

Singleton Pattern

U sva tri primera Singleton obrazac rešava isti problem na različitim domenima: obezbeđuje da postoji samo jedan objekat koji upravlja zajedničkim resursom ili informacijom. Kod biblioteke to su globalna podešavanja sistema, kod kopirnice centralni red štampe, a kod online prodavnice jedinstveni logger.

Zadatak 1 – Podešavanja sistema biblioteke

Škola razvija Java aplikaciju za upravljanje digitalnom bibliotekom. Potrebno je napraviti klasu koja čuva osnovna podešavanja sistema: naziv biblioteke, radno vreme i maksimalan broj knjiga koje jedan student može da pozajmi. Pošto cela aplikacija mora da koristi ista podešavanja, u sistemu sme da postoji samo jedna instanca te klase. Potrebno je omogućiti pregled i izmenu podešavanja, kao i demonstrirati u glavnom programu da svi delovi sistema koriste isti objekat.

Zahtevi:

- Primeni Singleton pattern nad klasom BibliotekaConfig.
- Dodaj attribute: nazivBiblioteke, radnoVreme i maksimalanBrojKnjiga.
- Napravi get/set metode i metodu za ispis svih podešavanja.
- U glavnom programu pokaži da dve promenljive referenciraju isti objekat.

1. BibliotekaConfig.java

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

    private String nazivBiblioteke;
    private String radnoVreme;
    private int maksimalanBrojKnjiga;

    private BibliotekaConfig() {
        nazivBiblioteke = "Gradska studentska biblioteka";
        radnoVreme = "08:00 - 20:00";
        maksimalanBrojKnjiga = 3;
    }

    public static synchronized BibliotekaConfig getInstance() {
        if (instance == null) {
            instance = new BibliotekaConfig();
        }
        return instance;
    }

    public String getNazivBiblioteke() {
        return nazivBiblioteke;
    }

    public void setNazivBiblioteke(String nazivBiblioteke) {
        this.nazivBiblioteke = nazivBiblioteke;
    }
}
```

```

public String getRadnoVreme() {
    return radnoVreme;
}

public void setRadnoVreme(String radnoVreme) {
    this.radnoVreme = radnoVreme;
}

public int getMaximalanBrojKnjiga() {
    return maksimalanBrojKnjiga;
}

public void setMaximalanBrojKnjiga(int maksimalanBrojKnjiga) {
    this.maksimalanBrojKnjiga = maksimalanBrojKnjiga;
}

public void ispisiPodesavanja() {
    System.out.println("Naziv biblioteke: " + nazivBiblioteke);
    System.out.println("Radno vreme: " + radnoVreme);
    System.out.println("Maksimalan broj knjiga: " + maksimalanBrojKnjiga);
}
}

```

2. MainBiblioteka.java

```

public class MainBiblioteka {
    public static void main(String[] args) {
        BibliotekaConfig config1 = BibliotekaConfig.getInstance();
        BibliotekaConfig config2 = BibliotekaConfig.getInstance();

        config1.setNazivBiblioteke("Univerzitetska biblioteka");
        config1.setRadnoVreme("07:30 - 21:00");
        config1.setMaximalanBrojKnjiga(5);

        System.out.println("Podešavanja preko prve reference:");
        config1.ispisiPodesavanja();

        System.out.println();
        System.out.println("Podešavanja preko druge reference:");
        config2.ispisiPodesavanja();

        System.out.println();
        System.out.println("Da li su reference iste? " + (config1 == config2));
    }
}

```

Singleton je ovde opravdan zato što aplikacija treba da koristi jedinstvena podešavanja biblioteke. Klasa ima privatni konstruktor kako se ne bi mogla napraviti nova instanca spolja, statičko polje instance čuva jedini objekat, a metoda getInstance() vraća tu istu instancu pri svakom pozivu. U glavnom programu se vidi da promene načinjene preko jedne reference odmah važe i za drugu, što potvrđuje da je u pitanju isti objekat.

Zadatak 2 – Štampa u kopirnici

Studentska kopirnica koristi jedan mrežni štampač na koji više korisnika šalje dokumenta za štampu. Potrebno je napraviti Java program u kome postoji samo jedan menadžer reda štampe. Menadžer treba da prima dokumenta u red, prikaže trenutno stanje reda i odštampa sledeći dokument. Zadatak treba da pokaže kako više delova programa koristi isti centralni red štampe.

Zahtevi:

- Primeni Singleton pattern nad klasom StampacManager.
- Napravi pomoćnu klasu Dokument sa nazivom i brojem strana.
- Omogući dodavanje dokumenta u red, prikaz reda i štampanje sledećeg dokumenta.
- U glavnom programu testiraj dodavanje više dokumenata i obradu reda.

1. Dokument.java

```
public class Dokument {
    private String naziv;
    private int brojStrana;

    public Dokument(String naziv, int brojStrana) {
        this.naziv = naziv;
        this.brojStrana = brojStrana;
    }

    public String getNaziv() {
        return naziv;
    }

    public int getBrojStrana() {
        return brojStrana;
    }

    @Override
    public String toString() {
        return naziv + " (" + brojStrana + " str.)";
    }
}
```

2. StampacManager.java

```
import java.util.LinkedList;
import java.util.Queue;

public class StampacManager {
    private static StampacManager instance;
    private Queue<Dokument> red;

    private StampacManager() {
        red = new LinkedList<>();
    }

    public static synchronized StampacManager getInstance() {
        if (instance == null) {
```

```

        instance = new StampacManager();
    }
    return instance;
}

public void dodajDokument(Dokument dokument) {
    red.add(dokument);
    System.out.println("Dodat u red: " + dokument);
}

public void stampajSledeci() {
    if (red.isEmpty()) {
        System.out.println("Red je prazan. Nema dokumenata za štampu.");
        return;
    }

    Dokument dokument = red.poll();
    System.out.println("Štampa se: " + dokument);
}

public void prikaziRed() {
    if (red.isEmpty()) {
        System.out.println("Red za štampu je prazan.");
        return;
    }

    System.out.println("Trenutni red za štampu:");
    for (Dokument dokument : red) {
        System.out.println("- " + dokument);
    }
}

public int getBrojDokumenata() {
    return red.size();
}
}

```

3. MainStampac.java

```

public class MainStampac {
    public static void main(String[] args) {
        StampacManager manager1 = StampacManager.getInstance();
        StampacManager manager2 = StampacManager.getInstance();

        manager1.dodajDokument(new Dokument("Seminarski rad", 12));
        manager1.dodajDokument(new Dokument("Prijava ispita", 2));
        manager2.dodajDokument(new Dokument("Ugovor o praksi", 4));

        System.out.println();
        manager1.prikaziRed();

        System.out.println();
        manager2.stampajSledeci();
        manager1.stampajSledeci();
    }
}

```

```
System.out.println();
manager1.prikaziRed();

System.out.println();
System.out.println("Da li su manager1 i manager2 isti objekat? " + (manager1 == manager2));
}
}
```

U ovom primeru Singleton je logičan jer u sistemu postoji samo jedan centralni red štampe. Ako bi postojalo više instanci klase StampacManager, svaki deo aplikacije bi mogao da vodi svoj poseban red, što bi dovelo do zabune i neusklađenosti. Zbog toga privatni konstruktor sprečava direktno pravljenje objekata, a statička metoda getInstance() obezbeđuje jedinstven pristup istoj instanci. Klasa Dokument predstavlja podatke o jednom poslu za štampu, dok StampacManager upravlja redom dokumenata.

Zadatak 3 – Logger u online prodavnici

Online prodavnica garderobe želi da vodi centralnu evidenciju važnih događaja u sistemu. Na primer, potrebno je zabeležiti kreiranje porudžbine, uspešnu uplatu i eventualne greške. Pošto svi moduli sistema treba da upisuju poruke na jedno isto mesto, potrebno je napraviti logger kao Singleton. Zadatak treba da pokaže da različite servisne klase koriste isti logger i da se svi zapisi čuvaju centralizovano.

Zahtevi:

- Primeni Singleton pattern nad klasom CentralniLogger.
- Logger treba da čuva sve poruke u listi i da omogući prikaz celog loga.
- Napravi dve servisne klase, na primer PorudzbinaServis i PlacanjeServis, koje koriste isti logger.
- U glavnom programu simuliraj nekoliko aktivnosti i prikaži kompletan log.

Rešenje

1. CentralniLogger.java

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;

public class CentralniLogger {
    private static CentralniLogger instance;
    private List<String> zapisi;

    private CentralniLogger() {
        zapisi = new ArrayList<>();
    }

    public static synchronized CentralniLogger getInstance() {
        if (instance == null) {
            instance = new CentralniLogger();
        }
        return instance;
    }

    public void evidentiraj(String poruka) {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd.MM.yyyy HH:mm:ss");
        String vreme = LocalDateTime.now().format(formatter);
        zapisi.add(vreme + " - " + poruka);
    }

    public void prikaziSveZapise() {
        if (zapisi.isEmpty()) {
            System.out.println("Nema evidentiranih poruka.");
            return;
        }

        System.out.println("LOG SISTEMA:");
        for (String zapis : zapisi) {
            System.out.println(zapis);
        }
    }
}
```

```

    }
}

public void obrisiLog() {
    zapisi.clear();
}
}

```

2. PorudzbinaServis.java

```

public class PorudzbinaServis {
    private CentralniLogger logger = CentralniLogger.getInstance();

    public void kreirajPorudzbinu(int idPorudzbine, String kupac) {
        logger.evidentiraj("Kreirana porudžbina #" + idPorudzbine + " za kupca: " + kupac);
    }

    public void otkaziPorudzbinu(int idPorudzbine) {
        logger.evidentiraj("Otkazana porudžbina #" + idPorudzbine);
    }
}

```

3. PlacanjeServis.java

```

public class PlacanjeServis {
    private CentralniLogger logger = CentralniLogger.getInstance();

    public void izvrsiPlacanje(int idPorudzbine, double iznos) {
        logger.evidentiraj("Uspešno plaćanje za porudžbinu #" + idPorudzbine +
            ", iznos: " + iznos + " RSD");
    }

    public void prijaviGresku(int idPorudzbine) {
        logger.evidentiraj("Greška pri plaćanju za porudžbinu #" + idPorudzbine);
    }
}

```

4. MainLogger.java

```

public class MainLogger {
    public static void main(String[] args) {
        PorudzbinaServis porudzbinaServis = new PorudzbinaServis();
        PlacanjeServis placanjeServis = new PlacanjeServis();

        porudzbinaServis.kreirajPorudzbinu(101, "Ana Petrović");
        placanjeServis.isvrsiPlacanje(101, 4599.99);
        porudzbinaServis.kreirajPorudzbinu(102, "Marko Jovanović");
        placanjeServis.prijaviGresku(102);
        porudzbinaServis.otkaziPorudzbinu(102);

        System.out.println();

        CentralniLogger logger1 = CentralniLogger.getInstance();
        CentralniLogger logger2 = CentralniLogger.getInstance();
    }
}

```

```
logger1.prikaziSveZapise();
System.out.println();
System.out.println("Da li logger1 i logger2 predstavljaju isti objekat? " +
(logger1 == logger2));
}
}
```

Centralni logger je tipičan primer za Singleton zato što svi delovi sistema treba da šalju poruke na jedno zajedničko mesto. Na taj način administratori ili programeri mogu da prate tok rada aplikacije bez rasutih logova po različitim objektima. Servisne klase PorudzbinaServis i PlacanjeServis ne kreiraju sopstvene logere, već preuzimaju istu instancu preko metode getInstance(). Time se dobija jedinstvena, centralizovana evidencija svih događaja u aplikaciji.

Factory Pattern i Abstract Factory Pattern

Kratko objašnjenje obrazaca

Factory Pattern se koristi kada klijent ne treba direktno da zna koju konkretnu klasu treba da instancira. Umesto toga, klijent traži objekat od fabrike, a fabrika odlučuje koju implementaciju treba napraviti.

U prva tri zadatka fabrika kreira po jedan konkretan proizvod iz grupe sličnih proizvoda: obaveštenje, opciju dostave ili kartu za prevoz.

Abstract Factory Pattern se koristi kada sistem treba da kreira porodicu međusobno povezanih objekata. Klijent ne bira pojedinačne klase, već bira celu fabriku, a fabrika zatim proizvodi kompatibilne objekte iste porodice.

U poslednja dva zadatka jedna fabrika kreira više povezanih objekata: komplet obroka ili skup UI elemenata za istu platformu.

Razlika između Factory i Abstract Factory obrasca

Obrazac	Šta kreira?	Primer iz dokumenta
Factory Pattern	Jedan objekat iz grupe sličnih objekata	Jedno obaveštenje: email, SMS ili push
Abstract Factory Pattern	Porodicu povezanih objekata	Komplet obroka: piće, glavno jelo i desert

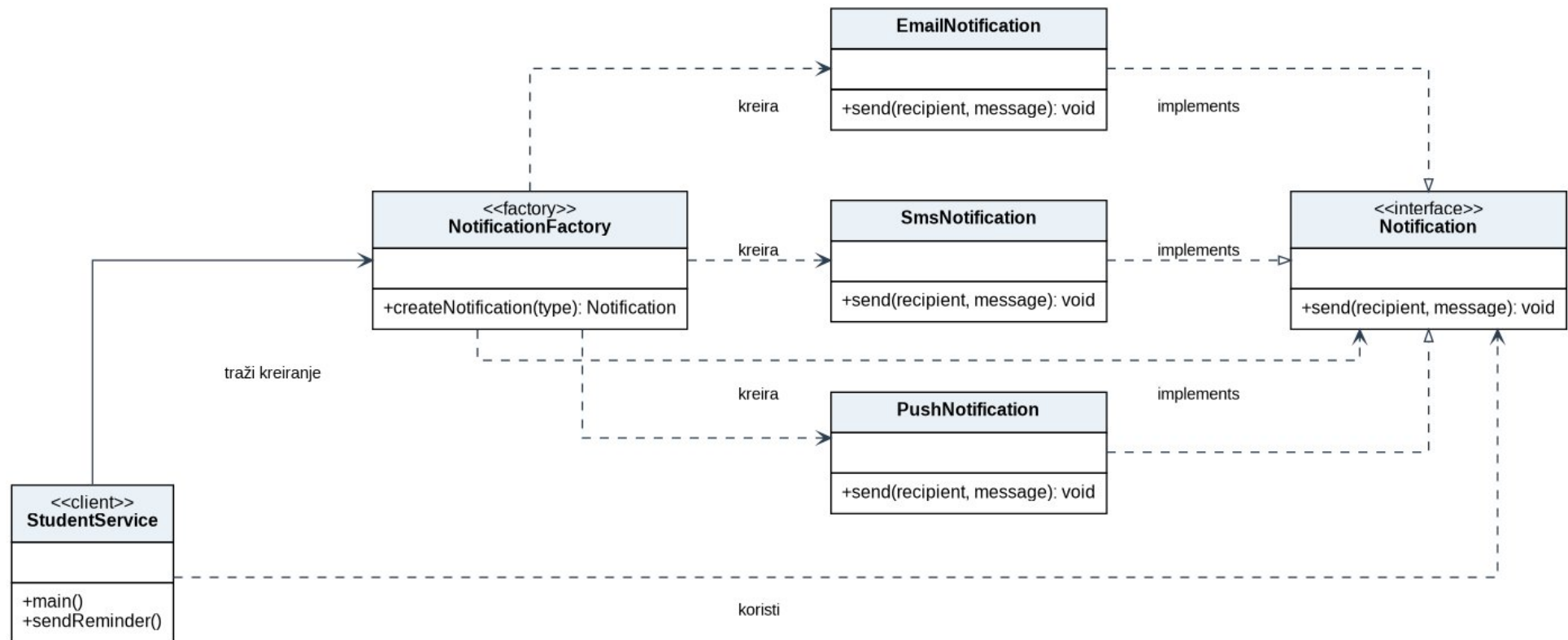
ZADATAK 1 – Factory Pattern: Sistem za slanje obaveštenja studentima

Studentska služba želi jednostavan sistem koji može da šalje različite vrste obaveštenja studentima: e-mail, SMS i push obaveštenje kroz mobilnu aplikaciju. U budućnosti se može pojaviti i novi kanal obaveštavanja, pa nije dobro da se svuda u programu pišu uslovi i direktno prave objekti konkretnih klasa.

Potrebno je napraviti rešenje u Javi u kome postoji zajednički interfejs Notification. Konkretno klase EmailNotification, SmsNotification i PushNotification treba da implementiraju taj interfejs. Posebna klasa NotificationFactory treba da na osnovu prosleđenog tipa obaveštenja vrati odgovarajući objekat. Klijent, odnosno glavna klasa programa, treba da koristi samo fabriku i interfejs, a ne konkretne klase.

Program treba da demonstrira slanje bar dva različita obaveštenja. Rešenje treba da pokaže zašto je Factory Pattern koristan kada objekat koji se kreira zavisi od izbora korisnika ili poslovnog pravila.

UML class diagram



Slika 1. UML class dijagram za Factory Pattern u sistemu obaveštenja

Rešenje koristi interfejs Notification, koji definiše zajedničku operaciju send(). Sve konkretne klase imaju isti javni način korišćenja, ali se ponašaju drugačije.

NotificationFactory centralizuje odluku o tome koja se klasa instancira. Ako se kasnije doda, na primer, ViberNotification ili WhatsAppNotification, promena će se uglavnom odnositi na novu klasu i fabriku, dok ostatak aplikacije može ostati isti.

Klijent dobija objekat tipa Notification i poziva send(), bez potrebe da zna da li iza toga stoji email, SMS ili push mehanizam.

Java kod

```
interface Notification {
    void send(String recipient, String message);
}

class EmailNotification implements Notification {
    @Override
    public void send(String recipient, String message) {
        System.out.println("EMAIL za " + recipient + ": " + message);
    }
}

class SmsNotification implements Notification {
    @Override
    public void send(String recipient, String message) {
        System.out.println("SMS za " + recipient + ": " + message);
    }
}

class PushNotification implements Notification {
    @Override
    public void send(String recipient, String message) {
        System.out.println("PUSH poruka za " + recipient + ": " + message);
    }
}

class NotificationFactory {
    public static Notification createNotification(String type) {
        if (type == null) {
            throw new IllegalArgumentException("Tip obaveštenja nije unet.");
        }

        switch (type.toUpperCase()) {
            case "EMAIL":
                return new EmailNotification();
            case "SMS":
                return new SmsNotification();
            case "PUSH":
```

```
        return new PushNotification();
    default:
        throw new IllegalArgumentException("Nepoznat tip obaveštenja: " + type);
    }
}

public class FactoryTask1Demo {
    public static void main(String[] args) {
        Notification notification = NotificationFactory.createNotification("EMAIL");
        notification.send("student@example.com", "Vaš termin konsultacija je potvrđen.");

        Notification urgentNotification = NotificationFactory.createNotification("SMS");
        urgentNotification.send("+38164111222", "Podsetnik: konsultacije počinju za 30 minuta.");
    }
}
```

Kratko objašnjenje koda

- Interfejs definiše zajedničko ponašanje svih konkretnih proizvoda.
- Konkretnе klase implementiraju interfejs i sadrže specifičnu logiku.
- Factory klasa sadrži metodu koja na osnovu tipa vraća odgovarajući objekat.
- Klijent koristi interfejs i fabriku, pa je manje zavisn od konkretnih klasa.

