**):

```
public class User
{
    protected String email;
}

public class Player extends User
{
    public Player(String email)
    {
        this.email = email;
    }
}

public class Referee extends User
{
    public Referee(String email)
    {
        this.email = email;
    }
}
```

Constructor  
pull-up

```
public class User
{
    protected String email;
    public User(String email)
    {
        this.email = email;
    }
}

public class Player extends User
{
    public Player(String email)
    {
        super(email);
    }
}

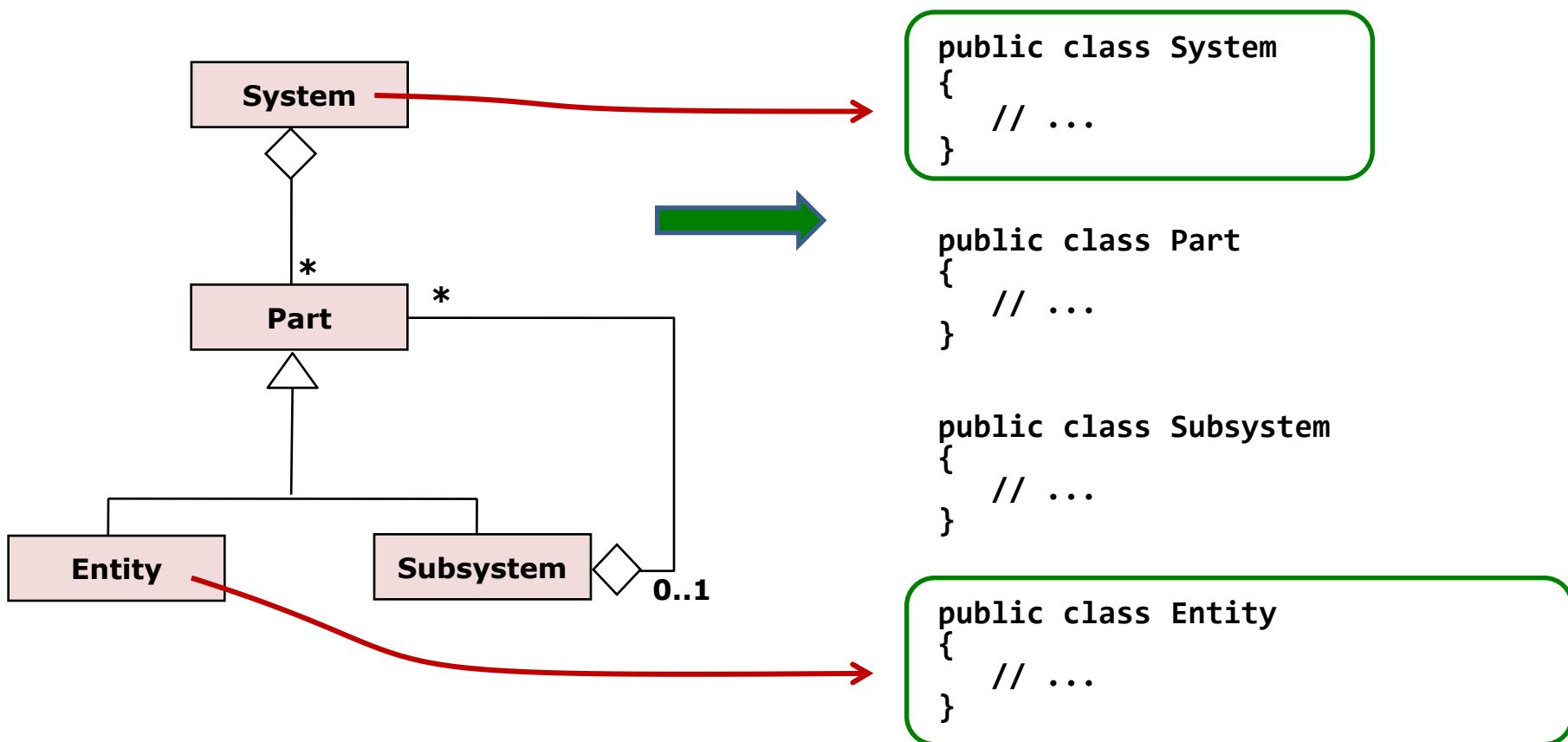
public class Referee extends User
{
    public Referee(String email)
    {
        super(email);
    }
}
```

