UNIVERZITET U BANJOJ LUCI
ELEKTROTEHNIČKI FAKULTET

Prof. dr Dražen Brđanin

OBJEKTNO-ORIJENTISANI 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

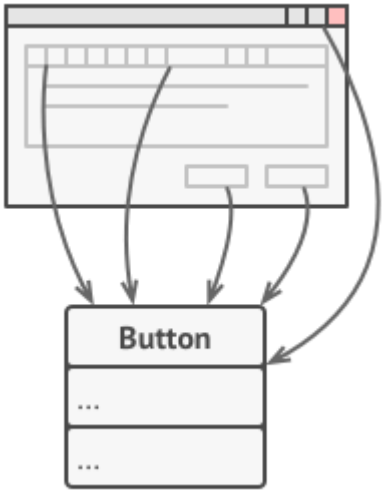
naziv	kratak opis
Command (Komanda)	**Inkapsulacija komande u objekat**
Memento (Podsjetnik, Podsjećanje)	**Pamćenje i restauriranje stanja objekta**
Iterator (Brojač)	**Sekvencijalni pristup elementima kolekcije**
State (Stanje)	**Promjena ponašanja objekta u zavisnosti od promjene stanja**
Observer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

Obrasci ponašanja - Komanda

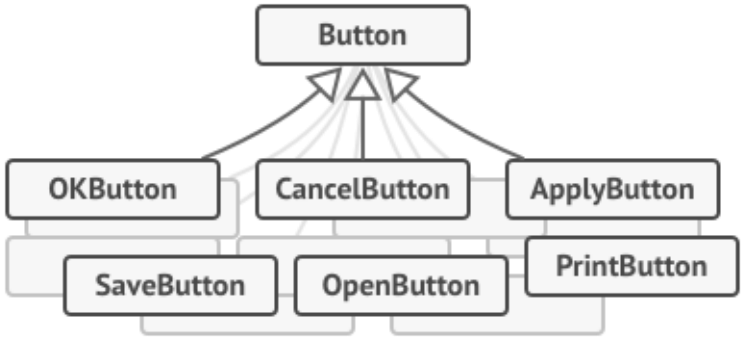
Motivacija za uvođenje projektnog obrasca Komanda

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 repezentujemo odgovarajućom potklasom



Ogroman broj potklasa

+ 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

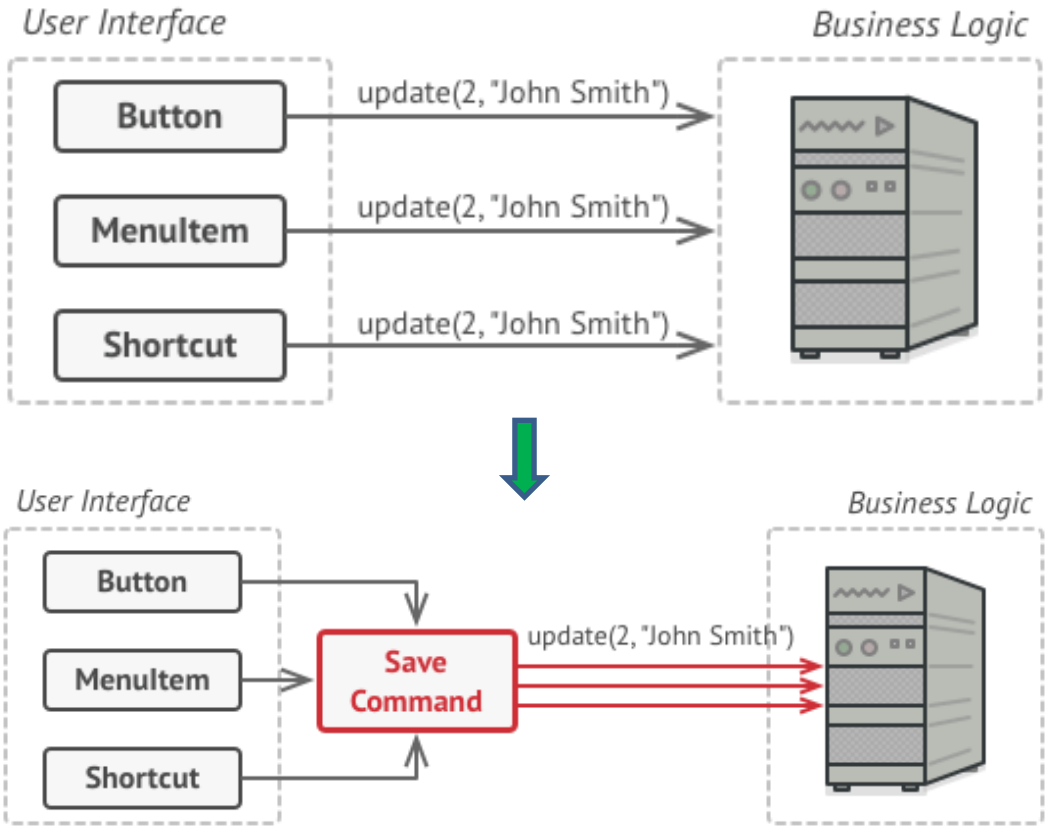
Obrasci ponašanja - Komanda

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

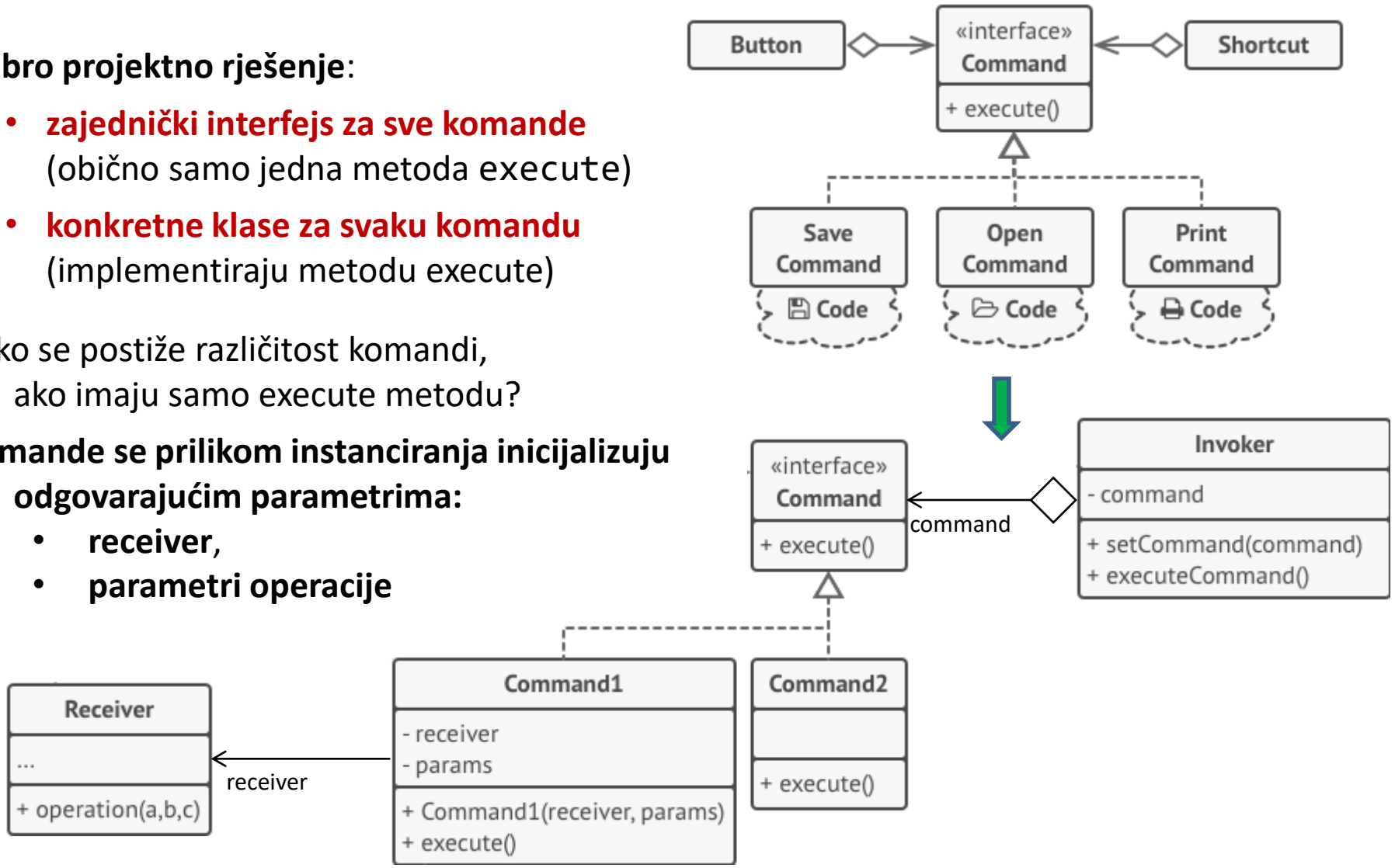
Dobro projektno rješenje:

- **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?

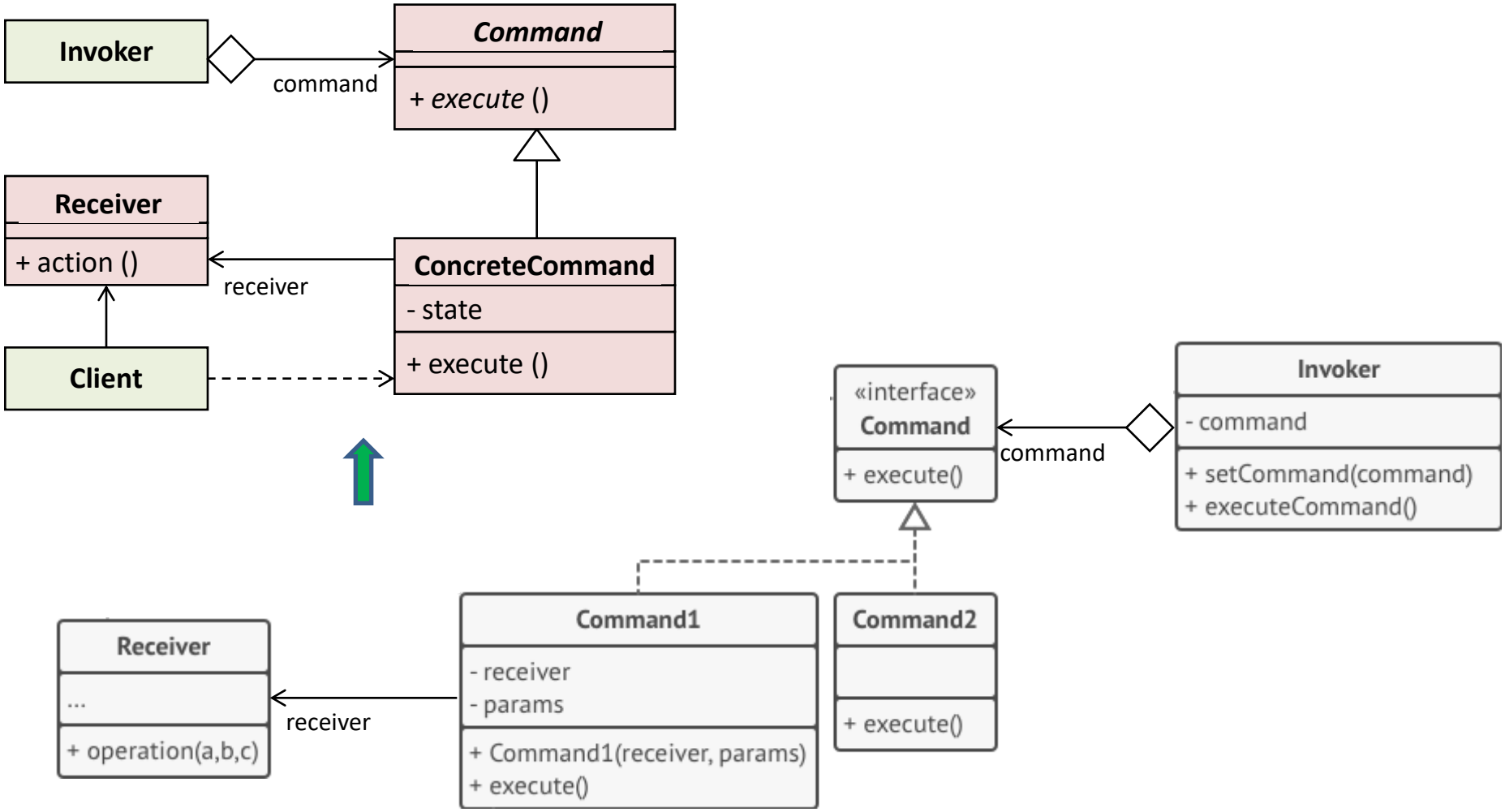
Komande se prilikom instanciranja inicijalizuju odgovarajućim parametrima:

- **receiver**,
- **parametri operacije**



Obrasci ponašanja - Komanda

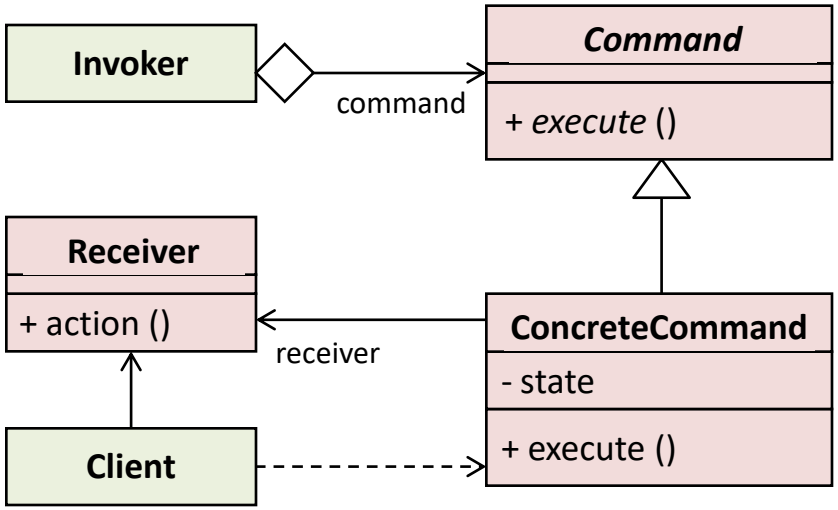
Motivacija za uvođenje projektnog obrasca Komanda



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
– implemetira **execute()** metodu pozivajući odgovarajuće operacije u klasi **Receiver**

receiver.action()

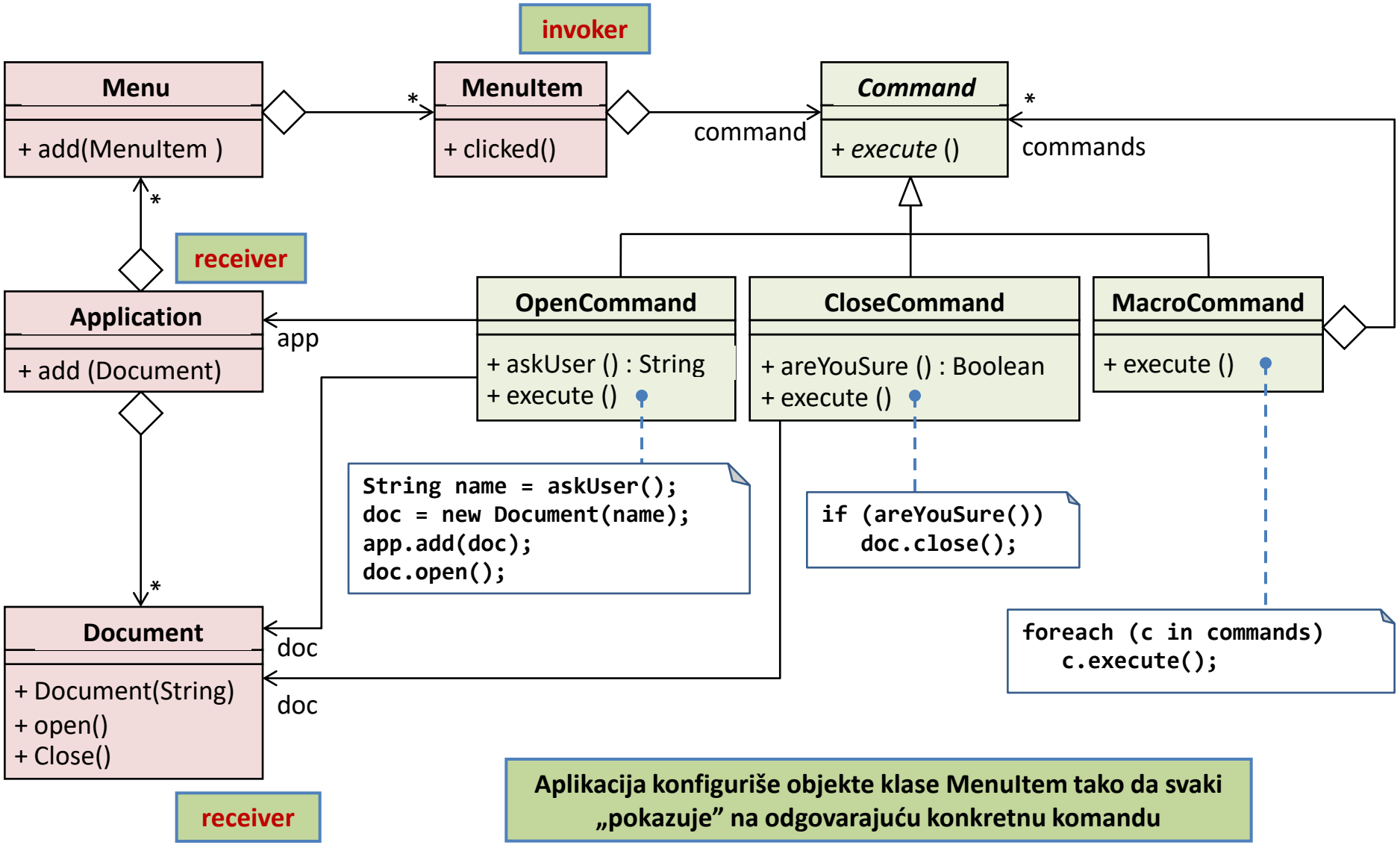
Client
– kreira objekat klase **ConcreteCommand** i postavlja objekat klase **Receiver** koji će prihvatiti njegove pozive