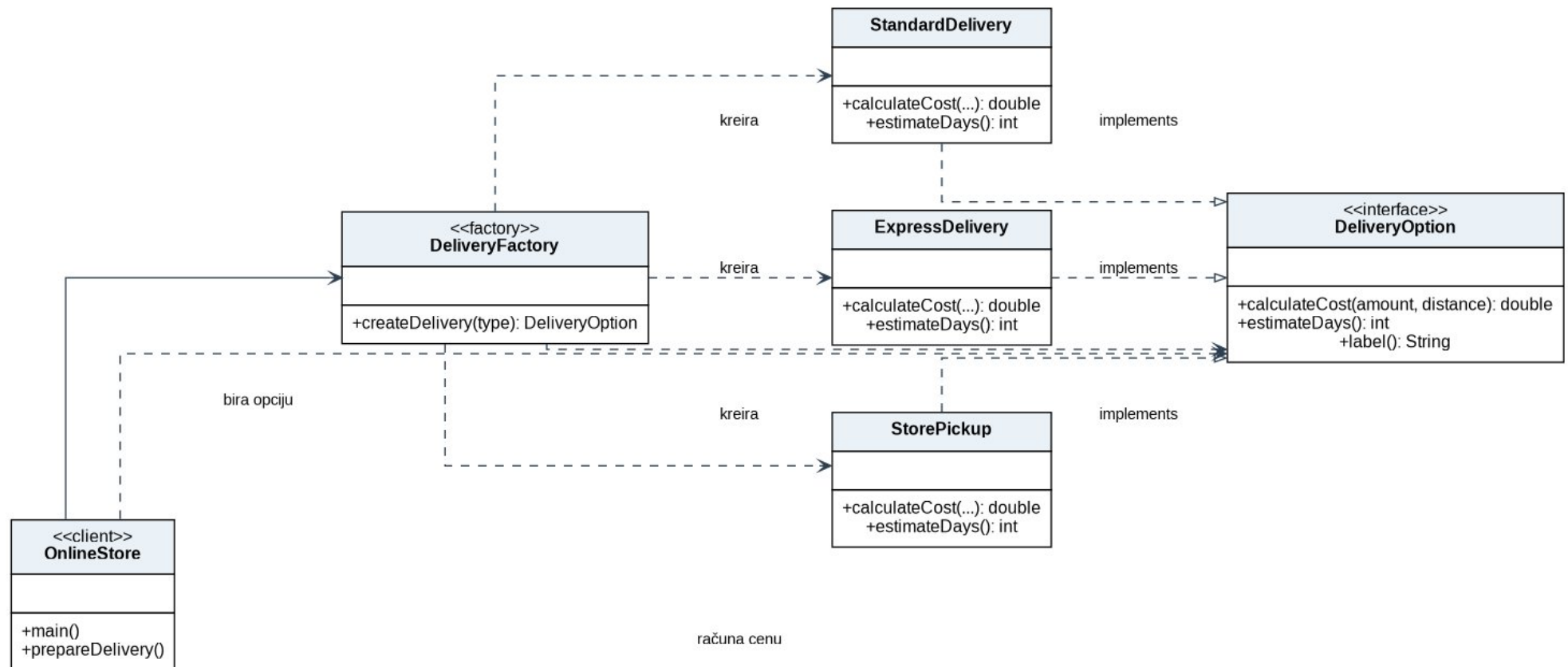
ZADATAK 2 – Factory Pattern: Sistem za izbor načina dostave u online prodavnici

Online prodavnica garderobe kupcima nudi više načina isporuke: standardnu dostavu, brzu dostavu i lično preuzimanje u prodavnici. Svaka opcija ima drugačiji način obračuna cene i različito očekivano vreme isporuke.

Potrebno je napraviti Java program u kome postoji interfejs `DeliveryOption` sa metodama za izračunavanje cene dostave, procenu broja dana isporuke i prikaz naziva opcije. Konkretno klase `StandardDelivery`, `ExpressDelivery` i `StorePickup` treba da implementiraju ovaj interfejs.

Klasa `DeliveryFactory` treba da na osnovu teksta koji korisnik izabere, na primer `STANDARD`, `EXPRESS` ili `PICKUP`, vrati odgovarajući objekat. Glavni program treba da prikaže naziv izabrane opcije, cenu dostave i očekivani broj dana do isporuke.

UML class diagram



Slika 2. UML class dijagram za Factory Pattern u izboru dostave
Materijal za vežbu: tekstualni zadaci, UML class dijagrami, Java rešenja i objašnjenja

U ovom zadatku sve opcije dostave imaju iste osnovne operacije, ali različitu poslovnu logiku. Zbog toga je dobro da postoji zajednički interfejs `DeliveryOption`.

`DeliveryFactory` skriva detalje kreiranja konkretne klase. Glavni program ne pravi direktno `new ExpressDelivery()`, već traži od fabrike da mu vrati odgovarajuću opciju dostave.

Prednost ovog pristupa je lakše proširenje sistema. Na primer, dodavanje međunarodne dostave može da se uradi dodavanjem klase `InternationalDelivery` i dopunom fabrike.

Java kod

```
interface DeliveryOption {
    double calculateCost(double orderAmount, double distanceKm);
    int estimateDays();
    String label();
}

class StandardDelivery implements DeliveryOption {
    @Override
    public double calculateCost(double orderAmount, double distanceKm) {
        return 350 + distanceKm * 20;
    }

    @Override
    public int estimateDays() {
        return 3;
    }

    @Override
    public String label() {
        return "Standardna dostava";
    }
}

class ExpressDelivery implements DeliveryOption {
    @Override
    public double calculateCost(double orderAmount, double distanceKm) {
        return 700 + distanceKm * 35;
    }

    @Override
    public int estimateDays() {
        return 1;
    }

    @Override
    public String label() {
        return "Brza dostava";
    }
}
```

```

}

class StorePickup implements DeliveryOption {
    @Override
    public double calculateCost(double orderAmount, double distanceKm) {
        return 0;
    }

    @Override
    public int estimateDays() {
        return 0;
    }

    @Override
    public String label() {
        return "Lično preuzimanje u prodavnici";
    }
}

class DeliveryFactory {
    public static DeliveryOption createDelivery(String type) {
        if (type == null) {
            throw new IllegalArgumentException("Tip dostave nije unet.");
        }

        switch (type.toUpperCase()) {
            case "STANDARD":
                return new StandardDelivery();
            case "EXPRESS":
                return new ExpressDelivery();
            case "PICKUP":
                return new StorePickup();
            default:
                throw new IllegalArgumentException("Nepoznata opcija dostave: " + type);
        }
    }
}

public class FactoryTask2Demo {
    public static void main(String[] args) {
        double orderAmount = 5400;
        double distanceKm = 8.5;

        DeliveryOption delivery = DeliveryFactory.createDelivery("EXPRESS");

        System.out.println("Izabrana opcija: " + delivery.label());
    }
}

```

```
System.out.println("Cena dostave: " + delivery.calculateCost(orderAmount, distanceKm) + " RSD");
System.out.println("Procena isporuke: " + delivery.estimateDays() + " dan(a)");
}
}
```

Kratko objašnjenje koda

- Interfejs definiše zajedničko ponašanje svih konkretnih proizvoda.
- Konkretne klase implementiraju interfejs i sadrže specifičnu logiku.
- Factory klasa sadrži metodu koja na osnovu tipa vraća odgovarajući objekat.
- Klijent koristi interfejs i fabriku, pa je manje zavisen od konkretnih klasa.

ZADATAK 3 – Factory Pattern: Automat za izdavanje karata za gradski prevoz

Tekst zadatka

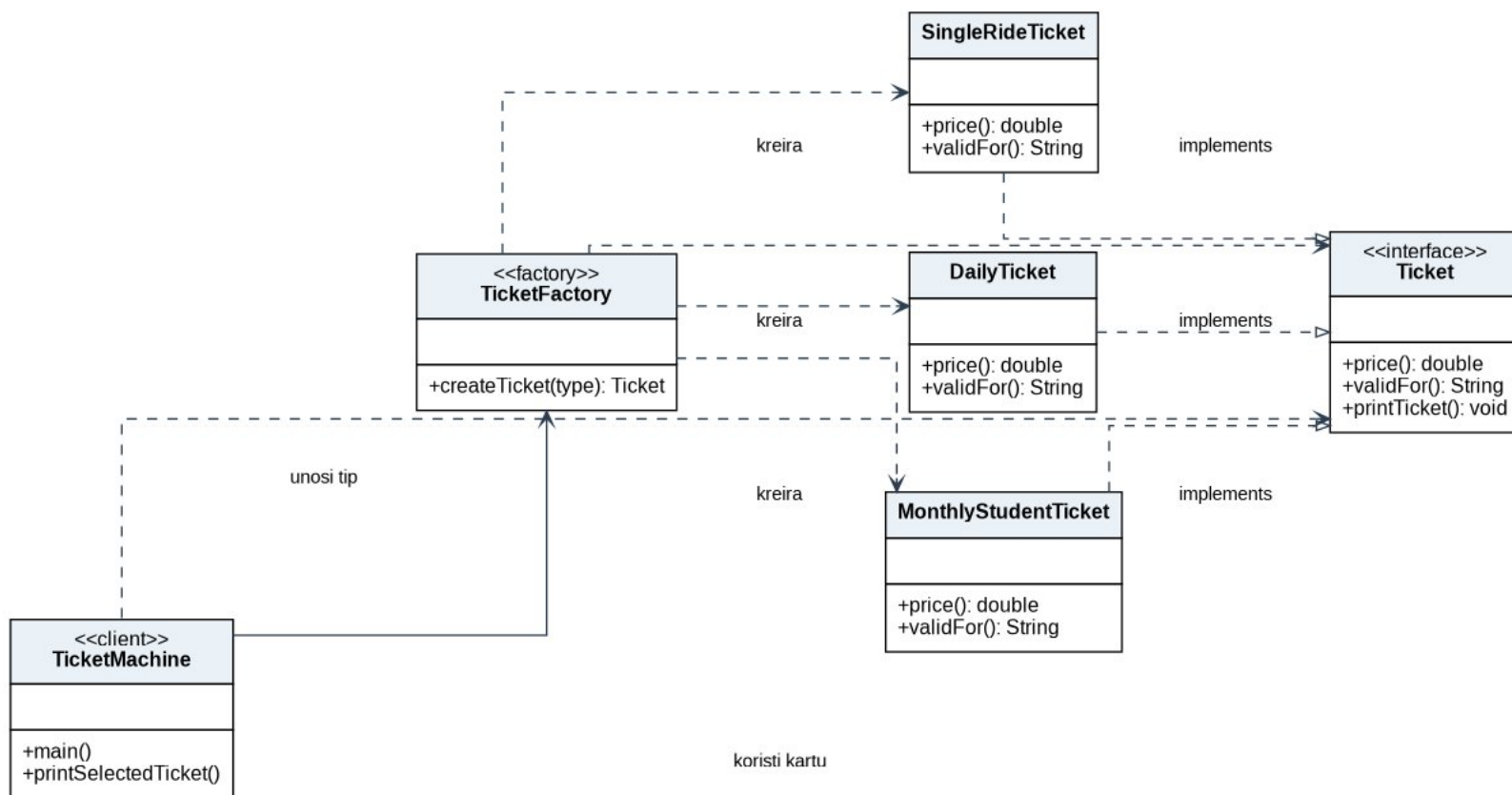
Gradsko preduzeće uvodi automat za prodaju karata za javni prevoz. Korisnik može izabrati kartu za jednu vožnju, dnevnu kartu ili mesečnu studentsku kartu. Svaka karta ima svoju cenu, period važenja i način ispisa informacija.

Potrebno je napraviti Java program sa interfejsom Ticket. Interfejs treba da sadrži metode price(), validFor() i printTicket(). Konkretne klase SingleRideTicket, DailyTicket i MonthlyStudentTicket treba da implementiraju taj interfejs.

Klasa TicketFactory treba da kreira odgovarajuću kartu na osnovu unetog tipa. Glavni program treba da zatraži jednu kartu od fabrike i prikaže informacije o toj karti.

UML class diagram

Java Design Patterns: Factory Pattern i Abstract Factory Pattern



Slika 3. UML class diagram za Factory Pattern u automatu za karte

Sve vrste karata imaju zajednički ugovor: mogu da vrate cenu, period važenja i da se ispišu na ekranu. Zato se koristi interfejs Ticket.

TicketFactory odvaja logiku izbora tipa karte od ostatka programa. Automat za prodaju karata ne mora da zna kako se pravi svaka konkretna karta, već koristi fabriku.

Ovo rešenje je pogodno za sisteme u kojima postoji više sličnih proizvoda, a izbor zavisi od korisničkog unosa ili nekog poslovnog pravila.

Java kod

```
interface Ticket {
    double price();
    String validFor();
    void printTicket();
}

class SingleRideTicket implements Ticket {

    @Override
    public double price() {
        return 90;
    }

    @Override
    public String validFor() {
        return "Jedna vožnja u trajanju do 90 minuta";
    }

    @Override
    public void printTicket() {
        System.out.println("Karta za jednu vožnju - " + price() + " RSD");
        System.out.println("Važi za: " + validFor());
    }
}

class DailyTicket implements Ticket {

    @Override
    public double price() {
        return 350;
    }

    @Override
    public String validFor() {
        return "Neograničen broj vožnji u toku jednog dana";
    }

    @Override
```

```

    public void printTicket() {
        System.out.println("Dnevna karta - " + price() + " RSD");
        System.out.println("Važi za: " + validFor());
    }
}

class MonthlyStudentTicket implements Ticket {

    @Override
    public double price() {
        return 1200;
    }

    @Override
    public String validFor() {
        return "Mesečna studentska karta";
    }

    @Override
    public void printTicket() {
        System.out.println("Mesečna studentska karta - " + price() + " RSD");
        System.out.println("Važi za: " + validFor());
    }
}

class TicketFactory {
    public static Ticket createTicket(String type) {
        if (type == null) {
            throw new IllegalArgumentException("Tip karte nije unet.");
        }

        switch (type.toUpperCase()) {
            case "SINGLE":
                return new SingleRideTicket();
            case "DAILY":
                return new DailyTicket();
            case "STUDENT_MONTHLY":
                return new MonthlyStudentTicket();
            default:
                throw new IllegalArgumentException("Nepoznat tip karte: " + type);
        }
    }
}

public class FactoryTask3Demo {

```

```
public static void main(String[] args) {  
    Ticket ticket = TicketFactory.createTicket("STUDENT_MONTHLY");  
    ticket.printTicket();  
}  
}
```

Kratko objašnjenje koda

- Interfejs definiše zajedničko ponašanje svih konkretnih proizvoda.
- Konkretnе klase implementiraju interfejs i sadrže specifičnu logiku.
- Factory klasa sadrži metodu koja na osnovu tipa vraća odgovarajući objekat.
- Klijent koristi interfejs i fabriku, pa je manje zavisn od konkretnih klasa.

ZADATAK 4 – Abstract Factory Pattern: Komplet obroka u studentskoj kafeteriji

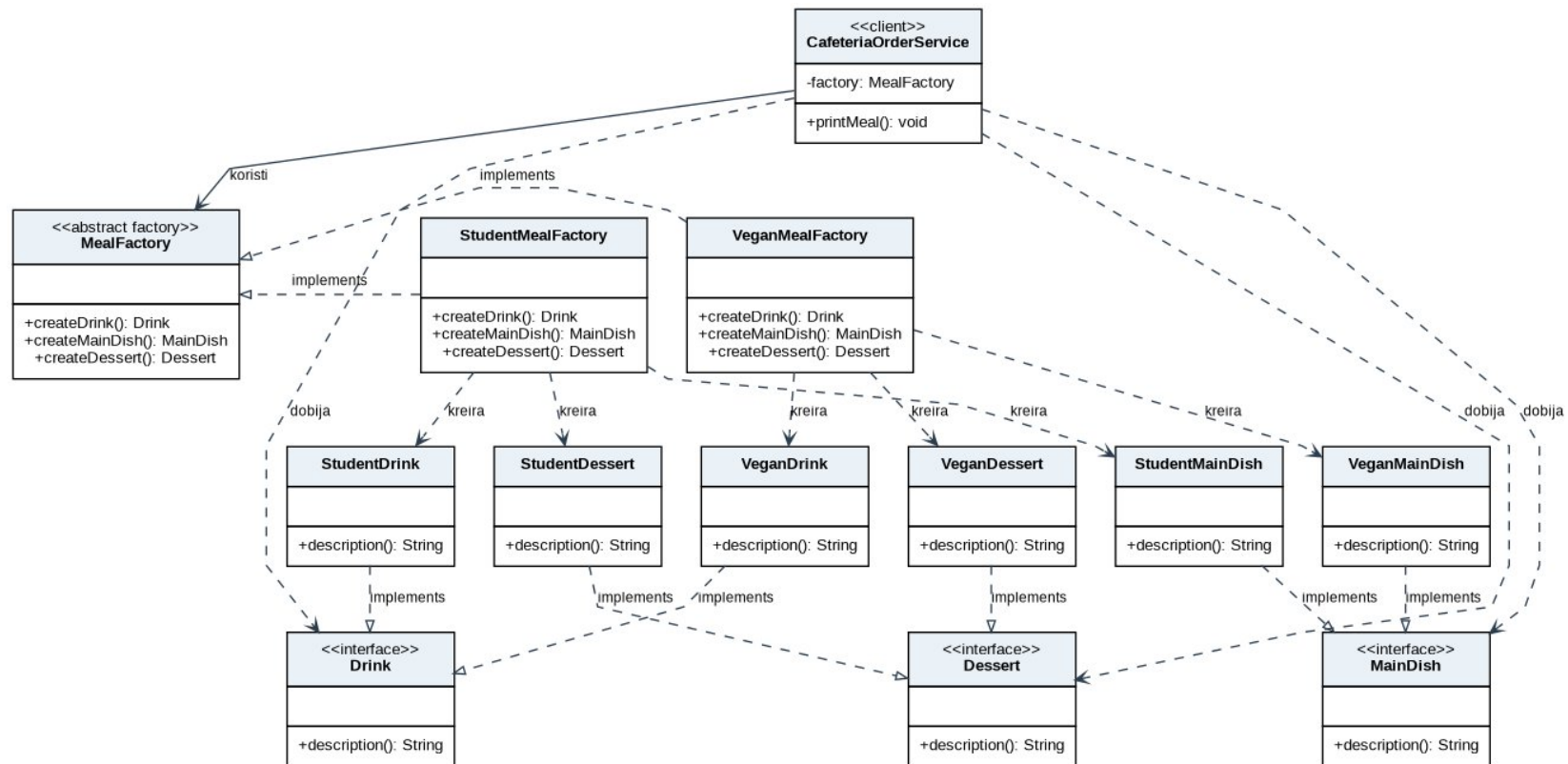
Tekst zadatka

Studentska kafeterija želi aplikaciju za naručivanje gotovih kompleta obroka. Jedan komplet se uvek sastoji od tri povezane stavke: pića, glavnog jela i deserta. Postoje različite porodice obroka, na primer studentski komplet i veganski komplet.

Potrebno je primeniti Abstract Factory Pattern. Treba definisati interfejse Drink, MainDish i Dessert kao apstraktne proizvode. Zatim treba napraviti konkretne proizvode za studentski komplet i konkretne proizvode za veganski komplet.

Interfejs MealFactory treba da ima metode createDrink(), createMainDish() i createDessert(). Konkretne fabrike StudentMealFactory i VeganMealFactory treba da kreiraju odgovarajuću porodicu proizvoda. Klasa CafeteriaOrderService treba da koristi samo apstraktnu fabriku i apstraktne proizvode, bez direktnog pozivanja konkretnih klasa.

UML class diagram



Slika 4. UML class dijagram za Abstract Factory Pattern u kafeteriji

Ovo nije samo izbor jednog objekta, već izbor cele porodice povezanih objekata. Ako korisnik izabere veganski komplet, piće, glavno jelo i desert treba da pripadaju istoj porodici. Zato se koristi Abstract Factory Pattern. MealFactory definiše koje proizvode svaka fabrika mora da kreira, dok StudentMealFactory i VeganMealFactory obezbeđuju konkretne kombinacije.

Klijent, odnosno CafeteriaOrderService, ne zna da li radi sa studentskim ili veganskim kompletom. On samo poziva metode fabrike i dobija kompatibilne proizvode.

