OMD – Övning vecka 3

Du förväntas göra lösningsförslag till uppgifterna före övningen, gärna tillsammans med andra kursdeltagare.

Lösningar till dessa uppgifter presenteras under övningen i läsvecka 3.

---

1 En mobiltelefonoperatör erbjuder olika slags abonnemang. Läraren i OMD har sagt att när beskrivningen av verkligheten använder "olika slags ..." skall man använda arv. Den uppmärksamme teknologen gör följande design på sommarjobbet på Comviq:

```
public abstract class Tariff {
    private String phoneNumber;
    protected List<Call> call = new ArrayList<Call>();
    public abstract Currency debit(Call call);
    // omissions
}

public class CashCard extends Tariff {
    public Currency debit(Call call) {
    // omissions
    }
}
public class KnockOut extends Tariff {
    public Currency debit(Call call) {
    // omissions
    }
}
```

Projektledaren har invändningar; när en abonnent vill byta abonnemangsform måste man skapa ett nytt `Tariff`-objekt och kopiera telefonnummer mm från det gamla abonnemanget. Gör en ny design utan denna olägenhet som utnyttjar designmönstret *Strategy*. Redovisa designen med ett klassdiagram.

2 En tidigare upplaga av en lärobok i objektorienterad programmering och Java innehåller följande kod:

```
...
private Application myApp;
private Button b1, b2, b3;
class ActionHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == b1) {
            myApp.action1();
        } else if (e.getSource() == b2) {
            myApp.action2();
        } else {
            myApp.action3();
        }
    }
}
```

Designen strider mot *Open-Closed Principle*; man kan inte lägga till ytterligare `Button`s utan att modifiera klassen. Gör om designen så att detta blir möjligt. Redovisa designen med ett klassdiagram.

3 Syftet med designmönstret *Singleton* är att förhindra att mer än en instans av klassen kan skapas. En bieffekt i den traditionella implementeringen är att objektet blir globalt åtkomligt.

Implementera *Singelton*-mönstret så att det bara begränsar antalet instanser av klassen, men inte automatiskt gör instansen globalt åtkomlig.

4 Följande två klasser innehåller duplicerad kod. Elimenera den genom att använda *Template Method*-mönstret. Lösningen redovisas som Java-kod.

```
public class StarList extends java.util.ArrayList {
    public String toString() {
        StringBuilder builder = new StringBuilder();
        for (int i = 0; i < size(); i++) {
            builder.append("* ");
            builder.append(get(i));
            builder.append('\n');
        }
        return builder.toString();
    }

public class EnumList extends java.util.ArrayList {
    public String toString() {
        StringBuilder builder = new StringBuilder();
        for (int i = 0; i < size(); i++) {
            builder.append(i + 1).append(". ");
            builder.append(get(i));
            builder.append('\n');
        }
        return builder.toString();
    }
```

5 I följande exempel används `null` för att representera okända föräldrar. Använd *Null Object*-mönstret för att få en bättre design där det inte behövs några if-satser. Lösningen redovisas med ett klassdiagram med generaliseringar, attribut och metoder samt Java-kod.

```
public class Person {
    private String name;
    private Person father, mother;
    public void printAncestors() {
        System.out.println(name);
        if (father != null) {
            father.printAncestors();
        }
        if (mother != null) {
            mother.printAncestors();
        }
    }
}
```

6 Med institutionens kaffeautomat kan man välja olika drycker och flera tillbehör genom att trycka på knappar. Använd *Decorator*-mönstret för att modellera möjligheten att välja kaffe eller choklad, med eller utan tillbehören mjölk och socker. Om man trycker flera på tillbehörsknapparna ökar dosen på tillbehöret.

Modellen skall testas med

```
public static void main(String[] args) {
     Drink drink = new Sugar(new Milk(new Milk(new Coffee())));
     System.out.println(drink);
     System.out.println(drink.cost());
}
```

med den förväntade utskriften

```
kaffe mjölk mjölk socker
7
```

Kostnaden för en dos kaffe är 5 kronor, en skvätt mjölk kostar 1 kr och sockret är gratis.

Gör ett klassdiagram för hela modellen och implementera klasserna Coffee och Milk och alla abstrakta klasser.

7 Du skall konstruera ett program som simulerar köer på en bank, post eller systembolagsbutik. I lokalen finns en könummerautomat och ett antal kassor eller betjäningsstationer.

När en kund ankommer får han ett könummer. Om någon av kassorna är ledig betjänas han omgående. Annars får kunden vänta. När någon av kassorna blir ledig blir kunden med lägst könummer betjänad. För att kunna göra beräkningar eller simuleringar av hur systemet fungerar behövs modeller för när kunder anländer och hur lång tid en betjäning tar. Vi antar att alla betjäningsstationer är ekvivalenta och att inga kunder lämnar lokalen utan att ha blivit betjänade. Vi antar att tidsavståndet mellan kundankomster är exponentialfördelad och att betjäningstiden är likformigt fördelad. (Du behöver inte känna till dessa begrepp; antag bara att det finns klasser vars objekt innehåller metoder som producerar slumpmässiga tider med de önskade egenskaperna).

Simuleringsprogrammet skall styras av händelser. De händelser som kan inträffa är att en kund ankommer till systemet och att en kund lämnar det efter att ha blivit betjänad. Vid dessa händelser förändras systemets tillstånd. Tillståndet beskrivs av en tidpunkt, antalet lediga kassor, en kö av kunder som väntar på betjäning och en mängd av händelser som skall inträffa. En kund som ankommer hamnar i kön om alla kassor är upptagna. Annars får han direkt betjäning vilket innebär att antalet lediga kassor minskar med 1. Med hjälp av slumptalsgeneratorn för betjäningstider kan man räkna ut när kunden kommer att lämna systemet. Denna framtida händelse lägges till händelsemängden.

När en kund lämnar systemet finns två alternativ. Om kön är tom ökas antalet lediga kassor med 1. Annars påbörjas betjäning av den kund som står först i kön och den händelse som innebär att han lämnar systemet tillfogas händelsemängden.

Man kan tänka sig att generera alla ankomster och betjäningstider innan simuleringen påbörjas, men det är mer rationellt att generera framtida händelser så sent som möjligt under simuleringen. Detta minskar programmets minnesbehov utan att man förlorar något i effektivitet.

Det är lämpligt att representera mängden av framtida händelser med en prioritetskö där prioriteten ges av tidpunkten när händelsen skall inträffa. Vi behöver nämligen komma åt framtida händelser i ordning efter den tidpunkt när de skall inträffa.

Kön av väntande kunder representeras naturligtvis med en kö-klass.

Antalet lediga kassor och aktuell tid representeras med enkla variabler. De enda tidpunkter som är intressanta är de som infaller när det inträffar någon händelse. Information om dessa finns i prioritetskön. Det kan därför vara lämpligt att hålla reda på tiden i anslutning till prioritetskön och låta den aktuella tiden vara lika med tidpunkten för senast inträffade händelse.

För att enkelt kunna identifiera kunder i utskrifter från programmet skall kunderna numreras löpande från 0.

Gör en design för simuleringsprogrammet som ett klass-diagram.