Invoker
– traži da mu komanda obradi zahtjev

Receiver
– izvršava operaciju na zahtjev komande

Obrasci ponašanja - Komanda

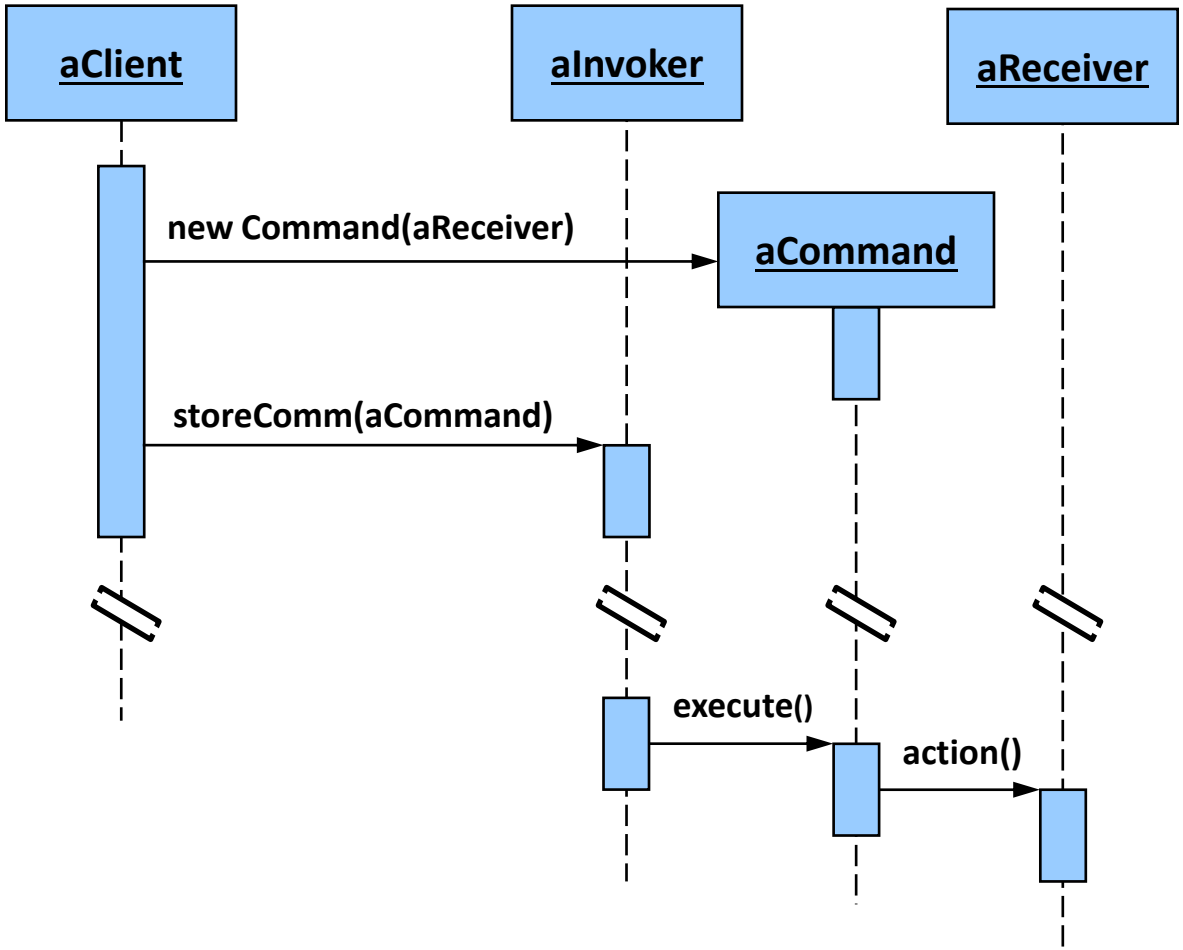
Primjer (Realizacija menija zasnovana na COMMAND obrascu):



Obrasci ponašanja - Komanda

Implementacioni detalji

– kolaboracija objekata



Klijent **kreira** ***komandu*** **i specifikuje njen** ***receiver*****.**

Klijent **konfiguriše** ***invoker*** **kreiranom** ***komandom*****.**

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

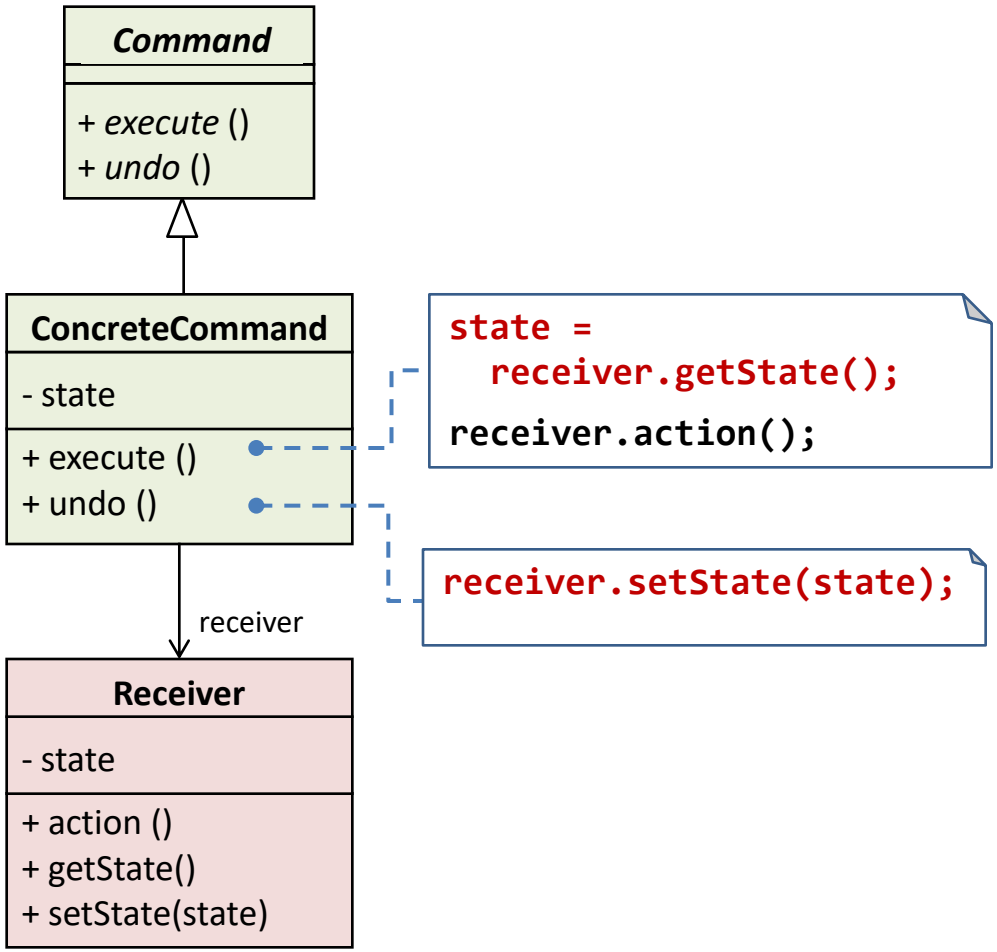
Komanda **inicira izvršavanje odgovarajuće operacije na** ***receiveru*****.**

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

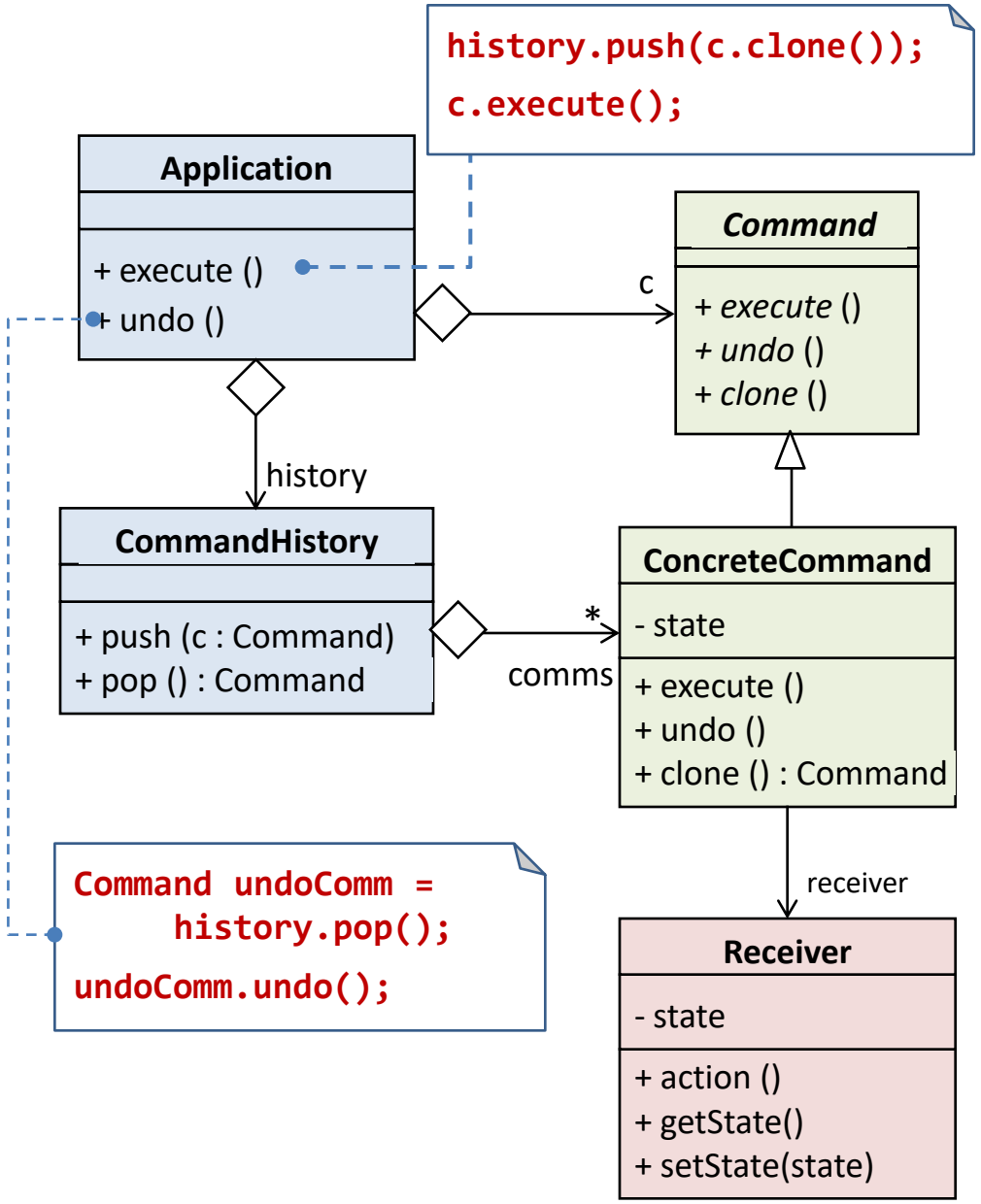


Obrasci ponašanja - Komanda

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

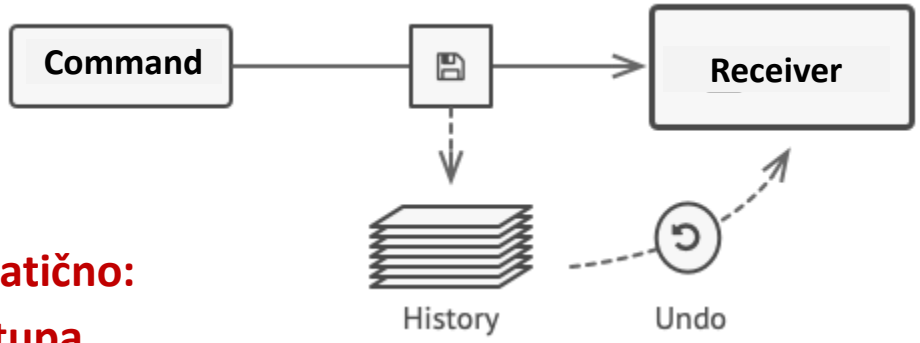


Obrasci ponašanja - Memento

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:

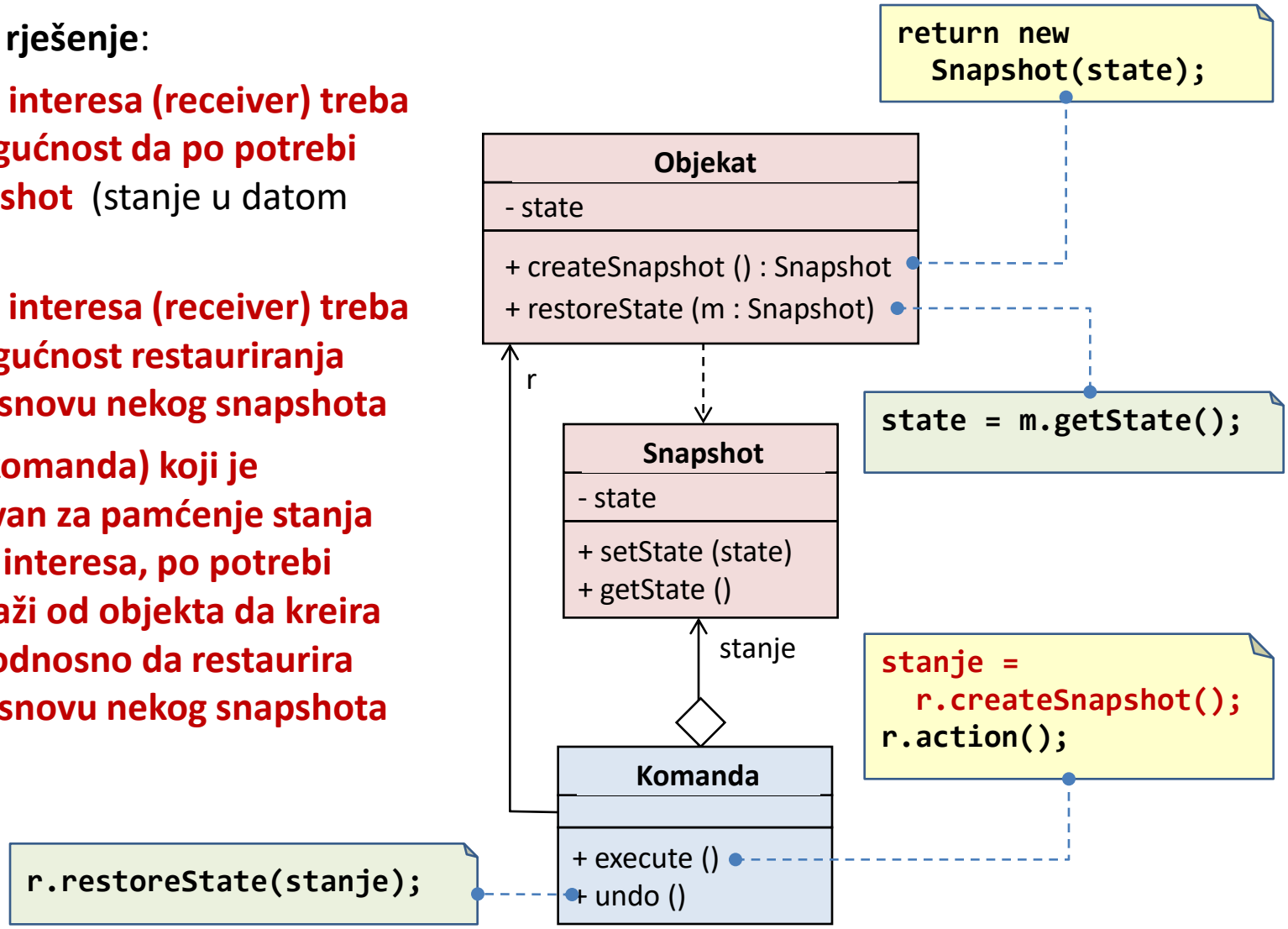
- **Možda objekat ne omogućava da neko pristupa njegovim atributima i čita stanje!**
- **Koje atribute treba memorisati i kako da to realizuju različite komande, pogotovo ako komande mogu da se primjenjuju nad različitim objektima? (svaka komanda mora da memoriše stanje receivera neposredno prije izvršenja)**