Java kod

```

interface Drink {
    String description();
}
  
```

```
interface MainDish {
    String description();
}

interface Dessert {
    String description();
}

class StudentDrink implements Drink {

    @Override
    public String description() {
        return "Limunada";
    }
}

class StudentMainDish implements MainDish {

    @Override
    public String description() {
        return "Sendvič sa piletinom";
    }
}

class StudentDessert implements Dessert {

    @Override
    public String description() {
        return "Muffin";
    }
}

class VeganDrink implements Drink {

    @Override
    public String description() {
        return "Ceđeni sok";
    }
}

class VeganMainDish implements MainDish {
    @Override
    public String description() {
        return "Salata sa leblebijama";
    }
}
```

```
class VeganDessert implements Dessert {
    @Override
    public String description() {
        return "Voćna salata";
    }
}

interface MealFactory {
    Drink createDrink();
    MainDish createMainDish();
    Dessert createDessert();
}

class StudentMealFactory implements MealFactory {
    @Override
    public Drink createDrink() {
        return new StudentDrink();
    }

    @Override
    public MainDish createMainDish() {
        return new StudentMainDish();
    }

    @Override
    public Dessert createDessert() {
        return new StudentDessert();
    }
}

class VeganMealFactory implements MealFactory {
    @Override
    public Drink createDrink() {
        return new VeganDrink();
    }

    @Override
    public MainDish createMainDish() {
        return new VeganMainDish();
    }

    @Override
    public Dessert createDessert() {
        return new VeganDessert();
    }
}
```

```
class CafeteriaOrderService {
    private final MealFactory factory;

    public CafeteriaOrderService(MealFactory factory) {
        this.factory = factory;
    }

    public void printMeal() {
        Drink drink = factory.createDrink();
        MainDish mainDish = factory.createMainDish();
        Dessert dessert = factory.createDessert();

        System.out.println("Piće: " + drink.description());
        System.out.println("Glavno jelo: " + mainDish.description());
        System.out.println("Desert: " + dessert.description());
    }
}

public class AbstractFactoryTask1Demo {
    public static void main(String[] args) {
        MealFactory factory = new VeganMealFactory();
        CafeteriaOrderService service = new CafeteriaOrderService(factory);
        service.printMeal();
    }
}
```

Kratko objašnjenje koda

- Više interfejsa predstavlja više apstraktnih proizvoda koji pripadaju istoj porodici.
- Apstraktna fabrika definiše metode za kreiranje svih povezanih proizvoda.
- Konkretna fabrike proizvode kompatibilne objekte iz iste porodice.
- Klijent radi sa apstraktnom fabrikom i apstraktnim proizvodima, pa može lako promeniti celu porodicu objekata.

ZADATAK 5 – Abstract Factory Pattern: UI elementi za web i mobilnu online prodavnicu

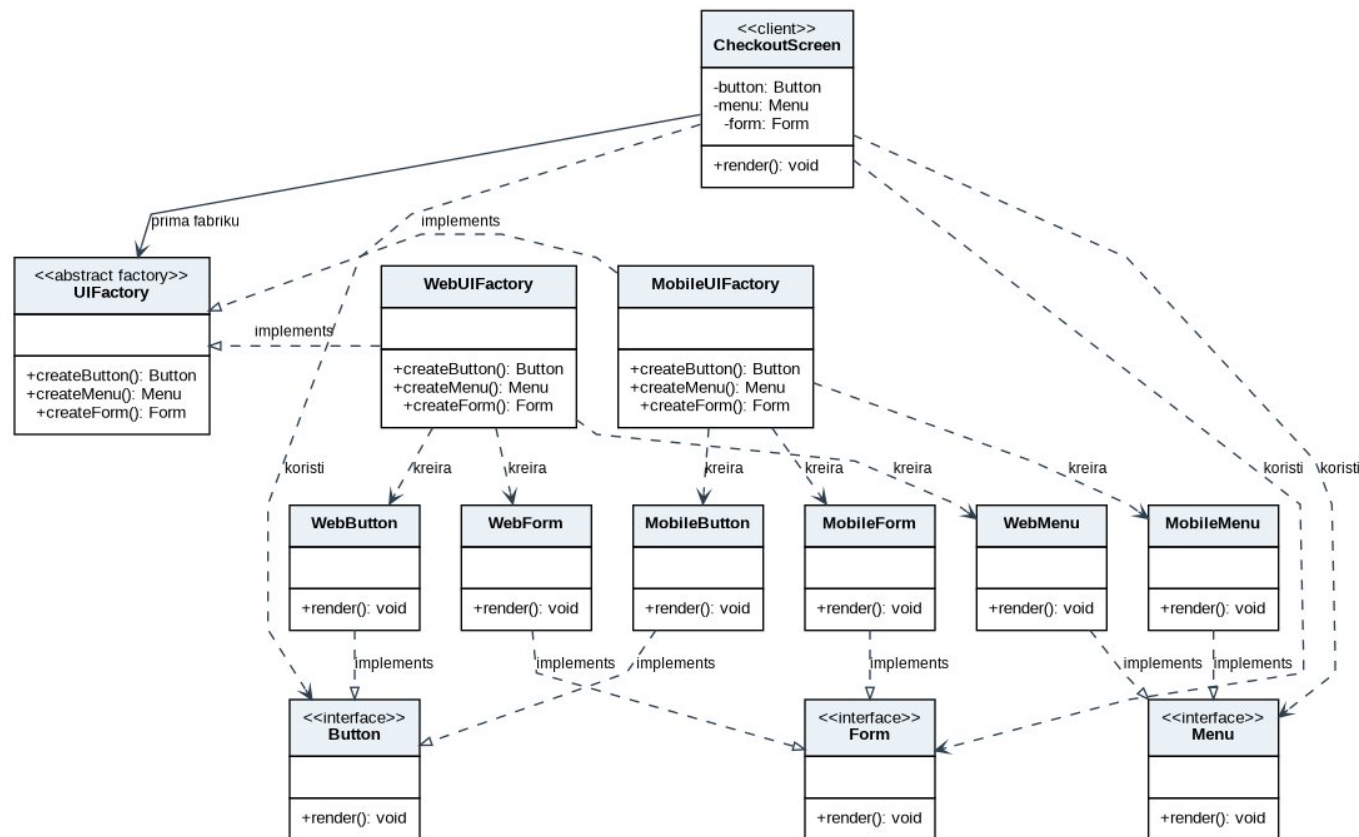
Tekst zadatka

Online prodavnica ima istu funkcionalnost na web sajtu i u mobilnoj aplikaciji, ali se elementi korisničkog interfejsa razlikuju. Na web verziji forma može imati više kolona, meni može biti horizontalan, a dugme šire. Na mobilnoj verziji forma treba da bude jednostavnija, meni prilagođen ikonama, a dugme veće za ekran osjetljiv na dodir.

Potrebno je primeniti Abstract Factory Pattern. Treba definisati apstraktne proizvode Button, Menu i Form. Zatim treba napraviti konkretne web proizvode WebButton, WebMenu i WebForm, kao i mobilne proizvode MobileButton, MobileMenu i MobileForm.

Interfejs UIFactory treba da kreira sva tri elementa. Konkretne fabrike WebUIFactory i MobileUIFactory treba da kreiraju kompatibilne elemente za svoju platformu. Klasa CheckoutScreen treba da koristi samo UIFactory i apstraktne proizvode, kako bi isti ekran mogao da se napravi za web ili mobilnu platformu bez izmene logike ekrana.

UML class diagram



U ovom primeru sistem ne bira jedan proizvod, već komplet elemenata korisničkog interfejsa. Dugme, meni i forma moraju biti usklađeni sa istom platformom.

Abstract Factory Pattern omogućava da se izabere cela porodica proizvoda: web porodica ili mobilna porodica. CheckoutScreen ne zavisi od konkretnih klasa, već od interfejsa.

Ako se kasnije doda desktop aplikacija, može se dodati DesktopUIFactory i konkretni DesktopButton, DesktopMenu i DesktopForm, bez menjanja glavne logike ekrana.

Java kod

```
interface Button {
    void render();
}

interface Menu {
    void render();
}

interface Form {
    void render();
}

class WebButton implements Button {

    @Override
    public void render() {
        System.out.println("Prikazujem široko web dugme za plaćanje.");
    }
}

class WebMenu implements Menu {

    @Override
    public void render() {
        System.out.println("Prikazujem horizontalni web meni.");
    }
}

class WebForm implements Form {

    @Override
    public void render() {
        System.out.println("Prikazujem web formu sa više kolona.");
    }
}
```

```
class MobileButton implements Button {
    @Override
    public void render() {
        System.out.println("Prikazujem veliko mobilno dugme za ekran osetljiv na dodir.");
    }
}

class MobileMenu implements Menu {
    @Override
    public void render() {
        System.out.println("Prikazujem mobilni meni sa ikonama.");
    }
}

class MobileForm implements Form {
    @Override
    public void render() {
        System.out.println("Prikazujem mobilnu formu sa jednim poljem po redu.");
    }
}

interface UIFactory {
    Button createButton();
    Menu createMenu();
    Form createForm();
}

class WebUIFactory implements UIFactory {
    @Override
    public Button createButton() {
        return new WebButton();
    }

    @Override
    public Menu createMenu() {
        return new WebMenu();
    }

    @Override
    public Form createForm() {
        return new WebForm();
    }
}

class MobileUIFactory implements UIFactory {
    @Override
    public Button createButton() {
```

```

        return new MobileButton();
    }

    @Override
    public Menu createMenu() {
        return new MobileMenu();
    }

    @Override
    public Form createForm() {
        return new MobileForm();
    }
}

class CheckoutScreen {
    private final Button button;
    private final Menu menu;
    private final Form form;

    public CheckoutScreen(UIFactory factory) {
        this.button = factory.createButton();
        this.menu = factory.createMenu();
        this.form = factory.createForm();
    }

    public void render() {
        menu.render();
        form.render();
        button.render();
    }
}

public class AbstractFactoryTask2Demo {
    public static void main(String[] args) {
        UIFactory factory = new MobileUIFactory();
        CheckoutScreen checkoutScreen = new CheckoutScreen(factory);
        checkoutScreen.render();
    }
}

```

Kratko objašnjenje koda

- Više interfejsa predstavlja više apstraktnih proizvoda koji pripadaju istoj porodici.
- Apstraktna fabrika definiše metode za kreiranje svih povezanih proizvoda.
- Konkretna fabrike proizvode kompatibilne objekte iz iste porodice.

- Klijent radi sa apstraktnom fabrikom i apstraktnim proizvodima, pa može lako promeniti celu porodicu objekata.

Zaključak

Factory Pattern je pogodan kada treba centralizovati kreiranje jednog objekta iz grupe sličnih objekata. U ovim primerima to su obaveštenje, dostava i karta za prevoz.

Abstract Factory Pattern je pogodan kada treba kreirati više povezanih objekata koji moraju pripadati istoj porodici. U ovim primerima to su komplet obroka i skup UI elemenata za određenu platformu.

U oba slučaja cilj je smanjenje zavisnosti između klijentskog koda i konkretnih klasa. Program postaje pregledniji, lakši za proširenje i jednostavniji za održavanje.

Zaključno, dok Factory Pattern rešava problem kreiranja jednog specifičnog proizvoda, Abstract Factory Pattern garantuje da će različiti delovi sistema uvek raditi harmonično kao jedna celina.

Builder & Prototype Design Patterns

Builder Pattern

Builder Pattern se koristi kada je potrebno napraviti složen objekat koji ima veliki broj obaveznih i opcionalnih delova. Umesto da se u konstruktor prosleđuje mnogo parametara, proces kreiranja objekta se deli na jasne korake. Na taj način kod postaje čitljiviji, smanjuje se mogućnost greške i lakše se prave različite varijante istog tipa objekta.

Klasična struktura Builder obrasca obično obuhvata Product, Builder interfejs, ConcreteBuilder klase, Director klasu i Client kod. Product je složen objekat koji se gradi. Builder definiše korake izgradnje. ConcreteBuilder izvršava konkretne korake. Director zna redosled koraka, ali ne mora da zna detalje implementacije. Client bira konkretnog buildera i pokreće proces.

Element	Opis	Šta student treba da prepozna
Product	Objekat koji se postepeno kreira.	Prepoznati klase kao što su plan putovanja, porudžbina ili program treninga.
Builder	Interfejs ili statička unutrašnja klasa koja definiše korake gradnje.	Razdvojiti proces kreiranja od same klase proizvoda.
ConcreteBuilder	Konkretna implementacija izgradnje objekta.	Omogućiti više varijanti istog proizvoda.
Director	Klasa koja određuje redosled koraka.	Koristi se kada postoji unapred poznat postupak kreiranja.

U svakodnevnom životu ovaj obrazac se može prepoznati kod kreiranja personalizovane porudžbine hrane, konfiguracije laptopa ili automobila, sastavljanja plana putovanja, izrade korisničkog profila ili definisanja programa treninga, jer se u svim tim primerima složen rezultat dobija postepenim izborom više opcija.

Prototype Pattern

Prototype Pattern se koristi kada želimo da novi objekat napravimo kopiranjem već postojećeg objekta, umesto da ga kreiramo od početka. Ovaj obrazac je koristan kada su objekti složeni, kada imaju mnogo podešavanja ili kada postoji više šablona koji se često ponavljaju.

Važno je razlikovati plitko i duboko kopiranje. Kod plitkog kopiranja novi objekat može deliti reference ka unutrašnjim objektima sa originalom. Kod dubokog kopiranja kopiraju se i unutrašnji objekti, pa izmene na kopiji ne utiču na original. U primerima u ovom dokumentu koristi se duboko kopiranje gde god objekat sadrži liste ili ugnježdene objekte.

Element	Opis	Šta student treba da prepozna
Prototype interfejs	Definiše metodu za kopiranje objekta.	Najčešće metoda copy(), cloneObject() ili clonePost().
ConcretePrototype	Konkretna klasa koja ume da napravi sopstvenu kopiju.	U kodu mora da se vidi kako se kopiraju polja.
Registry	Opciona klasa koja čuva prototipove po ključu.	Korisnik traži kopiju šablona umesto da zna detalje kreiranja.
Deep copy	Kopiraju se i unutrašnji objekti, ne samo glavna referenca.	Posebno važno za liste, klauzule, stilove i druge povezane objekte.

U svakodnevnom životu ovaj obrazac se može prepoznati kod kopiranja postojećeg dokumenta, dupliranja slajda u prezentaciji, pravljenja nove objave na osnovu prethodnog šablona, kopiranja formulara, oglasa ili plana aktivnosti, pri čemu se zadržava osnovna struktura, a menjaju samo konkretni detalji.

ZADATAK 1 – Builder: Sistem za kreiranje personalizovanog plana putovanja

- Turistička agencija želi aplikaciju koja zaposlenima omogućava da brzo kreiraju različite vrste aranžmana za klijente.
- Jedan plan putovanja treba da sadrži destinaciju, datume, prevoz, smeštaj, listu aktivnosti i informaciju da li je uključeno putno osiguranje.
- Agencija najčešće pravi dve varijante: kraći city break aranžman i porodični odmor. Obe varijante imaju iste osnovne delove, ali se razlikuju po vrednostima i aktivnostima.
- Potrebno je primeniti Builder Pattern tako da se složeni objekat TravelPlan ne kreira preko velikog konstruktora sa mnogo parametara iz korisničkog koda.

Zahtevi zadatka

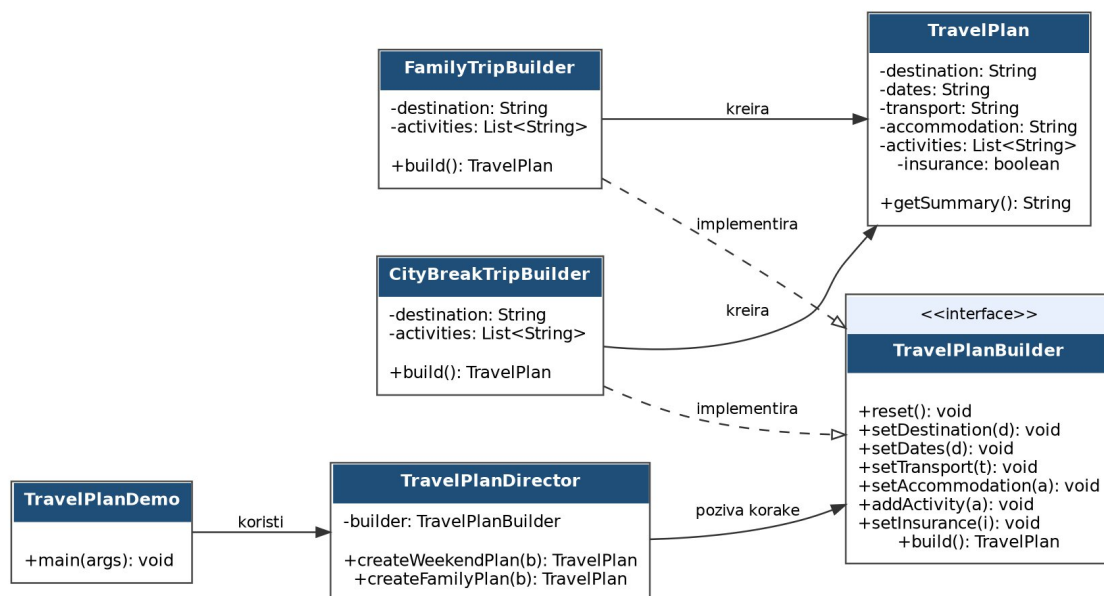
- Definirati klasu TravelPlan kao složen proizvod koji se kreira.
- Definirati TravelPlanBuilder interfejs sa koracima za podešavanje destinacije, datuma, prevoza, smeštaja, aktivnosti i osiguranja.
- Napraviti najmanje dve konkretne builder klase: CityBreakTripBuilder i FamilyTripBuilder.
- Napraviti Director klasu koja zna redosled koraka za tipične aranžmane.
- U main metodi kreirati dva različita plana putovanja i prikazati njihov sadržaj.

Tekstualno rešenje i objašnjenje

U ovom zadatku Builder Pattern je pogodan zato što plan putovanja ima više delova, a nisu svi jednako jednostavni za unos. Da se koristi samo konstruktor, poziv bi bio dug, nepregledan i podložan greškama. Builder razdvaja proces kreiranja od klase TravelPlan.

TravelPlan je Product. TravelPlanBuilder je interfejs koji definiše koje korake svaki builder mora da podrži. CityBreakTripBuilder i FamilyTripBuilder su ConcreteBuilder klase. TravelPlanDirector je klasa koja zna redosled popunjavanja podataka za uobičajene aranžmane. Client, odnosno main metoda, ne mora da zna sve detalje izgradnje.

- Asocijacija postoji između Director klase i Builder interfejsa, jer Director koristi builder za kreiranje plana.
- Implementacija postoji između konkretnih builder klasa i TravelPlanBuilder interfejsa.
- Veza kreiranja postoji između konkretnih buildera i klase TravelPlan, jer builder na kraju vraća gotov objekat.