# Direktni inženjering

Direktni inženjering: **Transformacija (mapiranje) modela u kod.**

## Mapiranje klasa

- direktno mapiranje u korespondentni kod koji sadrži odgovarajuće članove

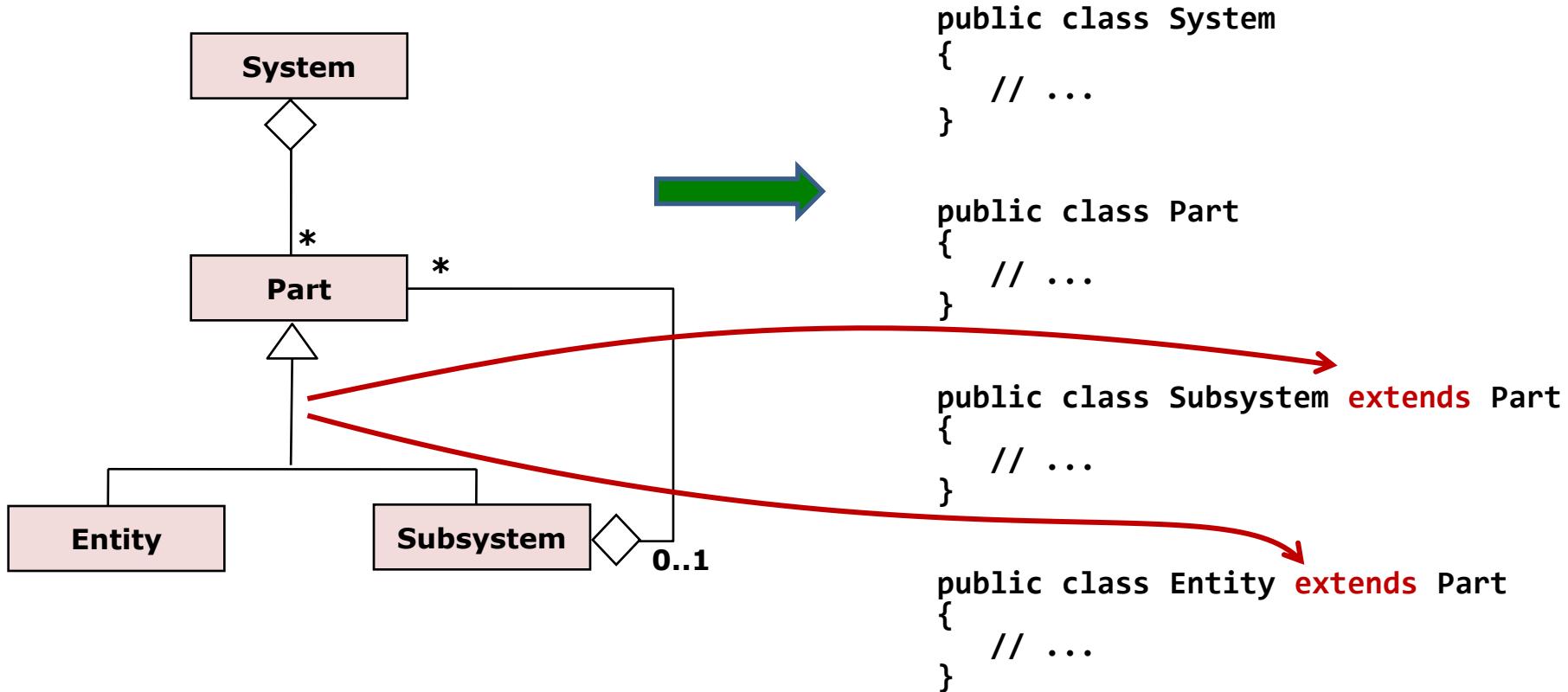


# Direktni inženjering

Direktni inženjering: Transformacija (mapiranje) modela u kod.

## Mapiranje klasa

- direktno mapiranje u korespondentni kod koji sadrži odgovarajuće članove

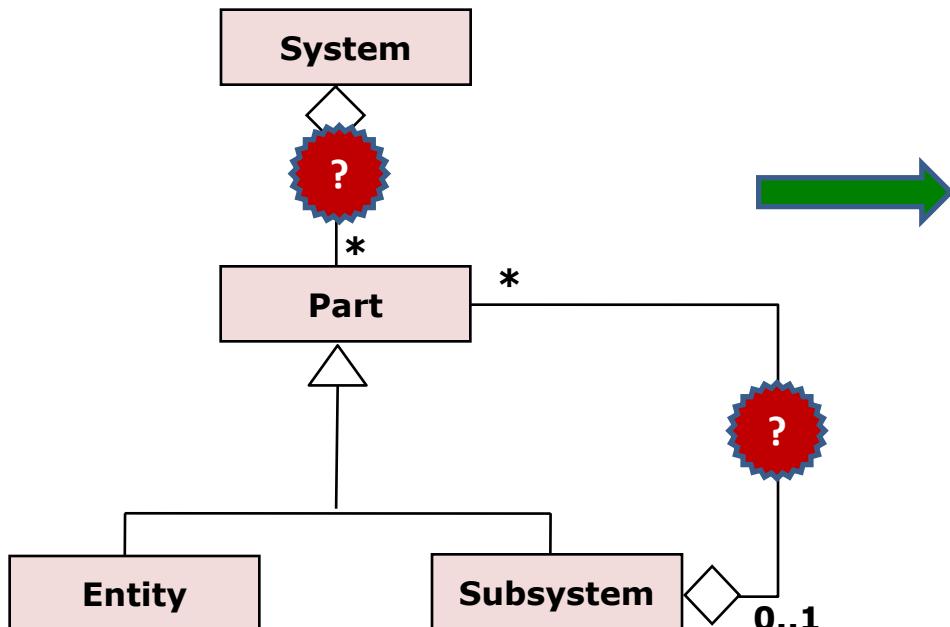


# Direktni inženjering

Direktni inženjering: **Transformacija (mapiranje) modela u kod.**

## Mapiranje klasa

- direktno mapiranje u korespondentni kod koji sadrži odgovarajuće članove



```
public class System
{
    // ...
}

public class Part
{
    // ...
}

public class Subsystem extends Part
{
    // ...
}

public class Entity extends Part
{
    // ...
}
```

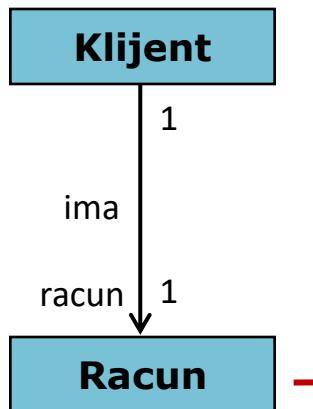
# Direktni inženjering

## Mapiranje asocijacija

- Programski jezici nemaju direktnu i potpunu podršku za UML asocijacije (asocijacije se realizuju referencama)
- Mapiranje:
  - unidirekciona / bidirekciona asocijacija (1:1, 1:\*, \*:\*)
  - association class (klasa pridružena asocijaciji)

1

### Mapiranje unidirekcionе asocijacije 1:1



```
public class Klijent
{
    private Racun racun;

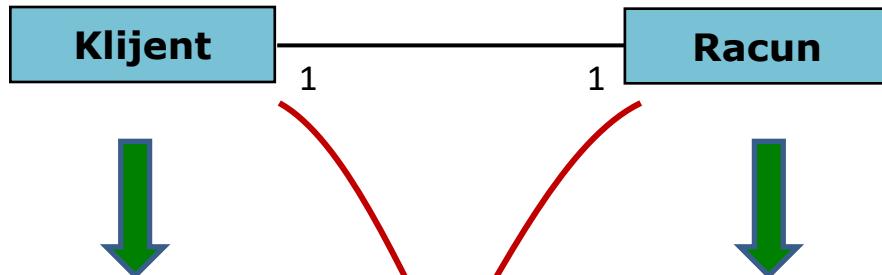
    public Klijent()
    {
        racun = new Racun();
    }

    public Racun getRacun()
    {
        return racun;
    }
}
```

