UNIVERZITET U BANJOJ LUCI
ELEKTROTEHNIČKI FAKULTET

Prof. dr Dražen Brđanin

OBJEKTNO-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** Obezbjeđuje 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** Obezbjeđuje 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

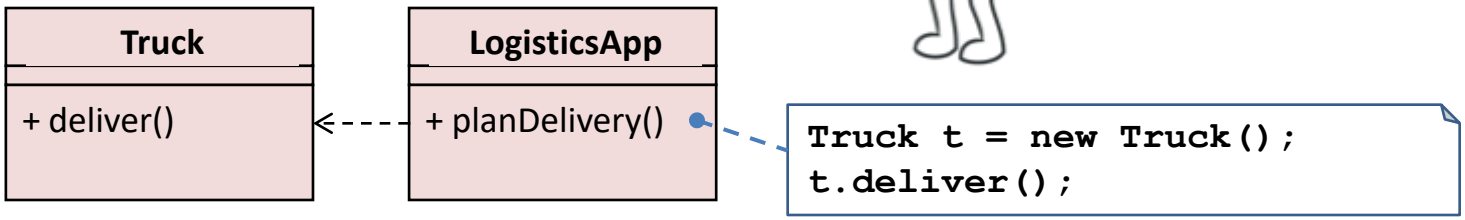
Motivacija za uvođenje fabričkog metoda

Pretpostavimo da aplikacija ima mogućnost manipulacije jednom vrstom objekata

(npr. aplikacija za organizaciju transporta, koja ima mogućnost organizacije prevoza robe kamionom)

Pretpostavimo da treba omogućiti manipulaciju dodatnim vrstama objekata

(npr. aplikacija treba da omogući i organizaciju prevoza robe brodom)



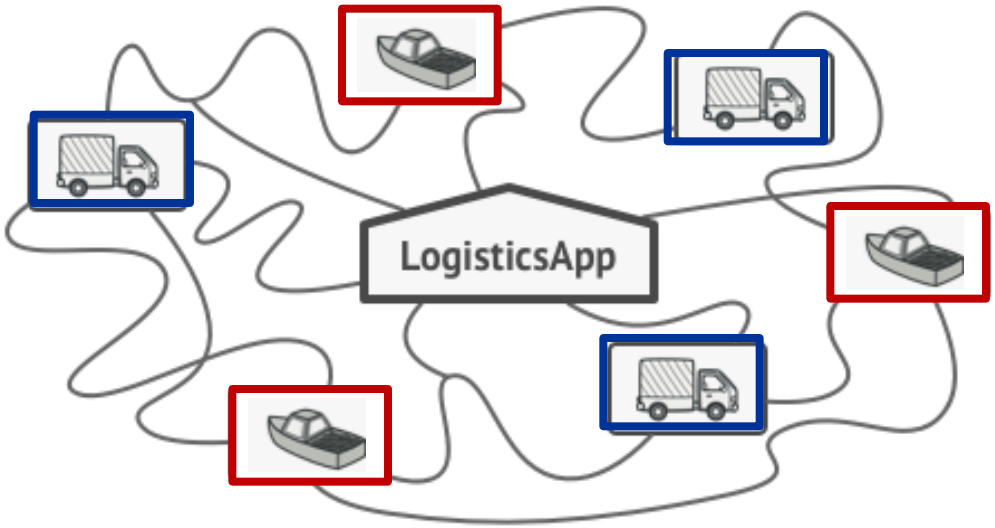
Kreacioni obrasci – Fabrički metod

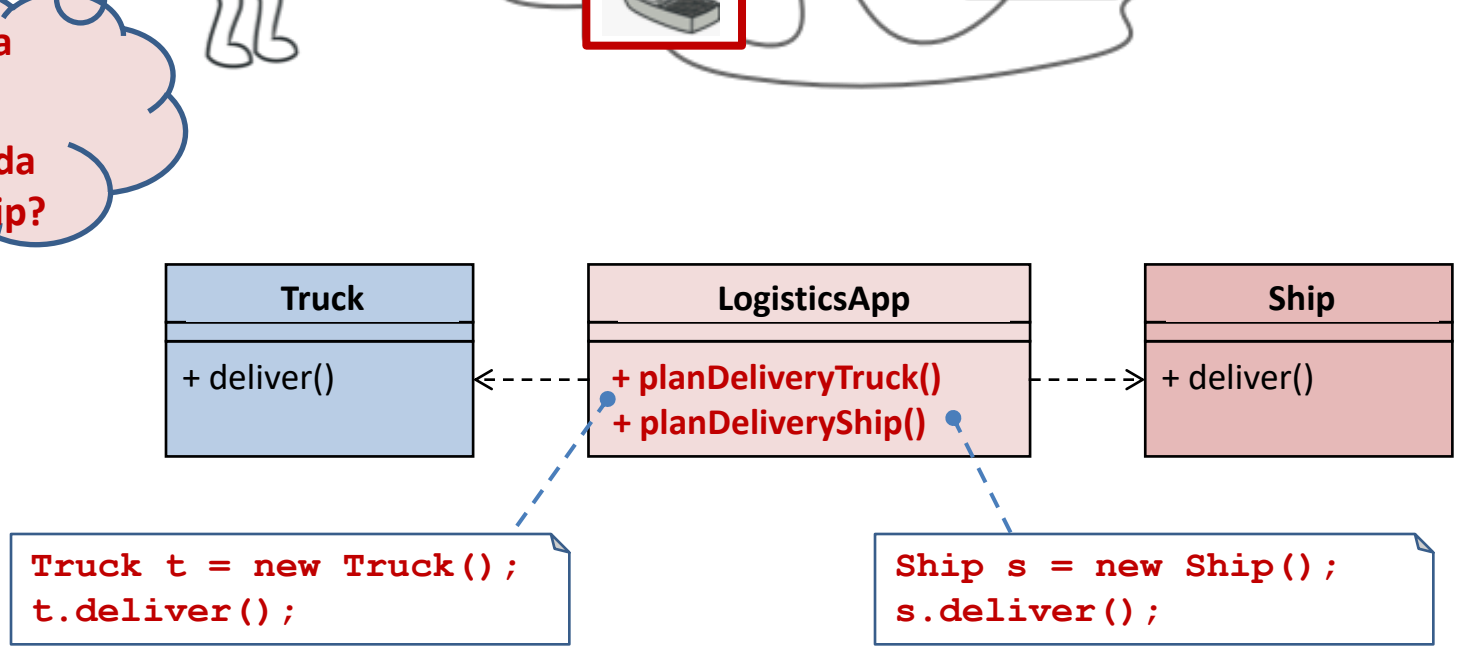
Motivacija za uvođenje fabričkog metoda

Kako omogućiti manipulaciju dodatnim vrstama objekata?

Dodatna klasa Ship?

Dodatna metoda planDeliveryShip?