Slika 1. UML class diagram za Builder Pattern - plan putovanja

Java kod rešenja

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

public class TravelPlanDemo {
    public static void main(String[] args) {
        TravelPlanDirector director = new TravelPlanDirector();

        TravelPlanBuilder cityBuilder = new CityBreakTripBuilder();
        TravelPlan weekend = director.createWeekendPlan(cityBuilder);
        System.out.println(weekend.getSummary());

        TravelPlanBuilder familyBuilder = new FamilyTripBuilder();
        TravelPlan familyTrip = director.createFamilyPlan(familyBuilder);
        System.out.println(familyTrip.getSummary());
    }
}

class TravelPlan {
    private final String destination;
    private final String dates;
    private final String transport;
    private final String accommodation;
    private final List<String> activities;
    private final boolean insurance;

    public TravelPlan(String destination, String dates, String transport,
        String accommodation, List<String> activities,
        boolean insurance) {
        this.destination = destination;
        this.dates = dates;
        this.transport = transport;
        this.accommodation = accommodation;
        this.activities = new ArrayList<>(activities);
        this.insurance = insurance;
    }

    public String getSummary() {
        return "Plan putovanja: " + destination + " | Datumi: " + dates +
            " | Prevoz: " + transport + " | Smeštaj: " + accommodation +
            " | Aktivnosti: " + activities +
            " | Osiguranje: " + (insurance ? "DA" : "NE");
    }
}

interface TravelPlanBuilder {
    void reset();
    void setDestination(String destination);
    void setDates(String dates);
    void setTransport(String transport);
    void setAccommodation(String accommodation);
    void addActivity(String activity);
    void setInsurance(boolean insurance);
    TravelPlan build();
}

class CityBreakTripBuilder implements TravelPlanBuilder {
    private String destination;
    private String dates;
    private String transport;
    private String accommodation;
    private final List<String> activities = new ArrayList<>();
    private boolean insurance;

    public void reset() {
        destination = "";
        dates = "";
        transport = "";
        accommodation = "";
        activities.clear();
        insurance = false;
    }
}
```



```

    public void setDestination(String destination) { this.destination = destination; }
    public void setDates(String dates) { this.dates = dates; }
    public void setTransport(String transport) { this.transport = transport; }
    public void setAccommodation(String accommodation) { this.accommodation = accommodation; }
    public void addActivity(String activity) { activities.add(activity); }
    public void setInsurance(boolean insurance) { this.insurance = insurance; }

    public TravelPlan build() {
        return new TravelPlan(destination, dates, transport, accommodation, activities, insurance);
    }
}

class FamilyTripBuilder implements TravelPlanBuilder {
    private String destination;
    private String dates;
    private String transport;
    private String accommodation;
    private final List<String> activities = new ArrayList<>();
    private boolean insurance;

    public void reset() {
        destination = "";
        dates = "";
        transport = "";
        accommodation = "";
        activities.clear();
        insurance = false;
    }

    public void setDestination(String destination) { this.destination = destination; }
    public void setDates(String dates) { this.dates = dates; }
    public void setTransport(String transport) { this.transport = transport; }
    public void setAccommodation(String accommodation) { this.accommodation = accommodation; }
    public void addActivity(String activity) { activities.add(activity); }
    public void setInsurance(boolean insurance) { this.insurance = insurance; }

    public TravelPlan build() {
        return new TravelPlan(destination, dates, transport, accommodation, activities, insurance);
    }
}

class TravelPlanDirector {
    public TravelPlan createWeekendPlan(TravelPlanBuilder builder) {
        builder.reset();
        builder.setDestination("Budimpešta");
        builder.setDates("petak - nedelja");
        builder.setTransport("autobus");
        builder.setAccommodation("hotel 3 zvezdice");
        builder.addActivity("obilazak centra grada");
        builder.addActivity("krstarenje Dunavom");
        builder.setInsurance(true);
        return builder.build();
    }

    public TravelPlan createFamilyPlan(TravelPlanBuilder builder) {
        builder.reset();
        builder.setDestination("Zlatibor");
        builder.setDates("7 dana");
        builder.setTransport("sopstveni prevoz");
        builder.setAccommodation("apartman sa kuhinjom");
        builder.addActivity("šetnja i izlet");
        builder.addActivity("aktivnosti za decu");
        builder.setInsurance(true);
        return builder.build();
    }
}

```

ZADATAK 2 - Builder: Online porudžbina hrane sa dodatnim opcijama

- Aplikacija za dostavu hrane omogućava kupcu da napravi porudžbinu iz izabranog restorana. Neke informacije su obavezne, na primer ime kupca, restoran, adresa i bar jedna stavka u porudžbini.
- Druge informacije su opcione: piće, napomena za dostavljača, prioriteta dostava i dodatne stavke. Nije dobro praviti mnogo različitih konstruktora za sve moguće kombinacije.
- Potrebno je primeniti Builder Pattern sa statičkom unutrašnjom Builder klasom i metodama koje vraćaju this.

Zahtevi zadatka

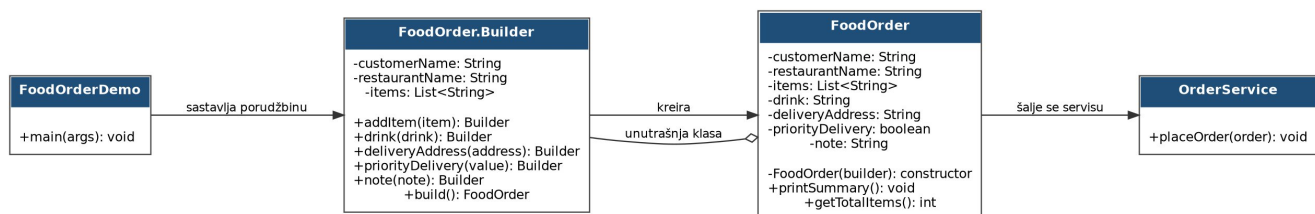
- Napraviti klasu FoodOrder koja predstavlja gotovu porudžbinu.
- U okviru klase FoodOrder napraviti statičku unutrašnju klasu Builder.
- Obavezna polja treba postaviti kroz konstruktor Builder klase, a opcione vrednosti kroz metode addItem(), drink(), deliveryAddress(), priorityDelivery() i note().
- Metoda build() treba da proveri da li je uneta bar jedna stavka i adresa za dostavu.
- Napraviti jednostavan OrderService koji prima gotovu porudžbinu i prikazuje njen sažetak.

Tekstualno rešenje i objašnjenje

Ovaj primer prikazuje čestu varijantu Builder obrasca. Umesto posebnog Director objekta koristi se statička unutrašnja Builder klasa. Ovakav pristup je pogodan kada objekat ima nekoliko obaveznih i mnogo opcionih podataka.

Fluent stil omogućava da se pozivi metoda nižu jedan za drugim, pa kod liči na čitljiv opis porudžbine. Metoda build() je centralno mesto za validaciju. Ako porudžbina nema stavke ili adresu, objekat se ne kreira.

- FoodOrder je Product i ima privatni konstruktor, pa se ne može napraviti direktno izvan klase.
- FoodOrder.Builder je graditelj koji prikuplja podatke i na kraju poziva private konstruktor klase FoodOrder.
- OrderService ne zna kako se porudžbina pravi; on dobija već validan objekat.



Slika 2. UML class dijagram za Builder Pattern - online porudžbina hrane

Java kod rešenja

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

public class FoodOrderDemo {
    public static void main(String[] args) {
        FoodOrder order = new FoodOrder.Builder("Ana Petrović", "Studentska menza Plus")
            .addItem("sendvič sa piletinom")
            .addItem("salata")
            .drink("limunada")
            .deliveryAddress("Bulevar oslobođenja 12, Niš")
            .priorityDelivery(true)
            .note("Pozvati kupca pre dolaska.")
            .build();

        OrderService.placeOrder(order);
    }
}

class FoodOrder {
    private final String customerName;
    private final String restaurantName;
    private final List<String> items;
    private final String drink;
    private final String deliveryAddress;
    private final boolean priorityDelivery;
    private final String note;

    FoodOrder(Builder builder) {
        this.customerName = builder.customerName;
        this.restaurantName = builder.restaurantName;
        this.items = builder.items;
        this.drink = builder.drink;
        this.deliveryAddress = builder.deliveryAddress;
        this.priorityDelivery = builder.priorityDelivery;
        this.note = builder.note;
    }

    void printSummary() {
        // Implementation
    }

    int getTotalItems() {
        // Implementation
    }
}
```

```

private final String restaurantName;
private final List<String> items;
private final String drink;
private final String deliveryAddress;
private final boolean priorityDelivery;
private final String note;

private FoodOrder(Builder builder) {
    this.customerName = builder.customerName;
    this.restaurantName = builder.restaurantName;
    this.items = new ArrayList<>(builder.items);
    this.drink = builder.drink;
    this.deliveryAddress = builder.deliveryAddress;
    this.priorityDelivery = builder.priorityDelivery;
    this.note = builder.note;
}

public int getTotalItems() {
    return items.size();
}

public void printSummary() {
    System.out.println("Kupac: " + customerName);
    System.out.println("Restoran: " + restaurantName);
    System.out.println("Stavke: " + items);
    System.out.println("Piće: " + drink);
    System.out.println("Adresa: " + deliveryAddress);
    System.out.println("Prioritetna dostava: " + (priorityDelivery ? "DA" : "NE"));
    System.out.println("Napomena: " + note);
}

public static class Builder {
    private final String customerName;
    private final String restaurantName;
    private final List<String> items = new ArrayList<>();
    private String drink = "bez pića";
    private String deliveryAddress = "nije uneta";
    private boolean priorityDelivery = false;
    private String note = "nema napomene";

    public Builder(String customerName, String restaurantName) {
        this.customerName = customerName;
        this.restaurantName = restaurantName;
    }

    public Builder addItem(String item) {
        items.add(item);
        return this;
    }

    public Builder drink(String drink) {
        this.drink = drink;
        return this;
    }

    public Builder deliveryAddress(String address) {
        this.deliveryAddress = address;
        return this;
    }

    public Builder priorityDelivery(boolean value) {
        this.priorityDelivery = value;
        return this;
    }

    public Builder note(String note) {
        this.note = note;
        return this;
    }

    public FoodOrder build() {
        if (items.isEmpty()) {
            throw new IllegalStateException("Porudžbina mora imati bar jednu stavku.");
        }
    }
}

```

```

    }
    if (deliveryAddress.equals("nije uneta")) {
        throw new IllegalStateException("Adresa za dostavu mora biti uneta.");
    }
    return new FoodOrder(this);
}
}
}
class OrderService {
    public static void placeOrder(FoodOrder order) {
        System.out.println("Porudžbina je primljena.");
        System.out.println("Broj stavki: " + order.getTotalItems());
        order.printSummary();
    }
}
}

```

ZADATAK 3 - Builder: Personalizovani program treninga u fitness aplikaciji

- Fitness aplikacija kreira programe treninga za korisnike različitog nivoa spremnosti. Program sadrži naziv, nivo, broj treninga nedeljno, listu vežbi, savet za ishranu i plan oporavka.
- Za početnike se pravi blaži program sa jednostavnijim vežbama, dok se za napredne korisnike pravi program sa više treninga i zahtevnijim vežbama.
- Potrebno je primeniti Builder Pattern sa Builder interfejsom, dve konkretne implementacije i Director klasom koja određuje redosled kreiranja programa.

Zahtevi zadatka

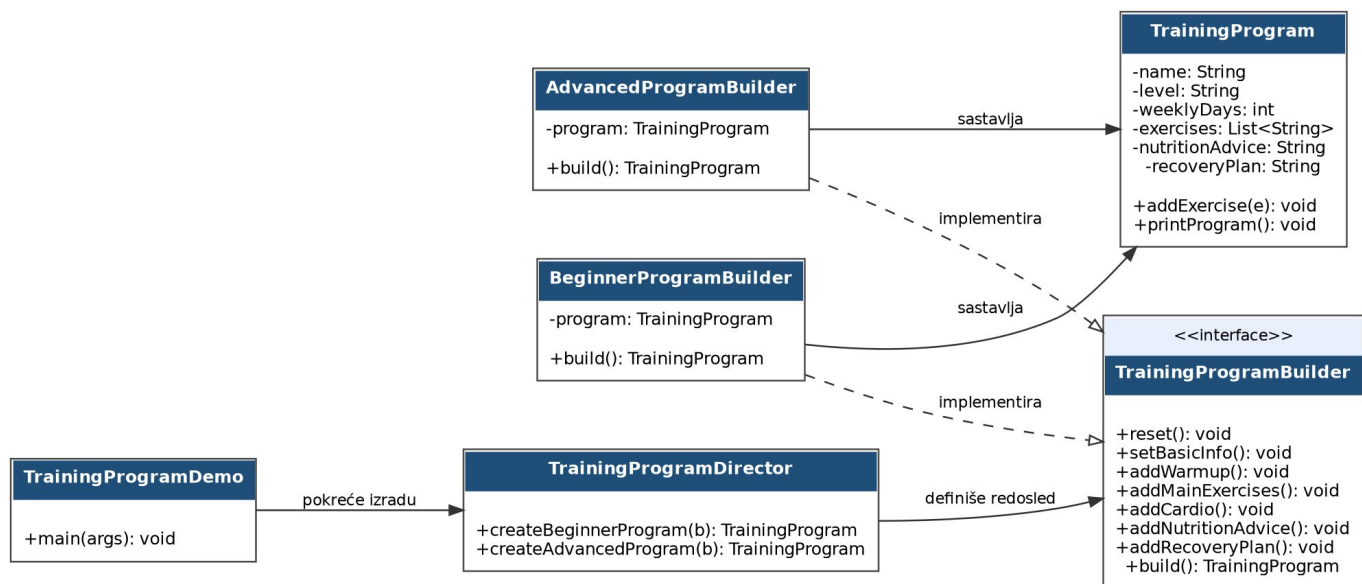
- Napraviti klasu TrainingProgram koja predstavlja gotov program treninga.
- Definirati TrainingProgramBuilder interfejs sa koracima za osnovne informacije, zagrevanje, glavne vežbe, kardio deo, ishranu i oporavak.
- Napraviti BeginnerProgramBuilder i AdvancedProgramBuilder.
- Napraviti TrainingProgramDirector koji koristi isti redosled koraka, ali može da radi sa različitim builderima.
- U main metodi prikazati program za početnika i program za naprednog korisnika.

Tekstualno rešenje i objašnjenje

Ovo rešenje pokazuje glavnu prednost Builder obrasca: isti proces izgradnje može dati različite rezultate. Director zna da program treba da ima osnovne informacije, zagrevanje, glavne vežbe, kardio, ishranu i oporavak. Međutim, Director ne mora da zna koje su tačno vežbe za početnike, a koje za napredne korisnike.

Promena konkretnog buildera menja rezultat, ali ne menja klijentski kod koji pokreće proces. To je korisno kada sistem kasnije treba proširiti novim tipovima programa, na primer programom za rehabilitaciju ili programom za mršavljenje.

- TrainingProgramBuilder je interfejs koji standardizuje korake izgradnje.
- BeginnerProgramBuilder i AdvancedProgramBuilder implementiraju iste metode, ali popunjavaju proizvod drugačijim vrednostima.
- TrainingProgramDirector koristi polimorfizam, jer radi sa tipom TrainingProgramBuilder, a ne sa konkretnom klasom.



Slika 3. UML class diagram za Builder Pattern - program treninga

Java kod rešenja

```

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

public class TrainingProgramDemo {
    public static void main(String[] args) {
        TrainingProgramDirector director = new TrainingProgramDirector();

        TrainingProgramBuilder beginnerBuilder = new BeginnerProgramBuilder();
        TrainingProgram beginner = director.createBeginnerProgram(beginnerBuilder);
        beginner.printProgram();

        TrainingProgramBuilder advancedBuilder = new AdvancedProgramBuilder();
        TrainingProgram advanced = director.createAdvancedProgram(advancedBuilder);
        advanced.printProgram();
    }
}

class TrainingProgram {
    private String name;
    private String level;
    private int weeklyDays;
    private final List<String> exercises = new ArrayList<>();
    private String nutritionAdvice;
    private String recoveryPlan;

    public void setName(String name) { this.name = name; }
    public void setLevel(String level) { this.level = level; }
    public void setWeeklyDays(int weeklyDays) { this.weeklyDays = weeklyDays; }
    public void addExercise(String exercise) { exercises.add(exercise); }
    public void setNutritionAdvice(String nutritionAdvice) { this.nutritionAdvice = nutritionAdvice; }
    public void setRecoveryPlan(String recoveryPlan) { this.recoveryPlan = recoveryPlan; }

    public void printProgram() {
        System.out.println("\nProgram: " + name);
        System.out.println("Nivo: " + level + ", broj treninga nedeljno: " + weeklyDays);
        System.out.println("Vežbe: " + exercises);
        System.out.println("Ishrana: " + nutritionAdvice);
        System.out.println("Oporavak: " + recoveryPlan);
    }
}

interface TrainingProgramBuilder {
    void reset();

```

```

void setBasicInfo();
void addWarmup();
void addMainExercises();
void addCardio();
void addNutritionAdvice();
void addRecoveryPlan();
TrainingProgram build();
}

class BeginnerProgramBuilder implements TrainingProgramBuilder {
    private TrainingProgram program;

    public void reset() { program = new TrainingProgram(); }

    public void setBasicInfo() {
        program.setName("Program za početnike");
        program.setLevel("početni");
        program.setWeeklyDays(3);
    }

    public void addWarmup() { program.addExercise("10 minuta laganog zagrevanja"); }
    public void addMainExercises() {
        program.addExercise("čučanj bez opterećenja");
        program.addExercise("sklekovi na povišenju");
        program.addExercise("vežbe za leđa sa elastičnom trakom");
    }

    public void addCardio() { program.addExercise("20 minuta brzog hodanja"); }
    public void addNutritionAdvice() { program.setNutritionAdvice("redovni obroci i dovoljan unos vode"); }
    public void addRecoveryPlan() { program.setRecoveryPlan("jedan dan odmora između treninga"); }
    public TrainingProgram build() { return program; }
}

class AdvancedProgramBuilder implements TrainingProgramBuilder {
    private TrainingProgram program;

    public void reset() { program = new TrainingProgram(); }

    public void setBasicInfo() {
        program.setName("Napredni program snage");
        program.setLevel("napredni");
        program.setWeeklyDays(5);
    }

    public void addWarmup() { program.addExercise("dinamičko zagrevanje i mobilnost"); }
    public void addMainExercises() {
        program.addExercise("mrtvo dizanje");
        program.addExercise("bench press");
        program.addExercise("zgibovi");
        program.addExercise("iskorak sa opterećenjem");
    }
    public void addCardio() { program.addExercise("intervalni kardio trening"); }
    public void addNutritionAdvice() { program.setNutritionAdvice("planiran unos proteina i praćenje kalorija"); }
    public void addRecoveryPlan() { program.setRecoveryPlan("aktivni oporavak, istezanje i kvalitetan san"); }
    public TrainingProgram build() { return program; }
}

class TrainingProgramDirector {
    public TrainingProgram createBeginnerProgram(TrainingProgramBuilder builder) {
        builder.reset();
        builder.setBasicInfo();
        builder.addWarmup();
        builder.addMainExercises();
        builder.addCardio();
        builder.addNutritionAdvice();
        builder.addRecoveryPlan();
        return builder.build();
    }

    public TrainingProgram createAdvancedProgram(TrainingProgramBuilder builder) {
        builder.reset();
        builder.setBasicInfo();
        builder.addWarmup();

```

```
builder.addMainExercises();  
builder.addCardio();  
builder.addNutritionAdvice();  
builder.addRecoveryPlan();  
return builder.build();  
}  
}
```

ZADATAK 4 - Prototype: Sistem za kopiranje šablona dokumenata u kanc.

- Administrativna služba svakodnevno pravi veliki broj sličnih dokumenata: ugovore, potvrde, saglasnosti i zapisnike. Svaki dokument ima naslov, tip dokumenta, ime klijenta i više standardnih klauzula.
- Umesto da zaposleni svaki put kreira dokument od početka, sistem treba da čuva unapred pripremljene šablone. Kada je potreban novi dokument, sistem pravi kopiju odgovarajućeg šablona, menja ime klijenta i po potrebi dodaje posebnu klauzulu.
- Potrebno je primeniti Prototype Pattern i obratiti pažnju na duboko kopiranje liste klauzula.

Zahtevi zadatka

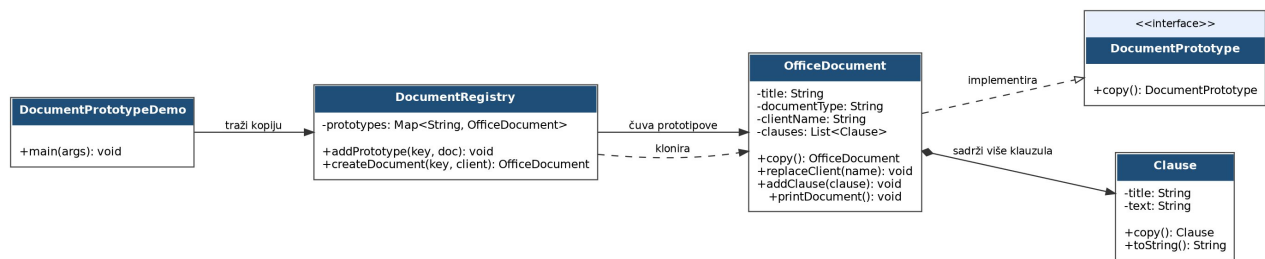
- Definirati DocumentPrototype interfejs sa metodom copy().
- Napraviti klasu OfficeDocument koja implementira DocumentPrototype.
- Napraviti klasu Clause koja predstavlja jednu klauzulu dokumenta.
- U metodi copy() klase OfficeDocument napraviti duboku kopiju svih klauzula.
- Napraviti DocumentRegistry koji čuva šablone dokumenata po ključu i vraća kopije za konkretnog klijenta.