# Direktni inženjering

2

## Mapiranje bidirekcione asocijacije 1:1



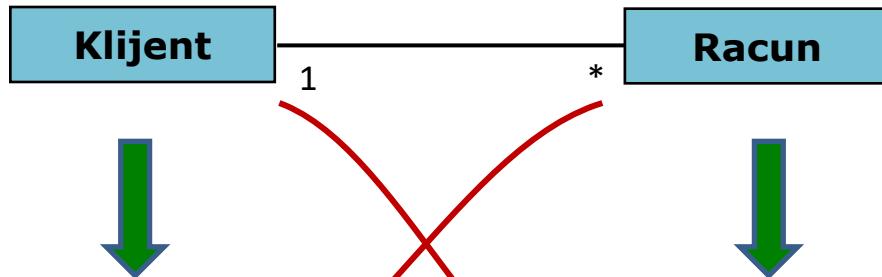
```
public class Klijent
{
    // racun se inicijalizuje
    // u konstruktoru
    // i nikad se ne mijenja
    private Racun racun;
    public Klijent()
    {
        racun = new Racun(this);
    }
    public Racun getRacun()
    {
        return racun;
    }
}
```

```
public class Racun
{
    // vlasnik racuna se inicijalizuje
    // u konstruktoru
    // i nikad se ne mijenja
    private Klijent klijent;
    public Racun(Klijent klijent)
    {
        this.klijent = klijent;
    }
    public Klijent getKlijent()
    {
        return klijent;
    }
}
```

# Direktni inženjering

3

## Mapiranje bidirekcione asocijacije 1:\*



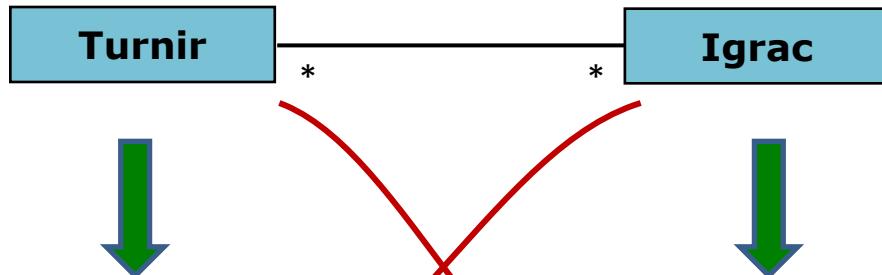
```
public class Klijent
{
    private Set racuni;
    public Klijent() {
        racuni = new HashSet();
    }
    public void addRacun(Racun r) {
        racuni.add(r);
        r.setKlijent(this);
    }
    public void removeRacun(Racun r) {
        racuni.remove(r);
        r.setKlijent(null);
    }
}
```

```
public class Racun
{
    private Klijent klijent;
    public void setKlijent(Klijent nk)
    {
        if (klijent != nk)
        {
            Klijent sk = klijent;
            klijent = nk;
            if (nk != null)
                nk.addRacun(this);
            if (sk != null)
                sk.removeRacun(this);
        }
    }
}
```

# Direktni inženjering

4

## Mapiranje bidirekcione asocijacije \*:\*



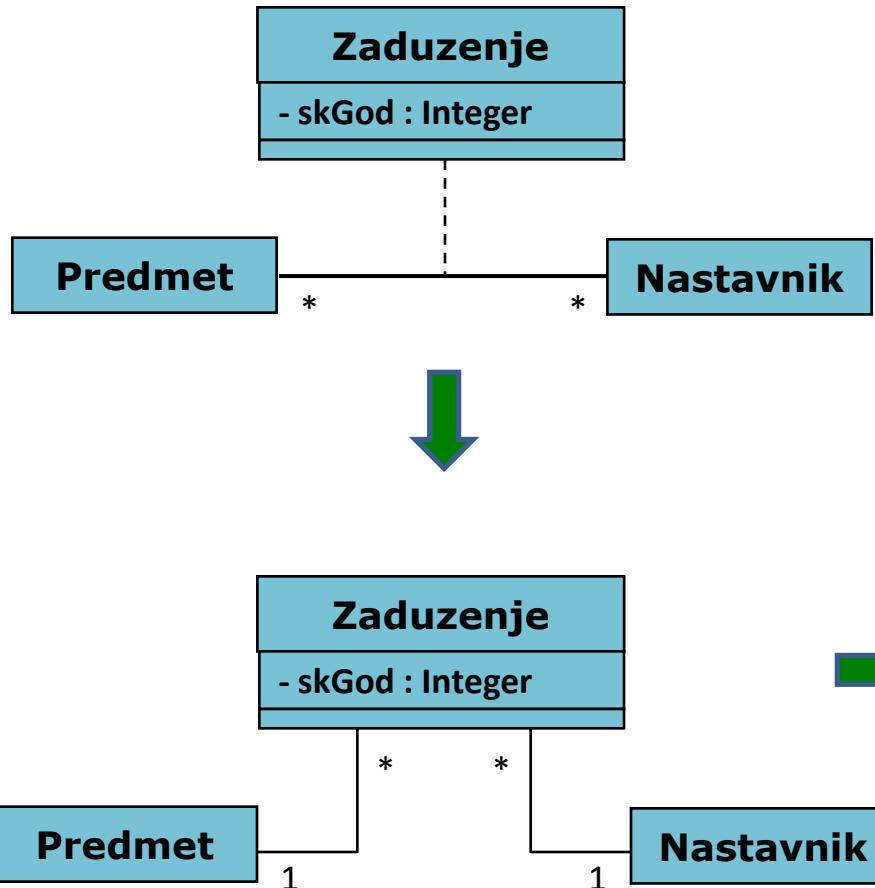
```
public class Turnir
{
    private List igraci;
    public Turnir()
    {
        igraci = new ArrayList();
    }
    public void addIgrac(Igrac i)
    {
        if (!igraci.contains(i))
        {
            igraci.add(i);
            i.addTurnir(this);
        }
    }
}
```

```
public class Igrac
{
    private List turniri;
    public Igrac()
    {
        turniri = new ArrayList();
    }
    public void addTurnir(Turnir t)
    {
        if (!turniri.contains(t))
        {
            turniri.add(t);
            t.addIgrac(this);
        }
    }
}
```