Obrasci ponašanja - Memento

Motivacija za uvođenje projektnog obrasca Memento

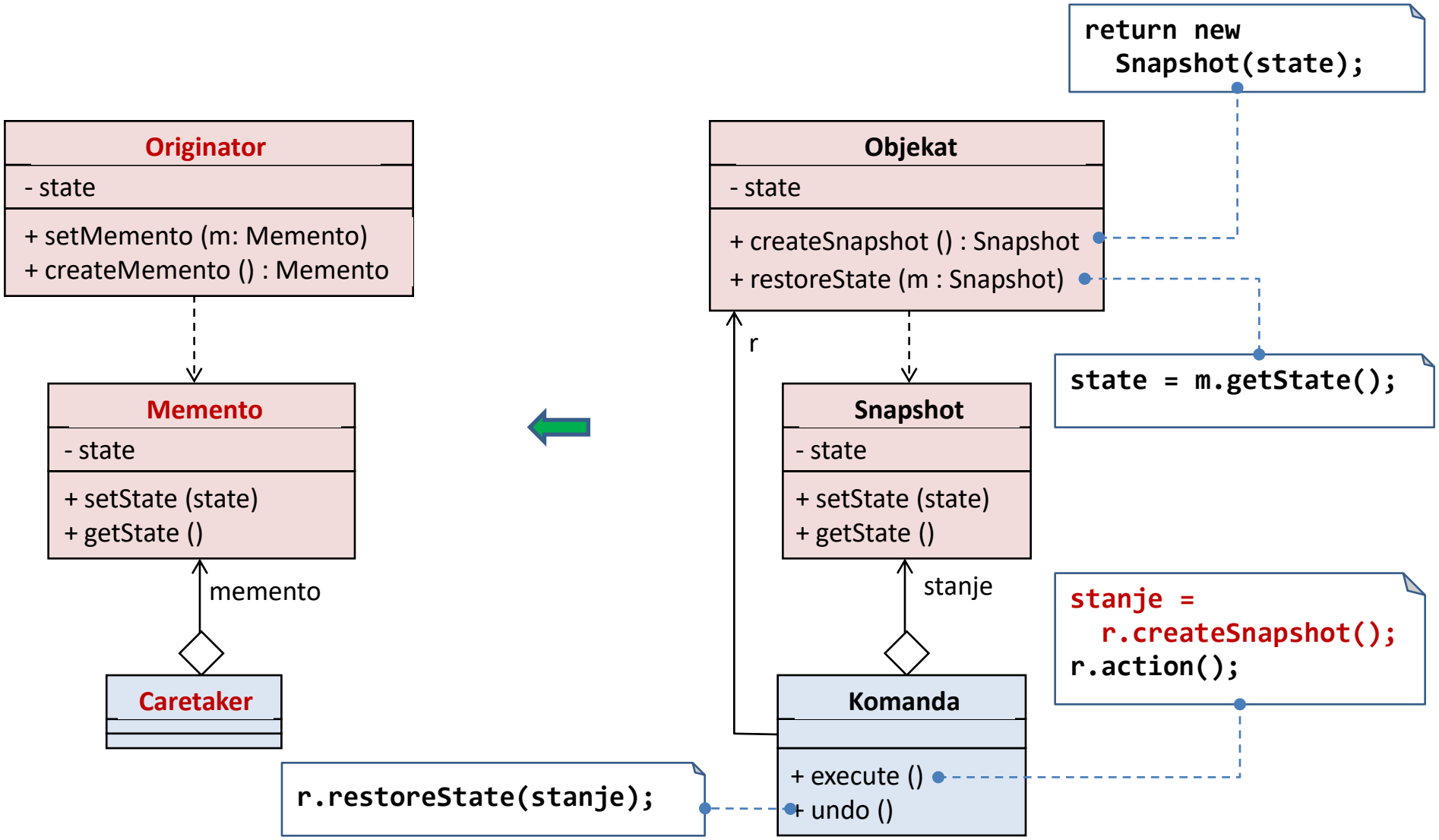
Dobro projektno rješenje:

- **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

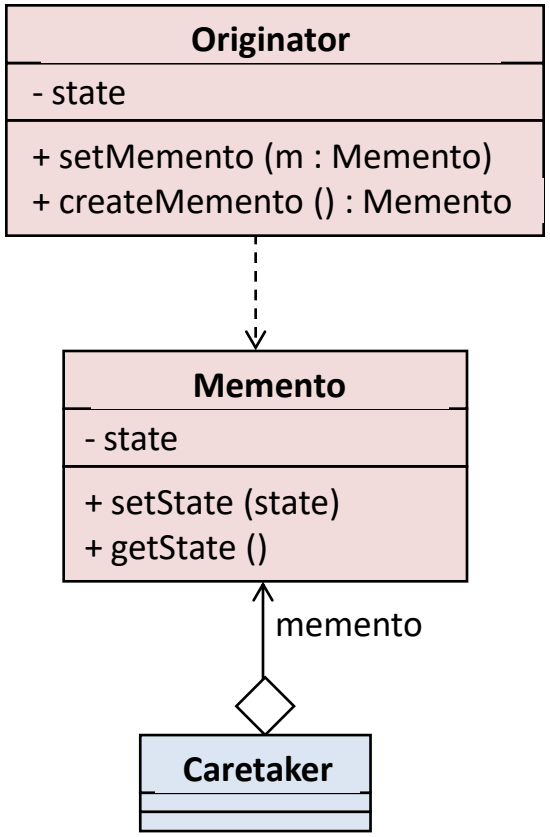
Motivacija za uvođenje projektnog obrasca Memento



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

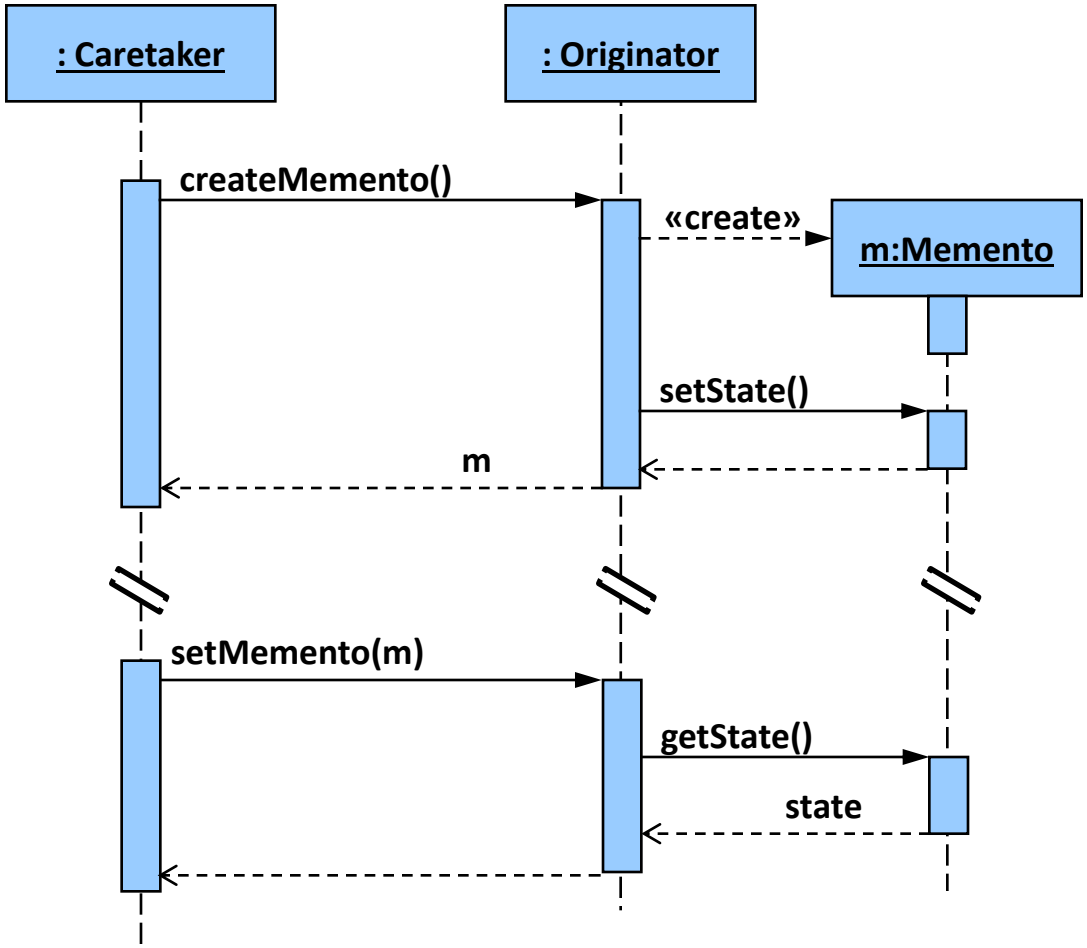
– Kreira Memento (snimak njegovog trenutnog stanja).
– Koristi Memento da resturira svoje unutrašnje stanje.

Caretaker

– Odgovoran za čuvanje Mementa.
– Nikada ne ispituje niti otvara sadržaj Mementa.

Obrasci ponašanja - Memento

Tipična kolaboracija

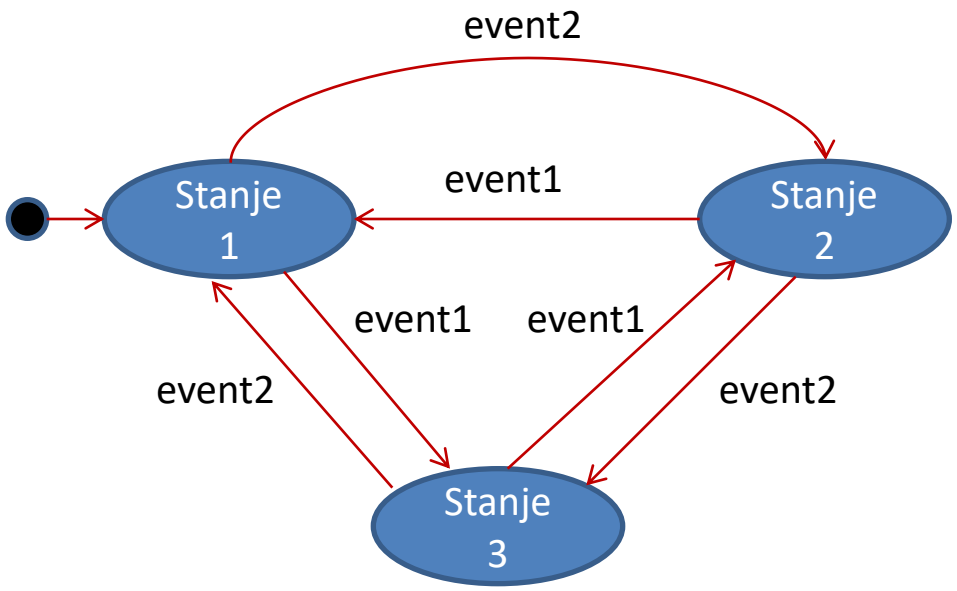


- Caretaker zahtijeva da originator kreira memento, kako bi se kasnije moglo vratiti staro stanje.
- Kad zatreba restauracija starog stanja, caretaker zahtijeva od originatora da restaurira stanje na osnovu mementa.
- Memento je pasivan – samo originator može da ga kreira i restaurira svoje stanje na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

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)

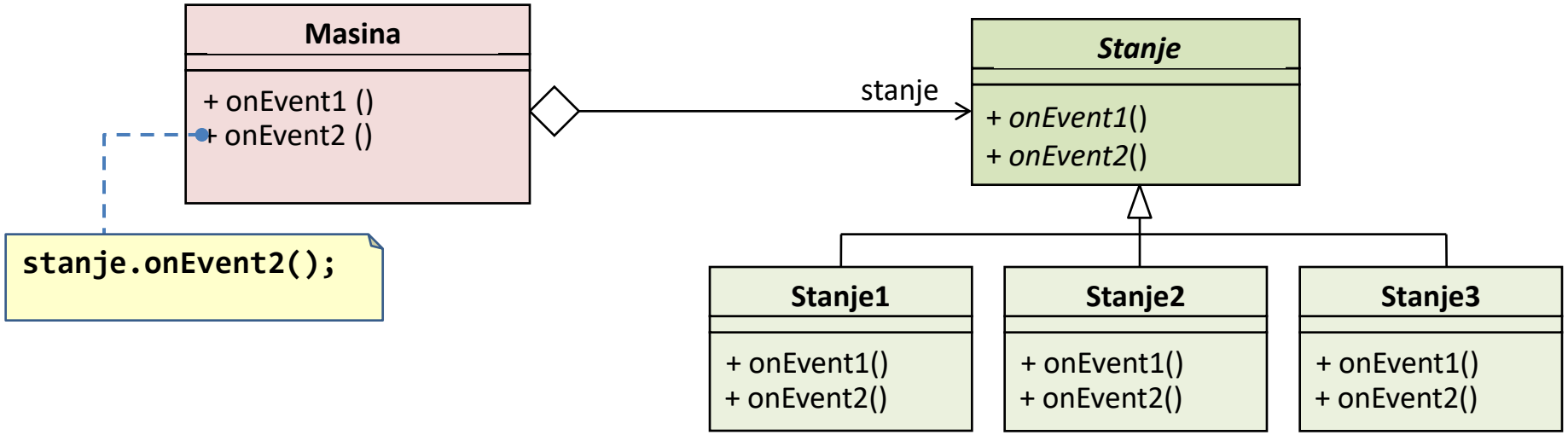


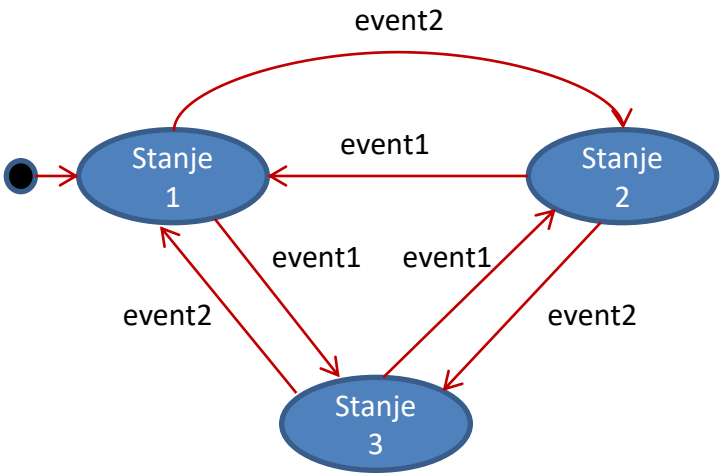
Ponašanje objekta na neki događaj zavisi od stanja u kojem se nalazi, što može da rezultuje metodama čija implementacija sadrži dosta grananja i provjera većeg broja uslova:

```
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





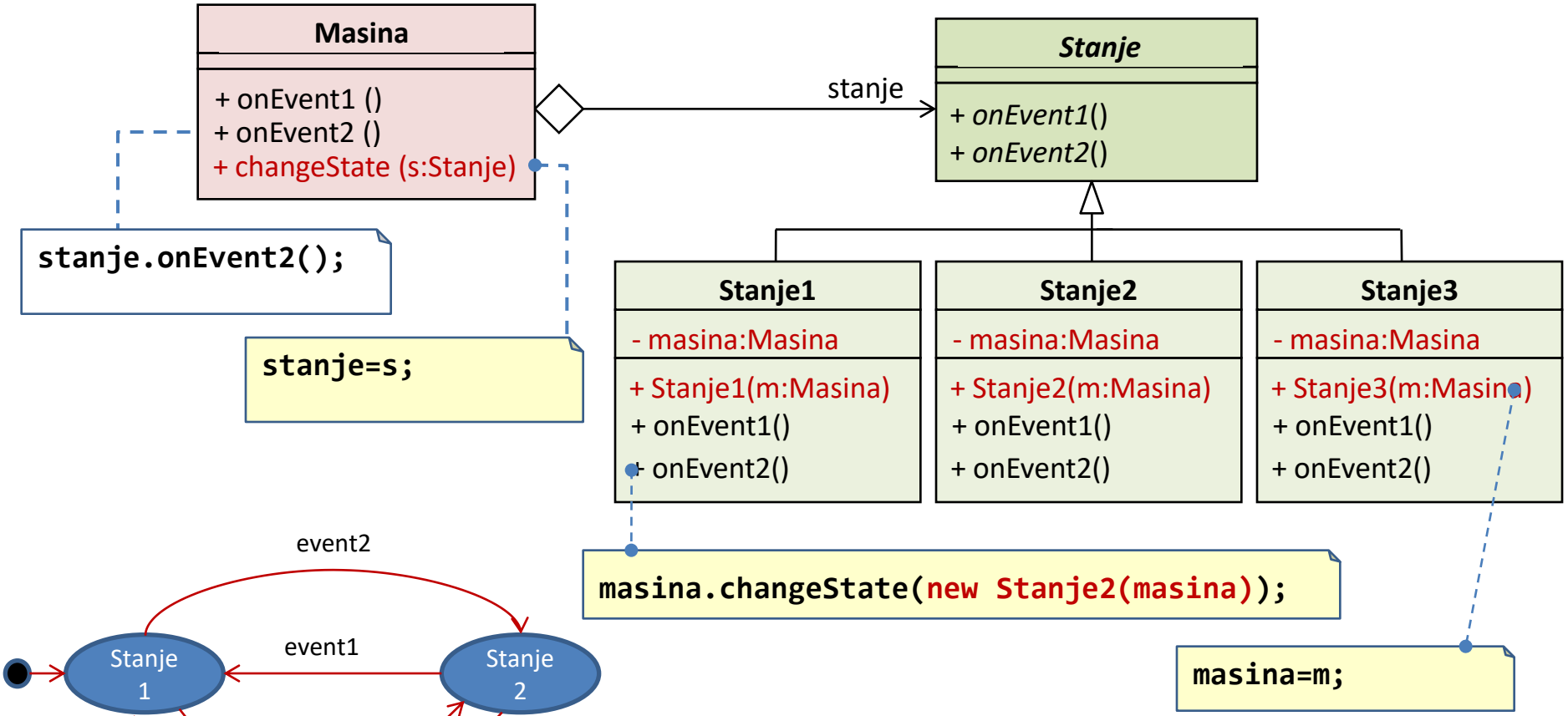
Dobro projektno rješenje jeste izvlačenje aplikativne logike iz klase koja reprezentuje objekat u posebne klase – za svako stanje po jedna klasa koja implementira ponašanje karakteristično za dato stanje.

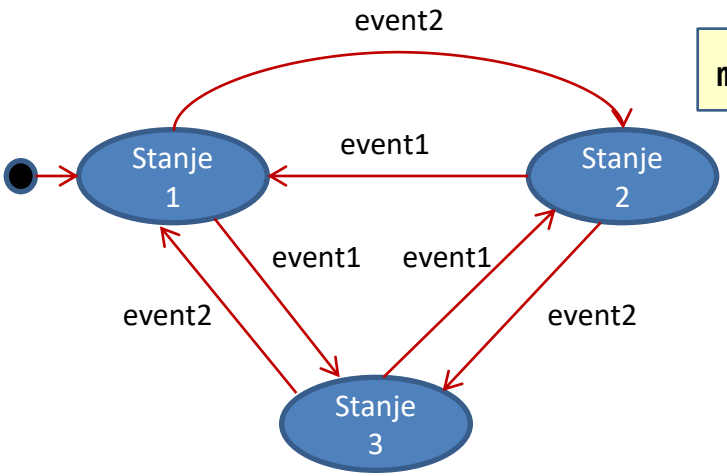
Objekat ima referencu na trenutno stanje i reagovanje na događaj svodi se na delegiranje poziva objektu koji reprezentuje trenutno stanje

