PROGRAMSKI ALATI ZA RAZVOJ SOFTVERA

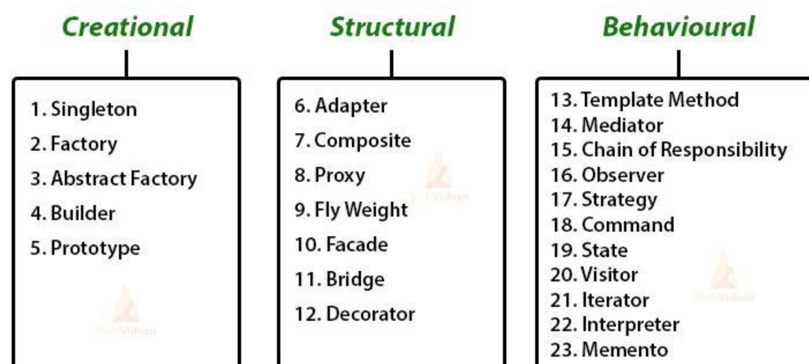
Vežba 5 (radi se dve nedelje)

Dizajn patterni

Dizajn paterni (šabloni) su standardizovana rešenja za softverske probleme, koja omogućavaju efikasnijii fleksibilniji razvoj aplikacija. Primena dizajn paterna omogućava programerima da se fokusiraju na visoke nivoe arhitekture, smanjujući potrebu za ponovljenim rešavanjem istih problema. Ovi šabloni pomažu u razvoju čistog i modularnog koda, koji je lakše održavati i proširivati. Koristeći dizajn paterne, timovi mogu obezbediti konzistentnost u razvoju aplikacija, što vodi ka boljim performansama, manjem broju grešaka i lakšoj saradnji među programerima. U ovoj vežbi ćemo se fokusirati na paterne u Javi, ali pokazaćemo i neke primere u Pythonu.

Postoje tri osnovne kategorije dizajn paterna:

1. **Kreacioni paterni (Creational Patterns)**: Ovi paterni fokusiraju se na način kreiranja objekata. Njihov cilj je da olakšaju kreiranje objekata tako da ne bude direktno zavisno od klase koju objekat instancira. Kreacioni paterni omogućavaju fleksibilnost u izboru vrste objekta, čineći kod manje zavisnim od specifičnih klasa.
2. **Strukturni paterni (Structural Patterns)**: Ovi paterni se bave organizacijom objekata i njihovim međusobnim odnosima, kako bi se stvorile veće, funkcionalne komponente bez previše složenosti. Ovi paterni omogućavaju fleksibilno spajanje objekata i klasa tako da se ponašaju kao jedan entitet, smanjujući vezanost između komponenti.
3. **Ponašajni paterni (Behavioral Patterns)**: Ponašajni paterni se bave načinima na koje objekti međusobno komuniciraju i kako se ponašanja objekata organizuju i distribuiraju. Ovi paterni olakšavaju deljenje odgovornosti i upravljanje tokovima kontrole među objektima, često smanjujući potrebu za složenim logičkim granama.



1. Singleton Pattern

Singleton je jedan od najjednostavnijih kreacionih dizajn paterna. Cilj ovog paterna je da osigura postojanje samo jedne instance klase tokom trajanja aplikacije i da pruži globalnu tačku pristupa toj instanci. Najčešće se koristi za implementaciju globalnih objekata kao što su konekcije s bazom podataka, konfiguracione postavke ili logovanje.

Osnovni princip Singletona je sledeći:

1. Konstruktori su privatni, čime se onemogućava direktno instanciranje klase.
2. Pristup instanci se omogućava kroz statički metod.
3. Često se koristi lazy initialization da bi se instanca kreirala tek kada je prvi put zatražena.

Prednosti Singletona:

1. Obezbeđuje da postoji samo jedna instanca klase tokom trajanja aplikacije, idealno za navedene resurse poput baza podataka i konfiguracija.
2. Omogućava deljenje iste instance između delova aplikacije.
3. Kreira instancu samo kada je potrebna, štedi memoriju i ubrzava pokretanje aplikacije.
4. Sprečava nepotrebno kreiranje objekata, što poboljšava performanse.
5. Pojednostavljuje praćenje i kontrolu resursa, poput logovanja ili mrežnih konekcija.

Scenario

Kao što je već napomenuto, zamislite da razvijate aplikaciju koja koristi bazu podataka. Treeba da obezbedite da postoji samo jedna instanca konektora ka bazi podataka, jer višestruke instanci mogu prouzrokovati probleme sa performansama ili konfliktima u podacima.