# Direktni inženjering

5

## Mapiranje vezne klase



```
public class Predmet
{
    private Set zaduzenja;
    // ...
}

public class Nastavnik
{
    private Set zaduzenja;
    // ...
}

public class Zaduzenje
{
    private Predmet predmet;
    private Nastavnik nastavnik;
    private Integer skGod;
    // ...
}
```

# Direktni inženjerинг

## Mapiranje kontrakta

- **Programski jezici uglavnom NEMAJU ugrađenu podršku za kontrakte** (automatska validacija ograničenja i podizanje odgovarajućeg izuzetka)
- **MANUELNA IMPLEMENTACIJA** – manuelna validacija ograničenja i podizanje odgovarajućeg izuzetka ako je kontrakt narušen

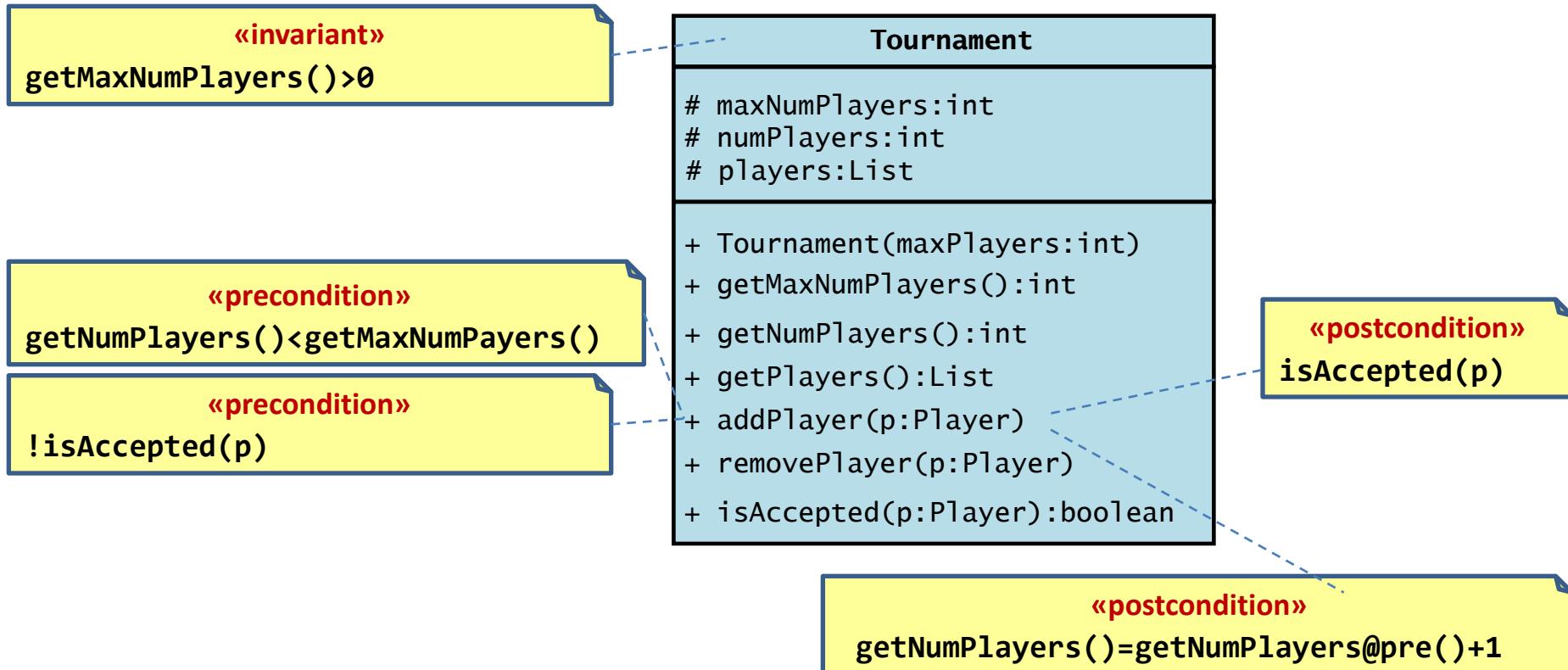
try – throw – catch

## Pravila za mapiranje kontrakta

- **Mapiranje preduslova**
  - provjera ispunjenosti preduslova na početku odgovarajuće metode (prije aplikativne logike),
  - ako neki preduslov nije ispunjen, podići odgovarajući izuzetak da bi korisnik znao koji preduslov nije ispunjen.
- **Mapiranje postuslova**
  - provjera ispunjenosti postuslova na kraju odgovarajuće metode (nakon aplikativne logike),
  - ako neki postuslov nije ispunjen, podići odgovarajući izuzetak da bi korisnik znao koji postuslov nije ispunjen.
- **Mapiranje invarijante**
  - provjera ispunjenosti invarijante tipično na kraju odgovarajuće metode (nakon aplikativne logike i provjere postuslova).

# Direktni inženjering

Primjer (mapiranje kontrakta): UML model sa ograničenjima



# Direktni inženjering

Primjer (mapiranje kontrakta): rezultantni kod

```
public class Tournament
{
    //...
    private List<Player> players;
    public void addPlayer(Player p)
        throws KnownPl, TooManyPls, UnknownPl,
               IllegalNumPls, IllegalMaxNumPls
    {
        // preduslov: !isAccepted(p)
        if(isAccepted(p))
        {
            throw new KnownPl(p);
        }

        // preduslov:
        // getNumPlayers() < maxNumPlayers
        if(getNumPlayers() == getMaxNumPlayers())
        {
            throw new TooManyPls(getNumPlayers());
        }

        // sacuvaj za provjeru postuslova
        int pre_getNumPlayers = getNumPlayers();
    }
}
```

```
// poslovna logika
players.add(p);
p.addTournament(this);

// postuslov: isAccepted(p)
if(!isAccepted(p))
{
    throw new UnknownPl(p);
}

// postuslov
// getNumPlayers()=getNumPlayers@pre()+1
if (getNumPlayers() != pre_getNumPlayers+1)
{
    throw new IllegalNumPls(getNumPlayers());
}

// invarijanta: maxNumPlayers > 0
if (getMaxNumPlayers() <= 0)
{
    throw new
        IllegalMaxNumPls(getMaxNumPlayers());
}
//...
}
```