Tekstualno rešenje i objašnjenje

Prototype Pattern je ovde pogodan zato što se dokumenti često ponavljaju, ali nisu potpuno isti. Šablon ugovora sadrži standardne klauzule koje ne treba unositi svaki put. Kada se napravi kopija, menja se samo ime klijenta i eventualno se dodaju posebne napomene.

Najvažniji deo rešenja je duboko kopiranje. Ako bi se kopirala samo referenca na listu klauzula, izmena jedne kopije mogla bi da promeni i drugi dokument ili originalni šablon. Zato svaka Clause klasa ima metodu copy(), a OfficeDocument.copy() pravi novu listu i nove objekte klauzula.

- DocumentPrototype je interfejs koji uvodi obavezu kopiranja.
- OfficeDocument je ConcretePrototype jer zna kako da napravi sopstvenu kopiju.
- DocumentRegistry je pomoćna klasa koja omogućava da se prototipovi pronalaze po nazivu, bez ručnog kreiranja.



Slika 4. UML class dijagram za Prototype Pattern - kancelarijski dokumenti

Java kod rešenja

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

public class DocumentPrototypeDemo {
    public static void main(String[] args) {
        OfficeDocument rentalTemplate = new OfficeDocument(
            "Šablon ugovora o zakupu", "ugovor", "[IME KLIJENTA]"
        );
        rentalTemplate.addClause(new Clause("Predmet ugovora", "Opisuje se stan koji se izdaje."));
        rentalTemplate.addClause(new Clause("Cena", "Navodi se mesečna zakupnina."));
        rentalTemplate.addClause(new Clause("Obaveze", "Definišu se obaveze zakupca i zakupodavca."));

        DocumentRegistry registry = new DocumentRegistry();
        registry.addPrototype("zakup", rentalTemplate);
    }
}
```



```

        OfficeDocument doc1 = registry.createDocument("zakup", "Marko Marković");
        doc1.addClause(new Clause("Posebna napomena", "Zakup počinje 1. juna."));
        OfficeDocument doc2 = registry.createDocument("zakup", "Jelena Jovanović");
        doc1.printDocument();
        doc2.printDocument();
    }
}

interface DocumentPrototype {
    DocumentPrototype copy();
}

class Clause {
    private final String title;
    private final String text;

    public Clause(String title, String text) {
        this.title = title;
        this.text = text;
    }

    public Clause copy() {
        return new Clause(title, text);
    }

    public String toString() {
        return title + ": " + text;
    }
}

class OfficeDocument implements DocumentPrototype {
    private String title;
    private String documentType;
    private String clientName;
    private final List<Clause> clauses = new ArrayList<>();

    public OfficeDocument(String title, String documentType, String clientName) {
        this.title = title;
        this.documentType = documentType;
        this.clientName = clientName;
    }

    public void replaceClient(String clientName) {
        this.clientName = clientName;
    }

    public void addClause(Clause clause) {
        clauses.add(clause);
    }

    public OfficeDocument copy() {
        OfficeDocument clone = new OfficeDocument(title, documentType, clientName);
        for (Clause clause : clauses) {
            clone.addClause(clause.copy());
        }
        return clone;
    }

    public void printDocument() {
        System.out.println("\nDokument: " + title);
        System.out.println("Tip: " + documentType);
        System.out.println("Klijent: " + clientName);
        for (Clause clause : clauses) {
            System.out.println(" - " + clause);
        }
    }
}

class DocumentRegistry {
    private final Map<String, OfficeDocument> prototypes = new HashMap<>();

    public void addPrototype(String key, OfficeDocument document) {

```

```

    prototypes.put(key, document);
}

public OfficeDocument createDocument(String key, String clientName) {
    OfficeDocument prototype = prototypes.get(key);
    if (prototype == null) {
        throw new IllegalArgumentException("Ne postoji šablon za ključ: " + key);
    }
    OfficeDocument copy = prototype.copy();
    copy.replaceClient(clientName);
    return copy;
}
}

```

ZADATAK 5 – Prototype: Kreiranje promotivnih objava za društvene mreže

- Tim za marketing organizuje humanitarni događaj i treba da pripremi promotivne objave za više platformi. Osnovna poruka kampanje, vizuelni stil i hashtagovi uglavnom su isti, ali se tekst i datum objave prilagođavaju platformi.
- Nije praktično da se svaka objava pravi od početka. Sistem treba da ima prototip objave za određenu kampanju, a zatim da pravi kopije za Instagram, LinkedIn i druge kanale.
- Potrebno je primeniti Prototype Pattern, uključiti Registry klasu i pokazati duboko kopiranje vizuelnog stila i liste hashtagova.

Zahtevi zadatka

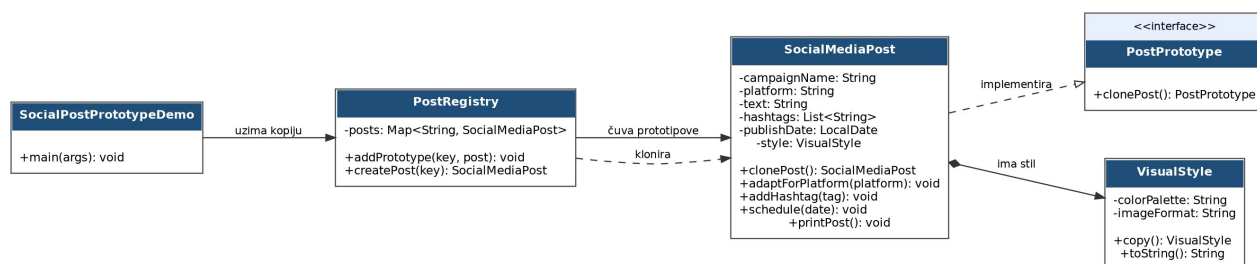
- Definisati PostPrototype interfejs sa metodom clonePost().
- Napraviti klasu SocialMediaPost koja sadrži naziv kampanje, platformu, tekst, hashtagove, datum i vizuelni stil.
- Napraviti klasu VisualStyle i obezbediti njeno kopiranje.
- Napraviti PostRegistry koji čuva prototipove objava.
- U main metodi klonirati istu osnovnu objavu i prilagoditi je za dve različite platforme.

Tekstualno rešenje i objašnjenje

Ovaj zadatak prikazuje praktičnu upotrebu Prototype obrasca u marketingu. Kada postoji osnovni šablon kampanje, mnogo je jednostavnije napraviti kopiju i prilagoditi je konkretnoj platformi nego svaki put kreirati novu objavu od nule.

Objava sadrži listu hashtagova i objekat VisualStyle. Zbog toga se mora voditi računa o dubokom kopiranju. Lista hashtagova se kopira u novu listu, a VisualStyle se kopira metodom copy(). Tako promene na LinkedIn objavi neće promeniti Instagram objavu ili originalni šablon.

- SocialMediaPost implementira PostPrototype i predstavlja konkretni prototip.
- PostRegistry omogućava centralno čuvanje prototipova kampanja.
- Metode adaptForPlatform(), addHashtag() i schedule() pokazuju kako se kopija prilagođava posle kloniranja.



Slika 5. UML class dijagram za Prototype Pattern - promotivne objave

Java kod rešenja

```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class SocialPostPrototypeDemo {
    public static void main(String[] args) {
        VisualStyle charityStyle = new VisualStyle("plava i bela", "kvadratna fotografija");
        SocialMediaPost template = new SocialMediaPost(
            "Humanitarni bazar", "Instagram",
            "Pridružite nam se i podržite lokalnu zajednicu.",
            charityStyle
        );
        template.addHashtag("#humanost");
        template.addHashtag("#donacije");

        PostRegistry registry = new PostRegistry();
        registry.addPrototype("bazar", template);

        SocialMediaPost instagramPost = registry.createPost("bazar");
        instagramPost.schedule(LocalDate.of(2026, 6, 1));
        instagramPost.addHashtag("#studenti");

        SocialMediaPost linkedInPost = registry.createPost("bazar");
        linkedInPost.adaptForPlatform("LinkedIn");
        linkedInPost.schedule(LocalDate.of(2026, 6, 2));

        instagramPost.printPost();
        linkedInPost.printPost();
    }
}

interface PostPrototype {
    PostPrototype clonePost();
}

class VisualStyle {
    private final String colorPalette;
    private final String imageFormat;

    public VisualStyle(String colorPalette, String imageFormat) {
        this.colorPalette = colorPalette;
        this.imageFormat = imageFormat;
    }

    public VisualStyle copy() {
        return new VisualStyle(colorPalette, imageFormat);
    }

    public String toString() {
        return colorPalette + ", format: " + imageFormat;
    }
}

class SocialMediaPost implements PostPrototype {
    private final String campaignName;
    private String platform;
    private String text;
    private final List<String> hashtags = new ArrayList<>();
    private LocalDate publishDate;
    private VisualStyle style;

    public SocialMediaPost(String campaignName, String platform, String text, VisualStyle style) {
        this.campaignName = campaignName;
        this.platform = platform;
        this.text = text;
        this.style = style;
    }

    public SocialMediaPost clonePost() {
```

```

        SocialMediaPost clone = new SocialMediaPost(campaignName, platform, text, style.copy());
        clone.hashtags.addAll(this.hashtags);
        clone.publishDate = this.publishDate;
        return clone;
    }

    public void adaptForPlatform(String platform) {
        this.platform = platform;
        if (platform.equalsIgnoreCase("LinkedIn")) {
            text = "Pozivamo partnere, nastavnike i studente da podrže akciju: " + text;
        } else if (platform.equalsIgnoreCase("Instagram")) {
            text = text + " Dodajte, donirajte i podelite objavu.";
        }
    }

    public void addHashtag(String hashtag) {
        hashtags.add(hashtag);
    }

    public void schedule(LocalDate date) {
        publishDate = date;
    }

    public void printPost() {
        System.out.println("\nKampanja: " + campaignName);
        System.out.println("Platforma: " + platform);
        System.out.println("Tekst: " + text);
        System.out.println("Hashtagovi: " + hashtags);
        System.out.println("Datum objave: " + publishDate);
        System.out.println("Vizuelni stil: " + style);
    }
}

class PostRegistry {
    private final Map<String, SocialMediaPost> posts = new HashMap<>();

    public void addPrototype(String key, SocialMediaPost post) {
        posts.put(key, post);
    }

    public SocialMediaPost createPost(String key) {
        SocialMediaPost prototype = posts.get(key);
        if (prototype == null) {
            throw new IllegalArgumentException("Ne postoji prototip objave za ključ: " + key);
        }
        return prototype.clonePost();
    }
}

```

1. Uvod u behavioral design patterns

Strategy i Observer pripadaju grupi obrazaca ponašanja, odnosno behavioral design patterns. Ovi obrasci ne rešavaju prvenstveno pitanje kako se objekti kreiraju, već kako objekti sarađuju, razmenjuju informacije i menjaju ponašanje tokom rada programa.

Strategy Pattern je koristan kada klasa može da izvršava isti zadatak na više različitih načina. Observer Pattern je koristan kada jedan objekat menja stanje, a više drugih objekata treba automatski da reaguje na tu promenu. Oba obrasca smanjuju direktnu zavisnost između klasa i čine program fleksibilnijim za proširenje.

- Strategy Pattern odgovara na pitanje: koju varijantu ponašanja želimo da primenimo u ovom trenutku?
- Observer Pattern odgovara na pitanje: koga sve treba obavestiti kada se promeni stanje važnog objekta?
- Kod oba obrasca važno je razdvojiti odgovornosti, tako da svaka klasa ima jasan i ograničen zadatak.

2. Strategy Pattern

Strategy Pattern se koristi kada postoji više algoritama ili više načina izvršavanja iste operacije, a program treba da može da izabere odgovarajući način rada bez menjanja glavne klase koja tu operaciju koristi. Umesto velikog broja if-else ili switch grananja, svaka varijanta ponašanja se izdvaja u posebnu klasu.

U svakodnevnom životu ovaj obrazac se može prepoznati kod izbora načina dostave hrane, obračuna različitih popusta u prodavnici, izbora rute u navigaciji, načina plaćanja, izbora paketa osiguranja ili podešavanja načina sortiranja proizvoda u online katalogu.

Klasična struktura Strategy obrasca obuhvata Strategy interfejs, više ConcreteStrategy klasa i Context klasu. Strategy interfejs definiše zajedničku operaciju. ConcreteStrategy klase predstavljaju konkretne algoritme. Context čuva referencu na izabranu strategiju i delegira izvršavanje toj strategiji.

Element	Opis	Šta student treba da prepozna
Strategy interfejs	Definiše zajedničku metodu koju sve strategije moraju da implementiraju.	Prepoznati ponašanje koje se menja: obračun cene, popusta, rute ili načina obrade.
ConcreteStrategy	Konkretna klasa koja implementira jednu varijantu algoritma.	Svaka strategija treba da ima jasno odvojenu logiku, bez mešanja sa drugim varijantama.
Context	Klasa koja koristi strategiju, ali ne zna detalje konkretne implementacije.	Prepoznati glavnu klasu koja treba fleksibilno da menja ponašanje.
Client	Kod koji bira konkretnu strategiju i prosleđuje je Context klasi.	Uočiti mesto gde se donosi odluka koju strategiju koristiti.

Kada koristiti Strategy: Kada više klasa sadrži slične algoritme, kada se ponašanje bira tokom izvršavanja programa ili kada if-else grananja postaju predugačka i teška za održavanje.

Kada ne koristiti Strategy: Kada postoji samo jedna jednostavna varijanta ponašanja i ne očekuje se proširenje. U tom slučaju dodatni interfejsi i klase mogu nepotrebno zakomplikovati program.

2.1. Opšta ideja Strategy obrasca

- Context ne izvršava sam algoritam, već poziva metodu iz Strategy interfejsa.
- Svaka ConcreteStrategy klasa može se menjati nezavisno od ostalih strategija.
- Dodavanje nove strategije najčešće ne zahteva izmenu postojećih strategija.
- Na ovaj način poštuje se Open/Closed princip: kod je otvoren za proširenje, ali zatvoren za nepotrebne izmene.

Zadatak 1 - Strategy Pattern: obračun dostave za online restoran

Tekst zadatka

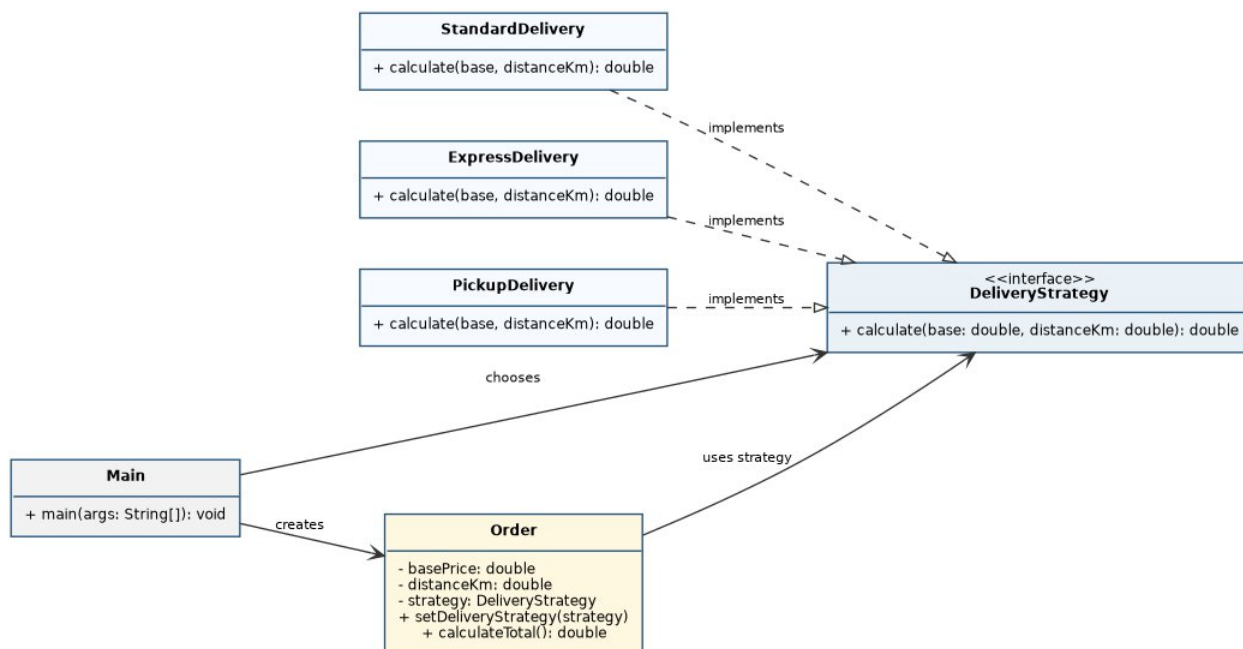
Online restoran želi jednostavan sistem za obračun ukupne cene porudžbine. Cena hrane je poznata, ali se cena dostave razlikuje u zavisnosti od izabranog načina isporuke. Korisnik može izabrati standardnu dostavu, brzu dostavu ili lično preuzimanje u lokalu.

- Standardna dostava dodaje cenu po kilometru.
- Brza dostava dodaje fiksnu naknadu i veću cenu po kilometru.
- Lično preuzimanje ne dodaje cenu dostave.
- Sistem treba da omogući promenu načina dostave bez promene klase koja predstavlja porudžbinu.

Ideja rešenja

Promenljivo ponašanje u ovom problemu je način obračuna ukupne cene. Zbog toga se pravi interfejs DeliveryStrategy, a svaka vrsta dostave predstavlja jednu konkretnu strategiju. Klasa Order ne zna detalje obračuna, već samo poziva metodu calculate nad trenutno izabranom strategijom.

UML Class Diagram



Slika 1. UML class dijagram za Strategy obrazac u sistemu dostave.

Java rešenje

```

interface DeliveryStrategy {
    double calculate(double basePrice, double distanceKm);
}

class StandardDelivery implements DeliveryStrategy {
    @Override
    public double calculate(double basePrice, double distanceKm) {
        return basePrice + distanceKm * 80;
    }
}

class ExpressDelivery implements DeliveryStrategy {
    @Override
    public double calculate(double basePrice, double distanceKm) {
        return basePrice + 300 + distanceKm * 120;
    }
}

class PickupDelivery implements DeliveryStrategy {
    @Override
    public double calculate(double basePrice, double distanceKm) {
        return basePrice;
    }
}
  
```