Primeri u Javi

```
public class DatabaseConnection {
    private static DatabaseConnection instance;

    private DatabaseConnection() {
        // Privatni konstruktor kako bi se sprečilo stvaranje instanci izvan klase
       System.out.println("Kreirana je konekcija sa bazom");
    }

    public static DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }
}
// Glavna klasa za demonstraciju
```

```
public class SingletonDemo {
    public static void main(String[] args) {
        // Dobijanje instance klase DatabaseConnection
        DatabaseConnection connection1 = DatabaseConnection.getInstance();
        DatabaseConnection connection2 = DatabaseConnection.getInstance();

        // Provera da li su obe instance iste
        if (connection1 == connection2) {
            System.out.println("Obe instance su identične.");
        } else {
            System.out.println("Instance nisu iste.");
        }
    }
}
```

2. Factory Pattern

Factory je kreacioni dizajn patern koji omogućava kreiranje objekata bez eksplicitnog navođenja njihovih konkretnih klasa. Fabrika definiše interfejs za kreiranje objekata, dok konkretne podklase odlučuju koji će se objekat instancirati. Cilj ovakve implementacije je da se izbegne direktno instanciranje objekata unutar klijentskog koda i da se poveća fleksibilnost i proširivost aplikacije.

Princip rada

1. Definiše se zajednički interfejs ili apstraktna klasa za objekte koji će se kreirati.
2. Implementira se „fabrika" koja sadrži metodu za kreiranje objekata na osnovu nekog kriterijuma (npr. tipa).
3. Klijent koristi fabriku za kreiranje objekata bez eksplicitnog pominjanja konkretnih klasa.

Scenario

Kod aplikacije koja kreira različite vrste vozila (automobili, bicikli, motocikli), Factory pattern vam omogućava da centralizujete logiku za kreiranje objekata i olakšate dodavanje novih tipova vozila.

Primer u Javi

```
abstract class Vehicle {
    abstract void drive();
}

class Car extends Vehicle {
    @Override
    void drive() {
        System.out.println("Driving a car");
    }
}
```

```java
class Bike extends Vehicle {
    @Override
    void drive() {
        System.out.println("Riding a bike");
    }
}

class VehicleFactory {
    public static Vehicle createVehicle(String type) {
        if (type.equalsIgnoreCase("car")) {
            return new Car();
        } else if (type.equalsIgnoreCase("bike")) {
            return new Bike();
        }
        return null;
    }
}

// Glavna klasa za demonstraciju Factory paterna
public class FactoryDemo {
    public static void main(String[] args) {
        // Kreiranje objekta tipa Car pomoću Factory metode
        Vehicle car = VehicleFactory.createVehicle("car");
        car.drive();  // Ispisuje: Driving a car

        // Kreiranje objekta tipa Bike pomoću Factory metode
        Vehicle bike = VehicleFactory.createVehicle("bike");
        bike.drive();  // Ispisuje: Riding a bike
    }
}
```

3. Abstract Factory Pattern

Abstract Factory je kreacioni dizajn patern koji omogućava kreiranje porodica povezanih objekata bez specificiranja njihovih konkretnih klasa. Patern je koristan kada aplikacija mora podržati više proizvoda iz različitih porodica, pri čemu svaki proizvod mora biti kompatibilan s ostalima. Abstract Factory omogućava lako proširivanje sistema dodavanjem novih porodica objekata bez modifikovanja postojećeg koda, čime se postiže veća fleksibilnost i održivost.

Ovaj patern je idealan kada želite da omogućite kreiranje objekata iz različitih platformi, ali sa zajedničkim interfejsom. Ovaj pristup poboljšava održavanje koda, jer omogućava lako menjanje implementacija objekata bez uticaja na ostatak sistema.

Princip rada

1. Definiše se interfejs za kreiranje porodica povezanih objekata.
2. Konkretne implementacije ovog interfejsa definišu stvarne porodice proizvoda.
3. Klijent koristi interfejs fabrike da dobije odgovarajuće objekte bez potrebe da zna njihove konkretne klase.

Scenario

Razmotrite aplikaciju za generisanje korisničkog interfejsa (UI) koja podržava različite operativne sisteme, kao što su Windows i MacOS. Abstract Factory pomaže u kreiranju porodica povezanih komponenti, poput dugmadi i menija, koji odgovaraju svakom stilu, bez eksplicitnog kodiranja konkretnih klasa.