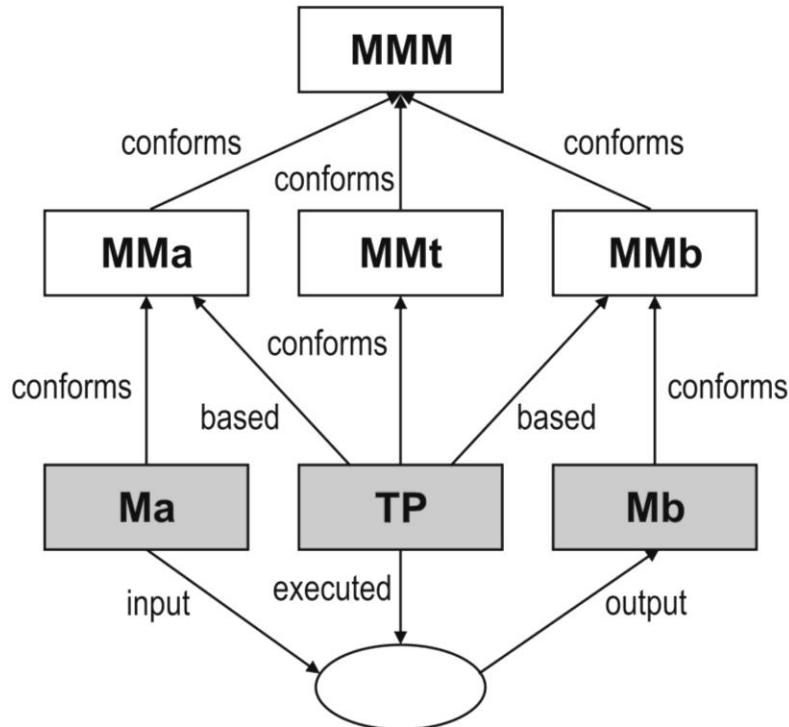
# Direktni inženjering

## Uticaj kontrakta na performanse:

- Dodatni kod za validaciju ograničenja usporava izvršavanje aplikacije
- U slučaju značajne degradacije performansi:
  - izbaciti provjeru u privatnim i zaštićenim metodama
  - ako su performanse i dalje nezadovoljavajuće:
    - redukovati provjeru postuslova i invarijanti
    - fokusirati se na validaciju ograničenja za perzistentne objekte
    - inkapsulirati minimalnu logiku za validaciju ograničenja u zasebne metode

# Model-driven Engineering (MDE)

## Transformacioni MDE obrazac



**Ma** – izvorni model

**Mb** – ciljni model

**TP** – transformacioni program

**MMa** – metamodel ulaznog modela

**MMb** – metamodel izlaznog modela

**MMt** – metamodel transformacionog programa

**MMM** – zajednički metamodel

### Vrste transformacija (po osnovu metamodela):

- **endogene transformacije ( $MMa=MMb$ )**
  - npr. UML CD → UML CD
- **egzogene transformacije ( $MMa \neq MMb$ )**
  - npr. BPMN → UML

### Vrste transformacija (po osnovu broja UI modela):

- **1:1, 1:\*, \*:1, \*:\***

### Vrste transformacija (po osnovu nivoa apstrakcije):

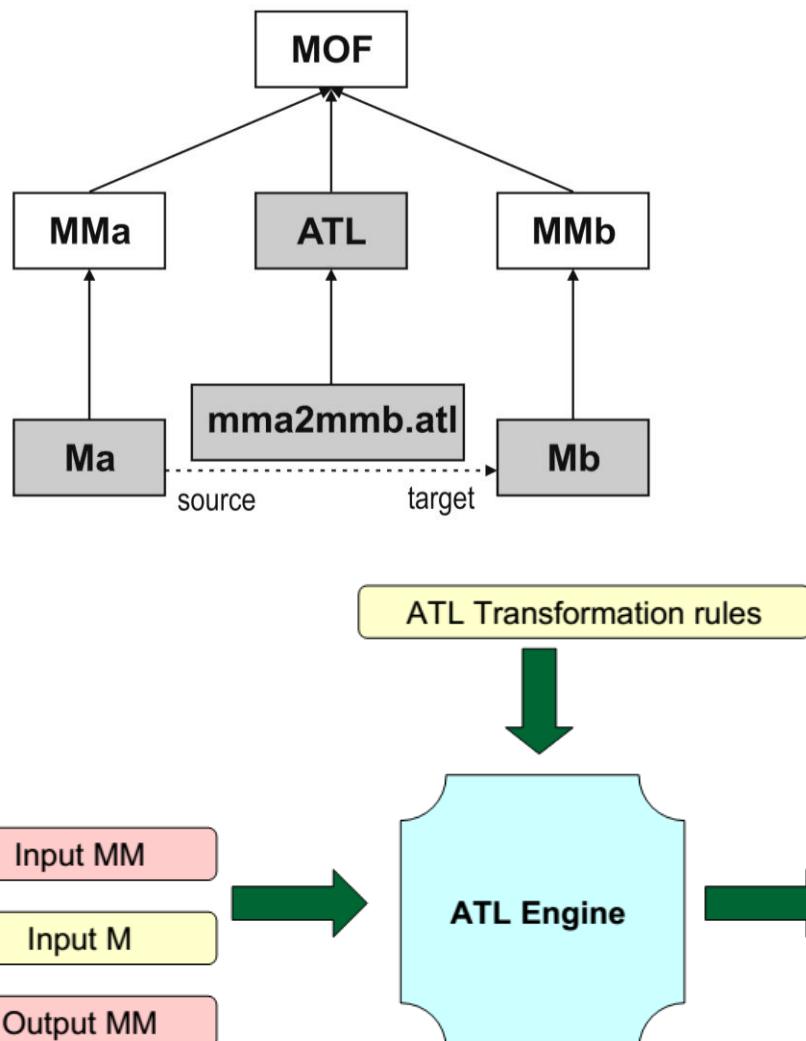
- **vertikalne (različiti nivoi apstrakcije)**
  - CIM → PIM → PSM
- **horizontalne (na istom nivou apstrakcije)**
  - npr. restrukturisanje dijagrama klase

### Specijalizovani transformacioni jezici

- **M2M:** ATL, QVT, TGG, JTL, ETL, ...
- **M2T:** XSLT, MOFScript, JET, Acceleo, ...