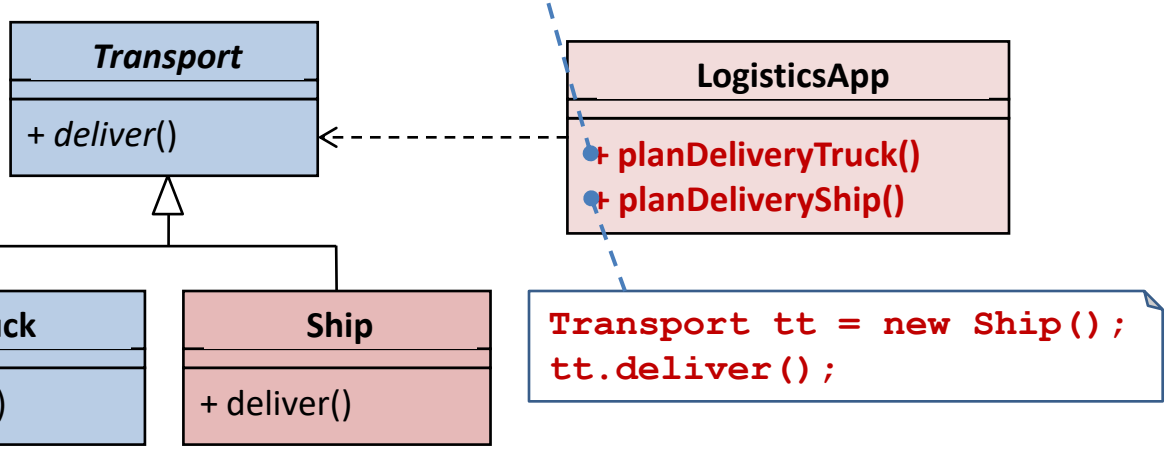
Kreacioni obrasci – Fabrički metod

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();
```

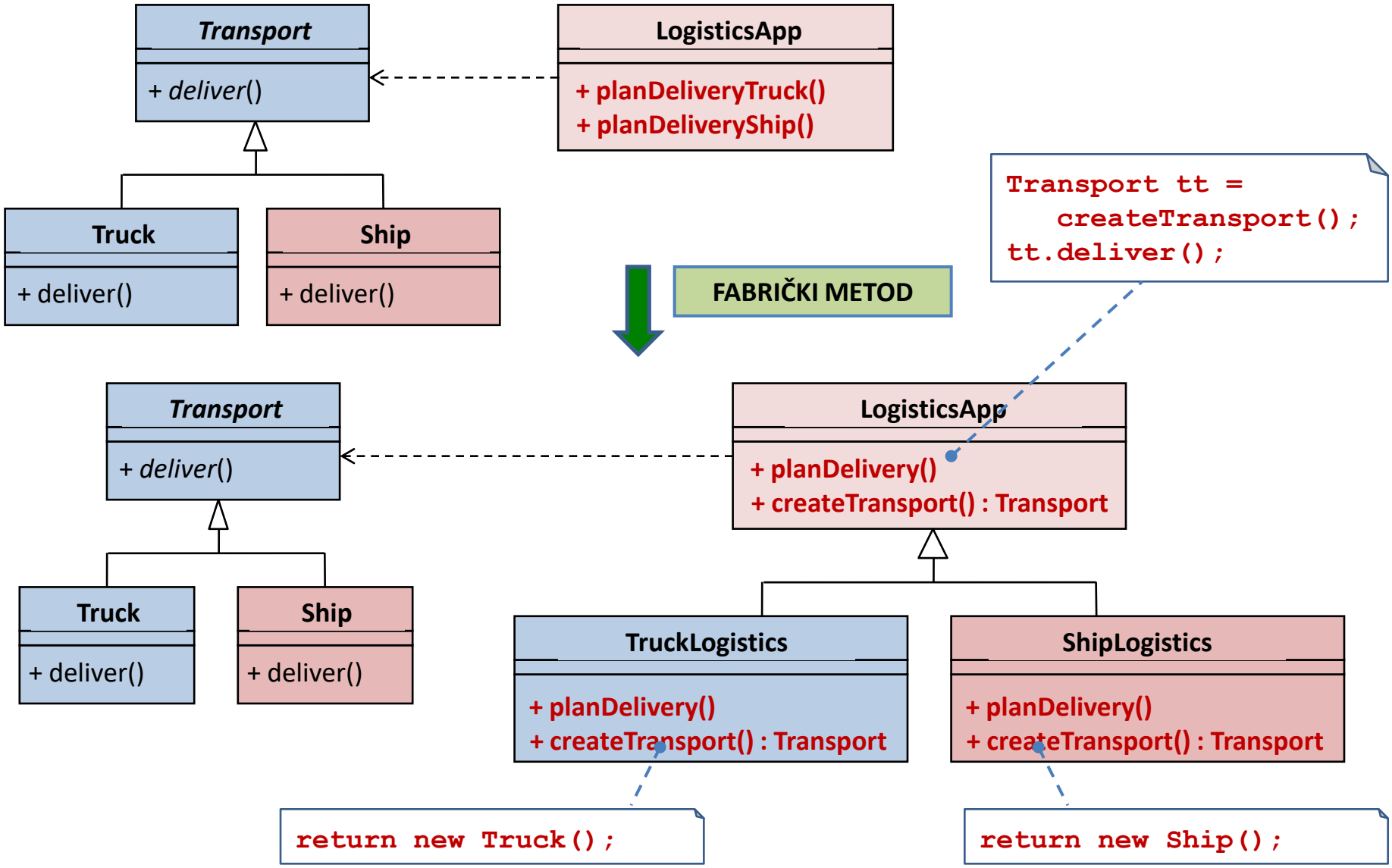
PROGRAMIRANJE PREMA INTERFEJSU


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

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

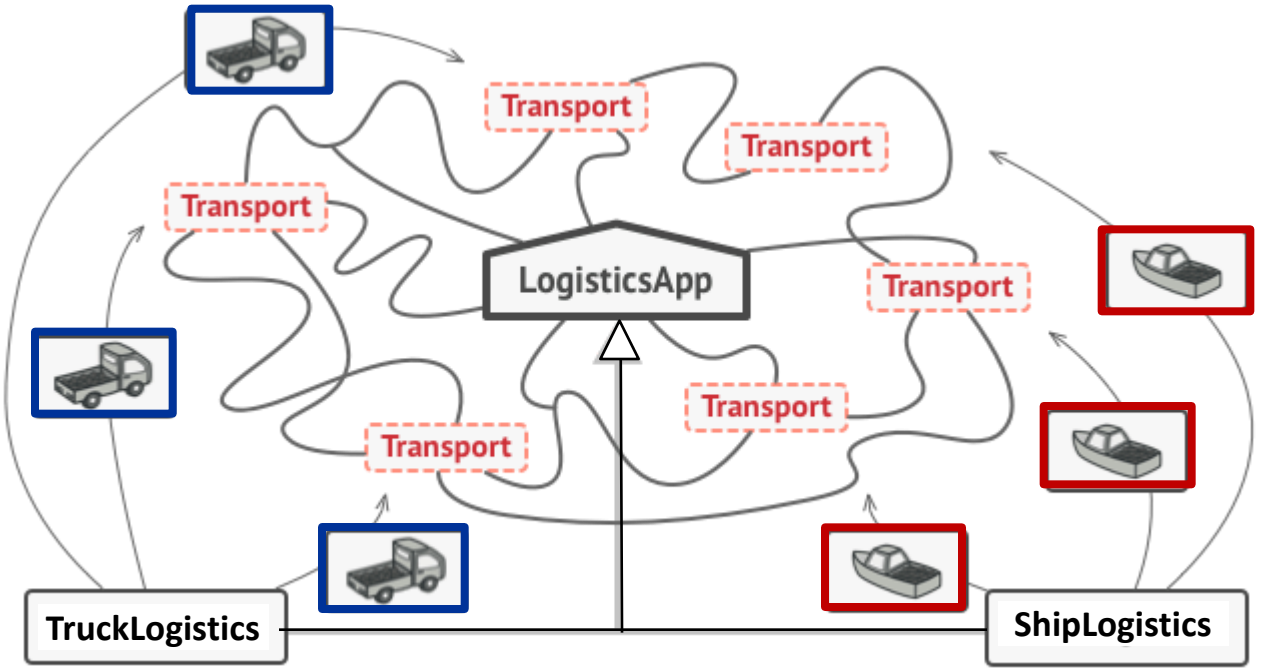
Kreacioni obrasci – Fabrički metod

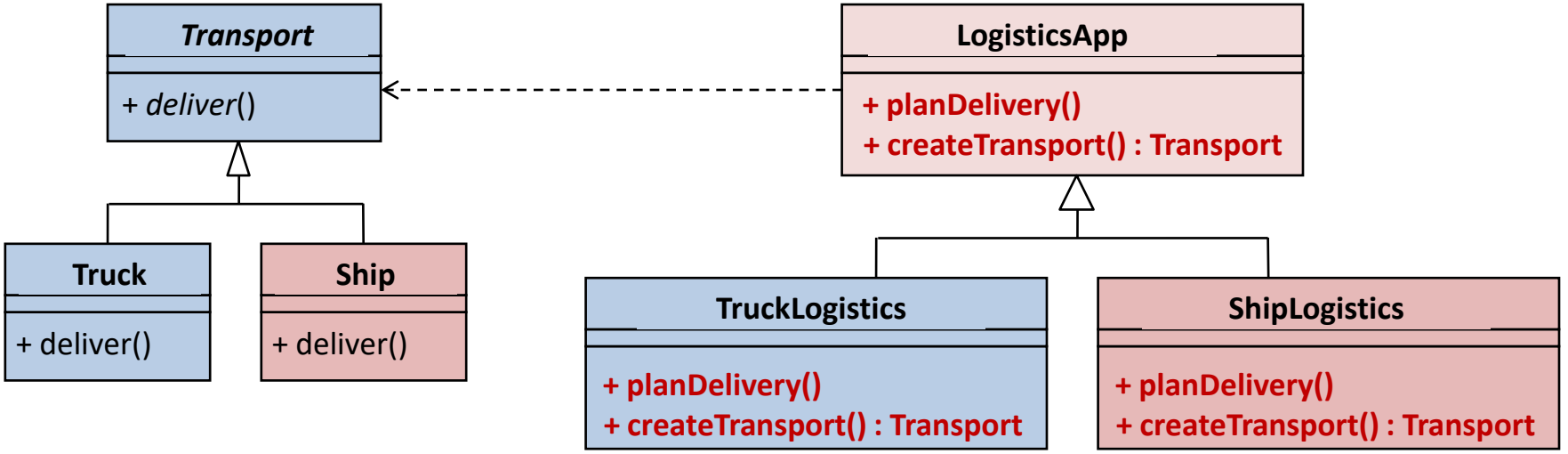
Motivacija za uvođenje fabričkog metoda



Kreacioni obrasci – Fabrički metod

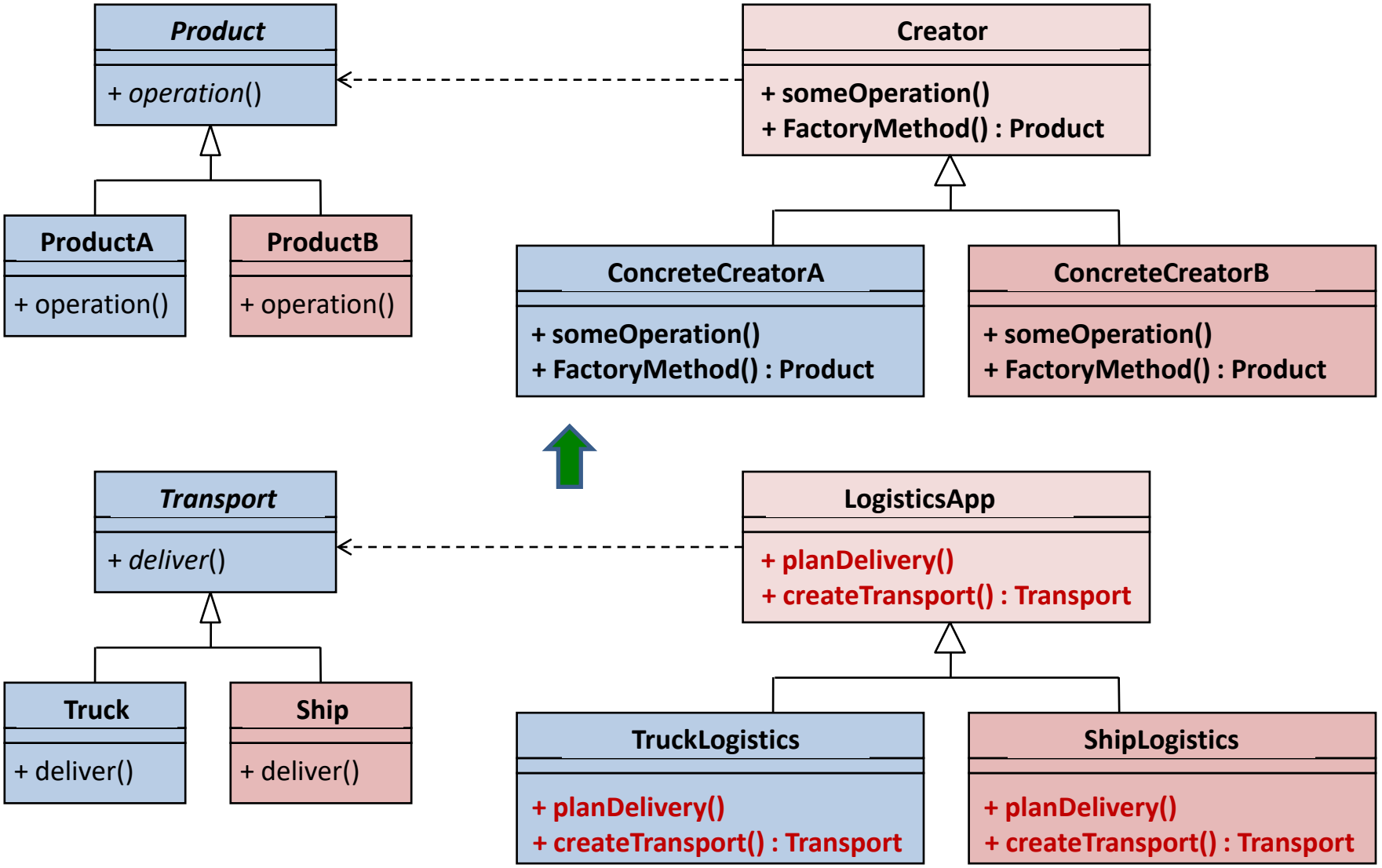
Motivacija za uvođenje fabričkog metoda





Kreacioni obrasci – Fabrički metod

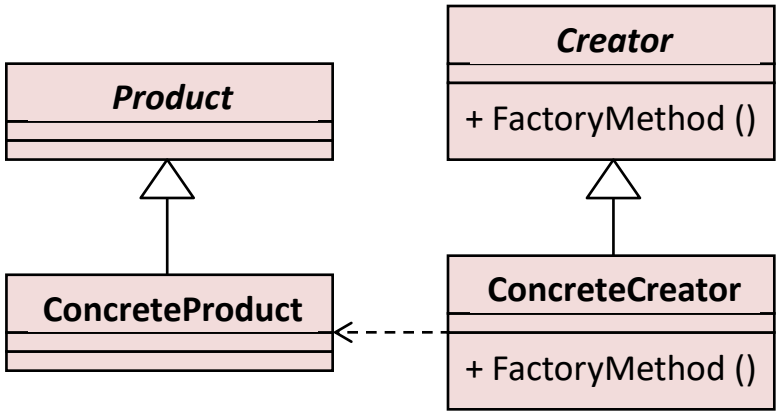
Motivacija za uvođenje fabričkog metoda



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 za instanciranje 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

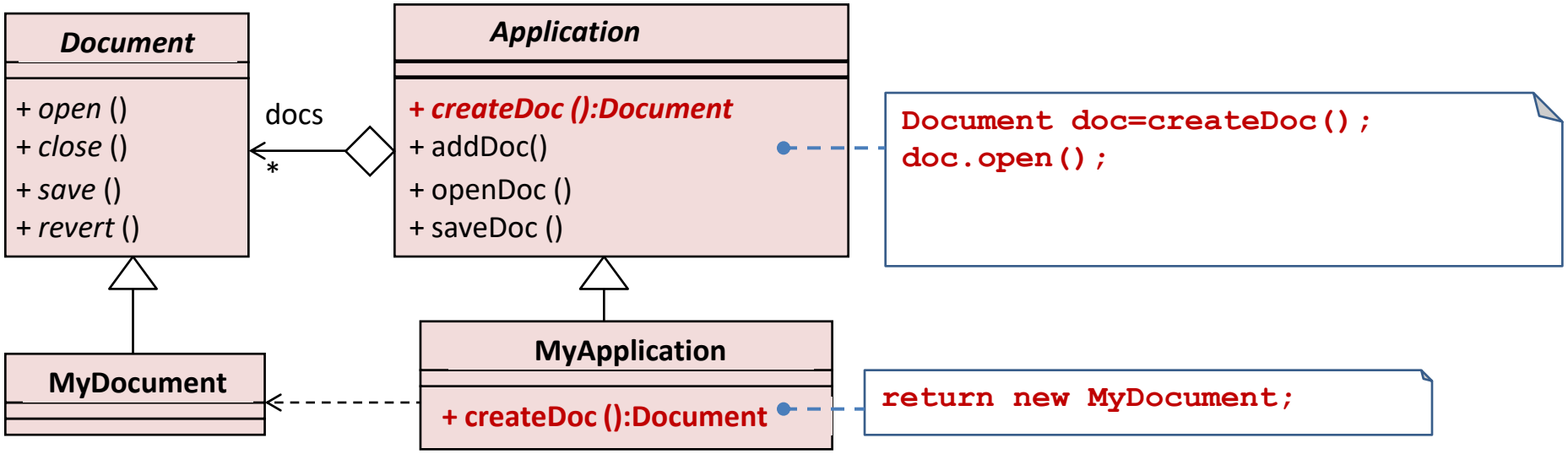
- redefiniše FactoryMethod kako bi vraćao instancu klase ConcreteProduct

