PROGRAMSKO INŽENJERSTVO

Design Patterns

Laboratorijske vježbe

Što su oblikovni obrasci?

Provjerena rješenja za česte probleme u dizajnu softvera

Creational

Mehanizmi kreiranja objekata koji povećavaju fleksibilnost

Singleton, Factory, Builder, Prototype

Structural

Kako sastaviti objekte i klase u veće strukture

Adapter, Decorator, Facade, Proxy

Behavioral

Algoritmi i raspodjela odgovornosti među objektima

Observer, Strategy, Command, Template

CREATIONAL

Kreacijski obrasci

Singleton

Osigurava samo jednu instancu klase s globalnim pristupom

Factory Method

Delegira kreiranje objekata podklasama

Builder

Konstrukcija kompleksnih objekata korak po korak

Prototype

Kloniranje postojećih objekata

CREATIONAL

Singleton Pattern

```
public class DatabaseConnection {
private static DatabaseConnection instance;
private DatabaseConnection() {
System.out.println("Konekcija");
}
public static DatabaseConnection
getInstance() {
if (instance == null)
instance = new DatabaseConnection();
return instance;
}
}
```

📌 Kada koristiti?

- Samo jedna instanca u aplikaciji
- Globalni pristup instanci
- Kontrola zajedničkih resursa

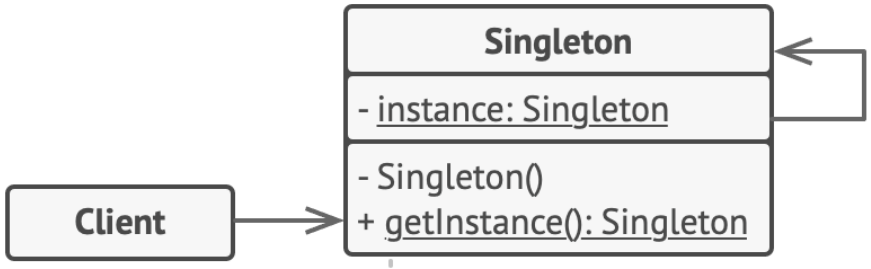
Primjeri

- **Logger** - sustav logiranja
- **Database** - connection pool
- **Config** - postavke aplikacije

⚠ Thread-safe: koristi synchronized

CREATIONAL

Singleton Pattern



1 The **Singleton** class declares the static method `getInstance` that returns the same instance of its own class.

The Singleton's constructor should be hidden from the client code. Calling the `getInstance` method should be the only way of getting the Singleton object.

```
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```

CREATIONAL

Factory Method Pattern

```
interface Animal {
String speak();
}
class Dog implements Animal {
public String speak() {
return "Vau!";
}
}
abstract class AnimalFactory {
abstract Animal createAnimal();
}
```

📌 Kada koristiti?

- Tip objekta nije poznat unaprijed
- Želimo delegirati kreiranje podklasama
- Potrebna fleksibilnost u kreiranju

Primjeri

- **DocumentFactory** - PDF, Word
- **UIFactory** - Button, TextField
- **LoggerFactory** - File, Console

💡 Koristi interface za maksimalnu fleksibilnost

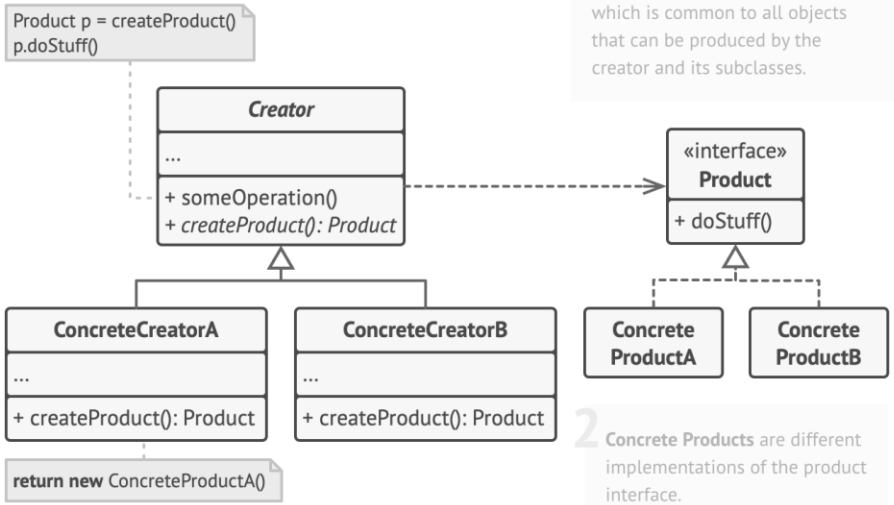
CREATIONAL

Factory Method Pattern

3 The **Creator** class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.

You can declare the factory method as abstract to force all subclasses to implement their own versions of the method. As an alternative, the base factory method can return some default product type.

Note, despite its name, product creation is **not** the primary responsibility of the creator. Usually, the creator class already has some core business logic related to products. The factory method helps to decouple this logic from the concrete product classes. Here is an analogy: a large software development company can have a training department for programmers. However, the primary function of the company as a whole is still writing code, not producing programmers.



1 The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.

2 **Concrete Products** are different implementations of the product interface.

4 **Concrete Creators** override the base factory method so it returns a different type of product.

Note that the factory method doesn't have to **create** new instances all the time. It can also return existing objects from a cache, an object pool, or another source.

CREATIONAL

Builder Pattern