```
stanje.onEvent();
```

Obrasci ponašanja - State

Motivacija za uvođenje projektnog obrasca State





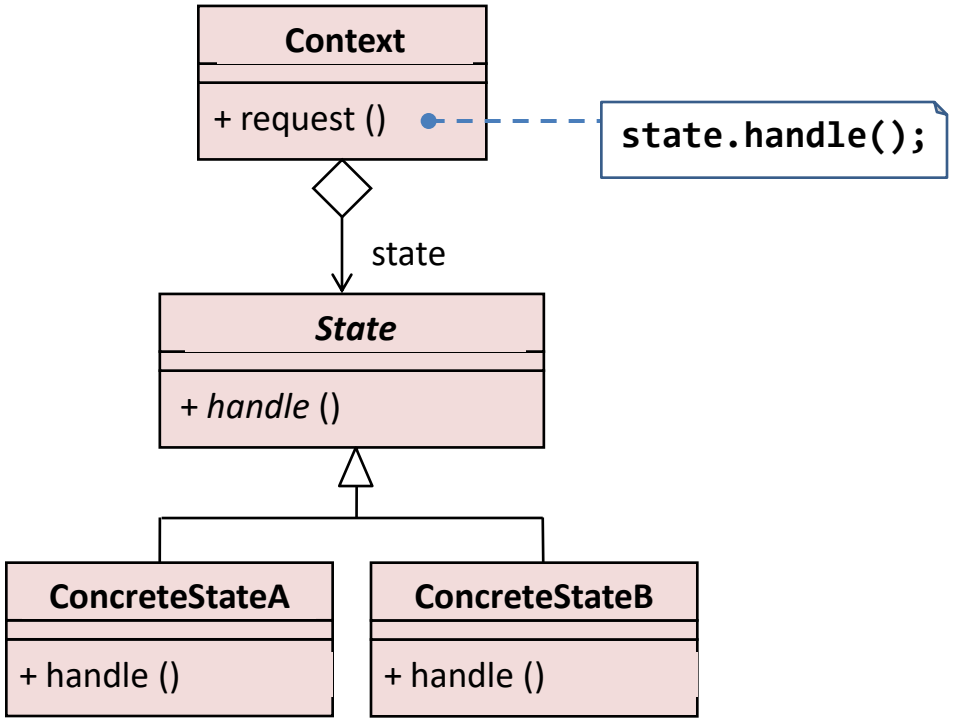
Jedan od načina da se realizuje promjena stanja:

•u metodi koja implementira reagovanje na događaj, instancira se novo stanje i pozove metoda u objektu koja mijenja stanje

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?

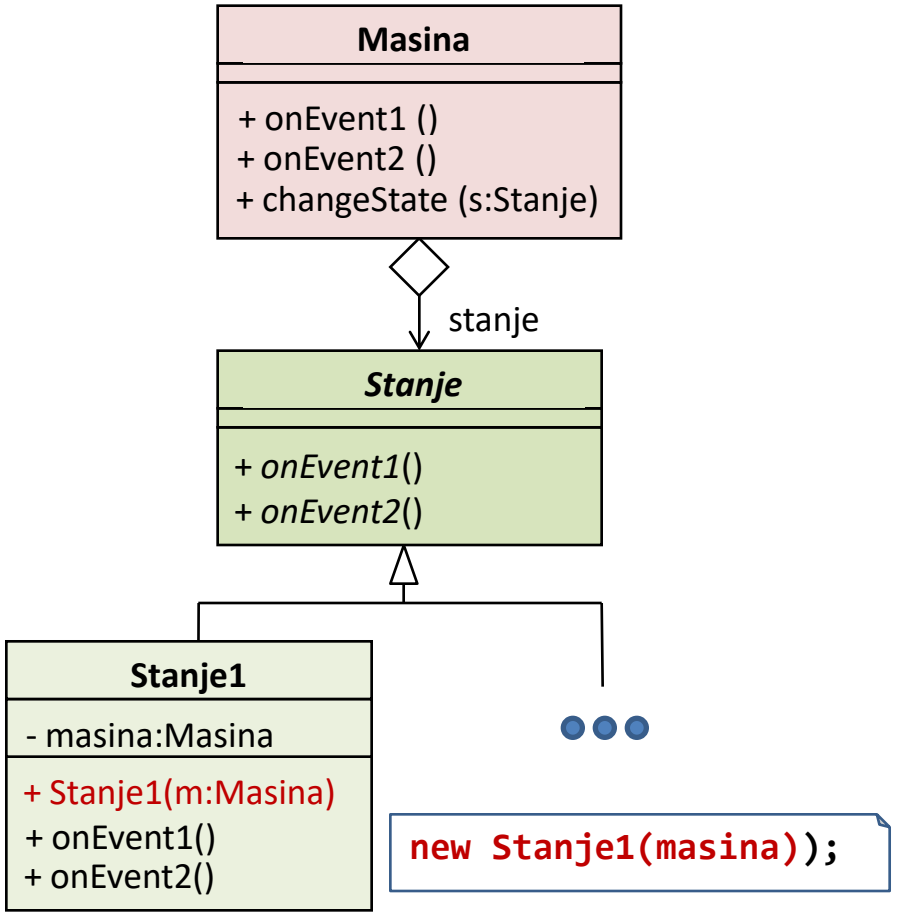
– Kad ponašanje objekta zavisi od stanja u kojem se nalazi i mora da se promijeni u toku izvršavanja zavisno od promjene stanja.
– Kad je implementacija operacija uveliko zavisna od stanja objekta (mnogo uslovnih iskaza, ...).

Obrasci ponašanja - State

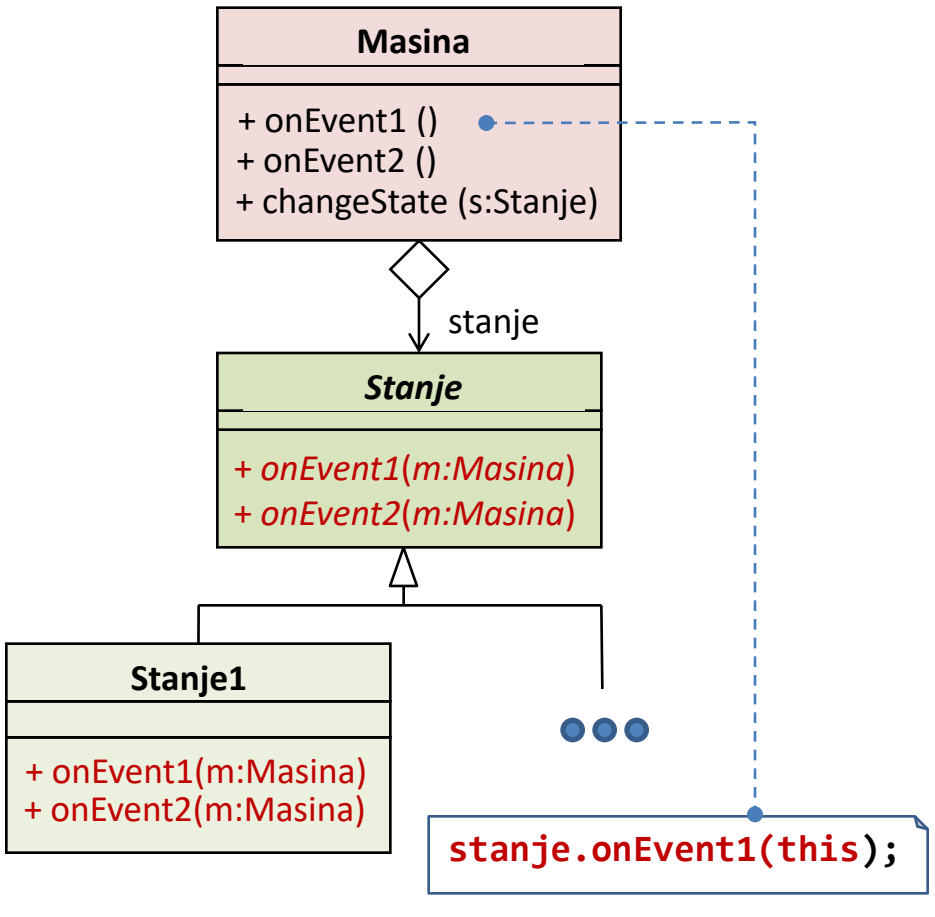
Implementacioni detalji – referenca na Context

– **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

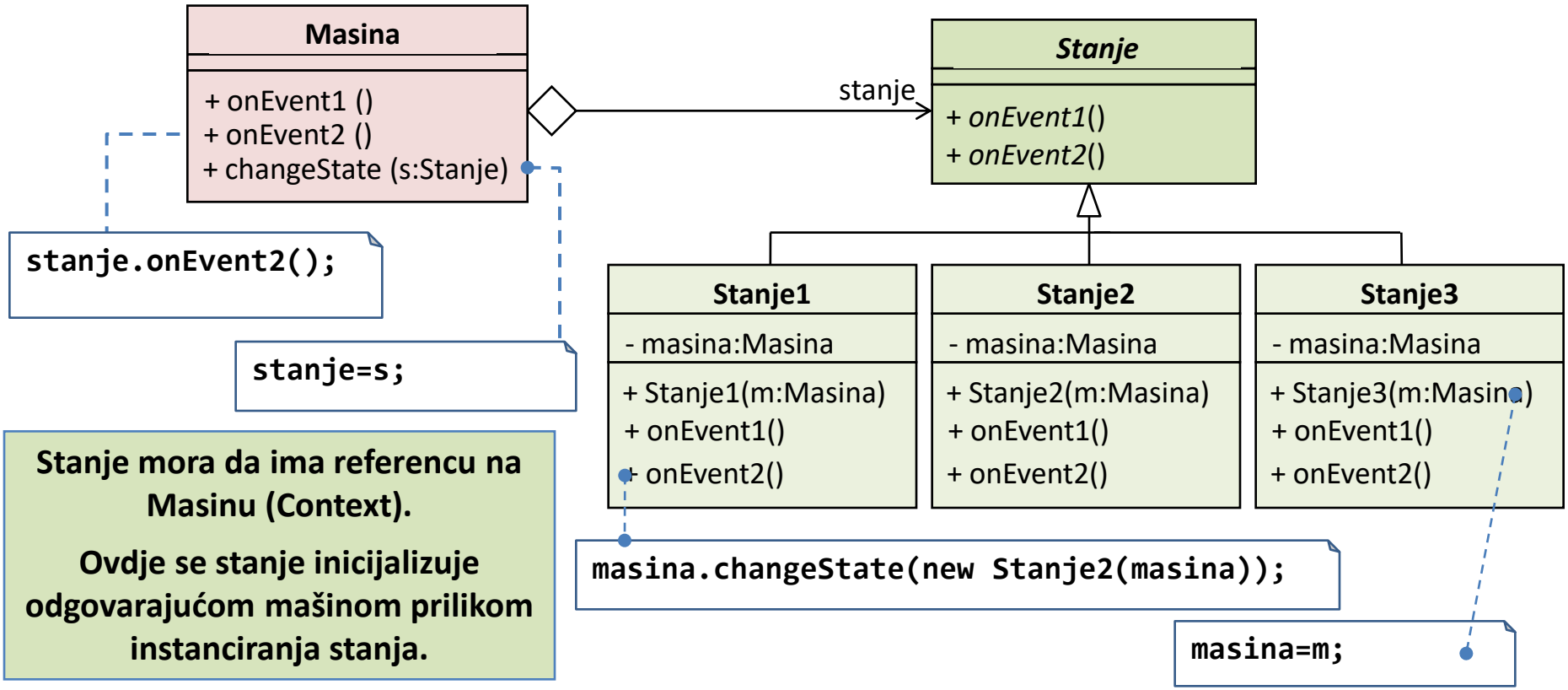
Implementacioni detalji – Promjene stanja

– STATE obrazac ne specifikuje odgovornost za promjenu stanja objekta

– Implementacija kriterijuma za promjenu stanja: **decentralizovano** / **centralizovano**

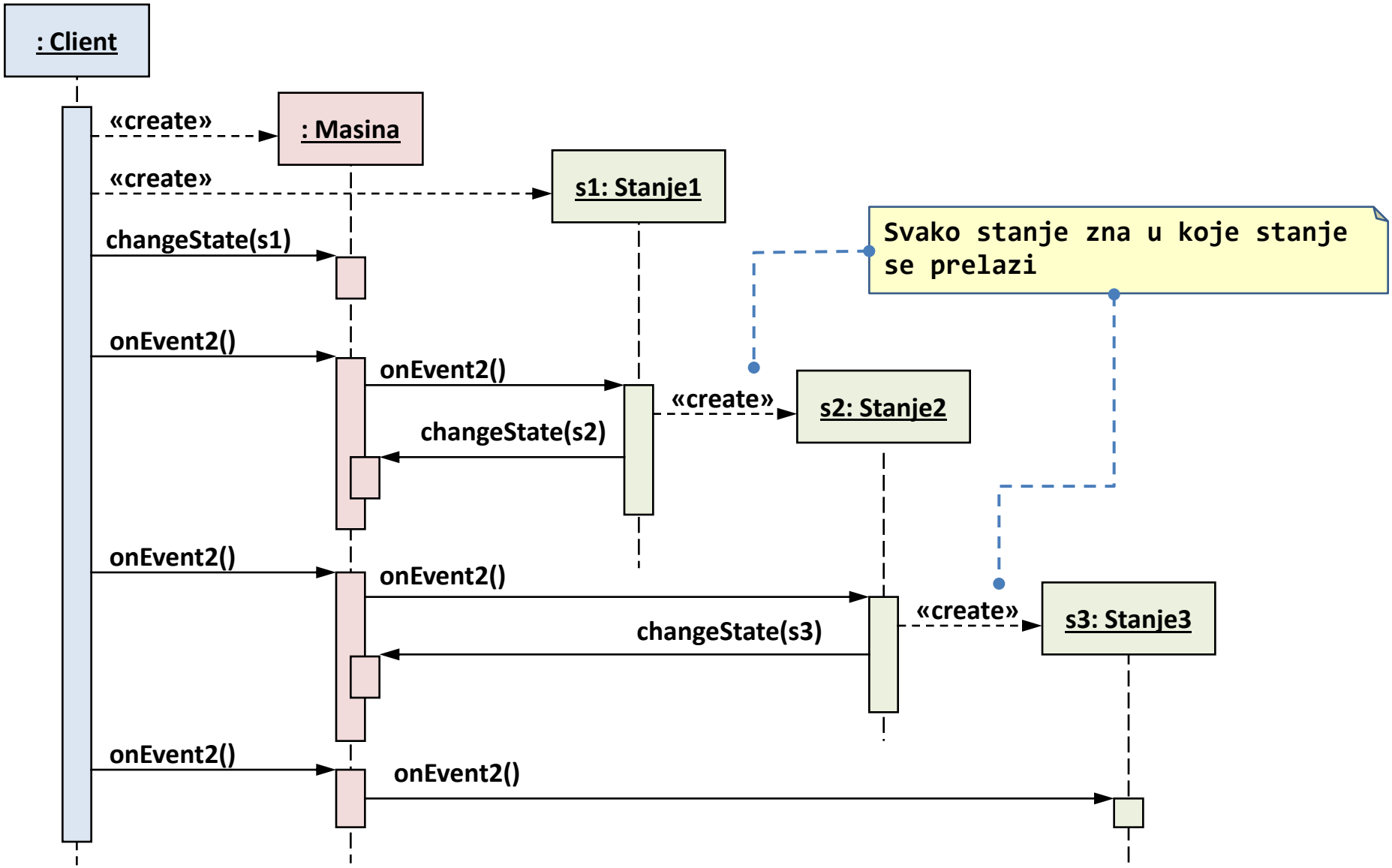
Decentralizovana kontrola promjene stanja

– svako konkretno stanje definiše kriterijume za tranziciju u sljedeće stanje



Obrasci ponašanja - State

Interakcija objekata u decentralizovanoj kontroli promjene stanja

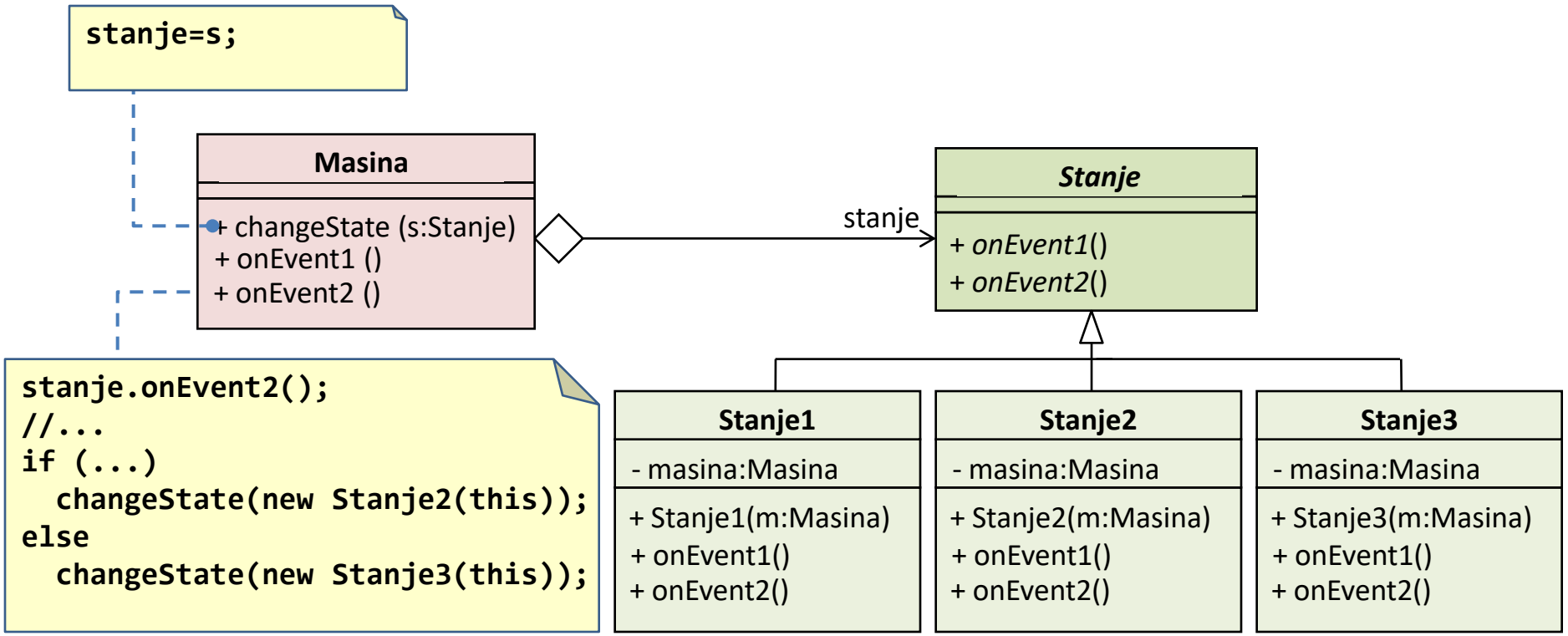


Obrasci ponašanja - State

Centralizovana kontrola promjene stanja

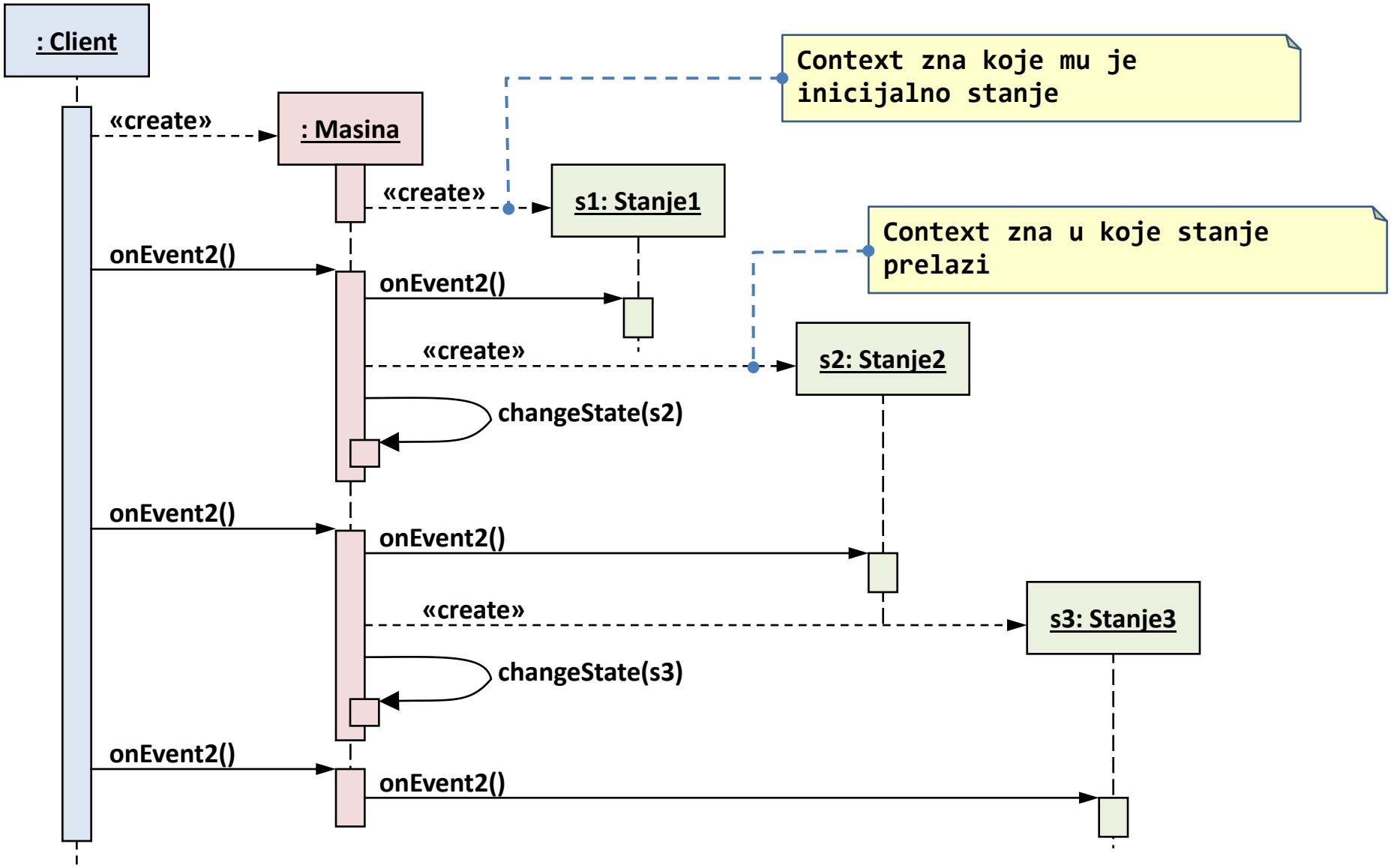
– kontrola promjene stanja realizuje se unutar Context klase
(kontekst instancira odgovarajuće stanje i sam vrši promjenu stanja)

– 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:

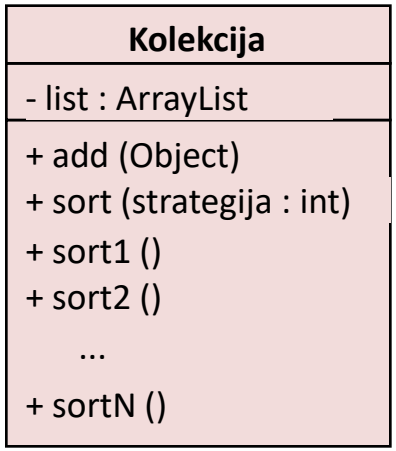
- **ako nemamo mogućnost modifikovanja klase i dodavanje odgovarajućih metoda;**
- **sav kod je u jednoj klasi (i reprezentacija kolekcije i aplikativna manipulacija;**
- **dodavanjem novih strategija klasa raste pa je teže održavanje; ...**