# ATL

## ATL transformacioni obrazac



### Osnovne karakteristike:

- **Hibridni M2M transformacioni jezik**
  - deklarativni elementi – **matched rules**
    - za jednostavnija mapiranja
  - imperativni elementi – **called rules**
    - za složenija mapiranja
- **Transformacioni program = ATL modul**
  - **ATL modul** = kolekcija transformacionih pravila

```
-- zaglavlje ATL modula  
module mma2mmbr  
create mb:mmbr from ma:mmbr
```

# ATL

## Struktura ATL modula (.atl)

### – header

- naziv modula
- deklaracija ulaznog i izlaznog modela

### – import (opciono)

- uključivanje biblioteka

### – helpers (opciono)

- definicija pomoćnih promjenljivih i funkcija

### – rules

- definicija transformacionih pravila

```
-- zaglavje ATL modula
-- @nsURI uml=http://www.eclipse.org/uml2/3.0.0/UML
module ad2cd;
create cd:uml from ad:uml;

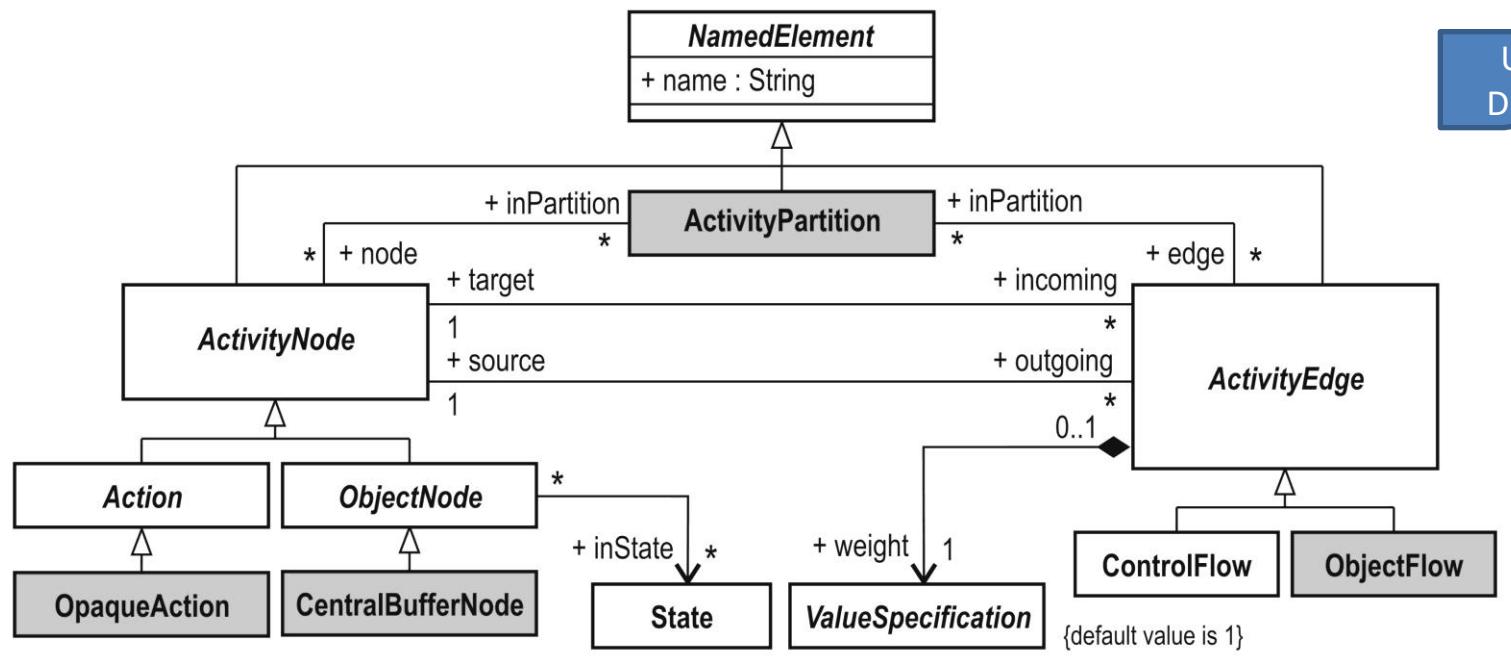
-- import
uses strings;

-- attribute helper
helper def: model : uml!Model = OclUndefined;

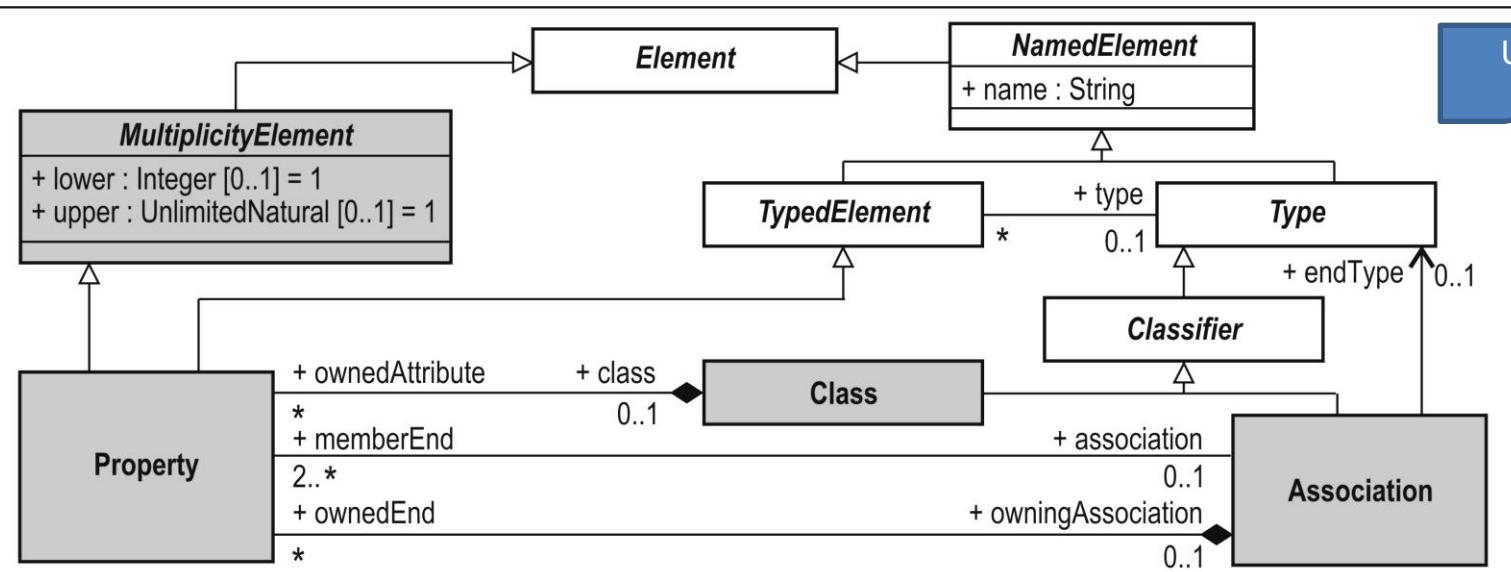
-- function helper
helper def: inkrement(i:Integer):Integer = i+1;

-- rule
rule ActivityPartition
{
    from
        s : uml!ActivityPartition
    to
        d : uml!Class ( name <- s.name )
}
```