```
class Pizza {
private String dough, sauce, topping;
static class Builder {
private Pizza pizza = new Pizza();
Builder dough(String d) {
pizza.dough = d; return this;
}
Pizza build() { return pizza; }
}
}
// Korištenje:
Pizza p = new Pizza.Builder()
.dough("thin").build();
```

📌 Kada koristiti?

- Objekt ima mnogo opcionalnih parametara
- Želimo čitljiv fluent API
- Korak-po-korak konstrukcija

Primjeri

- **StringBuilder** - Java
- **HttpRequest.Builder**
- **Lombok @Builder**

STRUCTURAL

Strukturni obrasci

Adapter
Omogućuje suradnju nekompatibilnih sučelja

Decorator
Dinamički dodaje nove funkcionalnosti objektu

Facade
Jednostavno sučelje za kompleksni podsustav

Proxy
Zamjenski objekt koji kontrolira pristup

Composite
Tretira pojedinačne objekte i kompozicije uniformno

Bridge
Odvaja apstrakciju od implementacije

STRUCTURAL

Adapter Pattern

```
interface MediaPlayer {
void play(String file);
}
class VlcPlayer {
void playVlc(String f) {...}
}
class VlcAdapter implements MediaPlayer {
private VlcPlayer vlc = new VlcPlayer();
public void play(String file) {
vlc.playVlc(file);
}
}
```

📌 Kada koristiti?

- Integracija legacy koda
- Korištenje external libraryja
- Nekompatibilna sučelja

Primjeri

- **Arrays.asList()** - Java
- **InputStreamReader**
- **JDBC Driver**

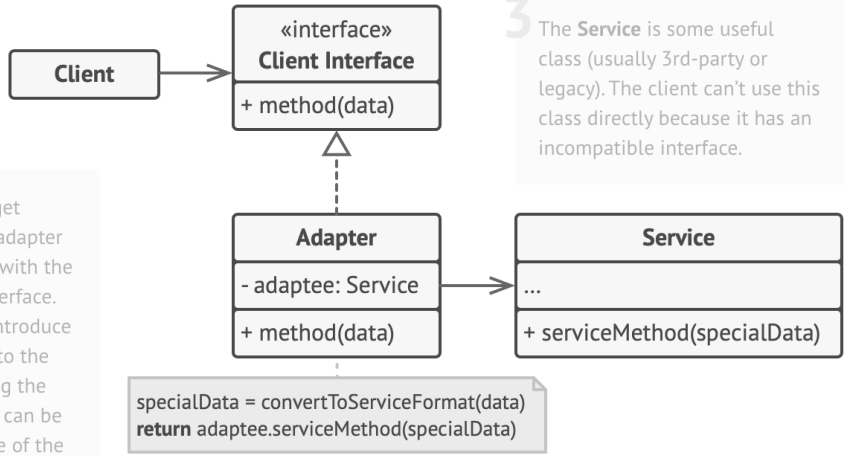
🕯️ Adapter = "prijevod" između sučelja

STRUCTURAL

Adapter Pattern

2 The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

1 The **Client** is a class that contains the existing business logic of the program.



3 The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

5 The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.

4 The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the adapter interface and translates them into calls to the wrapped service object in a format it can understand.

STRUCTURAL

Decorator Pattern