```
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;
        // ...
     }
  }
}
```

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

Obrasci ponašanja - Strategy

Motivacija za uvođenje projektnog obrasca Strategy




```
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;
        // ...
     }
  }
}
```

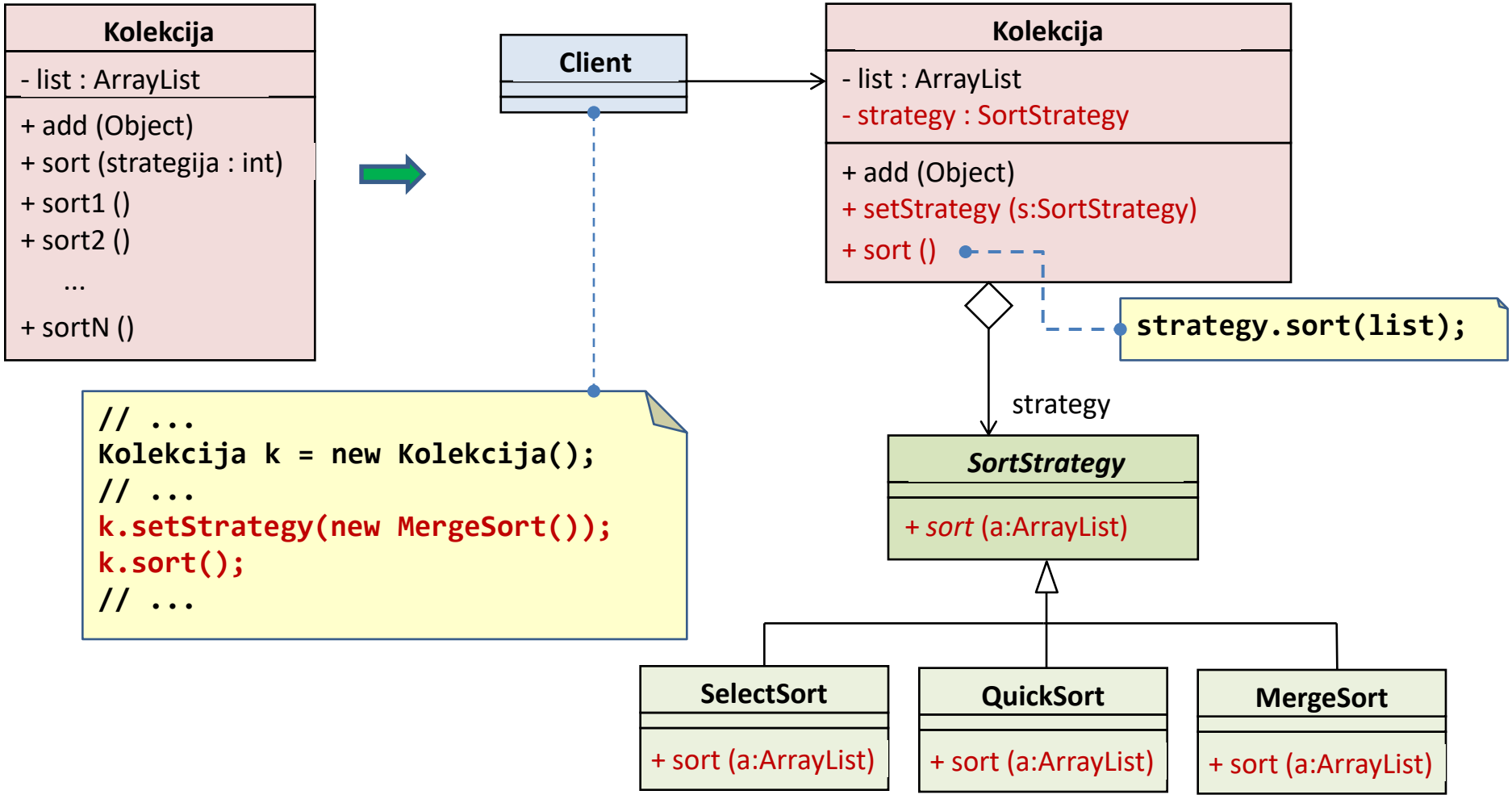
Problemi:

- **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; ...**

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

Obrasci ponašanja - Strategy

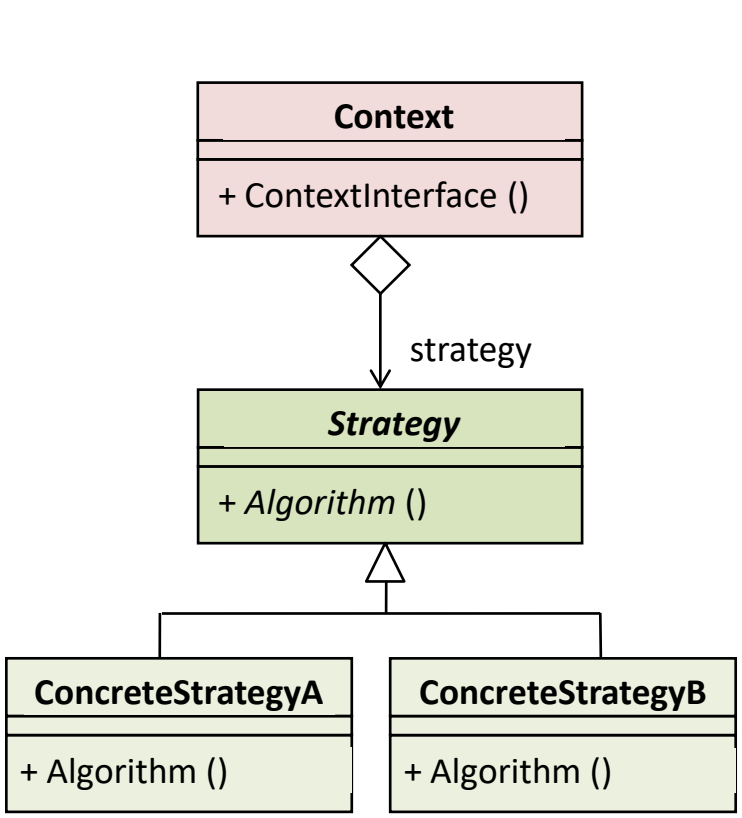
Motivacija za uvođenje projektnog obrasca Strategy



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

Obrasci ponašanja - Strategy

Motivacija za uvođenje projektnog obrasca Strategy

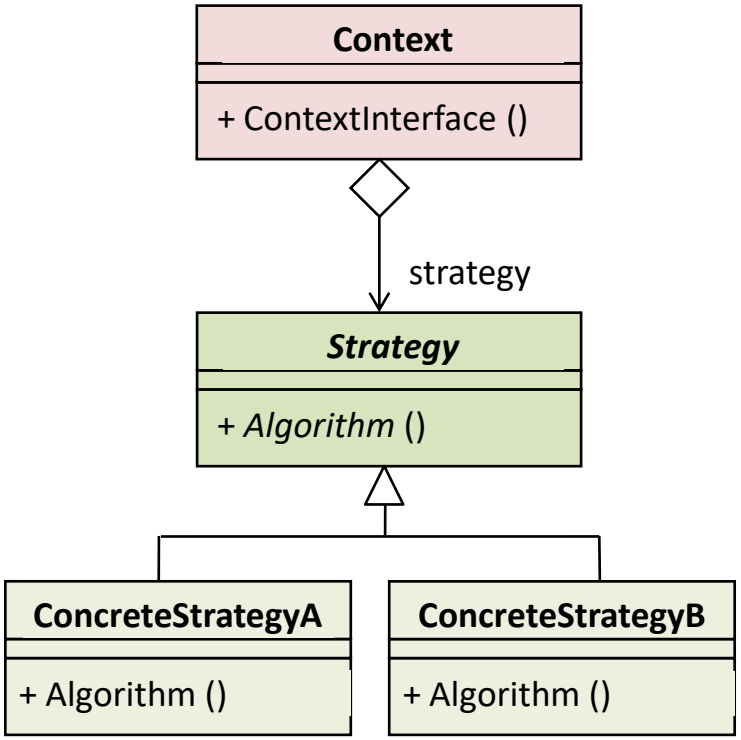


Dobro projektno rješenje: izdvajanje aplikativnog koda za svaku strategiju u posebnu klasu

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?

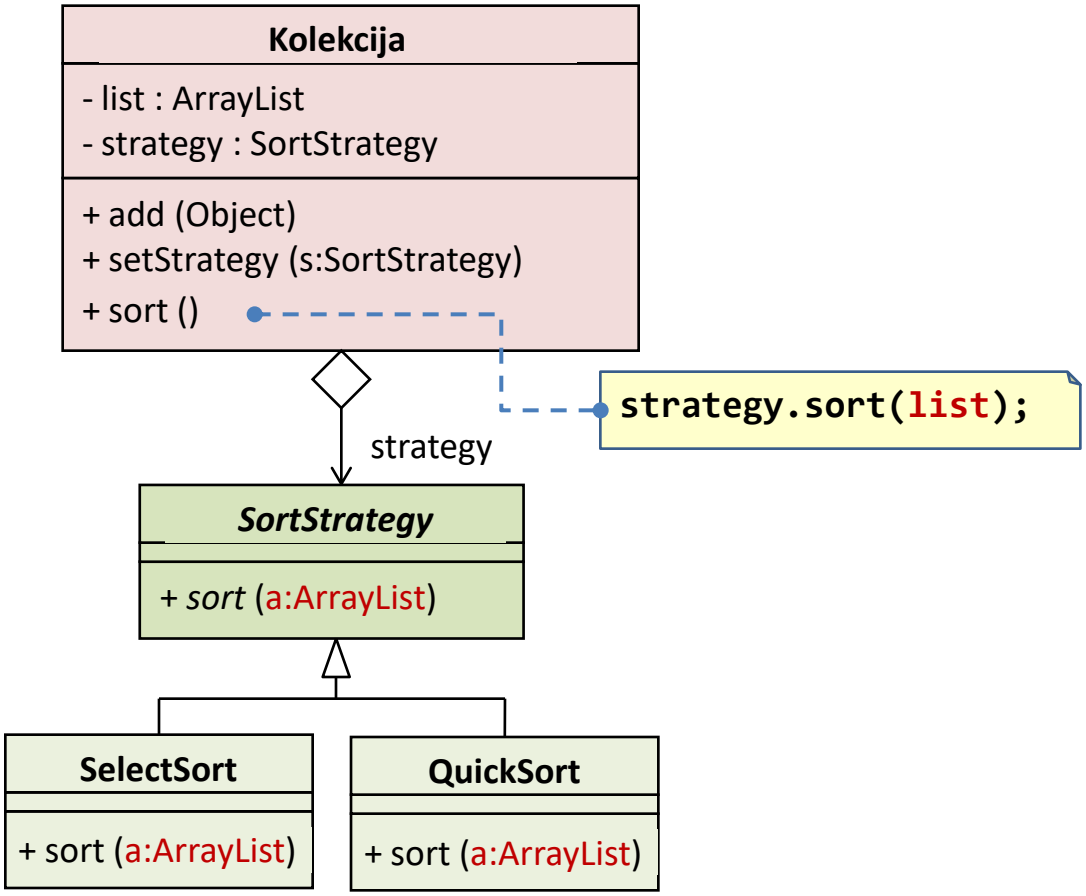
– Kad se više klasa razlikuje samo u ponašanju. Strategija omogućava konfigurisanje objekta jednim od više mogućih ponašanja.
– Kad postoji više varijacija nekog algoritma (s obzirom na to kakve su željene performanse).

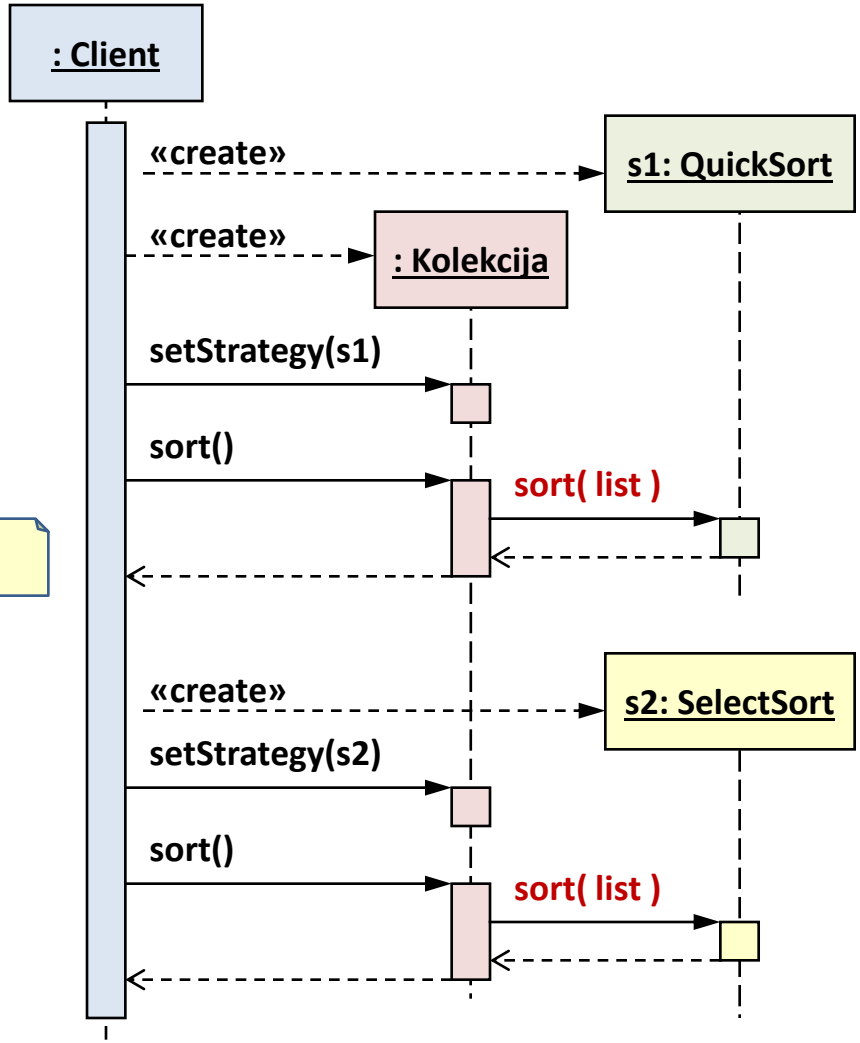
Obrasci ponašanja - Strategy

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