UML metamodel:  
Dijagram aktivnosti



UML metamodel:  
Dijagram klasa



# ATL

## Transformaciona pravila

### matched rules

- deklarativna pravila
- primjenjuju se za sve elemente ulaznog modela koji su odgovarajućeg (matched) tipa (specificuje se klauzulom **from**)
- na osnovu datog elementa ulaznog modela kreira odgovarajući(e) element(e) u izlaznom modelu (specificuje se klauzulom **to**)
- moguće i dodavanje imperativnog dijela (specificuje se klauzulom **do**)

```
-- matched rule
rule ActivityPartition
{
    from
        s : uml!ActivityPartition
    to
        d : uml!Class
            ( name <- s.name )
}
```

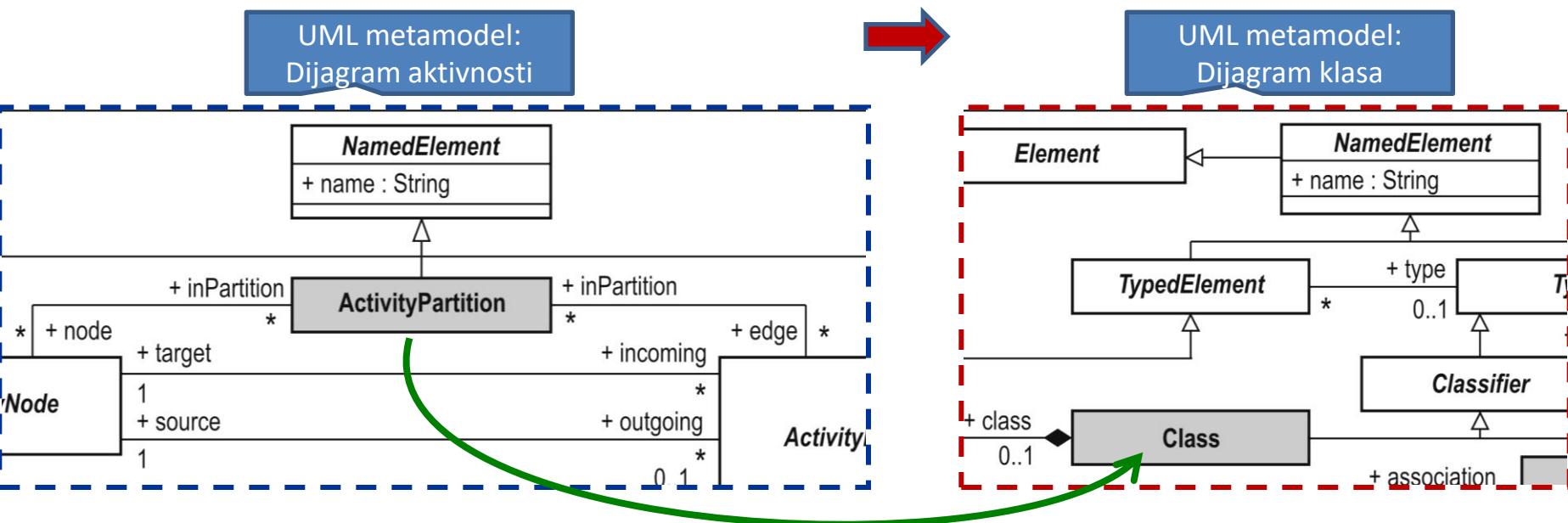
Naziv pravila

Tip elementa iz ulaznog modela na koji se primjenjuje pravilo

Tip elementa koji se kreira u izlaznom modelu

Inicijalizacija atributa generisanog elementa

```
-- matched rule
rule ActivityPartition
{
    from
        s : uml!ActivityPartition
    to
        d : uml!Class
            ( name <- s.name )
}
```



# ATL

## Transformaciona pravila called rules

- imperativna pravila
- izvršavaju se po pozivu (nemaju klauzulu **from**), iz imperativnog dijela matched pravila ili iz nekog drugog called pravila
- mogu da imaju argumente
- omogućavaju generisanje odgovarajućeg(ih) elemen(a)ta u izlaznom modelu
  - specifikuje se klauzulom **to**
- moguće i dodavanje imperativnog dijela
  - specifikuje se klauzulom **do**

```
-- matched rule koji poziva called rule
rule ActivityPartition
{
    from
        s : uml!ActivityPartition
    to
        d : uml!Class ( name <- s.name )
    do
    {
        thisModule.addPK(d);
    }
}
```

```
-- called rule
rule addPK (class : uml!Element)
{
    to
        pk : uml!Property ( name <- 'id' )
    do
    {
        class.ownedAttribute <- pk;
    }
}
```