```

}

class Order {
    private double basePrice;
    private double distanceKm;
    private DeliveryStrategy deliveryStrategy;

    public Order(double basePrice, double distanceKm) {
        this.basePrice = basePrice;
        this.distanceKm = distanceKm;
        this.deliveryStrategy = new StandardDelivery();
    }

    public void setDeliveryStrategy(DeliveryStrategy deliveryStrategy) {
        this.deliveryStrategy = deliveryStrategy;
    }

    public double calculateTotal() {
        return deliveryStrategy.calculate(basePrice, distanceKm);
    }
}

public class Main {
    public static void main(String[] args) {
        Order order = new Order(2500, 4.5);

        order.setDeliveryStrategy(new StandardDelivery());
        System.out.println("Standardna dostava: " + order.calculateTotal() + " RSD");

        order.setDeliveryStrategy(new ExpressDelivery());
        System.out.println("Brza dostava: " + order.calculateTotal() + " RSD");

        order.setDeliveryStrategy(new PickupDelivery());
        System.out.println("Preuzimanje u lokalu: " + order.calculateTotal() + " RSD");
    }
}

```

Objašnjenje rešenja

DeliveryStrategy je interfejs koji propisuje metodu calculate. Klase StandardDelivery, ExpressDelivery i PickupDelivery implementiraju istu metodu, ali svaka na svoj način računa ukupnu cenu. Klasa Order predstavlja Context jer čuva referencu na izabranu strategiju. Metodom setDeliveryStrategy može se promeniti ponašanje objekta bez menjanja koda klase Order.

- Asocijacija postoji između Order i DeliveryStrategy jer porudžbina koristi izabranu strategiju.
- Realizacija interfejsa postoji između konkretnih strategija i DeliveryStrategy interfejsa.
- Ovo rešenje uklanja potrebu da Order ima if-else grananja za svaki tip dostave.

Zadatak 2 - Strategy Pattern: obračun popusta u online prodavnici

Tekst zadatka

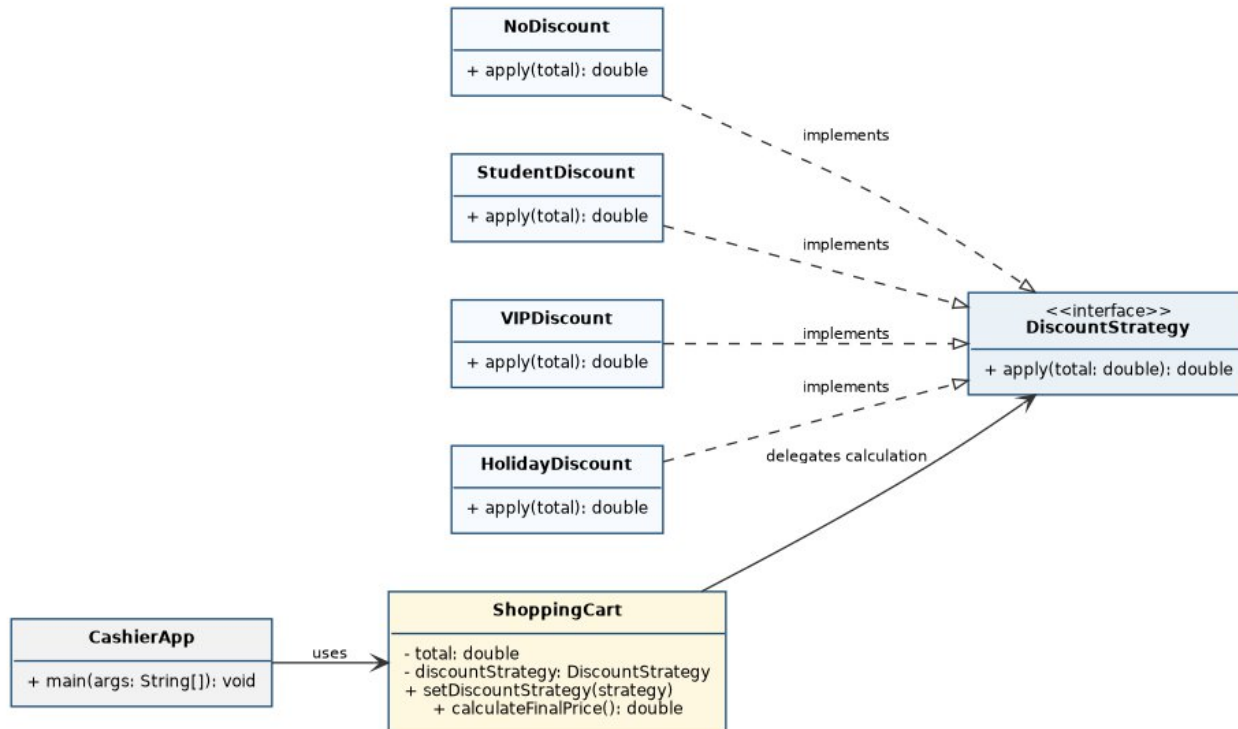
Online prodavnica garderobe želi sistem koji može da obračuna različite vrste popusta. Korpa ima početnu ukupnu cenu, ali se konačna cena razlikuje u zavisnosti od toga da li kupac nema popust, ima studentski popust, VIP popust ili praznični kupon.

- Kupac bez popusta plaća punu cenu.
- Studentski popust umanjuje cenu za 10%.
- VIP popust umanjuje cenu za 20% ako je kupovina veća od 10000 RSD, a inače za 10%.
- Praznični kupon umanjuje ukupnu cenu za fiksnih 1500 RSD.
- Sistem treba da omogući dodavanje novih vrsta popusta bez menjanja klase ShoppingCart.

Ideja rešenja

Obračun popusta je algoritam koji se menja. Zato se svaka vrsta popusta izdvaja u posebnu strategiju. ShoppingCart samo čuva ukupnu cenu i poziva izabranu DiscountStrategy. Kada se pojavi novi popust, dodaje se nova klasa koja implementira isti interfejs.

UML Class Diagram



Slika 2. UML class dijagram za Strategy obrazac u obračunu popusta.

Java rešenje

```

interface DiscountStrategy {
    double apply(double total);
}

class NoDiscount implements DiscountStrategy {
    @Override
    public double apply(double total) {
        return total;
    }
}

class StudentDiscount implements DiscountStrategy {
    @Override
    public double apply(double total) {
        return total * 0.90;
    }
}

class VIPDiscount implements DiscountStrategy {
    @Override
    public double apply(double total) {
        if (total >= 10000) {
            return total * 0.80;
        }
        return total * 0.90;
    }
}

class HolidayDiscount implements DiscountStrategy {

```



```

    @Override
    public double apply(double total) {
        return total - 1500;
    }
}

class ShoppingCart {
    private double total;
    private DiscountStrategy discountStrategy;

    public ShoppingCart(double total) {
        this.total = total;
        this.discountStrategy = new NoDiscount();
    }

    public void setDiscountStrategy(DiscountStrategy discountStrategy) {
        this.discountStrategy = discountStrategy;
    }

    public double calculateFinalPrice() {
        double finalPrice = discountStrategy.apply(total);
        return Math.max(finalPrice, 0);
    }
}

public class CashierApp {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart(12000);

        cart.setDiscountStrategy(new NoDiscount());
        System.out.println("Bez popusta: " + cart.calculateFinalPrice() + " RSD");

        cart.setDiscountStrategy(new StudentDiscount());
        System.out.println("Studentski popust: " + cart.calculateFinalPrice() + " RSD");

        cart.setDiscountStrategy(new VIPDiscount());
        System.out.println("VIP popust: " + cart.calculateFinalPrice() + " RSD");

        cart.setDiscountStrategy(new HolidayDiscount());
        System.out.println("Praznicni popust: " + cart.calculateFinalPrice() + " RSD");
    }
}

```

Objašnjenje rešenja

DiscountStrategy definiše zajedničku metodu apply. Klase NoDiscount, StudentDiscount, VIPDiscount i HolidayDiscount nude različite politike obračuna. ShoppingCart nije opterećena pravilima popusta, već samo delegira izračunavanje izabranoj strategiji.

- ShoppingCart je Context klasa jer koristi strategiju, ali ne zna detalje svake vrste popusta.
- Svaka ConcreteStrategy klasa može se testirati posebno.
- Dodavanje novog popusta, na primer BirthdayDiscount, ne zahteva promenu postojećih strategija.

Zadatak 3 - Strategy Pattern: izbor rute u gradskoj navigaciji

Tekst zadatka

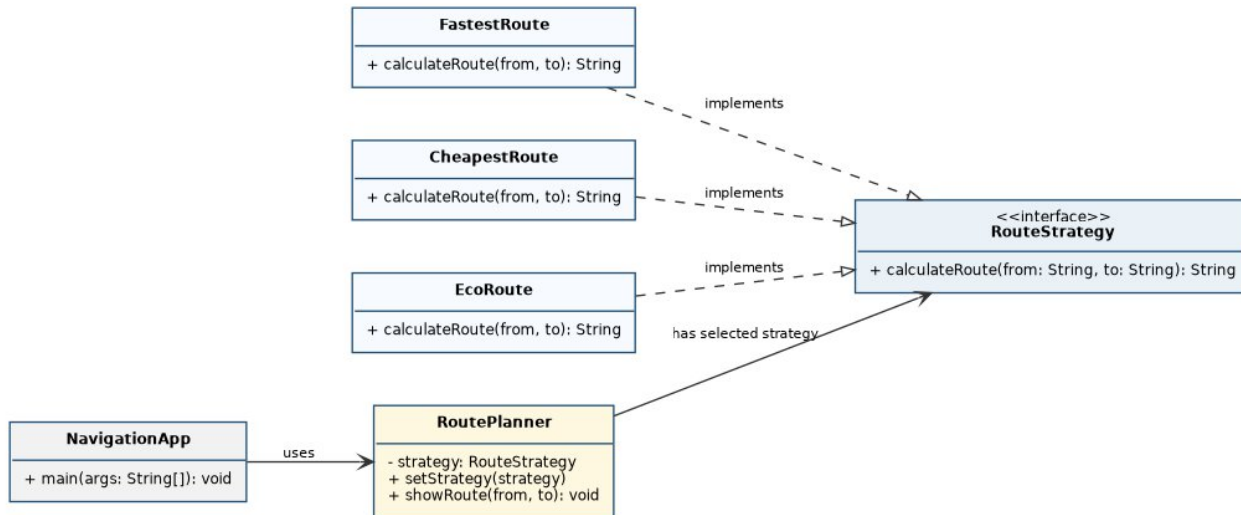
Aplikacija za gradsku navigaciju treba da predloži rutu od jedne lokacije do druge. Korisnik može izabrati najbržu rutu, najjeftiniju rutu ili ekološku rutu. Svaka ruta se računa drugačije, ali aplikacija treba da ima jedinstven način pozivanja izračunavanja.

- Najbrža ruta daje prednost vremenu putovanja.
- Najjeftinija ruta daje prednost nižoj ceni prevoza.
- Ekološka ruta daje prednost pešačenju, biciklu i deljenju prevoza.
- Korisnik tokom rada aplikacije može promeniti kriterijum izbora rute.

Ideja rešenja

Izbor rute je primer ponašanja koje se može menjati tokom rada programa. RoutePlanner ne treba da sadrži sve algoritme za rute, već treba da koristi interfejs RouteStrategy. Konkretne strategije implementiraju različite načine izračunavanja ili prikazivanja rute.

UML Class Diagram



Slika 3. UML class dijagram za Strategy obrazac u aplikaciji za navigaciju.

Java rešenje

```

interface RouteStrategy {
    String calculateRoute(String from, String to);
}

class FastestRoute implements RouteStrategy {
    @Override
    public String calculateRoute(String from, String to) {
        return "Najbrza ruta od " + from + " do " + to +
            ": auto-put i glavne saobraćajnice.";
    }
}

class CheapestRoute implements RouteStrategy {
    @Override
    public String calculateRoute(String from, String to) {
        return "Najjeftinija ruta od " + from + " do " + to +
            ": gradski prevoz i pesacenje.";
    }
}

class EcoRoute implements RouteStrategy {
    @Override
    public String calculateRoute(String from, String to) {
        return "Eko ruta od " + from + " do " + to +
            ": bicikl, pesacenje i deljenje prevoza.";
    }
}

class RoutePlanner {
    private RouteStrategy strategy;

    public RoutePlanner(RouteStrategy strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(RouteStrategy strategy) {
        this.strategy = strategy;
    }

    public void showRoute(String from, String to) {
        System.out.println(strategy.calculateRoute(from, to));
    }
}

public class NavigationApp {
    public static void main(String[] args) {
        RoutePlanner planner = new RoutePlanner(new FastestRoute());
        planner.showRoute("Centar", "Aerodrom");
        planner.setStrategy(new CheapestRoute());
        planner.showRoute("Centar", "Aerodrom");
    }
}
  
```

```
    planner.setStrategy(new EcoRoute());  
    planner.showRoute("Centar", "Aerodrom");  
}
```

Objašnjenje rešenja

RouteStrategy predstavlja opšti interfejs za računanje rute. FastestRoute, CheapestRoute i EcoRoute su konkretne strategije. RoutePlanner poseduje metodu setStrategy, pa može promeniti ponašanje bez kreiranja novog planera puta. Ovaj primer jasno pokazuje dinamičku zamenu algoritma tokom rada aplikacije.

- RoutePlanner zavisi od apstrakcije RouteStrategy, a ne od konkretne klase.
- Konkretne strategije se mogu dodavati bez promene RoutePlanner klase.
- U realnom sistemu svaka strategija bi mogla koristiti različite podatke: vreme, cenu, gužvu, emisiju CO2 ili dostupnost prevoza.

3. Observer Pattern

Observer Pattern se koristi kada jedan objekat, koji se često naziva Subject ili Observable, treba da obavesti više drugih objekata o promeni svog stanja. Ti drugi objekti se nazivaju Observer objekti. Umesto da Subject direktno poznaje sve detalje konkretnih klasa koje treba obavestiti, on radi preko zajedničkog Observer interfejsa.

U svakodnevnom životu ovaj obrazac se može prepoznati kod vremenskih upozorenja, obaveštenja na online kursu, praćenja statusa porudžbine, prijave na newsletter, notifikacija na društvenim mrežama, promena cena proizvoda ili sistema za praćenje slobodnih mesta na događaju.

Klasična struktura Observer obrasca obuhvata Subject klasu, Observer interfejs i više ConcreteObserver klasa. Subject čuva listu posmatrača, omogućava prijavu i odjavu posmatrača i poziva njihovu update metodu kada se dogodi promena.

Element	Opis	Šta student treba da prepozna
Subject / Observable	Objekat čije se stanje prati i koji šalje obaveštenja.	Prepoznati centralni izvor događaja, na primer kurs, stanica, prodavnica ili porudžbina.
Observer interfejs	Definiše update metodu koju svi posmatrači moraju da implementiraju.	Prepoznati zajednički način na koji različiti primaoci dobijaju obaveštenje.
ConcreteObserver	Konkretna klasa koja reaguje na obaveštenje.	Svaki primalac može reagovati drugačije na istu promenu.
Lista posmatrača	Subject čuva kolekciju prijavljenih Observer objekata.	Važno za dodavanje i uklanjanje pretplatnika tokom rada programa.

Kada koristiti Observer: Kada promena jednog objekta treba automatski da pokrene reakciju više drugih objekata, a ne želimo da objekti budu čvrsto povezani konkretnim klasama.

Kada ne koristiti Observer: Kada postoji samo jedan jednostavan primalac obaveštenja ili kada redosled izvršavanja mora biti strogo kontrolisan, jer veliki broj posmatrača može otežati praćenje toka programa.

3.1. Opšta ideja Observer obrasca

- Subject omogućava metodama subscribe/addObserver i unsubscribe/removeObserver upravljanje listom posmatrača.
- Kada se promeni stanje, Subject poziva notifyObservers.
- Svaki Observer dobija obaveštenje kroz metodu update.
- Subject ne mora da zna da li je posmatrač mobilna aplikacija, displej, email servis ili neka druga konkretna klasa.

Zadatak 4 - Observer Pattern: vremenska stanica i upozorenja građanima

Tekst zadatka

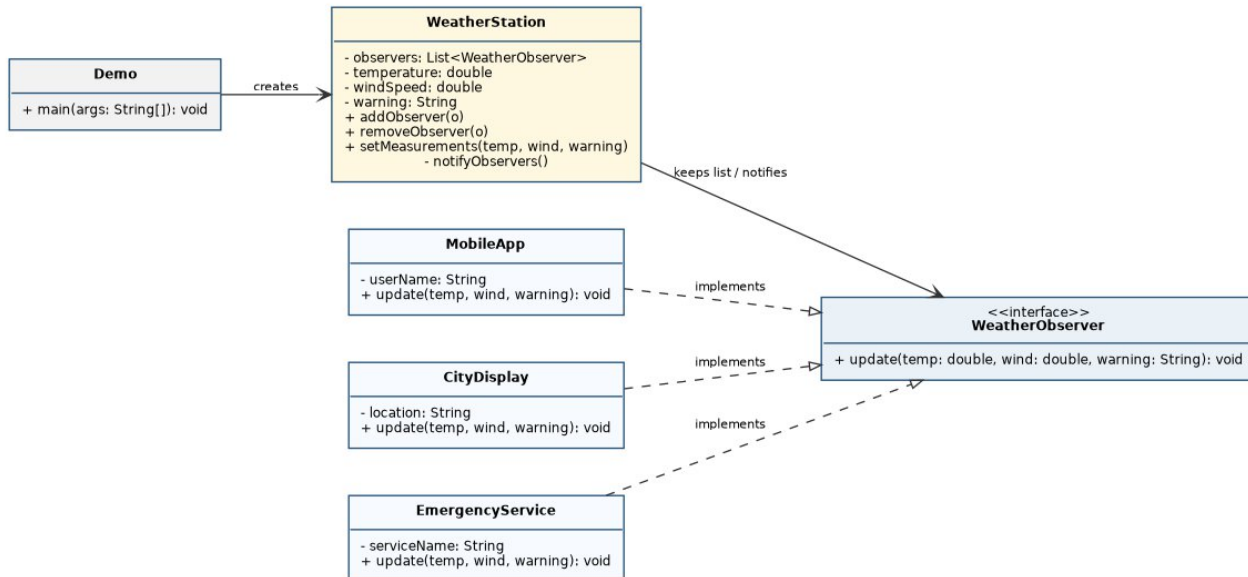
Grad želi jednostavan sistem za obaveštavanje građana o vremenskim promenama. Vremenska stanica meri temperaturu, brzinu vetra i eventualno upozorenje. Kada se podaci promene, obaveštenje treba da dobiju mobilna aplikacija korisnika, javni displej u gradu i služba za vanredne situacije.

- Mobilna aplikacija prikazuje sve osnovne vremenske podatke korisniku.
- Javni displej prikazuje temperaturu i upozorenje na određenoj lokaciji.
- Služba za vanredne situacije reaguje samo kada postoji posebno upozorenje.
- Sistem treba da omogući dodavanje novih primaoca obaveštenja bez promene klase WeatherStation.

Ideja rešenja

WeatherStation je Subject jer čuva stanje i obaveštava posmatrača. WeatherObserver je interfejs koji propisuje update metodu. MobileApp, CityDisplay i EmergencyService su konkretni posmatrači. Kada WeatherStation dobije nove podatke, poziva update nad svakim prijavljenim posmatračem.

UML Class Diagram



Slika 4. UML class dijagram za Observer obrazac u sistemu vremenskih upozorenja.

Java rešenje

```

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

interface WeatherObserver {
    void update(double temperature, double windSpeed, String warning);
}

class WeatherStation {
    private List<WeatherObserver> observers = new ArrayList<>();
    private double temperature;
    private double windSpeed;
    private String warning;

    public void addObserver(WeatherObserver observer) {
        observers.add(observer);
    }

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

    public void setMeasurements(double temperature, double windSpeed, String warning) {
        this.temperature = temperature;
        this.windSpeed = windSpeed;
        this.warning = warning;
        notifyObservers();
    }

    private void notifyObservers() {
        for (WeatherObserver observer : observers) {
            observer.update(temperature, windSpeed, warning);
        }
    }
}

class MobileApp implements WeatherObserver {
    private String userName;

    public MobileApp(String userName) {
        this.userName = userName;
    }

    @Override
    public void update(double temperature, double windSpeed, String warning) {
        System.out.println("Mobilna aplikacija za " + userName +
            ": temperatura " + temperature + " C, vetar " + windSpeed +
            " km/h, upozorenje: " + warning);
    }
}
  
```

```

    }
}

class CityDisplay implements WeatherObserver {
    private String location;

    public CityDisplay(String location) {
        this.location = location;
    }

    @Override
    public void update(double temperature, double windSpeed, String warning) {
        System.out.println("Displej na lokaciji " + location +
            ": " + temperature + " C, " + warning);
    }
}

class EmergencyService implements WeatherObserver {
    private String serviceName;

    public EmergencyService(String serviceName) {
        this.serviceName = serviceName;
    }