Primer u Javi

```
// Interfejs za dugme
interface Button {
    void render();
}

// Konkretna dugmad
class WindowsButton implements Button {
    public void render() {
        System.out.println("Rendering Windows Button");
    }
}

class MacOSButton implements Button {
    public void render() {
        System.out.println("Rendering MacOS Button");
    }
}
// Interfejs za meni
interface Menu {
    void render();
}
// Konkretni meniji
class WindowsMenu implements Menu {
    public void render() {
        System.out.println("Rendering Windows Menu");
    }
}
```

```java
class MacOSMenu implements Menu {
    public void render() {
        System.out.println("Rendering MacOS Menu");
    }
}

// Apstraktna fabrika
interface UIFactory {
    Button createButton();
    Menu createMenu();
}

// Konkretne fabrike
class WindowsUIFactory implements UIFactory {
    public Button createButton() {
        return new WindowsButton();
    }

    public Menu createMenu() {
        return new WindowsMenu();
    }
}

class MacOSUIFactory implements UIFactory {
    public Button createButton() {
        return new MacOSButton();
    }

    public Menu createMenu() {
        return new MacOSMenu();
    }
}

public class Main {
    public static void main(String[] args) {
        UIFactory uiFactory;

        // Pretpostavimo da aplikacija treba da kreira interfejs za Windows
        String platform = "Windows";
```

```
        if (platform.equalsIgnoreCase("Windows")) {
            uiFactory = new WindowsUIFactory();
        } else {
            uiFactory = new MacOSUIFactory();
        }

        // Kreiramo dugme i meni koristeći apstraktnu fabriku
        Button button = uiFactory.createButton();
        Menu menu = uiFactory.createMenu();

        // Renderovanje dugmeta i menija
        button.render();
        menu.render();
    }
}
```

4. Decorator Pattern

Decorator pattern je strukturni dizajn pattern koji omogućava dinamičko dodavanje novih funkcionalnosti objektima bez potrebe za promenom njihove strukture. Ovo je korisno kada ne želite da menjate već postojeći kod, ali želite da dodate nove sposobnosti objektima ili da prilagodite njihove ponašanja. U suštini, ovakav pattern omogućava da se nadograde objekti funkcionalnostima postepeno, kroz "dekoratore" koji se primenjuju na originalne objekte.

Glavna prednost Decorator patterna je njegova fleksibilnost u odnosu na nasleđivanje. Dok nasleđivanje može postati rigidno jer se nove funkcionalnosti moraju implementirati u podklasama, Decorator omogućava da se nove osobine dodaju objektima u vreme izvršavanja bez potrebe za stvaranjem velikog broja klasa.

Scenario:

Zamislite kafić koji nudi različite vrste kafa, a kupci mogu da dodaju mleko, šećer, šlag, itd. Umesto da kreirate klase za sve kombinacije kafe i dodatka (npr. EspressoSaMlekomISecerom), koristite decorator da dinamički dodate funkcionalnosti osnovnoj kafi.

Primer u Javi

```
interface Coffee {
    double cost(); // Metoda koja računa cenu kafe
}
class SimpleCoffee implements Coffee {
    @Override
    public double cost() {
        return 5.0; // Osnovna cena kafe
    }
}
```

```java
abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee; // Objekat koji ćemo obogatiti

    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }

    @Override
    public double cost() {
        return coffee.cost();
    }
}

class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double cost() {
        return super.cost() + 1.5; // Dodajemo cenu mleka
    }
}

class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double cost() {
        return super.cost() + 0.5; // Dodajemo cenu šećera
    }
}

public class Main {
    public static void main(String[] args) {
        Coffee coffee = new SimpleCoffee(); // Početna kafa
        System.out.println("Cost of simple coffee: " + coffee.cost());
```

```
        // Dodajemo mleko
        coffee = new MilkDecorator(coffee);
        System.out.println("Cost of coffee with milk: " + coffee.cost());

        // Dodajemo šećer
        coffee = new SugarDecorator(coffee);
        System.out.println("Cost of coffee with milk and sugar: " + coffee.cost());
    }
}
```

U ovom primeru, **SimpleCoffee** je osnovna klasa, dok su **MilkDecorator** i **SugarDecorator** dekoratori koji dodaju funkcionalnosti (mleko i šećer). Svaki dekorator poziva osnovnu cenu kafe i dodaje dodatne troškove.

5. Observer Pattern

Observer je jedan od ponašajnih dizajn paterna koji definiše relaciju "jedan-prema-više" između objekata. Kada jedan objekat (poznat kao subject) promeni svoje stanje, svi povezani objekti (observers) automatski bivaju obavešteni i ažurirani. Ovakav pristup pruža lakšu implementaciju događajno vođenih sistema gde različiti delovi aplikacije treba da reaguju na promene, čime se postiže bolja modularnost i fleksibilnost koda.