```
return new ConcreteProduct();
```

Kreacioni obrasci – Fabrički metod

Primjer:

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



Pretpostavimo 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 konkretne dokumente
- ***Application*** ne može da predvidi koji će se dokument instancirati (to zavisi od konkretne 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 se instanciraju
- Klasa prepušta odgovornost za instanciranje 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 konkretan 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

- obezbjeđuje 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);
  }
}
```

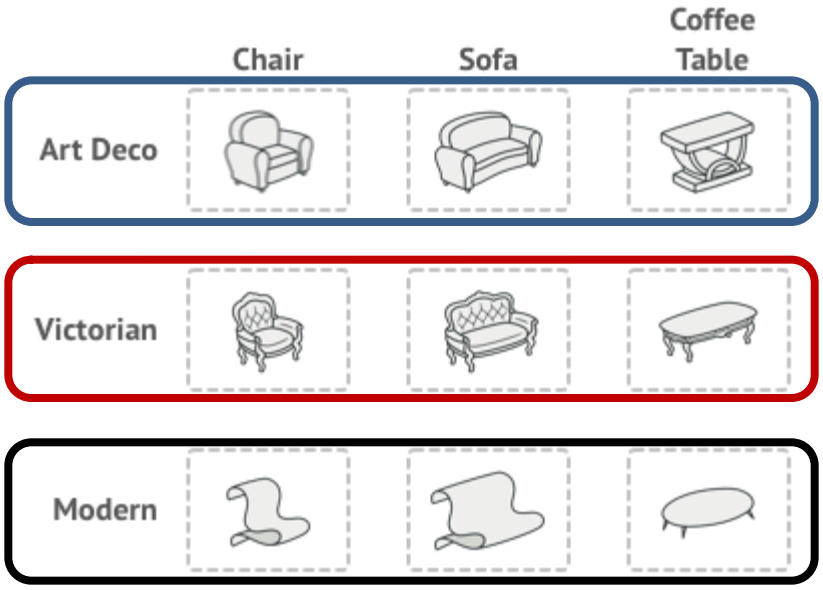
Kreacioni obrasci – Apstraktna fabrika

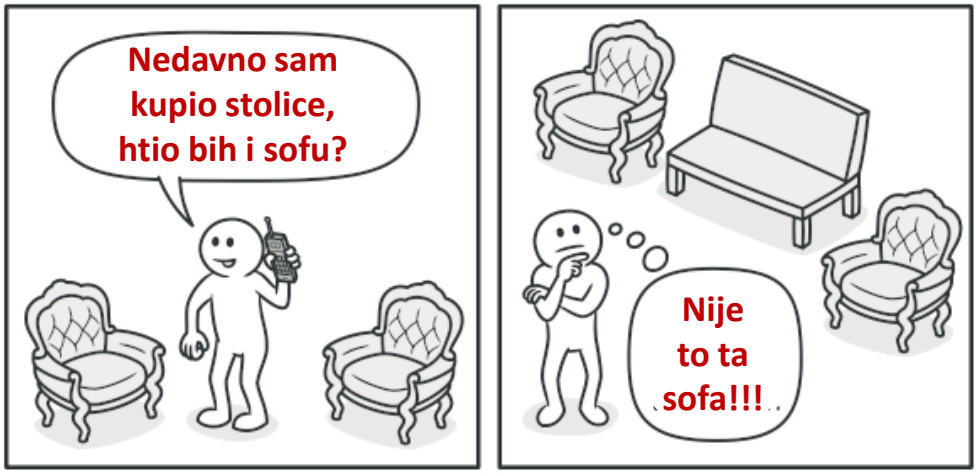
Motivacija za uvođenje apstraktne fabrike

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 fimilije





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

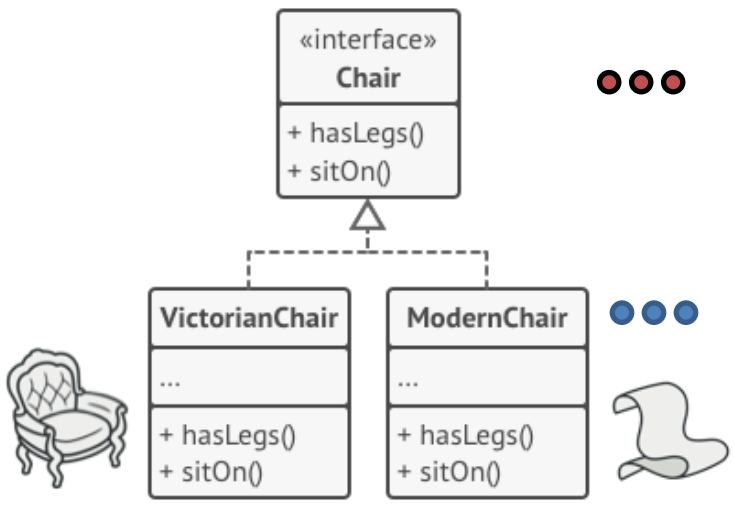
Arhitektura treba da je stabilna i da se ne mijenja dodavanjem novog proizvoda ili nove familije!

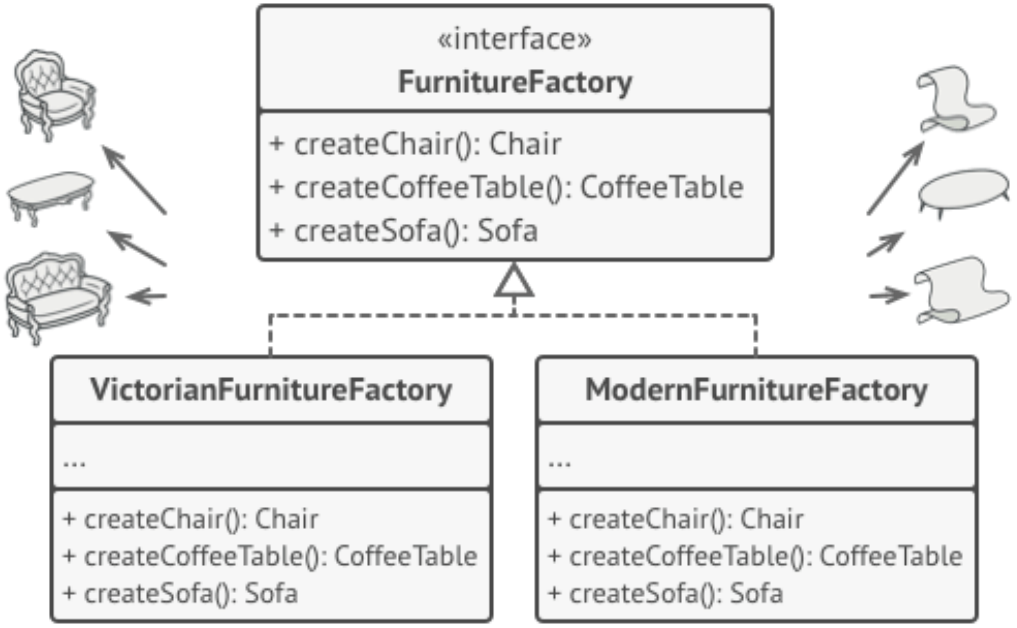
Kreacioni obrasci – Apstraktna fabrika

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 konkretne proizvode za svaki tip proizvoda

(npr. za Chair: VictorianChair, ModernChair, ...)

(npr. za Sofa: VictorianSofa, ModernSofa, ...)