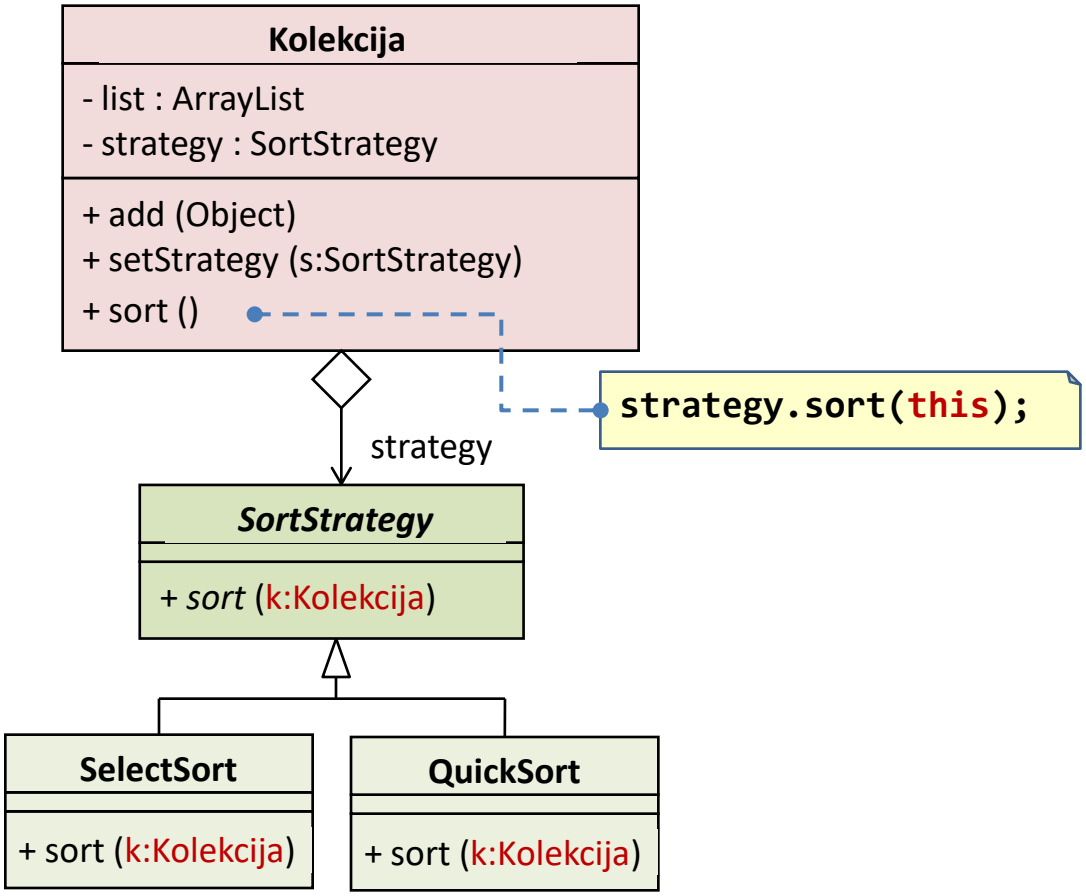


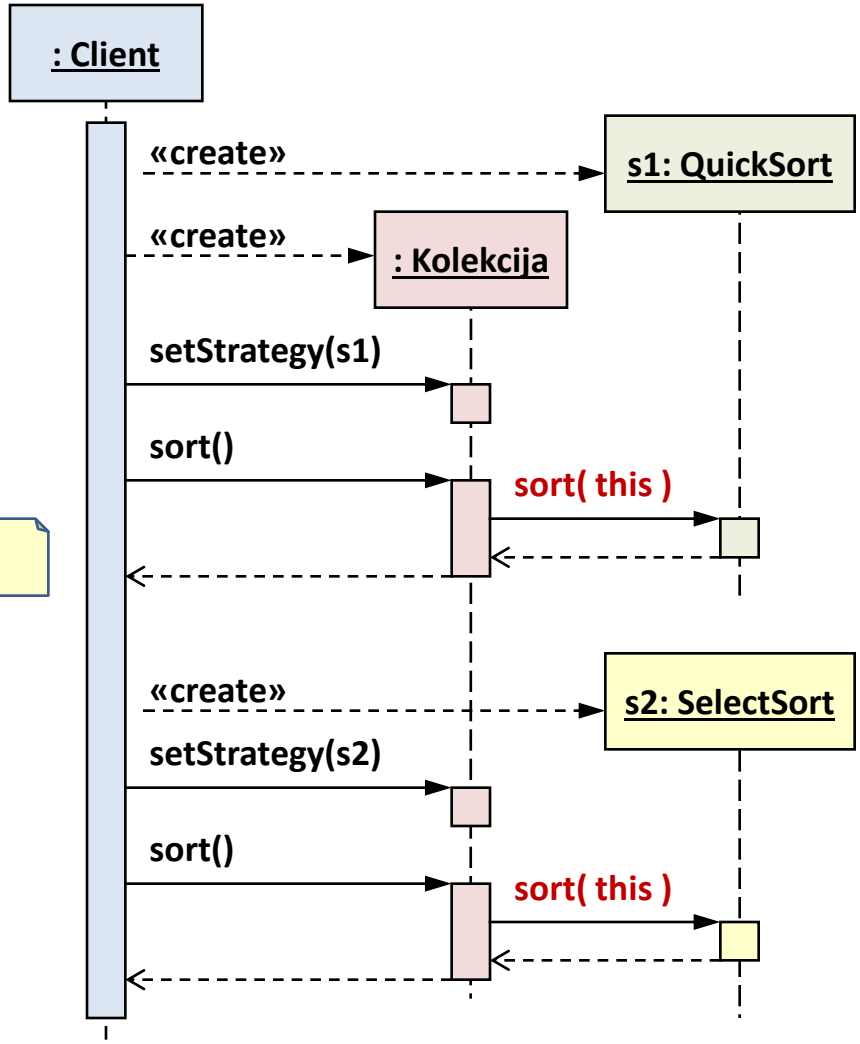


Obrasci ponašanja - Strategy

Implementacioni detalji – prosljeđivanje podataka u strategiju

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

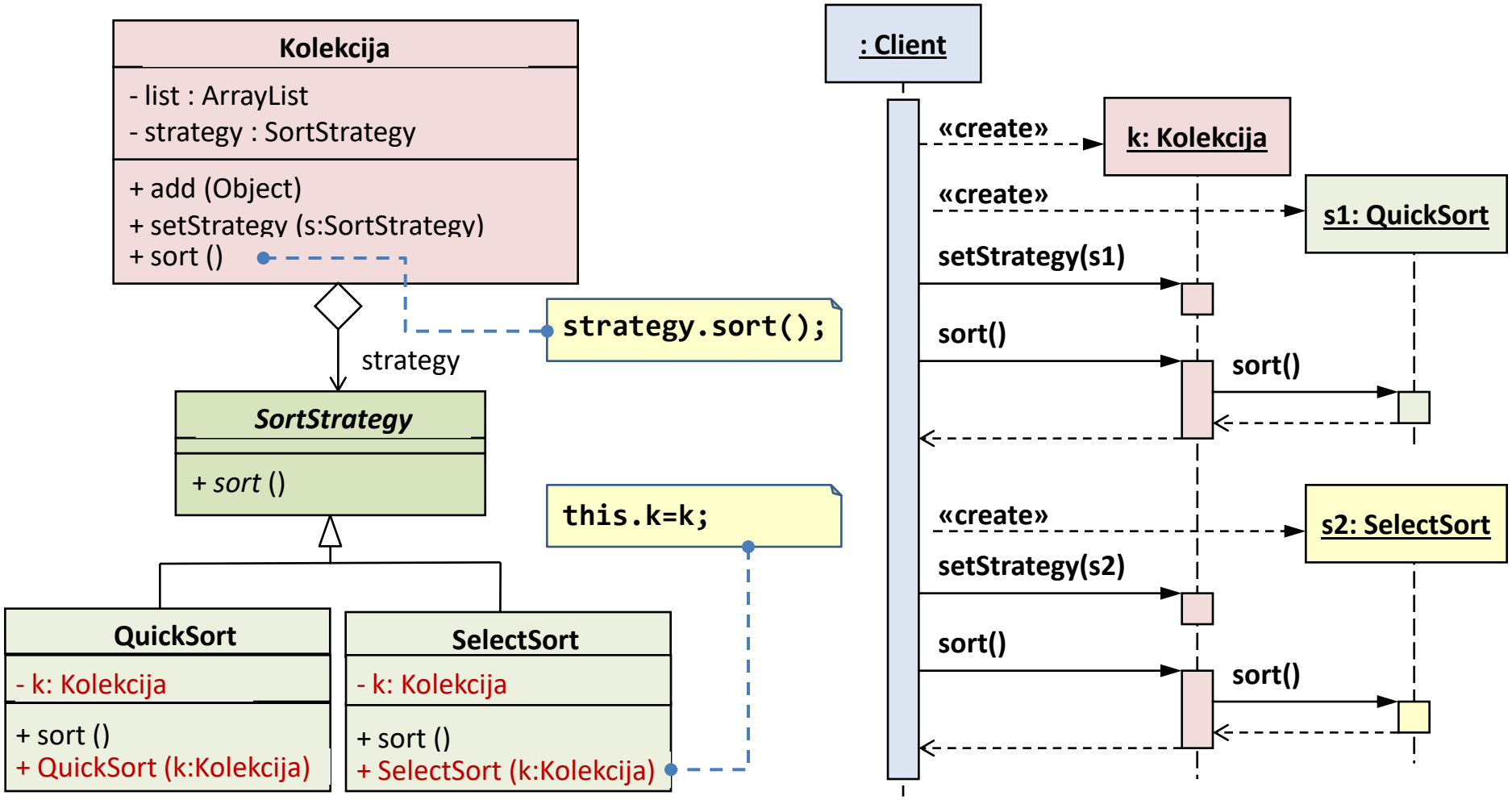




Obrasci ponašanja - Strategy

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

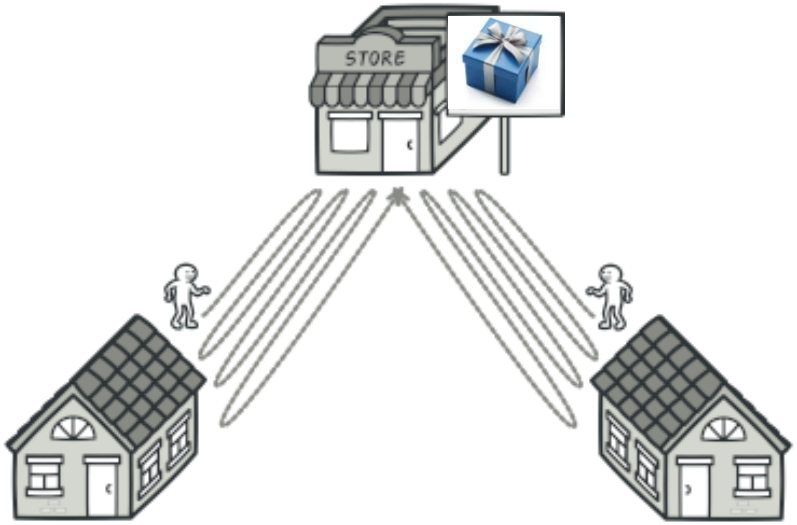
Motivacija za uvođenje projektnog obrasca Opserver

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)**



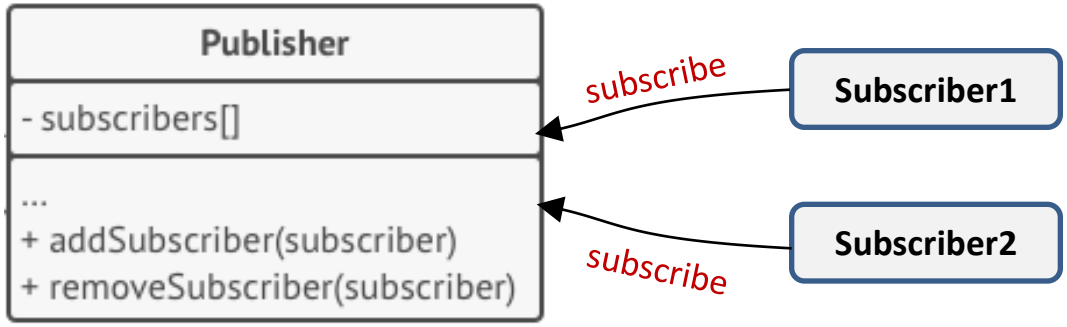
Problematično projektno rješenje

DOBRO PROJEKTNO RJEŠENJE

Obrasci ponašanja - Opserver

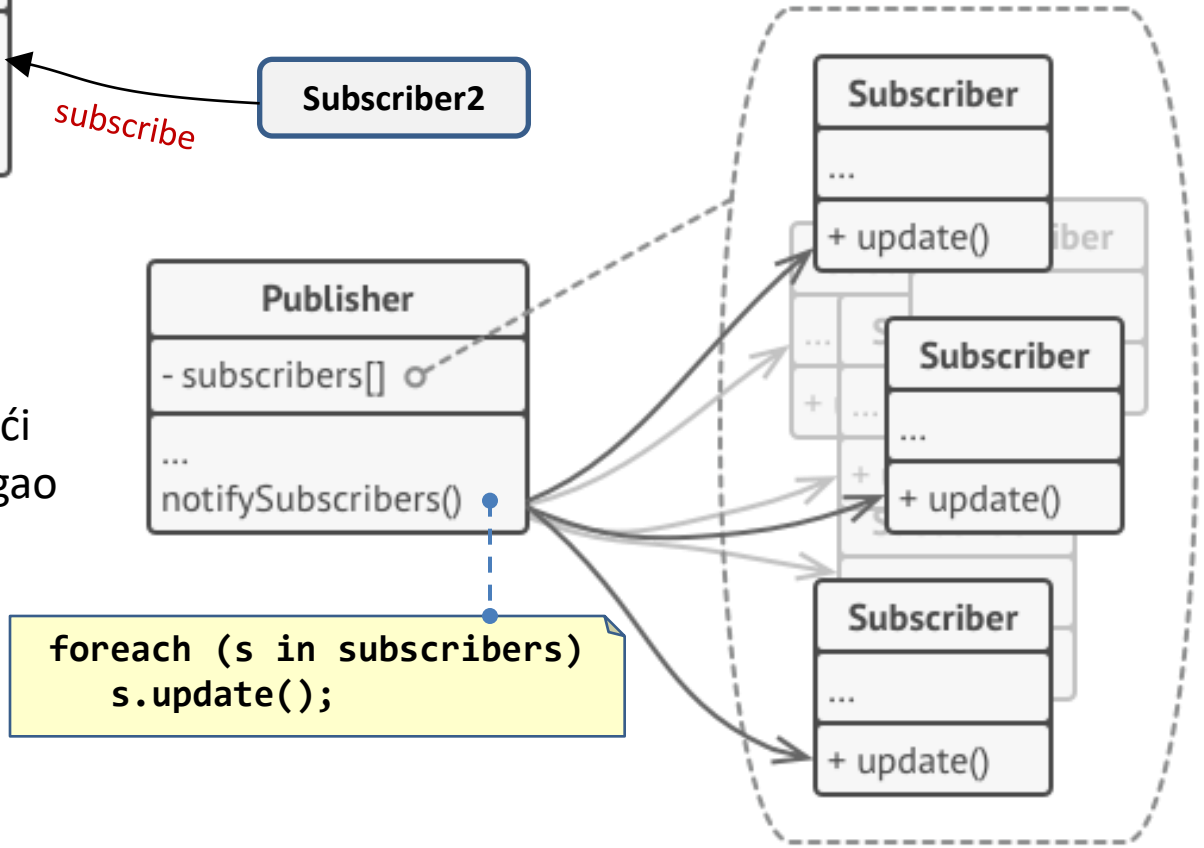
Motivacija za uvođenje projektnog obrasca Opserver

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



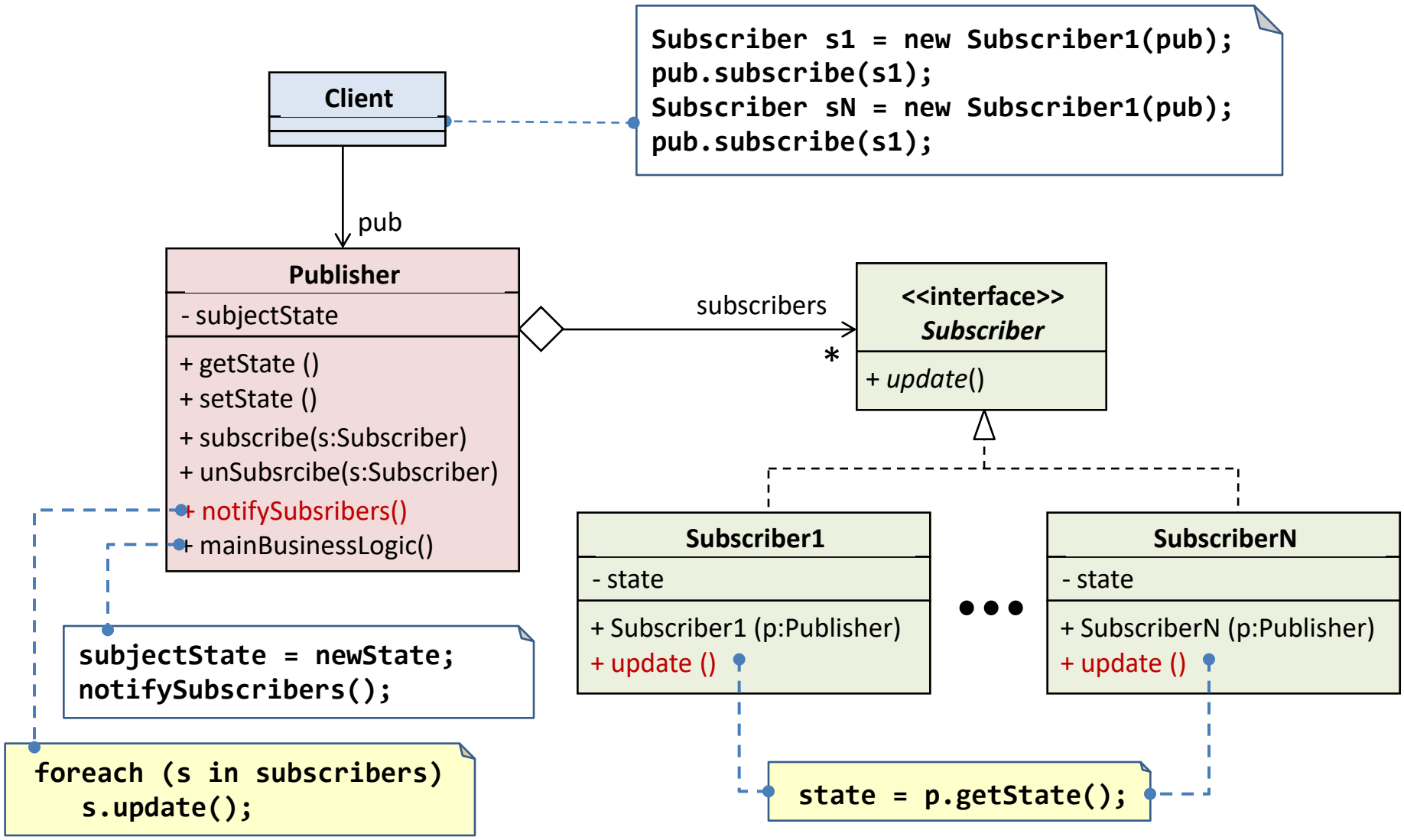
Subjekat održava registar prijavljenih posmatrača:
niz referenci +
jednostavan interfejs za prijavljivanje/odjavljivanje

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



Obrasci ponašanja - Opserver

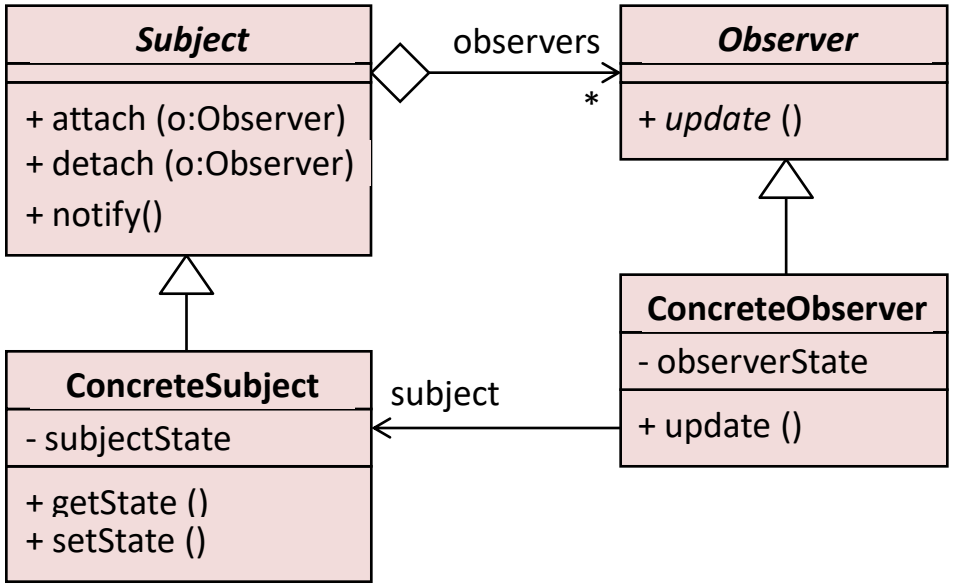
Motivacija za uvođenje projektnog obrasca Opserver



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

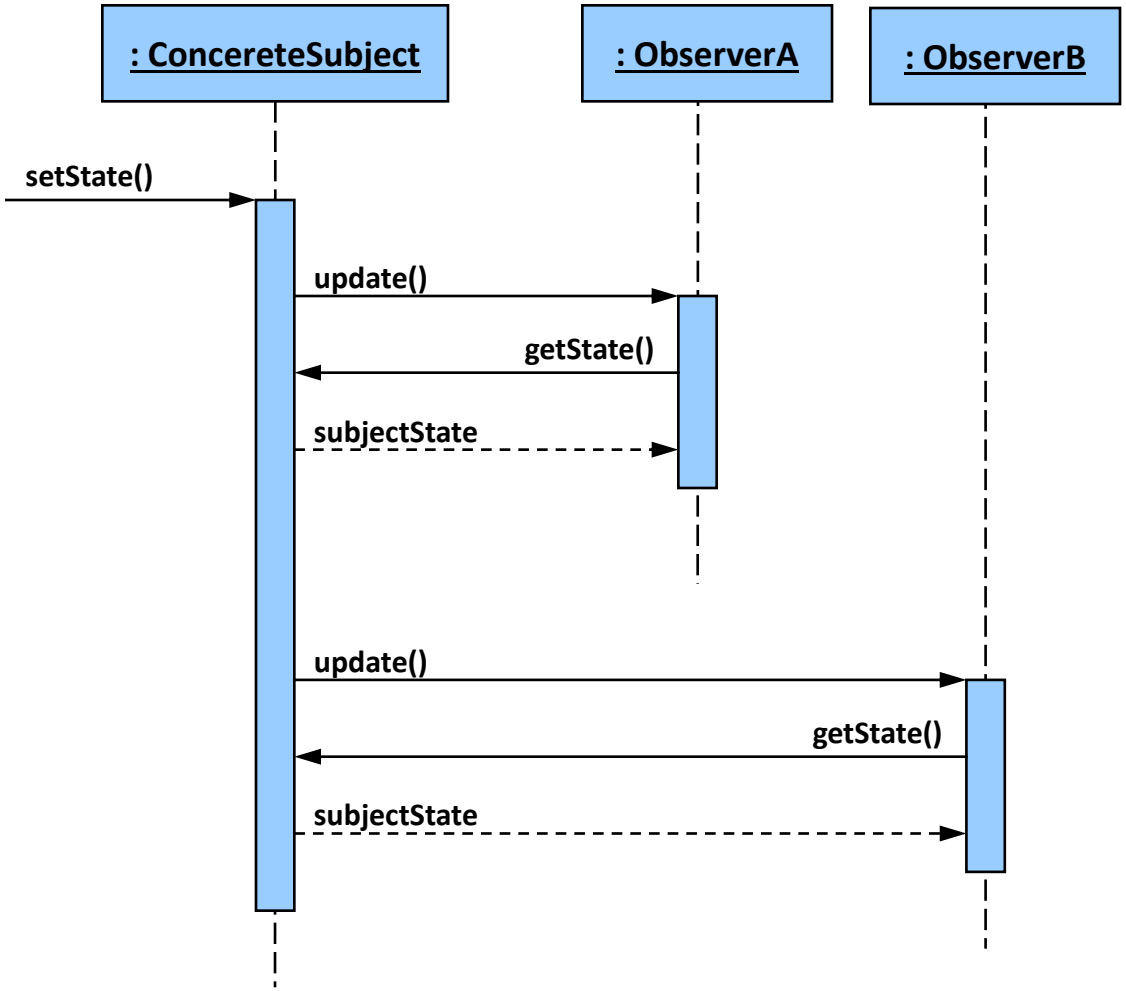
ConcreteSubject
– čuva stanje od interesa za konkretne opservere