Princip rada

1. **Subject (posmatrani objekat):** Objekat koji drži listu observera i obaveštava ih o promenama u svom stanju.
2. **Observers (posmatrači):** Objekti koji se prijavljuju na subject i implementiraju metod za ažuriranje kada subject promeni stanje.
3. **Mehanizam obaveštavanja:** Kada se stanje subject-a promeni, on poziva metod za ažuriranje na svim registrovanim observerima.

Scenario

Zamislite neku aplikaciju za vremensku prognozu. Postoji objekat WeatherStation (subject) koji periodično osvežava podatke o temperaturi, pritisku i vlažnosti. Na ovaj objekat su prijavljeni različiti observers, npr aplikacija za mobilni telefon, ekran za prikaz na javnim mestima i servis za slanje upozorenja putem email-a. Kada se podaci promene, svi observeri se automatski ažuriraju.

Primer u Javi

```
import java.util.ArrayList;
import java.util.List;

class WeatherStation {
    private List<Observer> observers = new ArrayList<>();
    private float temperature;
```

```java
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    public void setTemperature(float temperature) {
        this.temperature = temperature;
        notifyObservers();
    }

    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature);
        }
    }
}

// Observer
interface Observer {
    void update(float temperature);
}

// Concrete Observers
class PhoneDisplay implements Observer {
    @Override
    public void update(float temperature) {
        System.out.println("PhoneDisplay: Temperature updated to " + temperature + "°C");
    }
}
class PublicDisplay implements Observer {
    @Override
    public void update(float temperature) {
        System.out.println("PublicDisplay: Temperature updated to " + temperature + "°C");
    }
}
```

```
// Main klasa za demonstraciju Observer patterna
public class ObserverPatternDemo {
    public static void main(String[] args) {
        // Kreiramo WeatherStation objekat koji je subject
        WeatherStation station = new WeatherStation();

        // Kreiramo dva različita observera koji će se prijaviti na promene
        Observer phoneDisplay = new PhoneDisplay();
        Observer publicDisplay = new PublicDisplay();

        // Dodajemo observere u listu posmatrača WeatherStation objekta
        station.addObserver(phoneDisplay);
        station.addObserver(publicDisplay);

        // Postavljamo temperaturu, što izaziva obaveštavanje svih posmatrača
        station.setTemperature(25.0f);  // Svi posmatrači će biti obavešteni o ovoj promeni
        station.setTemperature(30.0f);  // Isto
    }
}
```

6. Strategy Pattern

Strategy je ponašajni dizajn patern koji omogućava dinamičko menjanje ponašanja objekta na osnovu njegove strategije. Ovaj patern omogućava jednostavno zamenu i primenu različitih algoritama bez potrebe za modifikovanjem samog objekta. U suštini, on omogućava izbor između različitih algoritama tokom izvođenja programa (runtime), čime se poboljšava fleksibilnost sistema. Prednost ovog paterna leži u mogućnosti lakog proširivanja sistema dodavanjem novih strategija, dok se postojeći kod ostavlja netaknutim.

Princip rada

- Strategy Pattern deli algoritme ili ponašanja u različite klase, nazvane strategije.
- Objekat koji koristi strategiju poziva odgovarajući metod u strategiji bez potrebe da zna koji je tačno algoritam implementiran.
- Svaka strategija implementira isti interfejs, pa može biti zamenjena u runtime-u, omogućavajući fleksibilnost u izboru algoritma.

Scenario

Zamislite aplikaciju koja obračunava popuste na proizvode. Na osnovu različitih strategija popusta, cena proizvoda se može promeniti na različite načine (popust u procentima, fiksni popust, popust na osnovu sezonskih akcija itd.). Umesto da implementirate svaku vrstu popusta u glavnoj klasi proizvoda, možete koristiti patern za upravljanje različitim strategijama popusta.