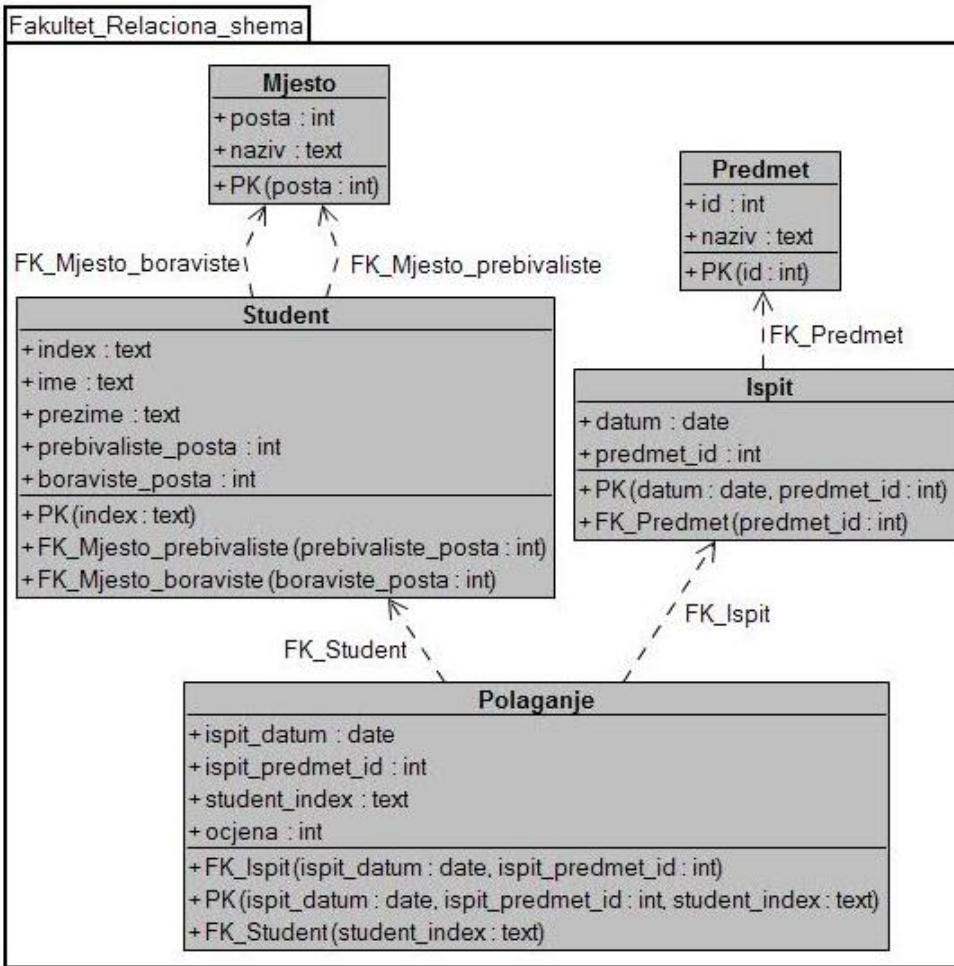
# Acceleo

## M2T transformacioni jezik

- Acceleo je Eclipse open-source generator kôda
- Acceleo implementira MOFM2T standard za transformaciju modela u tekst
- Acceleo je dio Eclipse fondacije od 2009. godine
- Acceleo omogućava da se kao polazni model za generisanje teksta koristi bilo kakav EMF baziran model (UML, SysML, ...)
- Acceleo je zasnovan na formalizmu OCL-a
- Generisanje koda zasnovano je primjeni obrazaca (template)

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/uml2/3.0.0/UML')]
[template public generateElement(aPackage : Package)]
[comment @main/]
[file (aPackage.name.concat('.ddl'), false, 'UTF-8')]
CREATE SCHEMA [aPackage.name];
  [for (aClass:Class | aPackage.ownedElement) after('\n')]
CREATE TABLE [aPackage.name.concat('.').concat(aClass.name)/]
(
  [for (aProperty:Property | aClass.ownedAttribute)
separator('\n') after('\n')]
    [aProperty.name/] [aProperty.type.name/][if (aProperty.lower>0)]
NOT NULL[/if],[/for]
  [for (aOperation:Operation | aClass.ownedOperation->
      select(o | o.name.startsWith('FK') or
o.name.equalsIgnoreCase('PK')) )
    separator(',',\n') after('\n')]
    [if (aOperation.name.equalsIgnoreCase('PK'))]
      PRIMARY KEY ([for (op:Parameter | aOperation.ownedParameter)
        separator(', ',')[op.name/][/for])][/if]
    [if (aOperation.name.startsWith('FK'))]
      CONSTRAINT [aOperation.name/]
        FOREIGN KEY ([for (op:Parameter | aOperation.ownedParameter)
        separator(', ',')[op.name/][/for])
        REFERENCES [for (aDependency:Dependency |
aPackage.ownedElement)]
        [if (aDependency.name.equalsIgnoreCase(aOperation.name))]
          [aPackage.name.concat('.')] [/][aDependency.supplier.name/][/if][/for]
        ]
      [for (r:Constraint | aOperation.ownedRule)] [r.name/][/for][/if]
      [/for]
    );
  [/for]
  [/file]
  [/template]
```

# Acceleo



```

CREATE SCHEMA Fakultet_Relaciona_shema;
CREATE TABLE Fakultet_Relaciona_shema.Student (
    index text NOT NULL,
    ime text NOT NULL,
    prezime text NOT NULL,
    prebivaliste_posta int NOT NULL,
    boraviste_posta int NOT NULL,
    PRIMARY KEY (index),
    CONSTRAINT FK_Mjesto_prebivaliste
        FOREIGN KEY (prebivaliste_posta)
        REFERENCES Fakultet_Relaciona_shema.Mjesto
        ON DELETE RESTRICT ON UPDATE CASCADE,
    CONSTRAINT FK_Mjesto_boraviste
        FOREIGN KEY (boraviste_posta)
        REFERENCES Fakultet_Relaciona_shema.Mjesto
        ON DELETE RESTRICT ON UPDATE CASCADE
);
CREATE TABLE Fakultet_Relaciona_shema.Predmet (
    id int NOT NULL,
    naziv text NOT NULL,
    PRIMARY KEY (id)
);
...
  
```