```
interface Coffee {
double cost();
}
class Espresso implements Coffee {
public double cost() { return 1.5; }
}
class MilkDecorator implements Coffee {
private Coffee coffee;
MilkDecorator(Coffee c) { coffee = c; }
public double cost() {
return coffee.cost() + 0.5;
}
}
```

📌 Kada koristiti?

- Dinamičko dodavanje funkcionalnosti
- Alternativa nasljeđivanju
- Kombiniranje ponašanja

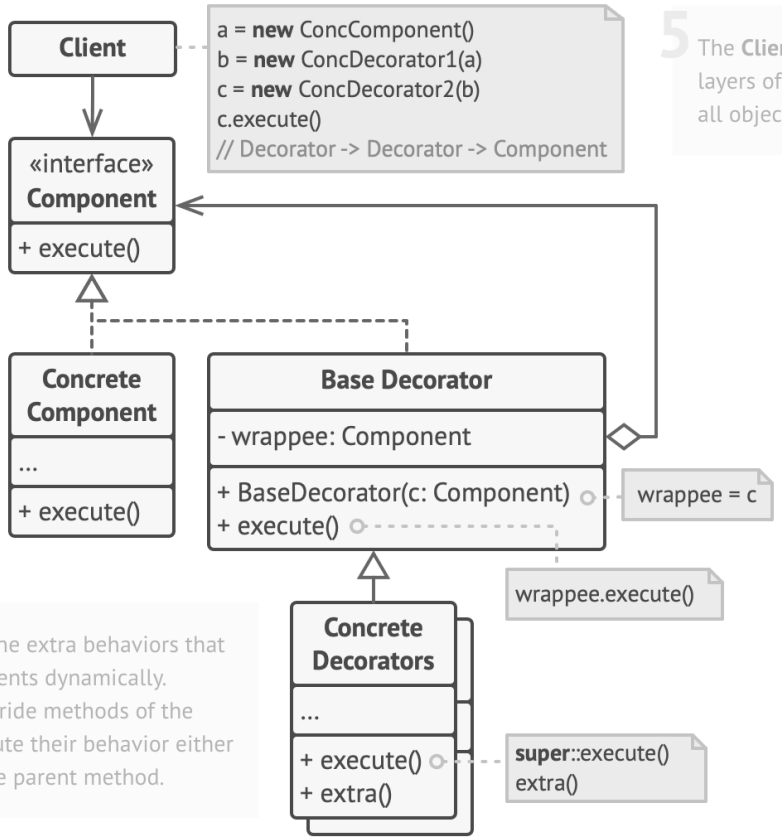
Primjeri

- **BufferedInputStream**
- **Collections.synchronizedList**
- **Spring @Transactional**

🎁 Decorator "omotava" objekt dodatnom funkcijom

STRUCTURAL

Decorator Pattern



1 The **Component** declares the common interface for both wrappers and wrapped objects.

2 **Concrete Component** is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.

3 The **Base Decorator** class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.

4 **Concrete Decorators** define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.

5 The **Client** can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.

STRUCTURAL

Facade Pattern

```
class CPU { void start() {...} }
class Memory { void load() {...} }
class HardDrive { void read() {...} }
class ComputerFacade {
private CPU cpu = new CPU();
private Memory mem = new Memory();
private HardDrive hd = new HardDrive();
void startComputer() {
cpu.start(); mem.load(); hd.read();
}
}
```

📌 Kada koristiti?

- Skrivanje kompleksnosti podsustava
- Jedna ulazna točka za više klasa
- Pojednostavljivanje API-ja

Primjeri

- **SLF4J** - logging facade
- **EntityManager** - JPA
- **RestTemplate** - Spring

🏠 Facade = "ulazna vrata" u kompleksni sustav

BEHAVIORAL

Obrasci ponašanja

Observer
Obavještava pretplaćene objekte o promjenama

Strategy
Definira obitelj algoritama koji se mogu izmjenjivati

Template
Definira kostur algoritma, koraci u podklasama

Command
Enkapsulira zahtjev kao objekt

State
Mijenja ponašanje ovisno o stanju

Iterator
Sekvencijalni pristup elementima

Mediator
Centralizira kompleksnu komunikaciju

Chain
Lanac rukovatelja zahtjevima

BEHAVIORAL

Observer Pattern