Primer u Javi

```
// Interfejs za strategiju popusta
interface DiscountStrategy {
    double applyDiscount(double price);
}

// Konkretne strategije popusta
class PercentageDiscount implements DiscountStrategy {
    private double percentage;

    public PercentageDiscount(double percentage) {
        this.percentage = percentage;
    }

    @Override
    public double applyDiscount(double price) {
        return price - (price * (percentage / 100));
    }
}

class FixedDiscount implements DiscountStrategy {
    private double discount;

    public FixedDiscount(double discount) {
        this.discount = discount;
    }

    @Override
    public double applyDiscount(double price) {
        return price - discount;
    }
}
class SeasonalDiscount implements DiscountStrategy {
    @Override
    public double applyDiscount(double price) {
        return price * 0.8;  // 20% popusta za sezonske akcije
    }
}
```

```
// Klasa koja koristi strategiju
class Product {
    private String name;
    private double price;
    private DiscountStrategy discountStrategy;

    public Product(String name, double price, DiscountStrategy discountStrategy) {
        this.name = name;
        this.price = price;
        this.discountStrategy = discountStrategy;
    }

    public double getPrice() {
        return discountStrategy.applyDiscount(price);
    }
}

// Main klasa
public class StrategyPatternDemo {
    public static void main(String[] args) {
        // Kreiramo proizvode sa različitim strategijama popusta
        Product product1 = new Product("Laptop", 1000, new PercentageDiscount(10));
                                                 // 10% popusta
        Product product2 = new Product("Phone", 800, new FixedDiscount(50));
                                                 // 50 fiksni popust
        Product product3 = new Product("TV", 1200, new SeasonalDiscount());
                                                 // 20% sezonski popust
        // Prikazivanje cena sa popustima
        System.out.println("Laptop cena sa popustom: " + product1.getPrice());
        System.out.println("Phone cena sa popustom: " + product2.getPrice());
        System.out.println("TV cena sa popustom: " + product3.getPrice());
    }
}
```

Neophodna literatura

Za detaljnije karakteristike opisanih patterna, pogledajte obavezno link u nastavku. Tu je za svaki pattern dato i teorijsko objašnjenje, i dijagrami, i primeri kodova. Obratite pažnju i na patterne kao što su **Composite, Builder, Prototype.**

https://refactoring.guru/design-patterns/java

Drugi link koji može pogledati je ovaj odličan YouTube kurs:
https://www.youtube.com/watch?v=pjDi4LVI8J4&ab_channel=AlphaBrainsCourses

Praktični primer

Treba da razvijete aplikaciju koja obaveštava studente kada su njihovi rezultati ispita objavljeni. Studenti mogu odabrati različite obavesti putem emaila ili mobilnih notifikacija. U ovom slučaju koristićemo observer kako bi omogućili studentima da se pretplate na obaveštenja o rezultatima.

```
// Apstraktni posmatrac (Observer)
interface Observer {
    void update(String message);
}
// Konkretni posmatrači (Studenti)
class EmailStudent implements Observer {
    @Override
    public void update(String message) {
        System.out.println("Obaveštenje putem Email-a: " + message);
    }
}

class MobileStudent implements Observer {
    @Override
    public void update(String message) {
        System.out.println("Obaveštenje putem Mobilnog App: " + message);
    }
}

// Apstraktna tema (Subject)
interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

// Konkretna tema (Ispitni sistem)
class ExamResultsSystem implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String examResults;

    public void setExamResults(String results) {
        this.examResults = results;
        notifyObservers(); // Obaveštavamo sve posmatrače
    }
    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }
```

```java
    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(examResults);
        }
    }
}

public class ObserverPatternExample {
    public static void main(String[] args) {
        ExamResultsSystem examSystem = new ExamResultsSystem();

        // Kreiranje studenata koji će primati obaveštenja
        Observer emailStudent = new EmailStudent();
        Observer mobileStudent = new MobileStudent();

        // Registracija studenata kao posmatrača
        examSystem.addObserver(emailStudent);
        examSystem.addObserver(mobileStudent);

        // Postavljanje rezultata ispita, što pokreće obaveštenja
        examSystem.setExamResults("Rezultati ispita su objavljeni!");
    }
}
```

Zadatak za samostalni rad

Treba da razvijete aplikaciju za upravljanje korisnicima u banci. Aplikacija mora da podrži različite tipove korisnika, kao što su "Standardni korisnici", "VIP korisnici" i "Preduzetnici". Svaki tip korisnika ima specifične privilegije, kao što su maksimalni iznos za povlačenje gotovine, dostupnost specijalnih usluga i pristup određenim bankovnim računima. Svaka vrsta korisnika treba da može da dobije specifičan paket usluga, ali želite da omogućite lako dodavanje novih vrsta korisnika bez modifikacije postojećeg koda.

Vaš zadatak je da osmislite sistem koji omogućava kreiranje različitih tipova korisnika, pri čemu će biti lako dodavati nove vrste korisnika sa specifičnim pravilima i uslovima, bez potrebe za menjanjem postojećeg koda koji se odnosi na korisnike.

Koji dizajn pattern biste primenili za rešavanje ovog problema? Objasnite zašto.