    @Override
    public void update(double temperature, double windSpeed, String warning) {
        if (!warning.equalsIgnoreCase("nema")) {
            System.out.println(serviceName + " prima hitno upozorenje: " + warning);
        }
    }
}

public class WeatherDemo {
    public static void main(String[] args) {
        WeatherStation station = new WeatherStation();
        station.addObserver(new MobileApp("Ana"));
        station.addObserver(new CityDisplay("Trg kralja Milana"));
        station.addObserver(new EmergencyService("Gradska sluzba za vanredne situacije"));
        station.setMeasurements(32.5, 18.0, "nema");
        station.setMeasurements(38.0, 45.0, "opasnost od oluje");
    }
}

```

Objašnjenje rešenja

WeatherStation čuva listu WeatherObserver objekata. Metode addObserver i removeObserver omogućavaju prijavu i odjavu posmatrača. Kada se pozove setMeasurements, stanica ažurira svoje podatke i zatim obaveštava sve posmatrače. Svaki posmatrač sam odlučuje kako će reagovati na primljene podatke.

- Veza WeatherStation prema WeatherObserver predstavlja asocijaciju prema listi posmatrača.
- MobileApp, CityDisplay i EmergencyService realizuju WeatherObserver interfejs.
- WeatherStation nije zavisna od konkretnih klasa posmatrača, što omogućava lako proširenje sistema.

Zadatak 5 - Observer Pattern: obaveštenja na online kursu

Tekst zadatka

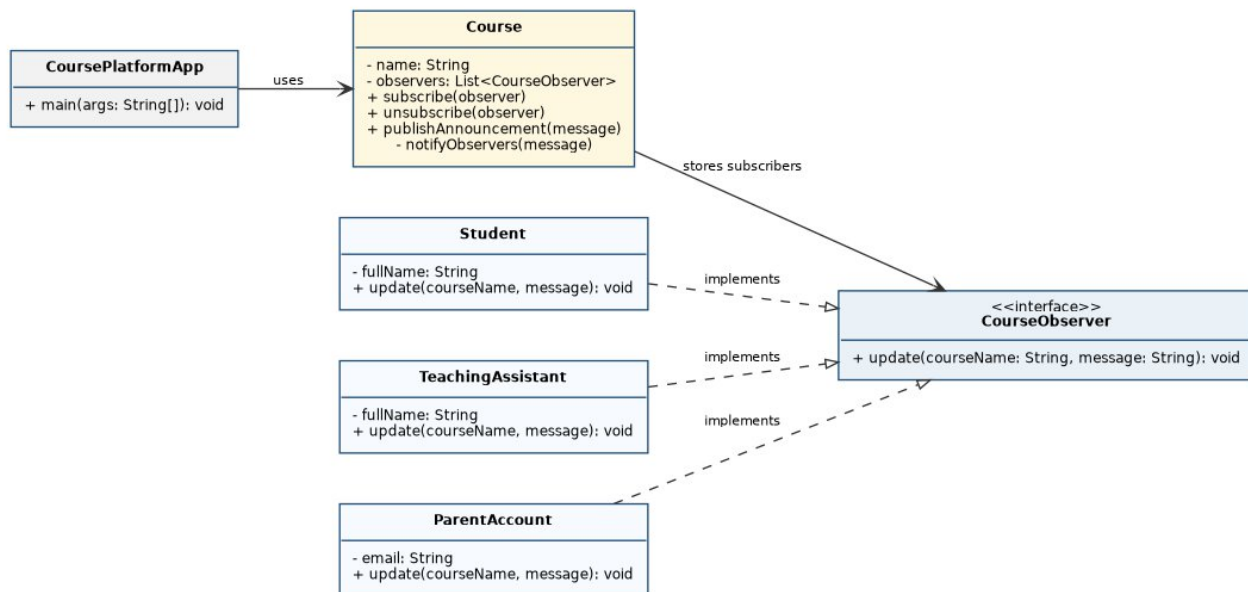
Platforma za online nastavu želi da automatski obaveštava sve zainteresovane korisnike kada nastavnik objavi novu informaciju na kursu. Obaveštenje mogu primiti studenti, asistent i roditeljski nalog. Neki korisnici mogu naknadno da se odjave sa obaveštenja.

- Kurs ima naziv i listu pretplaćenih korisnika.
- Student dobija obaveštenje kao poruku za praćenje nastave.
- Asistent dobija obaveštenje kako bi mogao da proveri materijale i pitanja studenata.
- Roditeljski nalog dobija opšte informacije, ali može biti odjavljen.
- Sistem treba da podrži više različitih primalaca bez menjanja klase Course.

Ideja rešenja

Course je Subject jer objavljuje informacije i obaveštava posmatrača. CourseObserver je interfejs za sve korisnike koji žele da dobiju obaveštenje. Student, TeachingAssistant i ParentAccount su različite implementacije posmatrača.

UML Class Diagram



Slika 5. UML class dijagram za Observer obrazac u platformi za online nastavu.

Java rešenje

```

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

interface CourseObserver {
    void update(String courseName, String message);
}

class Course {
    private String name;
    private List<CourseObserver> observers = new ArrayList<>();

    public Course(String name) {
        this.name = name;
    }

    public void subscribe(CourseObserver observer) {
        observers.add(observer);
    }

    public void unsubscribe(CourseObserver observer) {
        observers.remove(observer);
    }

    public void publishAnnouncement(String message) {
        System.out.println("\nNova objava na kursu: " + name);
        notifyObservers(message);
    }
}
  
```

```

    }

    private void notifyObservers(String message) {
        for (CourseObserver observer : observers) {
            observer.update(name, message);
        }
    }
}

class Student implements CourseObserver {
    private String fullName;

    public Student(String fullName) {
        this.fullName = fullName;
    }

    @Override
    public void update(String courseName, String message) {
        System.out.println("Student " + fullName + " dobija obavestenje za " +
            courseName + ": " + message);
    }
}

class TeachingAssistant implements CourseObserver {
    private String fullName;

    public TeachingAssistant(String fullName) {
        this.fullName = fullName;
    }

    @Override
    public void update(String courseName, String message) {
        System.out.println("Asistent " + fullName + " proverava objavu na kursu " +
            courseName + ": " + message);
    }
}

class ParentAccount implements CourseObserver {
    private String email;

    public ParentAccount(String email) {
        this.email = email;
    }

    @Override
    public void update(String courseName, String message) {
        System.out.println("Roditeljski nalog " + email +
            " dobija informaciju sa kursa " + courseName + ": " + message);
    }
}

public class CoursePlatformApp {
    public static void main(String[] args) {
        Course course = new Course("Softversko inzenjerstvo");
        Student milan = new Student("Milan Petrovic");
        Student sara = new Student("Sara Jovanovic");
        TeachingAssistant assistant = new TeachingAssistant("Milena Nikolic");
        ParentAccount parent = new ParentAccount("roditelj@example.com");
        course.subscribe(milan);
        course.subscribe(sara);
        course.subscribe(assistant);
        course.subscribe(parent);
        course.publishAnnouncement("Kolokvijum se odrzava u petak u 10h.");
        course.unsubscribe(parent);
        course.publishAnnouncement("Postavljeni su dodatni materijali za vezbu.");
    }
}

```

Objašnjenje rešenja

Course čuva listu objekata tipa CourseObserver. Kada se objavi nova informacija, metoda publishAnnouncement poziva notifyObservers i prosleđuje poruku svim prijavljenim korisnicima. Svaka konkretna klasa posmatrača drugačije prikazuje ili obrađuje istu poruku.

- Pretplata i odjava su deo odgovornosti Subject klase.
- CourseObserver omogućava da Course ne zavisi od konkretnih klasa Student, TeachingAssistant i ParentAccount.
- Novi tip posmatrača, na primer EmailNotification ili MobilePushNotification, može se dodati kao nova klasa koja implementira isti interfejs.

Poređenje Strategy i Observer obrasca

Kriterijum	Strategy Pattern	Observer Pattern
Glavno pitanje	Koji algoritam ili ponašanje treba koristiti?	Koga treba obavestiti kada se stanje promeni?
Centralna klasa	Context koristi jednu izabranu strategiju.	Subject čuva listu posmatrača.
Tip veze	Context delegira posao strategiji.	Subject obaveštava sve prijavljene posmatrače.
Promena tokom rada	Strategija se može zameniti u toku izvršavanja.	Posmatrači se mogu prijaviti ili odjaviti u toku izvršavanja.
Tipičan primer	Popust, dostava, ruta, način plaćanja.	Notifikacije, pretplate, status porudžbine, promene na kursu.

Kratak zaključak

Strategy i Observer su među najkorisnijim obrascima ponašanja jer rešavaju veoma česte probleme u aplikacijama. Strategy omogućava da se promenljivo ponašanje izdvoji iz glavne klase i zameni bez menjanja njenog koda. Observer omogućava da se promena stanja jednog objekta automatski prosledi većem broju zainteresovanih objekata.

Kod analize zadatka student najpre treba da prepozna da li se problem odnosi na izbor jednog od više algoritama ili na obaveštavanje više primalaca. Ako je naglasak na promenljivom načinu rada, najverovatnije je rešenje Strategy. Ako je naglasak na pretplatnicima, obaveštenjima i reakcijama na promenu stanja, najverovatnije je rešenje Observer.

Praktični savet: U UML dijagramima obavezno označiti interfejse, konkretne klase koje ih implementiraju i klasu koja koristi ili obaveštava te objekte. U Java kodu treba jasno da se vidi gde se koristi polimorfizam, jer je upravo on osnova oba obrasca.

Decorator Pattern

Decorator Pattern pripada grupi strukturnih projektnih obrazaca. Njegova osnovna ideja je da se postojećem objektu dinamički dodaju nove funkcionalnosti, bez menjanja njegove klase i bez pravljenja velikog broja podklasa. Umesto da se ponašanje proširuje nasleđivanjem, objekat se obavlja drugim objektom koji ima isti interfejs i dodaje dodatno ponašanje pre, posle ili oko osnovne funkcionalnosti.

U svakodnevnom životu ovaj obrazac se može prepoznati kod naručivanja kafe sa dodacima, formiranja paketa usluga u teretani, kreiranja poklon paketa, dodavanja opcija na automobil, izbora dodataka u online prodavnici ili proširivanja osnovne pretplate dodatnim pogodnostima. U svim tim situacijama postoji osnovni proizvod ili usluga, a korisnik naknadno bira dodatke koji menjaju opis, cenu ili ponašanje.

Decorator je posebno koristan kada broj kombinacija može brzo da poraste. Na primer, ako kafa može imati mleko, čokoladu, šlag i karamelu, bilo bi nepraktično praviti posebnu klasu za svaku kombinaciju. Umesto toga, svaka dopuna postaje poseban dekorator koji može da obavlja osnovnu kafu ili već dekorisanu kafu.

Element	Opis	Šta student treba da prepozna
Component	Zajednički interfejs ili apstraktna klasa za osnovni objekat i dekoratore.	Prepoznati zajedničke metode, kao što su <code>getOpis()</code> , <code>getCena()</code> ili <code>execute()</code> .
ConcreteComponent	Osnovna klasa koja predstavlja početni objekat bez dodatnih opcija.	Prepoznati osnovni proizvod ili uslugu, na primer osnovnu kafu, osnovnu članarinu ili osnovni poklon.
Decorator	Apstraktna klasa koja implementira isti interfejs kao Component i čuva referencu na objekat koji obavlja.	Uočiti da dekorator ima polje tipa Component i da poziva metode obavljenog objekta.
ConcreteDecorator	Konkretna dopuna koja proširuje ponašanje osnovnog objekta.	Prepoznati dodatke koji menjaju opis, cenu, pravila ili dodatne operacije.
Client	Kod koji kombinuje osnovni objekat i dekoratore.	Videti kako se objekat gradi lančano, na primer <code>new Slag(new Mleko(new OsnovnaKafa()))</code> .

Kada se koristi Decorator Pattern?

- Kada je potrebno dinamički dodavati funkcionalnosti pojedinačnim objektima, a ne celoj klasi.
- Kada postoji mnogo mogućih kombinacija dodataka, opcija ili ponašanja.
- Kada bi rešenje kroz nasleđivanje dovelo do velikog broja klasa i komplikovane hijerarhije.
- Kada želimo da osnovna klasa ostane jednostavna i zatvorena za izmene, ali otvorena za proširenje.
- Kada više dodataka treba da se mogu kombinovati u različitom redosledu i u različitim varijantama.

Prednosti i ograničenja

Prednosti	Ograničenja
Funkcionalnosti se dodaju bez menjanja postojeće klase.	Može nastati veći broj malih klasa ako postoji mnogo različitih dekoratora.
Izbegava se eksplozija podklasa i previše složeno nasleđivanje.	Kod može biti teže pratiti ako postoji veliki lanac dekoratora.
Podržava kombinovanje dodataka u runtime toku izvršavanja.	Potrebno je dobro razumeti zajednički interfejs i referencu na obavijeni objekat.
Dobro podržava princip otvoreno za proširenje, zatvoreno za izmene.	Debugovanje može biti teže jer se pozivi prenose kroz više slojeva.

Zadatak 1 - Porudžbina kafe sa dodacima

Tekst zadatka

Kafić želi jednostavan informacijski sistem za formiranje porudžbine kafe. Kupac najpre bira osnovni napitak, na primer espresso ili latte, a zatim može da doda različite dodatke kao što su mleko, čokolada, šlag ili karamela. Svaki dodatak menja opis porudžbine i povećava ukupnu cenu. Sistem treba da omogući da se dodaci kombinuju slobodno, bez kreiranja posebne klase za svaku moguću kombinaciju kafe i dodataka.

Zahtevi za studente

- Prepoznati osnovni interfejs Napitak sa metodama getOpis() i getCena().
- Napraviti osnovnu klasu OsnovnaKafa koja predstavlja konkretan napitak.
- Napraviti apstraktni dekorator DodatakDekorator koji čuva referencu na objekat tipa Napitak.
- Implementirati najmanje četiri konkretna dekoratora: Mleko, Cokolada, Slag i Karamela.
- U Main klasi prikazati najmanje dve različite porudžbine sa različitim kombinacijama dodataka.
- Objasniti zašto je Decorator pogodniji od pravljenja posebnih klasa za svaku kombinaciju dodataka.

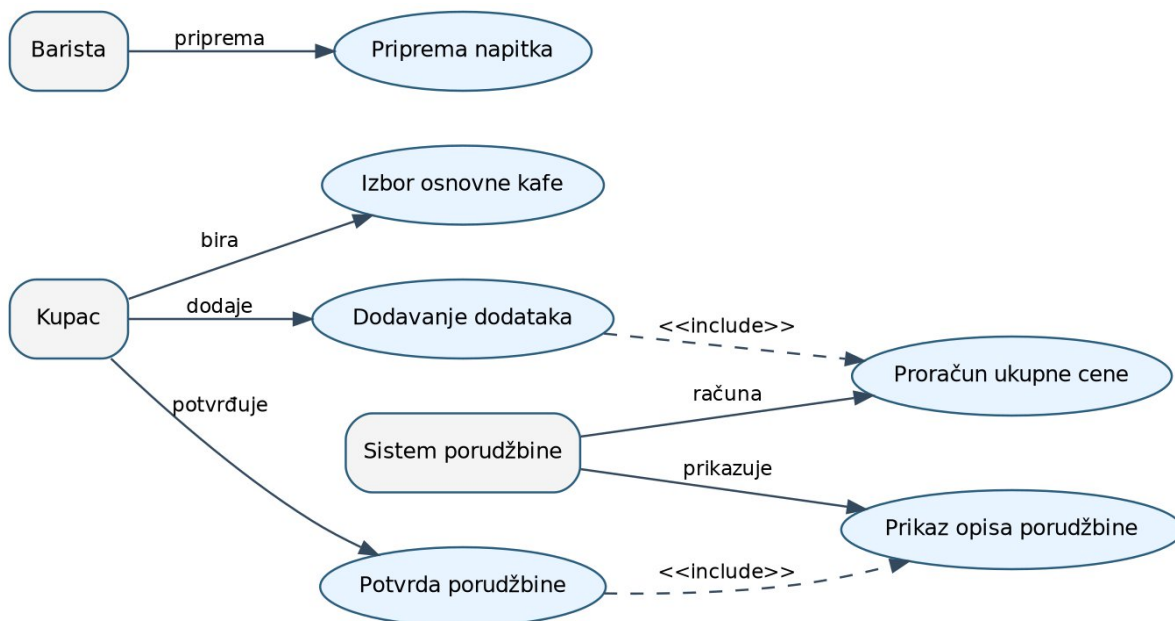
Rešenje i objašnjenje

U ovom rešenju Napitak predstavlja zajednički interfejs za osnovnu kafu i za sve dodatke. To je ključno za Decorator obrazac, jer osnovni objekat i dekoratori moraju imati iste metode. Zbog toga se svaki dodatak može posmatrati kao novi Napitak, iako u sebi čuva prethodni napitak.

OsnovnaKafa je konkretna komponenta i predstavlja početni objekat. DodatakDekorator je apstraktna klasa koja takođe implementira Napitak, ali u sebi ima referencu na drugi Napitak. Konkretni dekoratori kao što su Mleko, Cokolada, Slag i Karamela pozivaju metode obavljenog objekta i zatim dodaju svoj opis i cenu.

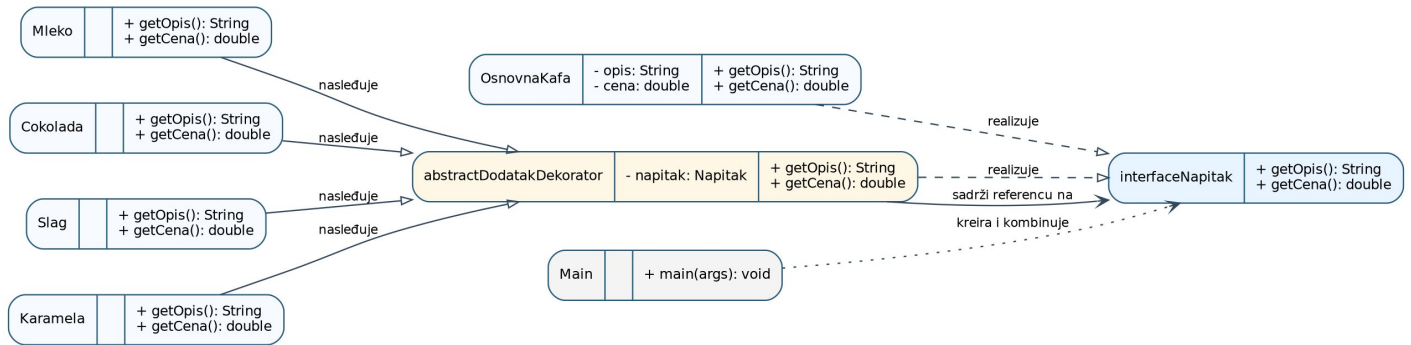
Na ovaj način korisnik može da napravi porudžbinu korak po korak. Ako se kasnije pojavi novi dodatak, na primer cimet, dovoljno je dodati novu klasu Cimet koja nasleđuje DodatakDekorator. Ne mora se menjati OsnovnaKafa niti postojeći dekoratori.

Use Case dijagram



Use Case dijagram za dati scenario.

UML Class diagram



UML Class diagram koji prikazuje strukturu Decorator obrasca.

```

interface Napitak {
    String getOpis();
    double getCena();
}

class OsnovnaKafa implements Napitak {
    private String opis;
    private double cena;

    public OsnovnaKafa(String opis, double cena) {
        this.opis = opis;
        this.cena = cena;
    }

    @Override
    public String getOpis() {
        return opis;
    }

    @Override
    public double getCena() {
        return cena;
    }
}

abstract class DodatakDekorator implements Napitak {
    protected Napitak napitak;

    public DodatakDekorator(Napitak napitak) {
        this.napitak = napitak;
    }
}

class Mleko extends DodatakDekorator {
    public Mleko(Napitak napitak) {
        super(napitak);
    }

    @Override
    public String getOpis() {
        return napitak.getOpis() + ", mleko";
    }

    @Override
    public double getCena() {
        return napitak.getCena() + 40;
    }
}
    
```

```

    }
}

class Cokolada extends DodatakDekorator {
    public Cokolada(Napitak napitak) {
        super(napitak);
    }

    @Override
    public String getOpis() {
        return napitak.getOpis() + ", čokolada";
    }

    @Override
    public double getCena() {
        return napitak.getCena() + 60;
    }
}

class Slag extends DodatakDekorator {
    public Slag(Napitak napitak) {
        super(napitak);
    }

    @Override
    public String getOpis() {
        return napitak.getOpis() + ", šlag";
    }

    @Override
    public double getCena() {
        return napitak.getCena() + 50;
    }
}

class Karamela extends DodatakDekorator {
    public Karamela(Napitak napitak) {
        super(napitak);
    }

    @Override
    public String getOpis() {
        return napitak.getOpis() + ", karamela";
    }

    @Override
    public double getCena() {
        return napitak.getCena() + 70;
    }
}

public class Main {
    public static void main(String[] args) {
        Napitak porudzbina = new OsnovnaKafa("Espresso", 180);
        porudzbina = new Mleko(porudzbina);
        porudzbina = new Cokolada(porudzbina);
        porudzbina = new Slag(porudzbina);

        System.out.println("Porudžbina: " + porudzbina.getOpis());
        System.out.println("Ukupna cena: " + porudzbina.getCena() + " RSD");

        Napitak drugaPorudzbina = new Karamela(new OsnovnaKafa("Latte", 240));
        System.out.println("Porudžbina: " + drugaPorudzbina.getOpis());
        System.out.println("Ukupna cena: " + drugaPorudzbina.getCena() + " RSD");
    }
}

```

```
}  
}
```

Kratko objašnjenje koda

- Napitak je Component, jer definiše zajedničko ponašanje za osnovni objekat i dekoratore.
- OsnovnaKafa je ConcreteComponent, jer predstavlja stvarni osnovni objekat koji se dekorše.
- DodatakDekorator je apstraktni Decorator i sadrži referencu na Napitak.
- Mleko, Cokolada, Slag i Karamela su ConcreteDecorator klase.
- Svaki dekorator proširuje opis i cenu, ali ne menja kod osnovne kafe.
- Dekoratori se mogu kombinovati u različitom redosledu, pa se od iste osnovne kafe mogu napraviti različite varijante bez kreiranja posebne klase za svaku moguću kombinaciju dodataka.

Zadatak 2 - Fleksibilna članarina u teretani

Tekst zadatka

Teretana želi sistem za kreiranje fleksibilnih članarina. Svaki korisnik može izabrati osnovnu mesečnu članarinu, a zatim dodati opcione usluge kao što su lični trener, individualni plan ishrane, grupni treninzi ili pristup spa zoni. Svaka dodatna usluga povećava mesečnu cenu i dodaje novi opis paketa. Sistem treba da podrži različite kombinacije usluga bez pravljenja posebne klase za svaki mogući paket članarine.

Zahtevi za studente

- Napraviti interfejs Clanarina sa metodama getOpis() i getMesecnaCena().
- Napraviti osnovnu klasu OsnovnaClanarina koja predstavlja početni paket.
- Napraviti apstraktni dekorator DodatakClanarine koji čuva referencu na objekat tipa Clanarina.
- Implementirati konkretne dekoratore LicniTrener, PlanIshrane, GrupniTreninzi i SpaZona.
- U Main klasi prikazati formiranje standardnog i premium paketa.
- Objasniti kako se pomoću Decorator obrasca izbegava pravljenje klasa kao što su StandardSaTrenerom, StandardSaTreneromISpaZonom i slično.

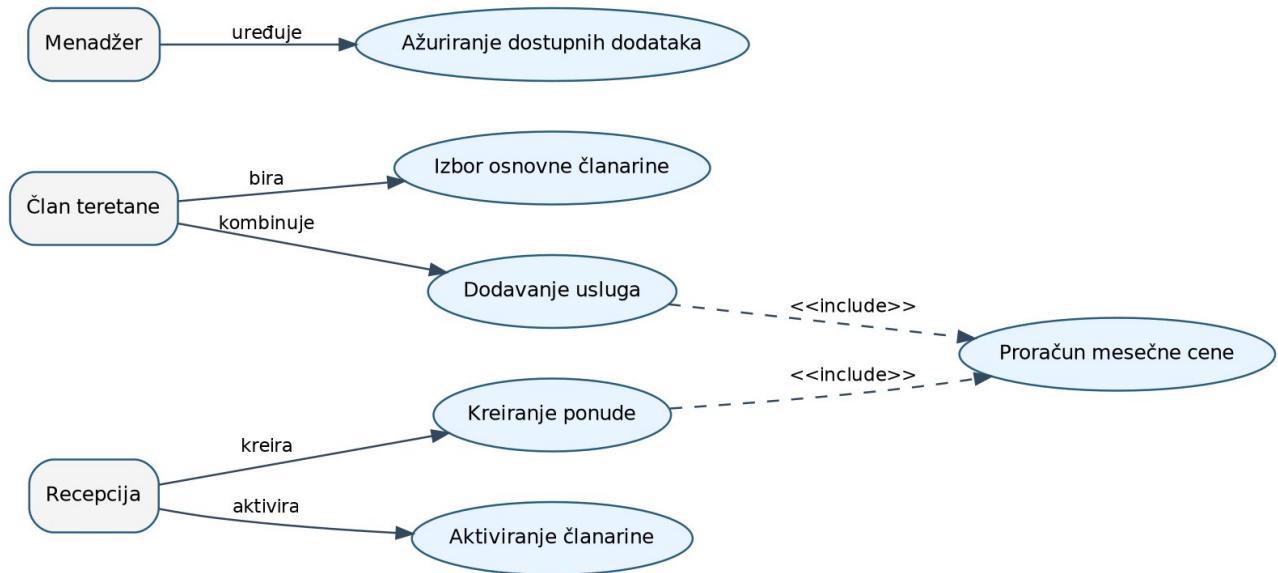
Rešenje i objašnjenje

U ovom primeru osnovna članarina predstavlja početni proizvod, dok se sve dodatne usluge ponašaju kao dekoratori. Interfejs Clanarina omogućava da se i osnovni paket i svi dodaci koriste na isti način, preko metoda getOpis() i getMesecnaCena().

Apstraktna klasa DodatakClanarine čuva referencu na postojeću članarinu. Kada se doda novi dekorator, on ne briše prethodne informacije, već ih proširuje. Na primer, ako se na osnovnu članarinu doda grupni trening, zatim plan ishrane i spa zona, svaki novi dekorator dodaje svoj deo opisa i svoju cenu.

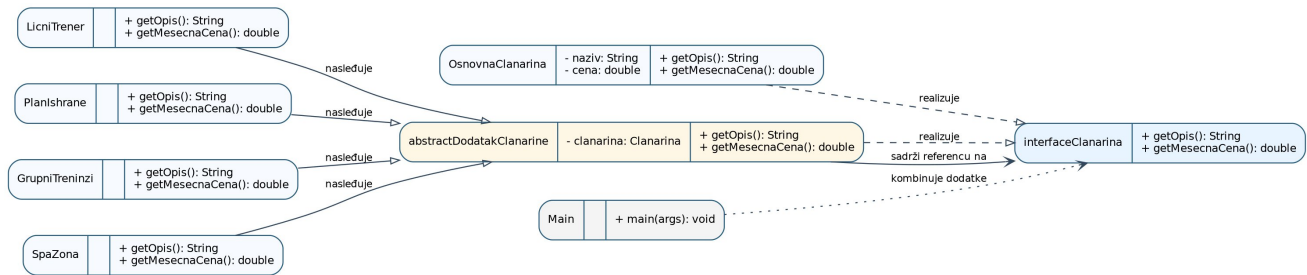
Ovakvo rešenje je pogodno za realne sisteme jer teretana može često uvoditi nove dodatne usluge. Umesto menjanja postojećih klasa, dodaje se nova klasa dekoratora, čime se smanjuje rizik od greške u već testiranom kodu.

Use Case dijagram



Use Case dijagram za dati scenario.

UML Class dijagram



UML Class dijagram koji prikazuje strukturu Decorator obrasca.

```

interface Clanarina {
    String getOpis();
    double getMesecnaCena();
}

class OsnovnaClanarina implements Clanarina {
    private String naziv;
    private double cena;

    public OsnovnaClanarina(String naziv, double cena) {
        this.naziv = naziv;
        this.cena = cena;
    }

    @Override
    public String getOpis() {
        return naziv;
    }

    @Override
    public double getMesecnaCena() {
        return cena;
    }
}
  
```

```

}

abstract class DodatakClanarine implements Clanarina {
    protected Clanarina clanarina;

    public DodatakClanarine(Clanarina clanarina) {
        this.clanarina = clanarina;
    }
}

class LicniTrener extends DodatakClanarine {
    public LicniTrener(Clanarina clanarina) {
        super(clanarina);
    }

    @Override
    public String getOpis() {
        return clanarina.getOpis() + ", lični trener";
    }

    @Override
    public double getMesecnaCena() {
        return clanarina.getMesecnaCena() + 4500;
    }
}

class PlanIshrane extends DodatakClanarine {
    public PlanIshrane(Clanarina clanarina) {
        super(clanarina);
    }

    @Override
    public String getOpis() {
        return clanarina.getOpis() + ", plan ishrane";
    }

    @Override
    public double getMesecnaCena() {
        return clanarina.getMesecnaCena() + 1800;
    }
}

class GrupniTreninzi extends DodatakClanarine {
    public GrupniTreninzi(Clanarina clanarina) {
        super(clanarina);
    }

    @Override
    public String getOpis() {
        return clanarina.getOpis() + ", grupni treninzi";
    }

    @Override
    public double getMesecnaCena() {
        return clanarina.getMesecnaCena() + 1200;
    }
}

class SpaZona extends DodatakClanarine {
    public SpaZona(Clanarina clanarina) {
        super(clanarina);
    }

    @Override

```



```

public String getOpis() {
    return clanarina.getOpis() + ", spa zona";
}

@Override
public double getMesečnaCena() {
    return clanarina.getMesečnaCena() + 2000;
}
}

public class Main {
    public static void main(String[] args) {
        Clanarina paket = new OsnovnaClanarina("Standard mesečna članarina", 3500);
        paket = new GrupniTreninzi(paket);
        paket = new PlanIshrane(paket);
        paket = new SpaZona(paket);

        System.out.println("Izabrani paket: " + paket.getOpis());
        System.out.println("Mesečna cena: " + paket.getMesečnaCena() + " RSD");

        Clanarina premium = new LicniTreners(
            new SpaZona(new OsnovnaClanarina("Premium članarina", 5500))
        );
        System.out.println("Izabrani paket: " + premium.getOpis());
        System.out.println("Mesečna cena: " + premium.getMesečnaCena() + " RSD");
    }
}

```

Kratko objašnjenje koda

- Clanarina je zajednički interfejs i omogućava da se svi paketi tretiraju jednako.
- OsnovnaClanarina čuva naziv i osnovnu cenu paketa.
- DodatakClanarine je apstraktni dekorator koji povezuje dodatnu uslugu sa postojećim paketom.
- Svaki konkretni dekorator dodaje jednu realnu uslugu i uvećava mesečnu cenu.
- Lančanim obavijanjem objekta dobijaju se različite kombinacije bez eksplozije klasa.
- Na ovaj način korisnik može fleksibilno da prilagodi članarinu svojim potrebama, bez izmene postojećih klasa.

Zadatak 3 - Personalizovani poklon paket

Tekst zadatka

Online prodavnica poklona želi sistem za kreiranje personalizovanih poklon paketa. Kupac najpre bira osnovni poklon, na primer knjigu, set kozmetike ili poklon korpu. Nakon toga može dodati ukrasno pakovanje, čestitku sa porukom, dodatne čokoladice i brzu dostavu. Svaki dodatak menja opis paketa i povećava ukupnu cenu. Sistem treba da omogući slobodno kombinovanje dodataka i jednostavno proširenje novim opcijama.

Zahtevi za studente

- Napraviti interfejs Poklon sa metodama getOpis() i getCena().
- Napraviti klasu OsnovniPoklon kao konkretnu komponentu.
- Napraviti apstraktni dekorator DekoracijaPoklona koji čuva referencu na objekat tipa Poklon.
- Implementirati dekoratore UkrasnoPakovanje, Cestitka, Cokoladice i BrzaDostava.
- U Main klasi prikazati jedan složeni poklon paket i jedan jednostavniji poklon.
- Uočiti šta bi se promenilo ako bi prodavnica uvela novi dodatak, na primer personalizovanu kutiju ili balone.

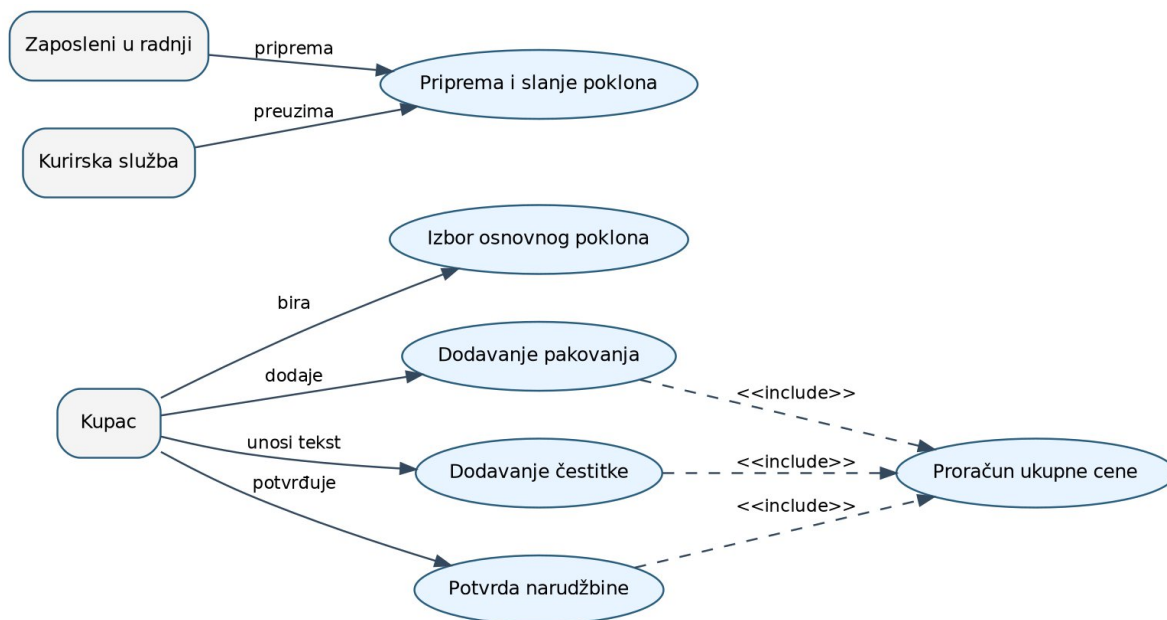
Rešenje i objašnjenje

Ovaj zadatak pokazuje Decorator Pattern na primeru personalizacije poklona. Osnovni poklon je konkretna komponenta, dok se ukrasno pakovanje, čestitka, čokoladice i brza dostava dodaju kao dekoratori. Svaki dekorator zadržava prethodno stanje paketa i dodaje svoju informaciju.

Posebno je važno što dekorator Čestitka ima dodatni atribut tekst. To pokazuje da dekorator ne mora samo da doda fiksnu cenu i opis, već može imati i sopstvene podatke. Ipak, on i dalje poštuje isti interfejs Poklon i može se kombinovati sa drugim dekoratorima.

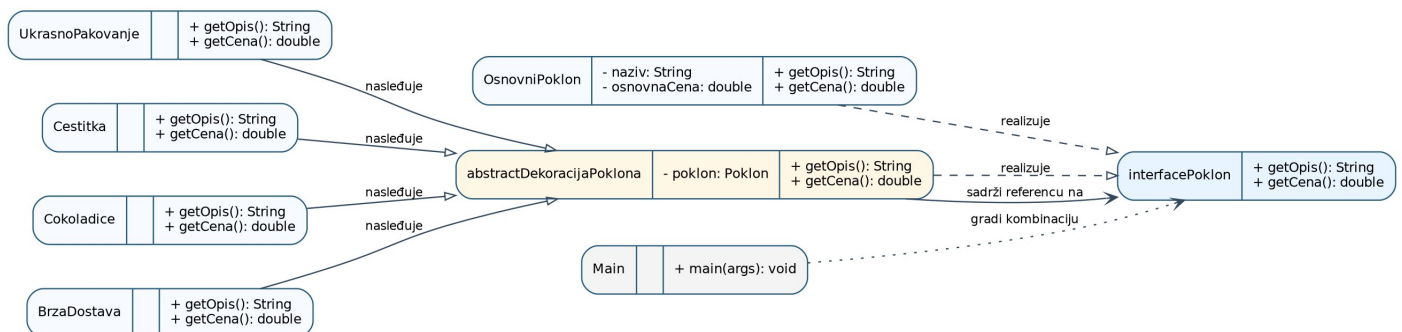
Rešenje je fleksibilno jer prodavnica može uvesti nove dodatke bez izmene postojećih klasa. Ako se uvede personalizovana kutija, dodaje se nova klasa koja nasleđuje DekoracijaPoklona i implementira svoje verzije metoda getOpis() i getCena().

Use Case diagram



Use Case dijagram za dati scenario.

UML Class diagram



UML Class dijagram koji prikazuje strukturu Decorator obrasca.

```
interface Poklon {
    String getOpis();
```

```

    double getCena();
}

class OsnovniPoklon implements Poklon {
    private String naziv;
    private double osnovnaCena;

    public OsnovniPoklon(String naziv, double osnovnaCena) {
        this.naziv = naziv;
        this.osnovnaCena = osnovnaCena;
    }

    @Override
    public String getOpis() {
        return naziv;
    }

    @Override
    public double getCena() {
        return osnovnaCena;
    }
}

abstract class DekoracijaPoklona implements Poklon {
    protected Poklon poklon;

    public DekoracijaPoklona(Poklon poklon) {
        this.poklon = poklon;
    }
}

class UkrasnoPakovanje extends DekoracijaPoklona {
    public UkrasnoPakovanje(Poklon poklon) {
        super(poklon);
    }

    @Override
    public String getOpis() {
        return poklon.getOpis() + ", ukrasno pakovanje";
    }

    @Override
    public double getCena() {
        return poklon.getCena() + 350;
    }
}

class Cestitka extends DekoracijaPoklona {
    private String tekst;

    public Cestitka(Poklon poklon, String tekst) {
        super(poklon);
        this.tekst = tekst;
    }

    @Override
    public String getOpis() {
        return poklon.getOpis() + ", čestitka: \"" + tekst + "\"";
    }

    @Override
    public double getCena() {
        return poklon.getCena() + 150;
    }
}

```

```

class Cokoladice extends DekoracijaPoklona {
    public Cokoladice(Poklon poklon) {
        super(poklon);
    }

    @Override
    public String getOpis() {
        return poklon.getOpis() + ", dodatne čokoladice";
    }
    @Override
    public double getCena() {
        return poklon.getCena() + 500;
    }
}

class BrzaDostava extends DekoracijaPoklona {
    public BrzaDostava(Poklon poklon) {
        super(poklon);
    }

    @Override
    public String getOpis() {
        return poklon.getOpis() + ", brza dostava";
    }
    @Override
    public double getCena() {
        return poklon.getCena() + 600;
    }
}

public class Main {
    public static void main(String[] args) {
        Poklon rodjendanskiPoklon = new OsnovniPoklon("Set za negu lica", 3200);
        rodjendanskiPoklon = new UkrasnoPakovanje(rodjendanskiPoklon);
        rodjendanskiPoklon = new Cestitka(rodjendanskiPoklon, "Srećan rođendan!");
        rodjendanskiPoklon = new Cokoladice(rodjendanskiPoklon);
        rodjendanskiPoklon = new BrzaDostava(rodjendanskiPoklon);

        System.out.println("Poklon: " + rodjendanskiPoklon.getOpis());
        System.out.println("Ukupna cena: " + rodjendanskiPoklon.getCena() + " RSD");

        Poklon jednostavanPoklon = new UkrasnoPakovanje(
            new OsnovniPoklon("Knjiga", 1400)
        );
        System.out.println("Poklon: " + jednostavanPoklon.getOpis());
        System.out.println("Ukupna cena: " + jednostavanPoklon.getCena() + " RSD");
    }
}

```

Kratko objašnjenje koda

- Poklon je Component i definiše zajedničke metode za osnovni poklon i sve dodatke.
- OsnovniPoklon je ConcreteComponent, jer predstavlja početnu verziju proizvoda.
- DekoracijaPoklona je Decorator i čuva referencu na već postojeći Poklon.
- UkrasnoPakovanje, Cestitka, Cokoladice i BrzaDostava su konkretni dekoratori.
- Dodaci se mogu slagati u različitom redosledu, a ukupna cena se računa prolaskom kroz lanac dekoratora.