– šalje notifikaciju opserverima kad promijeni stanje

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>`**
- **Jedan 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

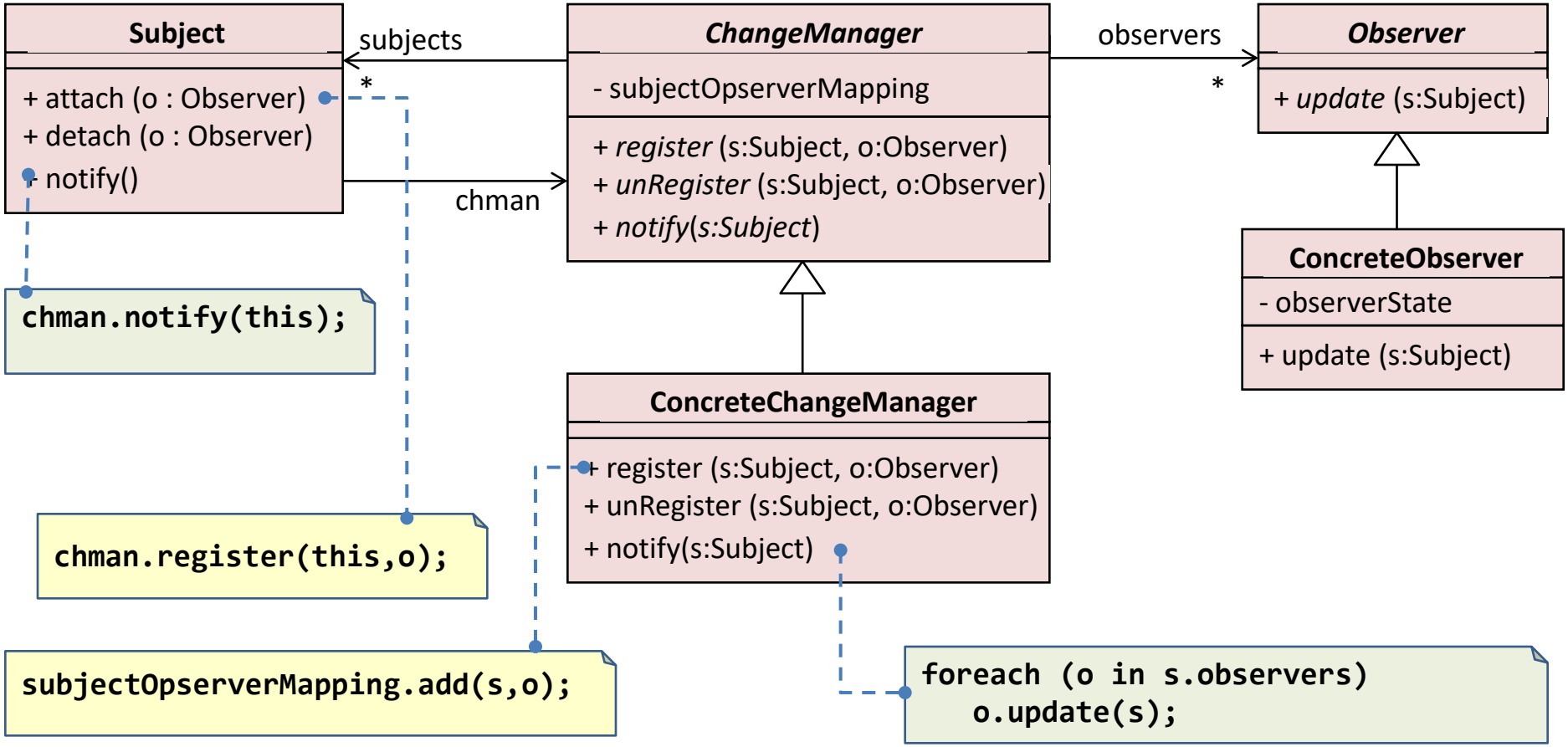
- **„push" model**:
  - subjekat šalje sve informacije opserveru o promjeni stanja
  - subjekat „zna" šta treba opserveru
- **„pull" model**:
  - subjekat samo obavještava operver da je došlo do promjene stanja, a opserver sve potrebne informacije o stanju traži od subjekta
  - može biti neefikasno ako opserver treba da provjeri mnogo atributa koji reprezentuju stanje subjekta
- **hibridni model:** između push i pull

ChangeManager

- U slučaju kompleksnih veza subjekti↔opserveri
- ChangeManager je poseban objekat koji upravlja optimalnim ažuriranjem opservera
- Uloge:
  - mapiranje subjekat→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

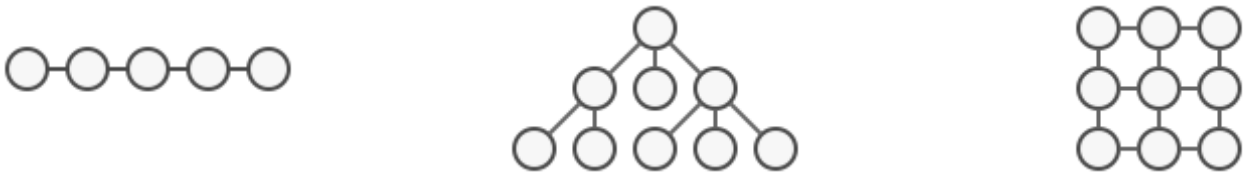
Primjer (primjena ChangeManagera):



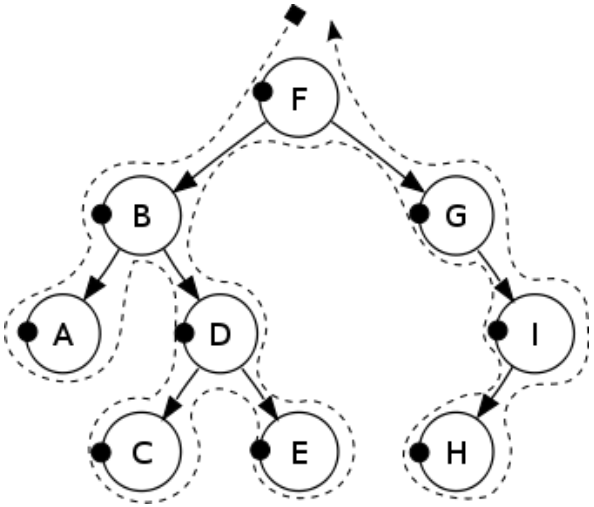
Obrasci ponašanja - Iterator

Motivacija za uvođenje projektnog obrasca Iterator

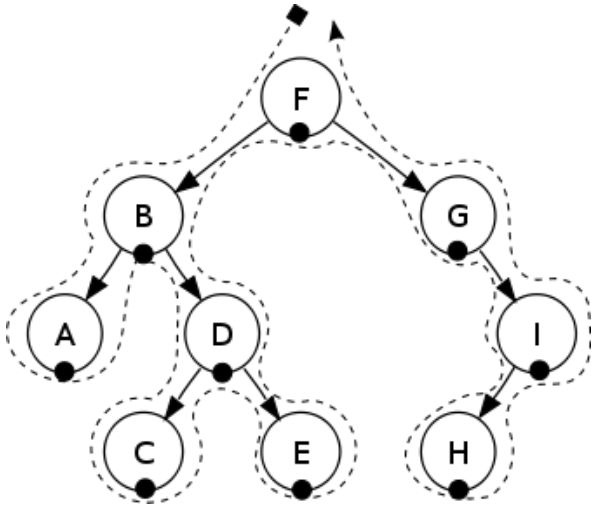
Aplikacije često manipulišu različitim kolekcijama podataka (liste, stabla, grafovi, ...)



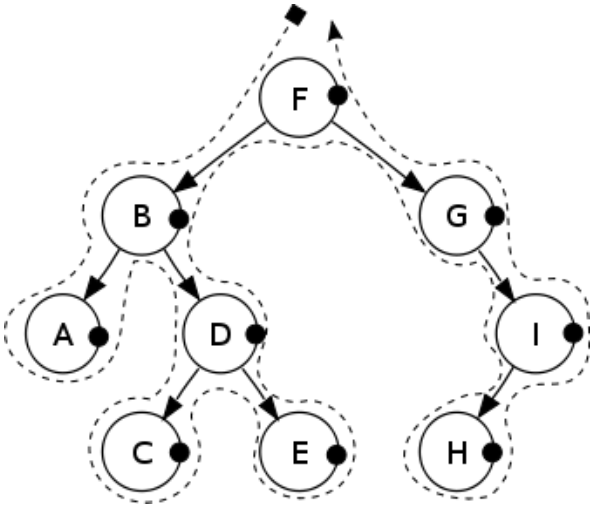
Često su potrebni različiti načini obilaska kolekcije