```
interface Observer {
void update(String news);
}
class NewsAgency {
private List<Observer> observers = new ArrayList<>();
void addObserver(Observer o) {
observers.add(o);
}
void setNews(String news) {
observers.forEach(o -> o.update(news));
}
}
```

📌 Kada koristiti?

- Publish-subscribe scenariji
- Event handling sustavi
- Reaktivne aplikacije

Primjeri

- **PropertyChangeListener**
- **RxJava Observable**
- **Spring Events**

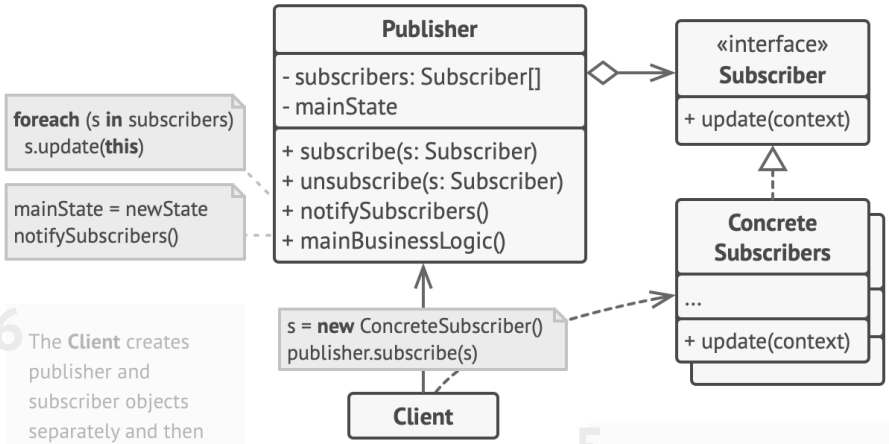
"Pretplata" na promjene stanja objekta

BEHAVIORAL

Observer Pattern

1 The **Publisher** issues events of interest to other objects. These events occur when the publisher changes its state or executes some behaviors. Publishers contain a subscription infrastructure that lets new subscribers join and current subscribers leave the list.

2 When a new event happens, the publisher goes over the subscription list and calls the notification method declared in the subscriber interface on each subscriber object.



3 The **Subscriber** interface declares the notification interface. In most cases, it consists of a single `update` method. The method may have several parameters that let the publisher pass some event details along with the update.

4 **Concrete Subscribers** perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher isn't coupled to concrete classes.

5 Usually, subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method. The publisher can pass itself as an argument, letting subscriber fetch any required data directly.

6 The **Client** creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

BEHAVIORAL

Strategy Pattern

```
interface PaymentStrategy {
void pay(int amount);
}
class CreditCard implements PaymentStrategy {
public void pay(int amt) {
System.out.println("Card: " + amt);
}
}
class ShoppingCart {
private PaymentStrategy strategy;
void setStrategy(PaymentStrategy s) {
strategy = s;
}
}
```

📌 Kada koristiti?

- Više algoritama za isti zadatak
- Izbor algoritma u runtime
- Zamjena za switch/if-else

Primjeri

- **Comparator** - sortiranje
- **Validator** - validacija
- **Compression** - ZIP, RAR

🔁 Mijenja algoritam "u hodu" bez promjene klijenta

BEHAVIORAL

Template Method Pattern

```
abstract class DataMiner {
// Template method
final void mine() {
openFile();
extractData();
parseData();
closeFile();
}
abstract void openFile();
abstract void extractData();
void parseData() { ... }
void closeFile() { ... }
}
```

📌 Kada koristiti?

- Isti algoritam, različiti koraci
- Definiranje "kostura" operacije
- Kontrola redoslijeda koraka

Primjeri

- **HttpServlet** - doGet, doPost
- **JUnit @Before** - setUp
- **AbstractList**

📋 final metoda osigurava redoslijed koraka

Koji pattern odabrati?

Singleton vs Factory
Singleton: jedna instanca | Factory: fleksibilno kreiranje više tipova

Builder vs Factory
Builder: kompleksni objekti korak-po-korak | Factory: jednostavno kreiranje

Adapter vs Decorator
Adapter: promjena sučelja | Decorator: dodavanje funkcionalnosti

Facade vs Adapter
Facade: pojednostavljenje API-ja | Adapter: kompatibilnost sučelja

Strategy vs Template
Strategy: cijeli algoritam | Template: samo koraci algoritma

Observer vs Mediator
Observer: 1-to-many obavijesti | Mediator: centralizirana komunikacija

💡 Savjet: Kombinacija patterna često daje najbolje rezultate!

PROGRAMSKO INŽENJERSTVO

Projektni zadatak

Implementirajte aplikaciju koristeći minimalno 3 design patterna iz različitih kategorija

1x Creational | 1x Structural | 1x Behavioral

- Svaki član tima mora u svom dijelu projekta implementirati po jedan oblikovni obrazac iz svake od kategorije (I3 – 1 bod i I8 – 1 bod)