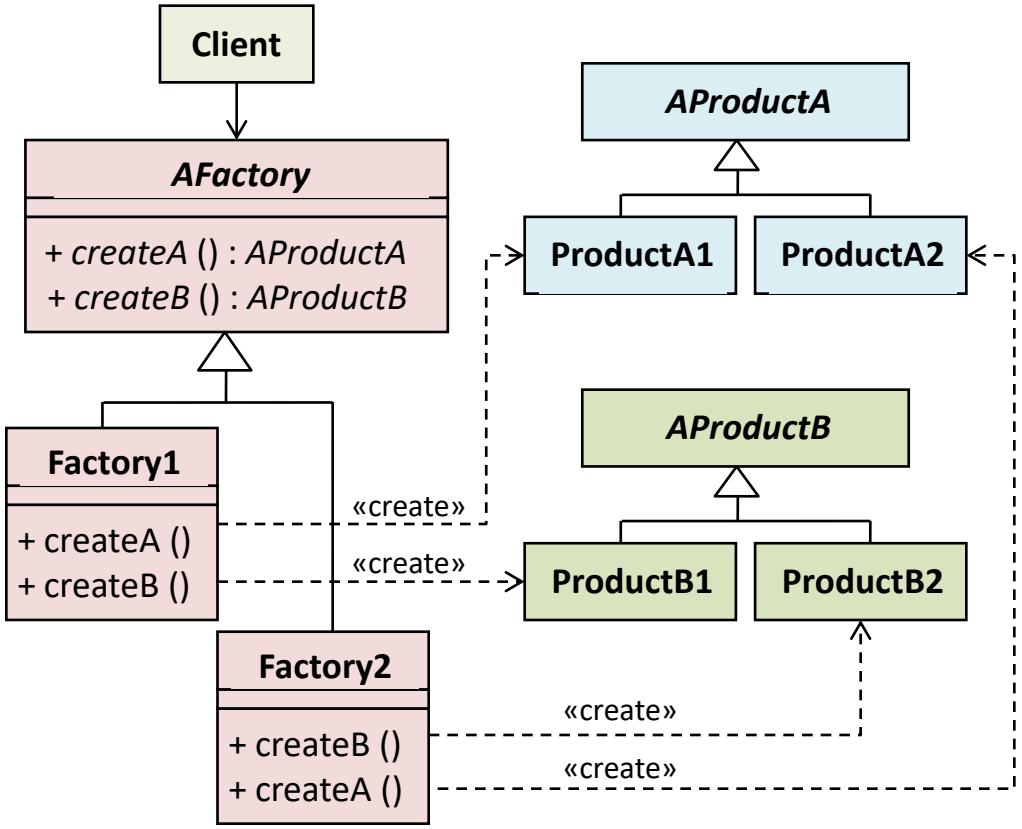
Definisati klase koje reprezentuju konkretne fabrike za svaku familiju proizvoda

(npr. VictorianFactory, ModernFactory, ...)

Kreacioni obrasci – Apstraktna fabrika

Abstract Factory
(Apstraktna fabrika)

- **Kreira instance nekoliko familija klasa.**
- **Obezbjeđuje 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)

- implemetira operacije za kreiranje određene vrste konkretnih objekata

AbstractProduct (AProductA, AProductB)

- deklariše interfejs za određenu vrstu objekata

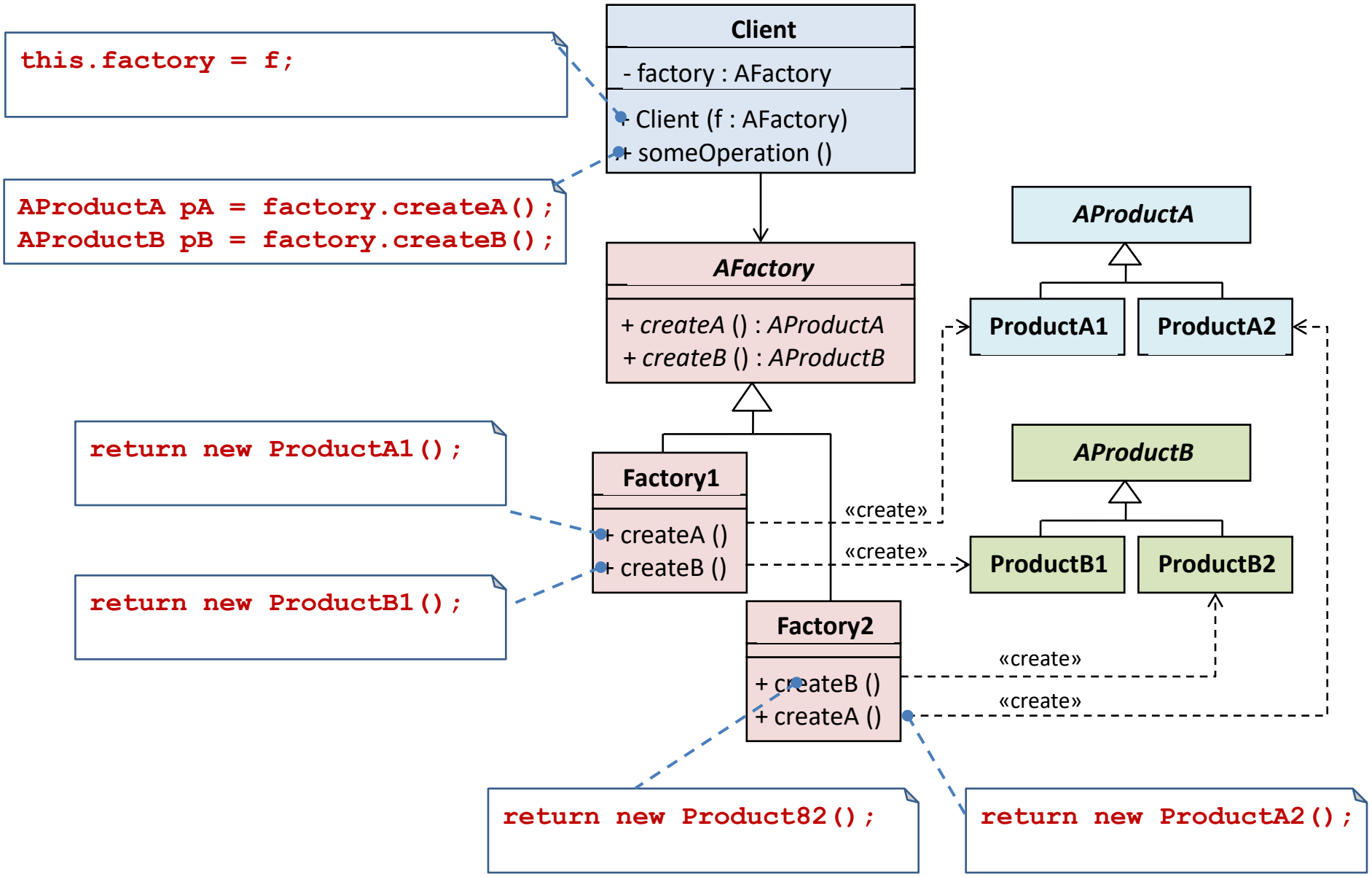
ConcreteProduct (ProductA1, ProductA2)

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

Motivacija za apstraktnu fabriku:

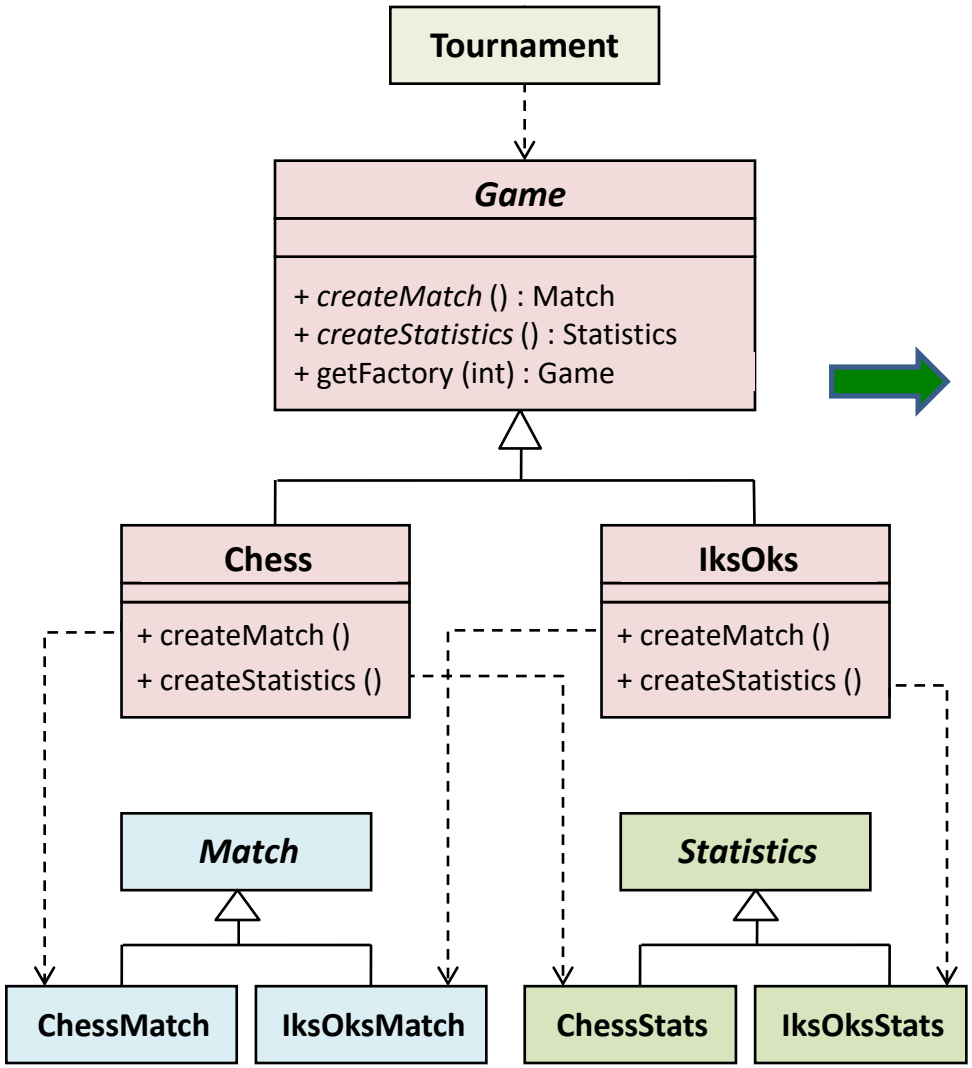
Npr. GUI koji treba da je portabilan na različite OS.
Npr. upravljački sistem nezavisan od ugrađene opreme.

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 sistemski 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.

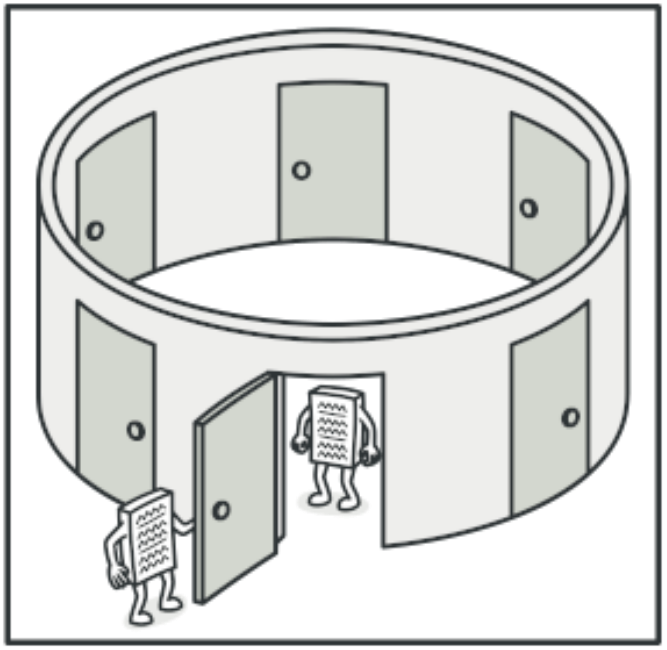
Kreacioni obrasci – Singleton

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

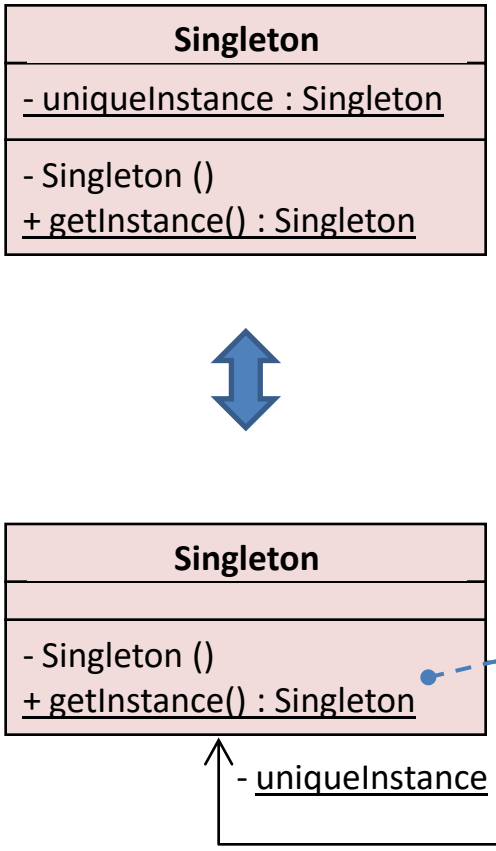
Konstruktor nije public, nego private (ili protected), tako da izraz new Singleton() predstavlja *compile-time* grešku

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)**
- **Obezbjeđuje odgovarajući pristup jedinstvenoj instanci klase.**



Singleton

- definiše statičku operaciju **getInstance()** koja obezbjeđuje 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 singltonu 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

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

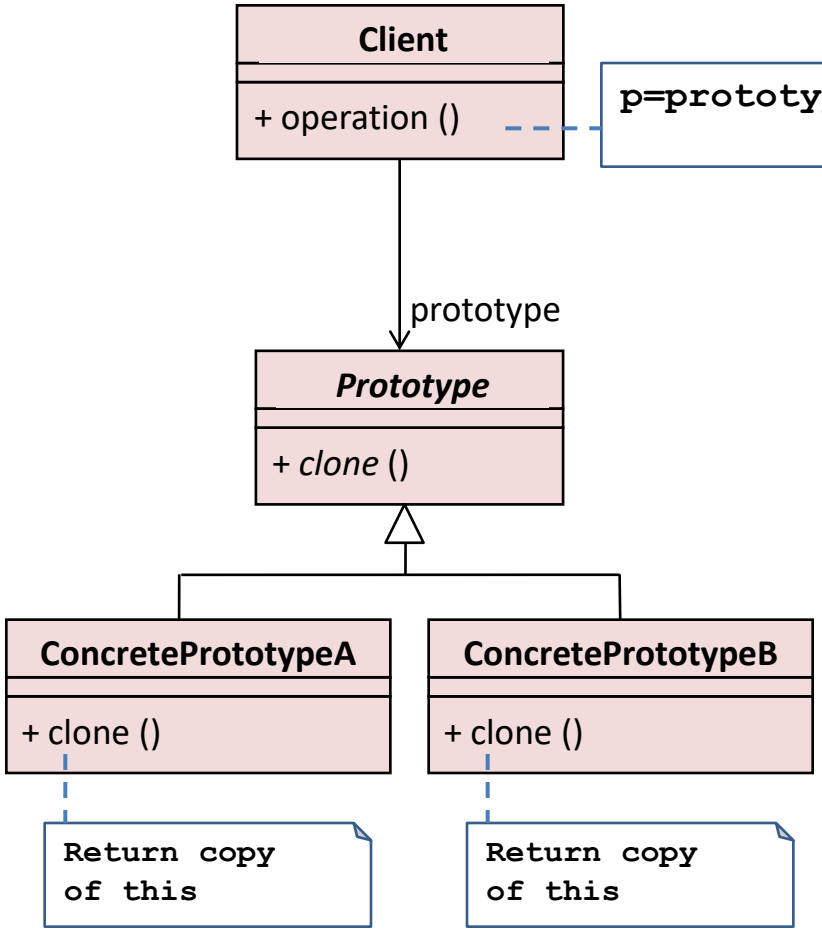
Objekat koji može da se klonira, naziva se prototip

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

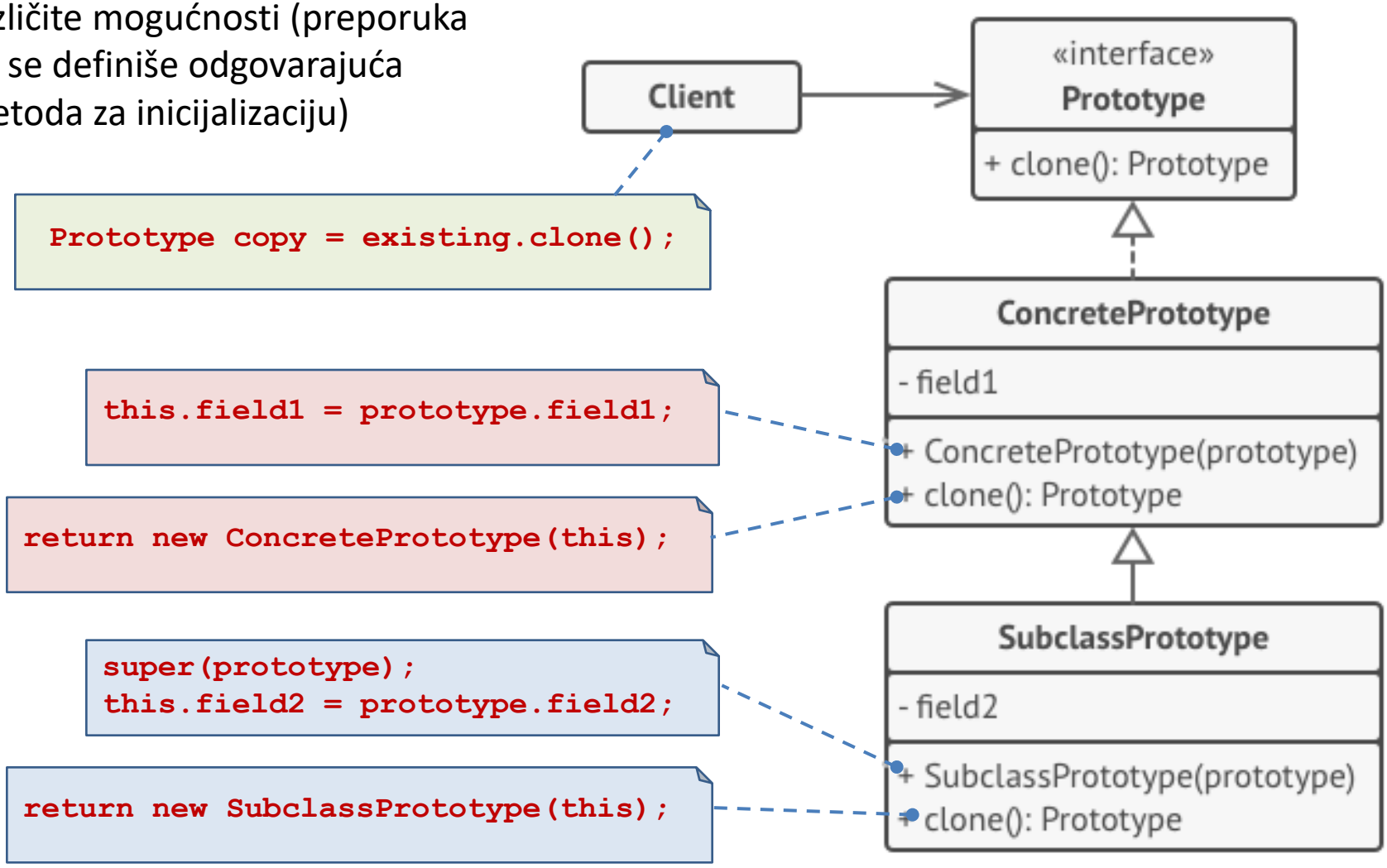
Kreacioni obrasci – Prototip

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

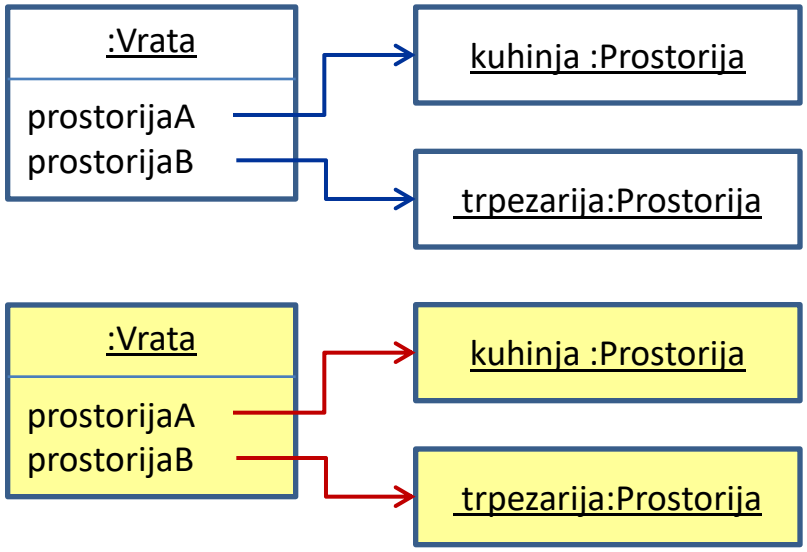


Kreacioni obrasci – Prototip

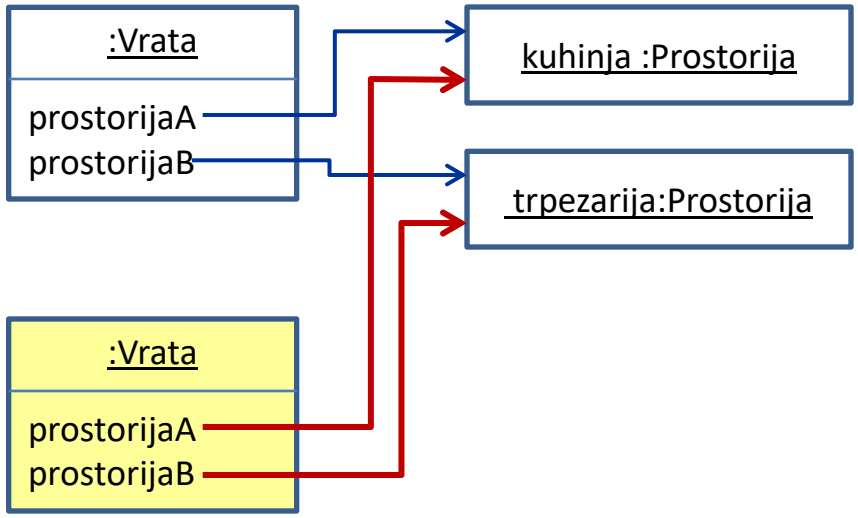
Implementacioni detalji

– **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

Primjer (*Shallow clone*):

```
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());
   }
}
```

```
Biblioteka
Biblioteka
```

Primjer (*Deep clone*):

```
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());
  }
}
```

```
Biblioteka
Racunovodstvo
```

Kreacioni obrasci – Prototip

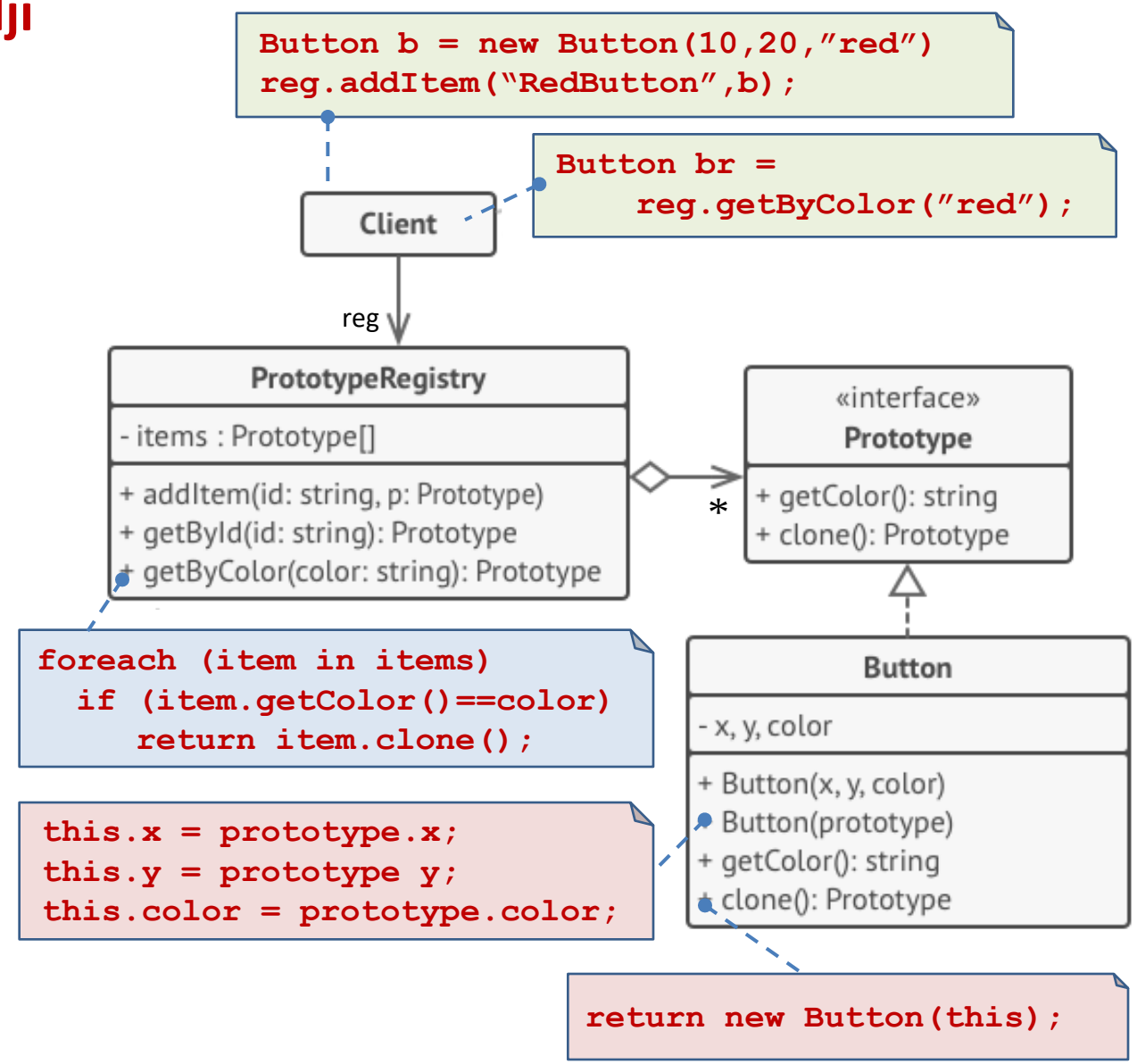
Implementacioni detalji

– Preporuka da se koristi **prototype manager** **(registar prototipova):**

  – **registar 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 strukturom, 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 konkretan 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

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

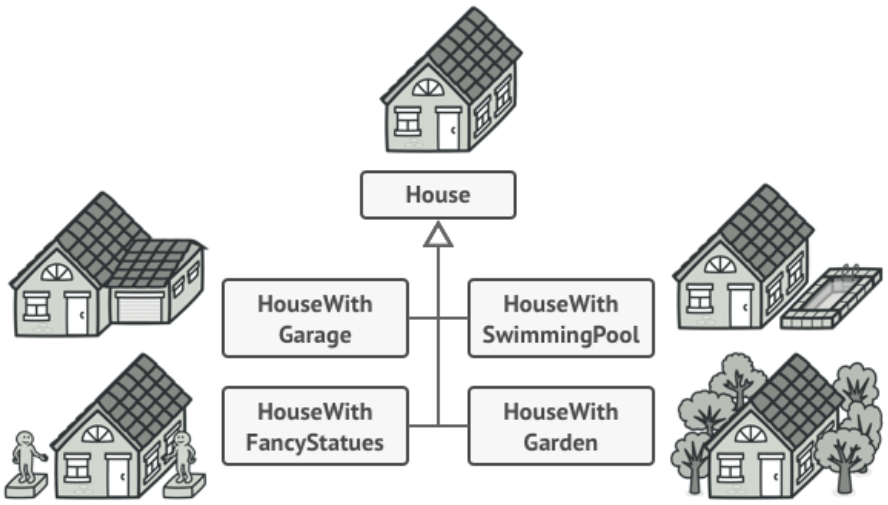
Takva inicijalizacija obično se izvodi unutar ogromnog konstruktora ili je “razbacana” u klijentskom kodu.

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

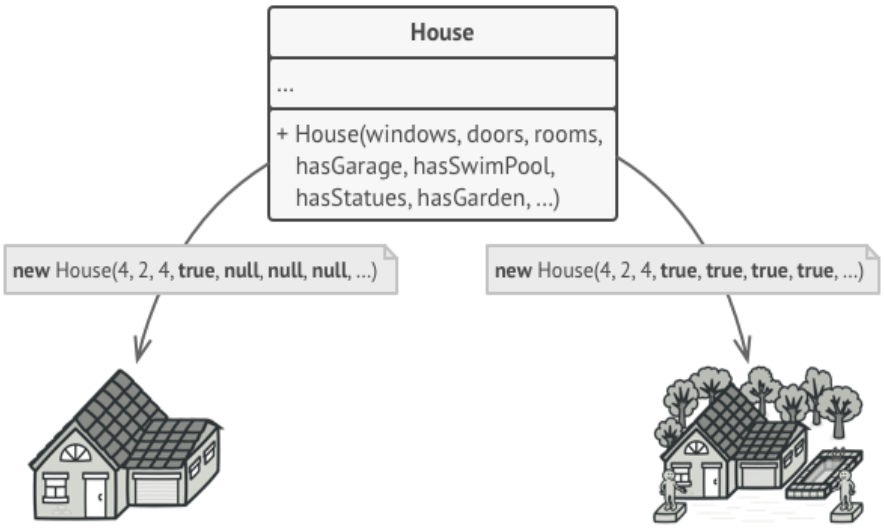
Pretpostavimo objekat House (osnovni model + garaža + bazen + ventilacija + grijanje + ...)

Hijerarhija klasa za sve moguće konfiguracije?

Nova potklasa za novu konfiguraciju?



Jedna osnovna klasa sa ogromnim kontruktorom i brojnim argumentima?



Kreacioni obrasci – Builder

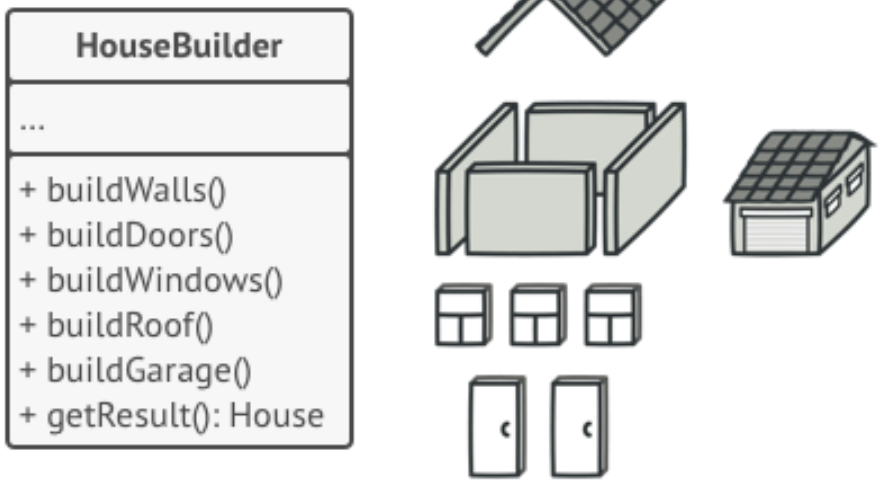
Motivacija za uvođenje buildera

Dobro projektno rješenje za izgradnju složenih objekata jeste **izdvajanje aplikativne logike za proces izgradnje** objekata iz klase koja reprezentuje objekat **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

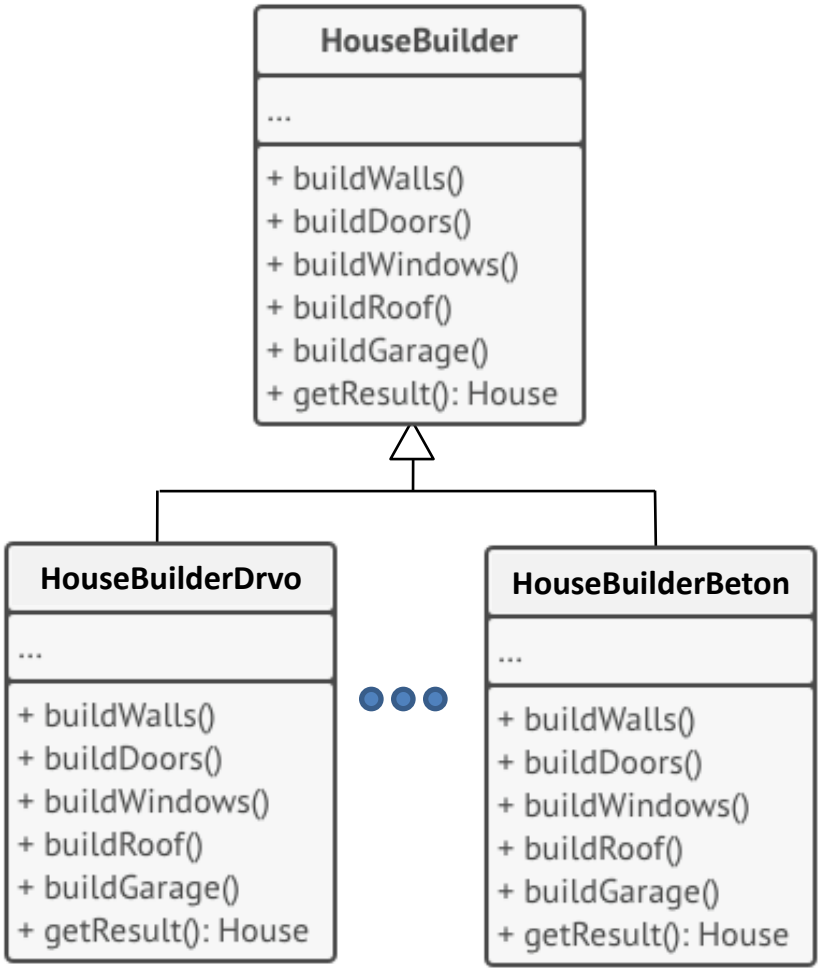
Builder je kreacioni obrazac koji omogućava formiranje složenih objekata korak po korak.



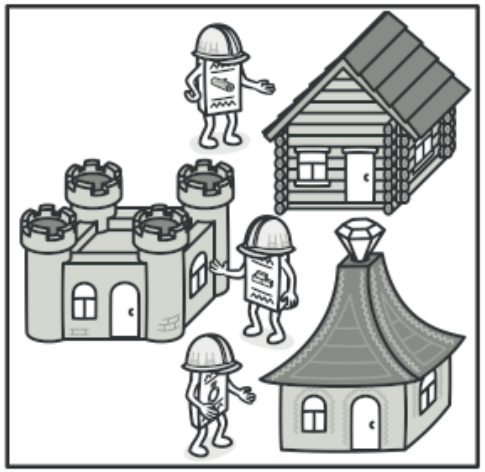
Kreacioni obrasci – Builder

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

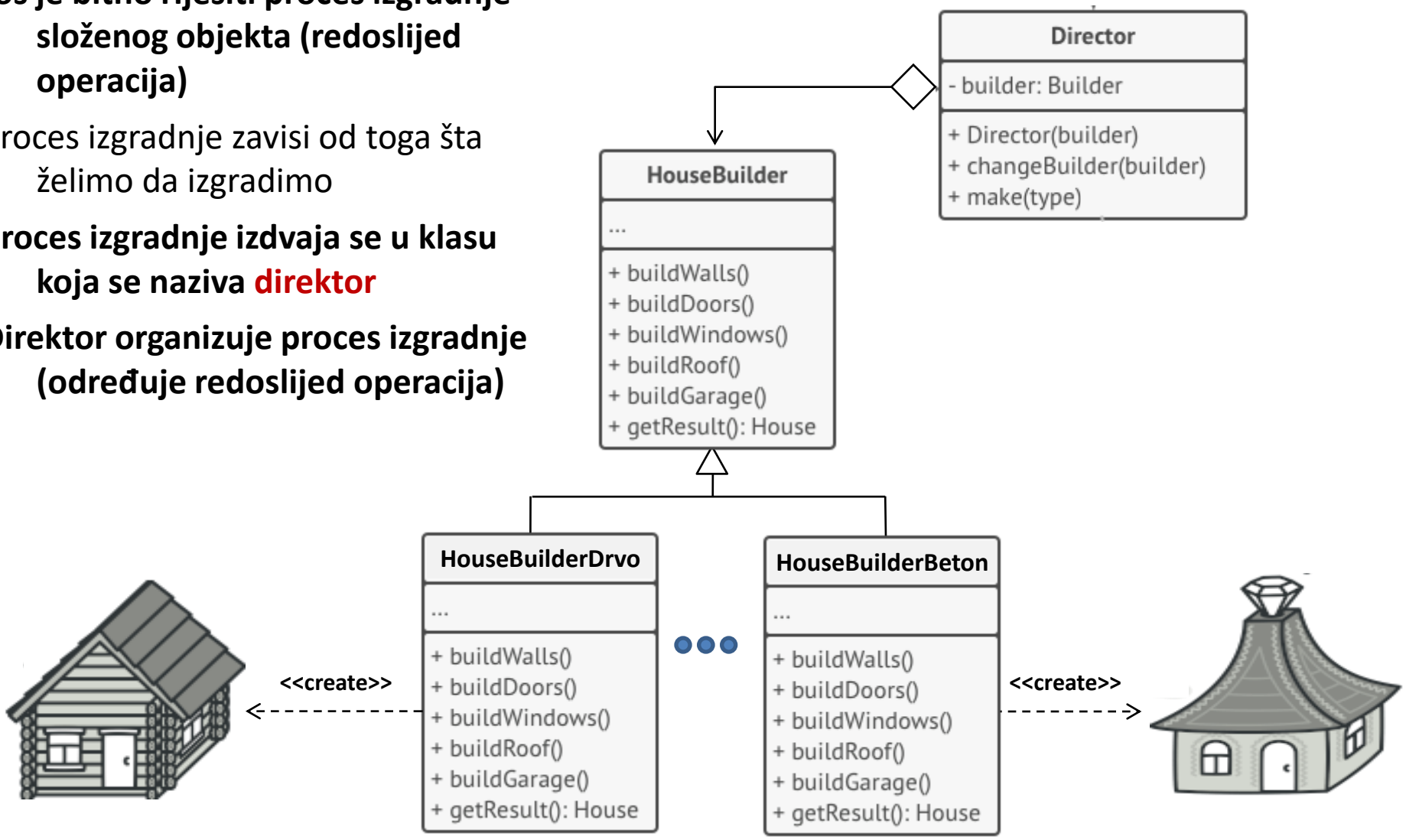
Motivacija za uvođenje buildera

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

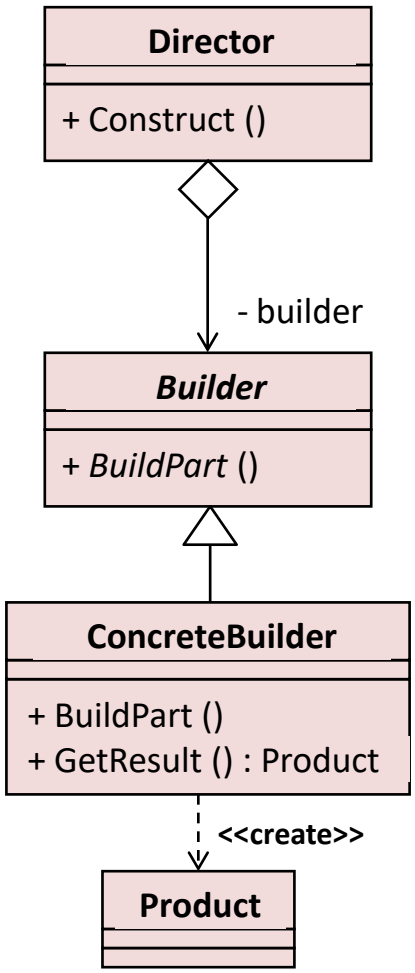
Direktor organizuje proces izgradnje (određuje redoslijed operacija)



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
- Obezbjeđuje interfejs za pribavljanje objekata klase Product

Director

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

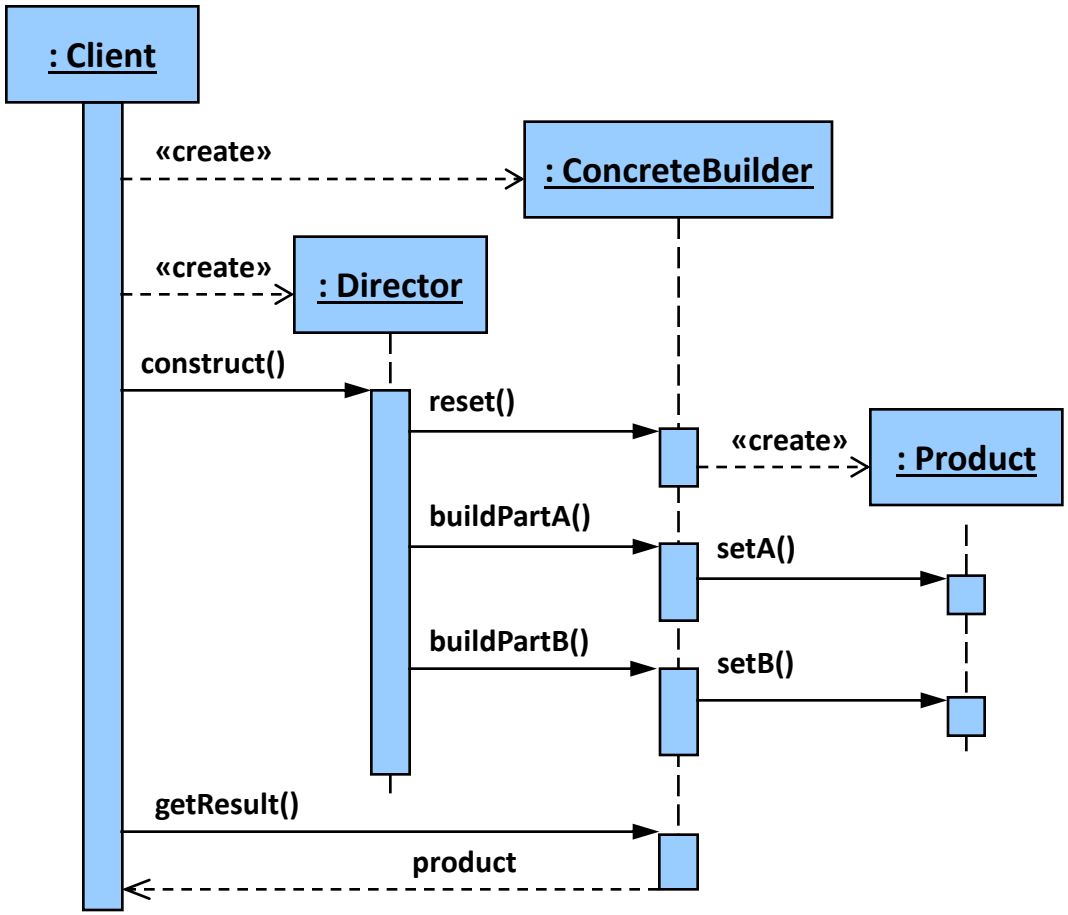
Product

- 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 sukcesivno dodavanje novog dijela)

– **BUILDER obrazac se uobičajeno koristi za kreiranje kompozicija** (COMPOSITE obrazac)

Apstraktna klasa za proizvode?

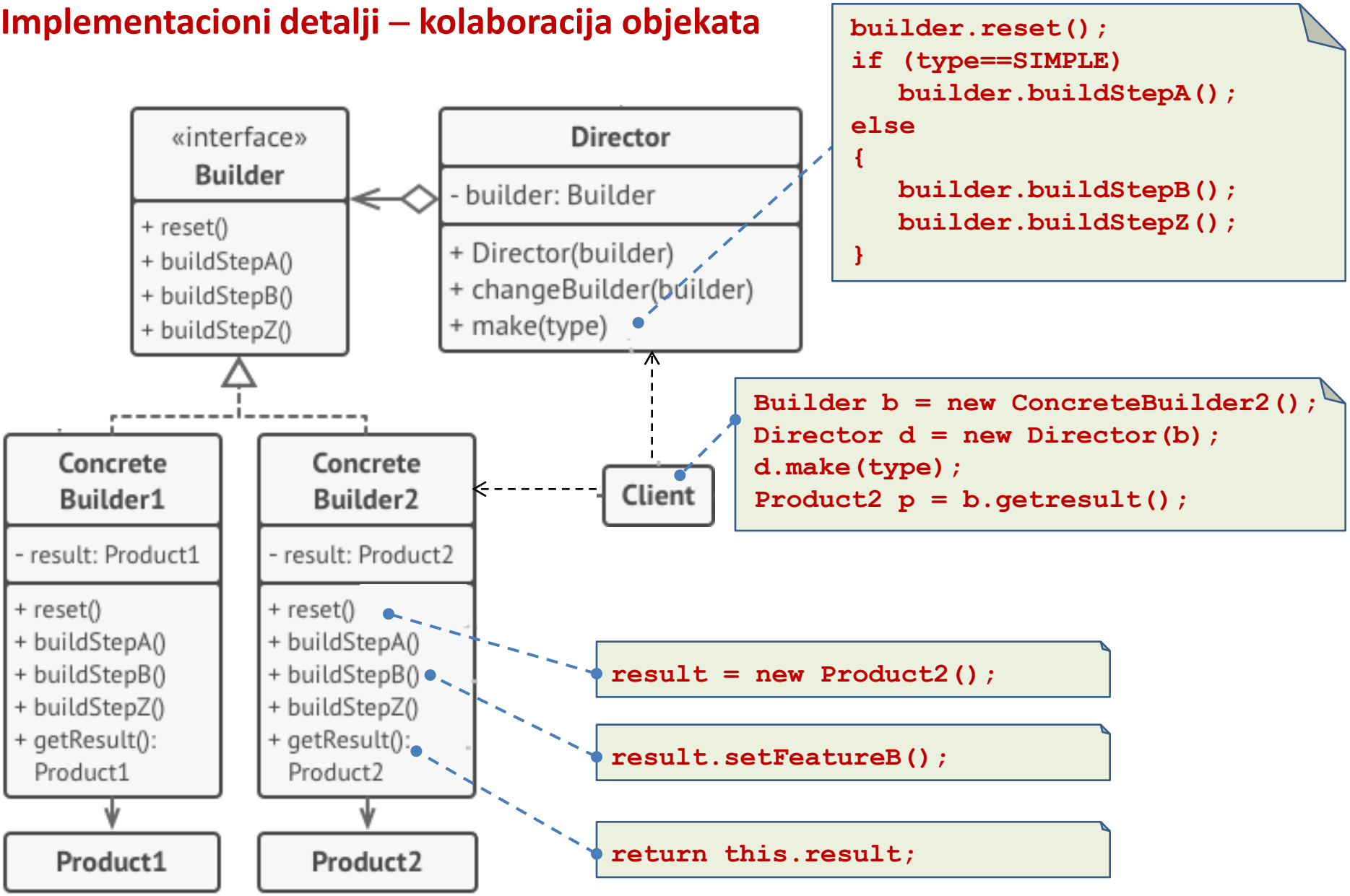
– može da se uvede apstraktna klasa za proizvode, u cilju formiranja odgovarajuće hijerarhije klasa

– često builderi generišu veoma različite proizvode, pa nekad apstraktni proizvodi nemaju smisla

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

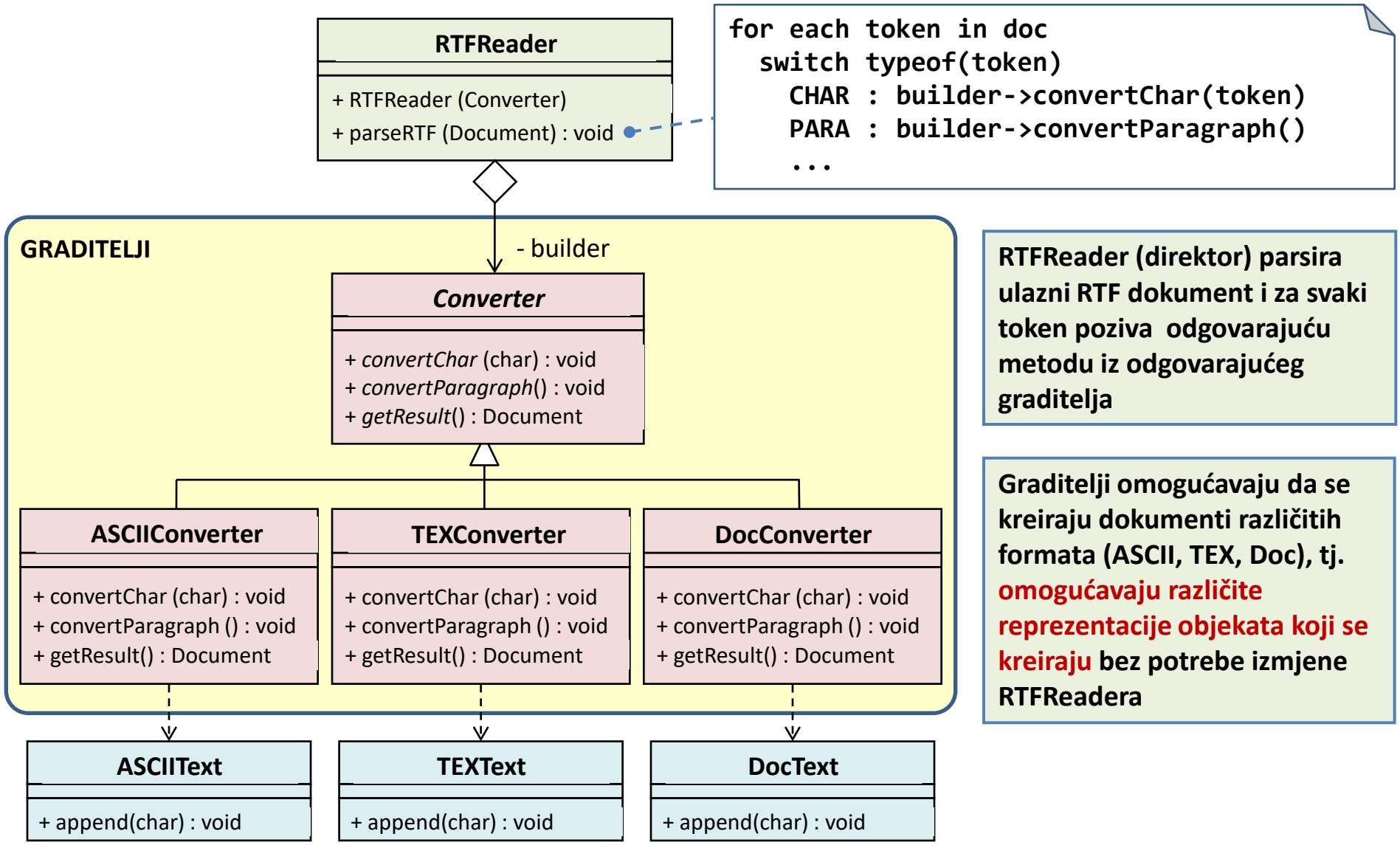
Kreacioni obrasci – Builder

Implementacioni detalji – kolaboracija objekata



Kreacioni obrasci – Builder

Primjer primjene BUILDER obrasca:

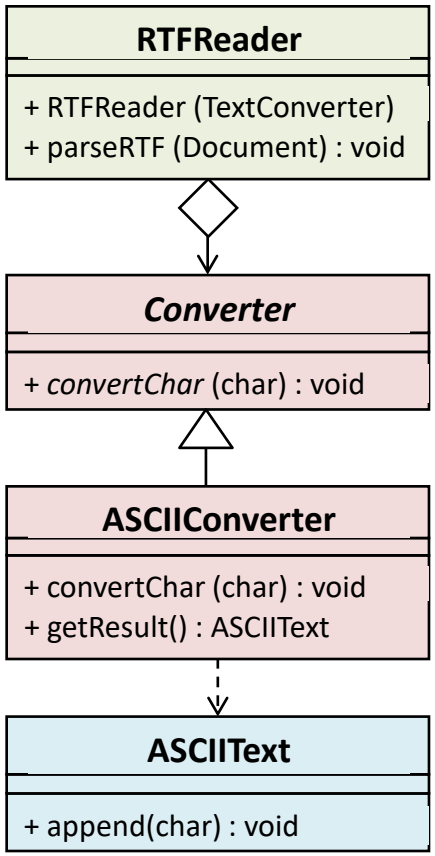


RTFReader (direktor) parsira ulazni RTF dokument i za svaki token poziva odgovarajuću metodu iz odgovarajućeg graditelja

Graditelji omogućavaju da se kreiraju dokumenti različitih formata (ASCII, TEX, Doc), tj. omogućavaju različite reprezentacije objekata koji se kreiraju bez potrebe izmjene RTFReadera

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.