(npr. za binarno stablo – obilazak po dubini: pre-order, in-order, post-order)



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

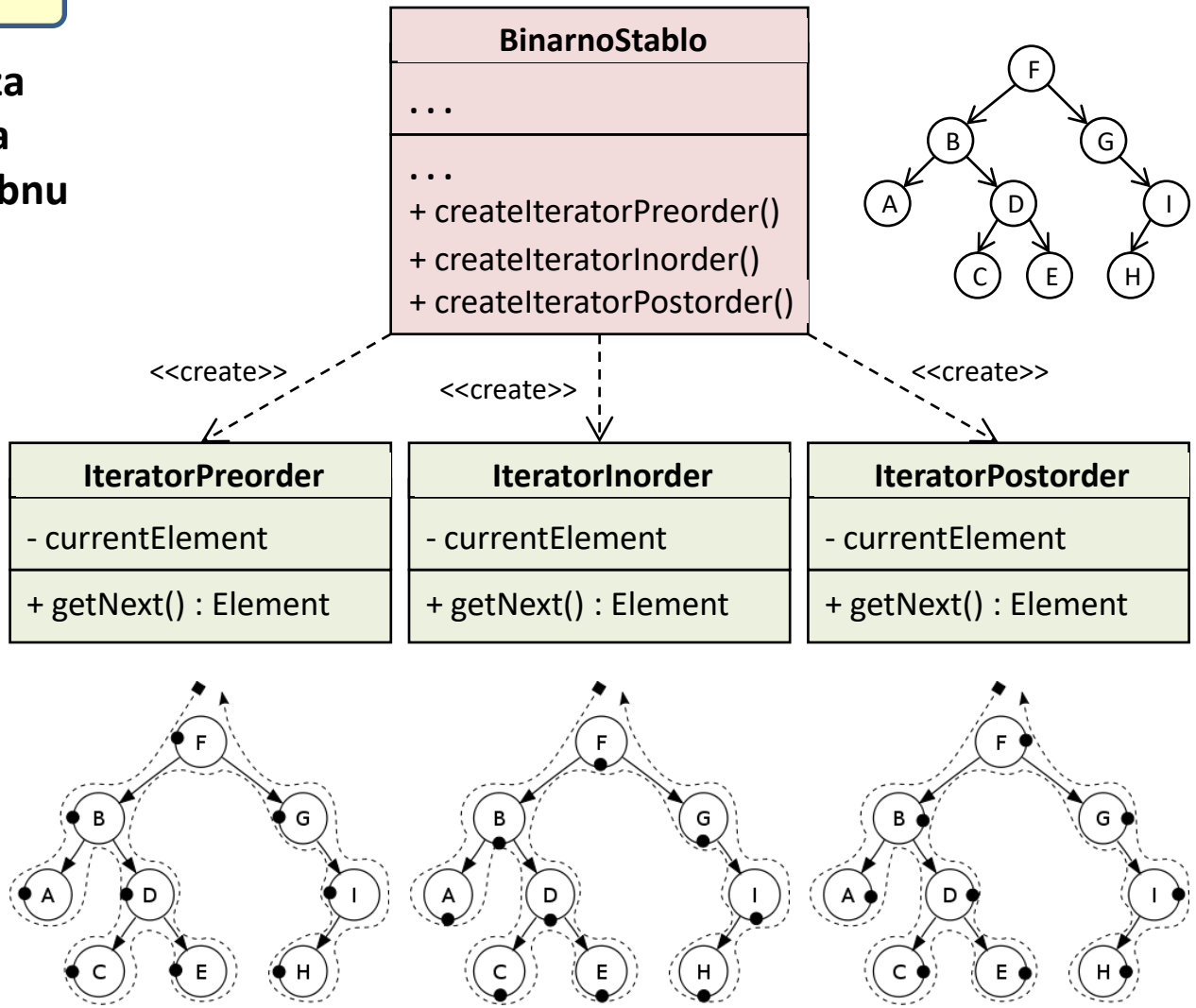
DOBRO PROJEKTNO RJEŠENJE

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



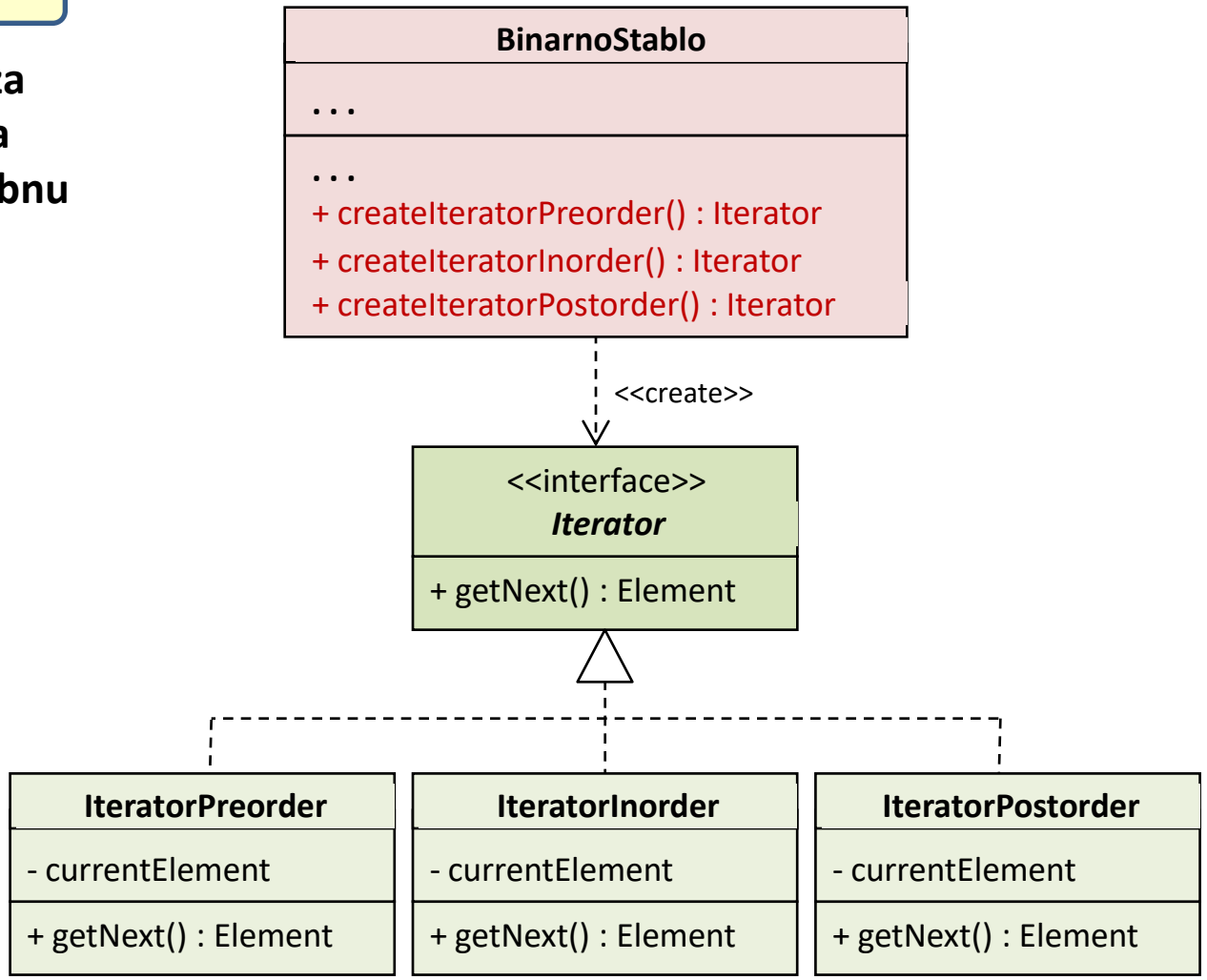
Obrasci ponašanja - Iterator

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



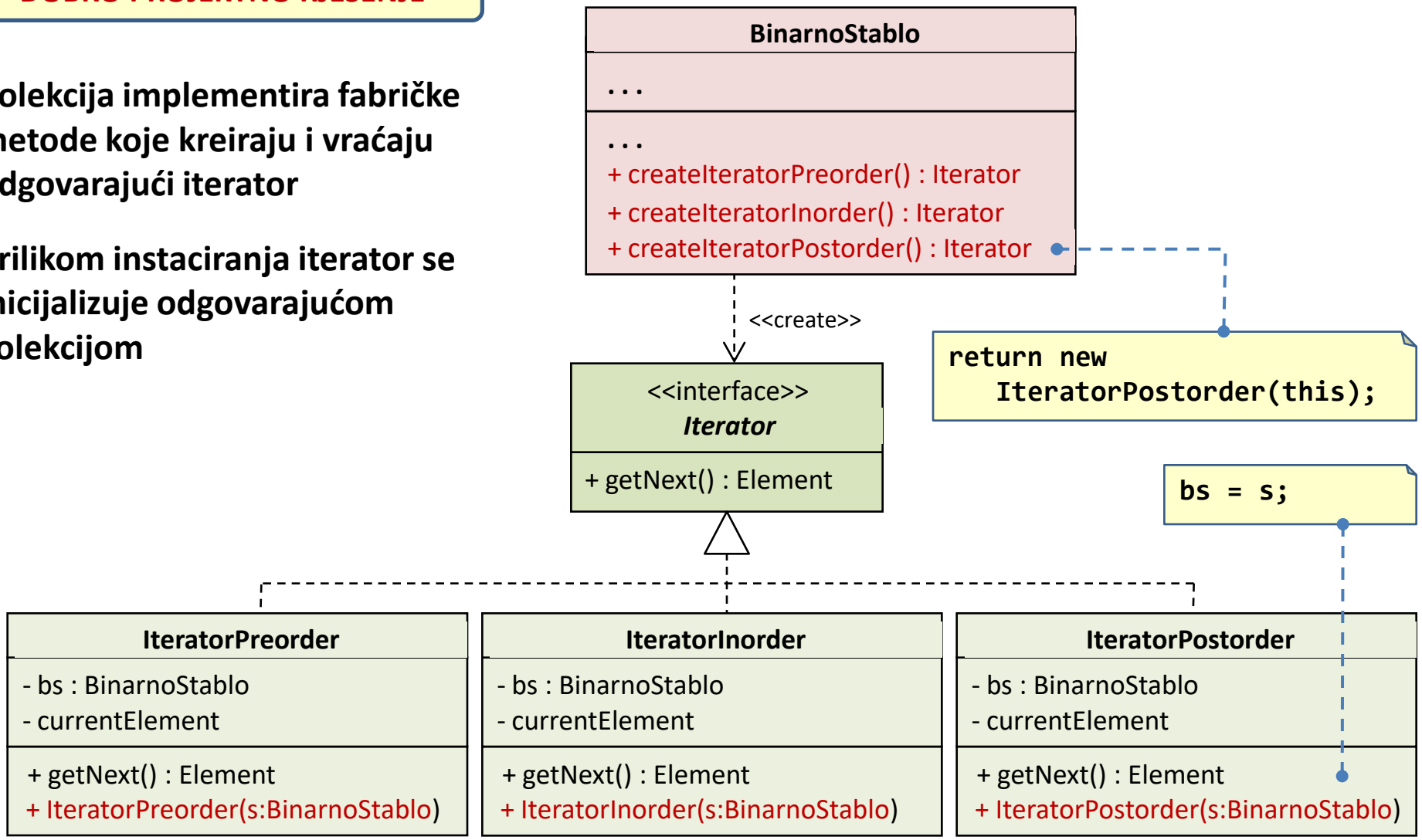
Obrasci ponašanja - Iterator

Motivacija za uvođenje projektnog obrasca Iterator

DOBRO PROJEKTNO RJEŠENJE

Kolekcija implementira fabričke metode koje kreiraju i vraćaju odgovarajući iterator

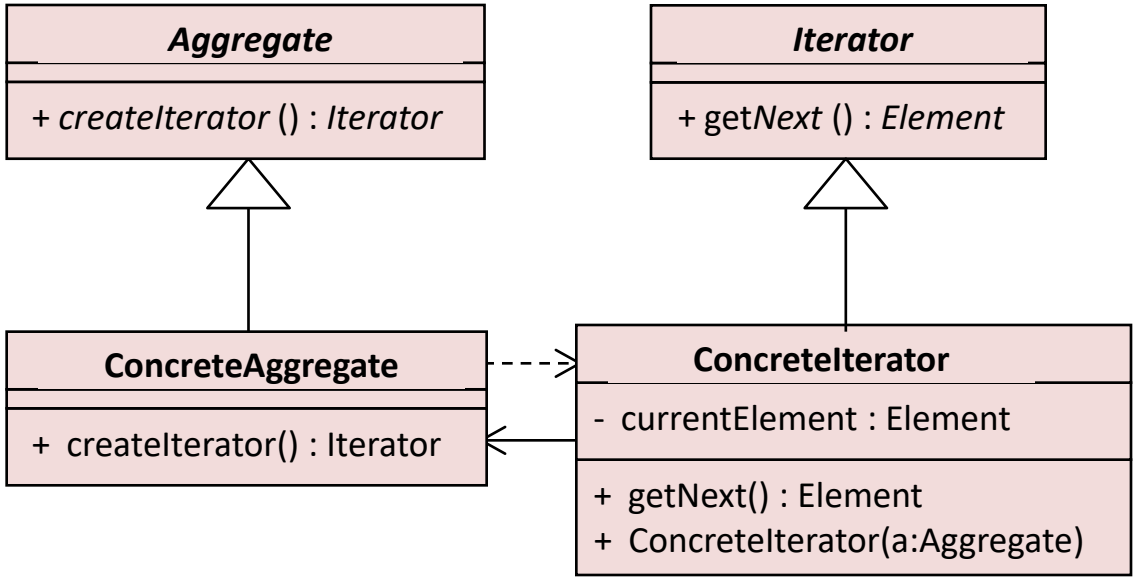
Prilikom instaciranja iterator se inicijalizuje odgovarajućom kolekcijom



Obrasci ponašanja - Iterator

Iterator
(Brojač)

– **Obezbjeđuje sekvencijalni pristup elementima objekta nastalog agregacijom bez potrebe za pristupanjem njegovoj stvarnoj reprezentaciji.**



Iterator

– Deklariše interfejs za pristupanje elementima i kretanje od jednog do drugog

ConcreteIterator

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

Aggregate

– Deklariše interfejs za kreiranje objekata tipa Iterator

ConcreteAggregate

– Implemetira interfejs za kreiranje Iterator objekata i vraća reference na odgovarajuće ConcreteIterator objekte

Obrasci ponašanja - Iterator

Primjer: