Tentamen i Objektorienterad modellering och design (OMD) Helsingborg

EDAF25

30 maj 2018, 8 – 13

- Skriv bara på ena sidan av pappret – tentorna kommer att scannas in, och endast framsidorna rättas.
- Skriv **inte** med färgpenna – enda tillåtna färg är svart/blyerts.
- Skriv tydligt – om texten inte går att läsa kan du inte få några poäng.
- Sätt identitet och sidnummer på varje inlämnat blad, kontrollera att sidnumret på din sista sida är samma som det antal blad du markerar på omslagspappret.

Tentamen består av 4 uppgifter med totalt 30 poäng. För godkänt betyg kommer att krävas högst hälften av poängen. Vid bedömningen kommer hänsyn att tas till lösningens kvalitet. UML-diagram skall ritas i enlighet med UML-häftet.

Hjälpmedel:
- Andersson: UML–syntax
- Holm: Java snabbreferens

Uppgift 1 14 P.

Denna uppgift innehåller ett antal deluppgifter, där varje deluppgift har ett påstående och en anledning. För varje deluppgift, svara med ett av följande alternativ:

A Både påståendet och anledningen är korrekta uttalanden och anledningen förklarar påståendet på ett korrekt sätt.

B Både påståendet och anledningen är korrekta uttalanden, men anledningen förklarar inte påståendet.

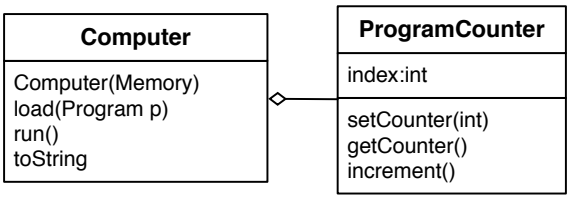
C Påståendet är ett korrekt uttalande, men anledningen är ett felaktigt uttalande.

D Påståendet är felaktigt, men anledningen är ett korrekt uttalande.

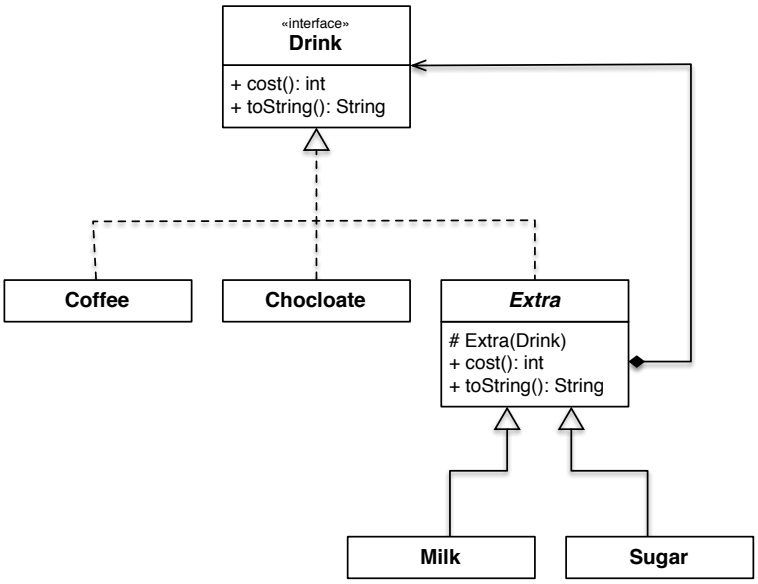
E Både påståendet och anledningen är felaktiga uttalanden.

Svara inte i uppgiftshäftet, utan skriv ditt svar på ett vanligt lösningspapper. Varje korrekt besvarad deluppgift ger 1.0 poäng. Vi vill inte att ni skall chansa, så ni får *minuspoäng* (-0.25 poäng) på en deluppgift om ni svarar fel, men aldrig minuspoäng totalt sett på hela uppgift 1.

	Påstående	Anledning
T1	Med hjälp av TDD förbättras kvaliteten på designen.	TDD är en testteknik.
T2	DRY handlar inte bara om att ta bort duplicerad information	DRY handlar också om att välja rimliga källor till informationen.
T3	Figur 1 visar ett exempel på brott mot DIP.	I exemplet har man separerat hög- och lågnivåansvar vad gäller datorns programexekvering.
T4	Figur 2 visar ett exempel på dekoratörsmönstret (Decorator).	Subklasserna `Milk` och `Sugar` i Figur 2 är specialiseringar av den abstrakta klassen `Extra`. Ytterligare tillbehör kan läggas till utan att någon förändring behöver göras i klassen `Extra`.
T5	Genom att kapsla in ett anrop i ett objekt kan man använda det för att t.ex. parametrisera andra objekt med olika anrop, logga, köa skapa undo.	Kompositmönstret låter användaren behandla hela samlingar och enskilda objekt på ett enhetligt sätt.
T6	Alla komponenter i en komposit måste implementera samma interface	Kompositmönstret kapslar in beteendet för olika tillstånd i separata klasser.
T7	Objektdiagram är inte särskilt användbara för att visa en sekvens av händelser.	Syftet med ett objektdiagram är att visa relationer mellan en uppsättning objekt.
T8	ISP säger att ett interface ska vara sådant att dess användare/klienter inte ska vara tvingade att bero på metoder som de inte använder.	Det omvända leder till brott mot SRP.
T9	Vill man ändra värde på ett attribut av en omuterbar typ måste man skapa ett nytt objekt.	En omuterbar klass är en klass vars instanser inte kan förändras.
T10	Klassen `SafeContainer` i Figur 3 är muterbar.	Klassen `SafeContainer` är final, dess attribut är privata och det finns inga metoder i klassen som förändrar attributen.
T11	Generalisering bör undvikas om man vill ha kod som är lätt att återanvända.	Generalisering bryter mot LSP
T12	Klassen `View` i Javakoden i Figur 4 måste tillhandahålla en observatörsiterator som kan anropas av `Switch` vid en förändring av dess tillstånd.	Javakoden i Figur 4 visar ett exempel på hur oberver-observable mönstret har tillämpats för att separera `Switch` från `View`.
T13	Tillståndsmönstret (State pattern) kan vara lämpligt att använda om ett objekts beteende beror på dess tillstånd och det därav behöver ändra betende dynamiskt.	Tillståndsmönstret gör koden lätt att förstå och underhålla.
T14	Javakoden i Figur 4 visar ett exempel på MVC mönstret där klassen `Switch` utgör modell och `Control` och `View` utgör kontroll respektive vy.	Klassen `View` har inget direkt beroende till klassen `Switch` och klassen Switch har inget direkt beroende till klassen `Control`. I stället finns indirekta kopplingar via lyssnare och observerramverket.



Figur 1: Teoriuppgift T3



Figur 2: Teoriuppgift T4

```
public final class SafeContainer {
    private final Object object;
    public SafeContainer(Object object) {
        this.object = object;
    }
    public String toString() {
        return object.toString();
    }
}
```

Figur 3: SafeContainer Teoriuppgift T12

```
public class Switch extends Observable {
    private boolean on = false;
    public void toggle() {
        on = !on;
        setChanged();
        notifyObservers();
    }
    public boolean isOn() {
        return on;
    }
}

public class Control implements ActionListener {
    private Switch switch_;
    public Control(Switch switch_, View view) {
        this.switch_ = switch_;
        view.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e) {
        switch_.toggle();
    }
}

public class View extends JButton implements Observer {
    private Switch switch_;
    public View(Switch switch_) {
        super("OFF");
        this.switch_ = switch_;
        switch_.addObserver(this);
    }
    public void update(Observable o, Object arg) {
        setLabel(switch_.isOn() ? "ON" : "OFF");
    }
}
```

Figur 4: MVC Teoriuppgift T14

Uppgift 2 6 P.

Följande kodexempel [1] är ett exempel på en stel design (Rigidity).

```
public class Account {
  private String type;
  private String accountNumber;
  private int amount;

  public Account(String type,String accountNumber,
                   int amount) {
    this.amount=amount;
    this.type=type;
    this.accountNumber=accountNumber;
  }

  public void debit(int debit) throws Exception {
    if(amount <= 500){
        throw new Exception("Mininum balance shuold be over
                                500");
    }
    amount = amount-debit;
    System.out.println("Now amount is" + amount);
  }

  public void transfer(Account from, Account to,
                       int creditAmount) throws Exception {
    if(from.amount <= 500){
        throw new Exception("Mininum balance shuold be over
                                500");
    }
    to.amount = amount+creditAmount;
  }

  public void sendWarningMessage(){
    if(amount <= 500){
        System.out.println("amount should be over 500");
    }
  }
}
```

a) Vad innebär det att en design är stel? och på vilket sätt är designen i exemplet stel?

[1] https://dzone.com/articles/code-smell-shot-surgery

b) En orsak till att en design börjar ruttna kan vara att grundläggande designprinciper inte följts. Nämn en grundläggande princip som har missats i exemplet? På vilket sätt?

c) På samma sätt kan konsekvensen av ruttnande kod vara att det inte går att följa grundläggande designprinciper. Ge exempel på ett scenario där en grundläggande designprincip kommer att brytas vid en vidareutveckling av koden i exemplet.

d) Föreslå en refaktorisering som åtgärdar problemet. Svara med text och/eller javakod.

Uppgift 3 6 P.

Nedan en implementering av sorteringsalgoritmen BubbleSort:

```
public class BubbleSorter {
    public static void sort(int[] intArray) {
        int n = intArray.length;

        for(int i=0; i < n; i++){
            for(int j=1; j < (n-i); j++){
                compareAndSwap(intArray, j-1){
            }
        }
    }

    private static void swap(int[] array, int index){
        int temp = array[index];
        array[index] = array[index+1];
        array[index+1] = temp;
    }

    private static void compareAndSwap(int[] array, int index){
        if(array[index] > array[index+1]){
            swap(array, index);
        }
    }
}
```

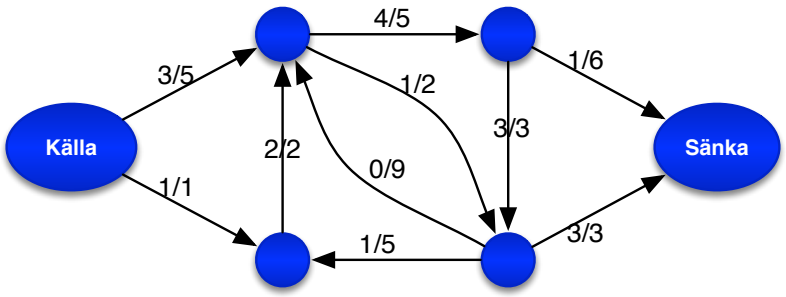
Klassen `BubbleSort` känner till hur man med hjälp av Bubble-sort algoritmen kan sortera en array av integers. De två hjälpmetoderna, tar hand om detaljerna kring vektorer och integers och tillhandahåller de mekanismer som algoritmen behöver.

a) Använd Mallmetodmönstret (*Template Method*) för att öppna upp för möjligheten att sortera olika slags objekt. Svara med en java-implementering av den uppdaterade klassen `BubbleSorter` och en specialisering `IntBubbleSorter`. Specialiseringen ska kapsla in representationen av såväl objekten (int) som behållaren (array).

b) Rita ett sekvensdiagram som visar anropet av metoden `sort(intArray)` i din lösning.

Uppgift 4 4 P.

Nedan visas ett nätverksflöde. Värdet x/y på bågarna ska tolkas som att bågen har flödet `x` och kapaciteten `y`.



a) Rita residualgrafen som associeras med nätverksflödet i figuren.

b) Exekvera en iteration av Ford-Fulkersons algoritm på nätverksflödet i figuren. Använd en heuristik som väljer den största utökande vägen i varje iteration. Visa tydligt den utökande vägen och värdet på varje båge före och efter ökningen.

Lösningar

Lösningsförslag till uppgift 1

T1 C TDD (Test driven development) är en utvecklingspraxis snarare än en testteknik. En effekt av denna praxis är att utvecklaren tvingas att se på koden från användarens perspektiv och utveckla testbar/anropsbar kod, d.v.s. modulen som ska testas behöver frikopplas från sin omgivning. Låg koppling är ett tecken på en god design.

T2 A, DRY- handlar inte bara om att ta bort duplicerad information det handlar också om att välja rimliga källor till informationen. Hur fördelar vi systemets funktionalitet på ett bra sätt? Varje bit information ska återfinnas på ett rimligt ställe.

T3 B båda påståenden är sanna, DIP (Dependency Inversion Principle) handlar dock om koppling (coupling) snarare än om sammanhang (cohesion). DIP säger att beroenden inte ska gå mot konkreta klasser utan mot abstraktioner. För att göra högnivåklassen `Computer` mindre beroende av lågnivåklassen `ProgramCounter` skulle man kunna införa en abstrakt klass som båda beror på. Det vill säga beroendet vänds (DIP) från den konkreta implementationen av `ProgramCounter` mot en abstraktion av dess ansvar.

T4 B Båda påståenden är sanna. Dock beskriver anledningen arv i generalla termer och förklarar inte påståendet. Signifikant för dekoratörsmönstret är kompositstrukturen och delegationen till ursprungsobjektet (i det här fallet kaffe eller choklad).

T5 B, Både påstående och anledning är korrekta uttalanden men orelaterade. Det ena beskriver kommandomönstret och det andra beskriver kompositmönstret.

T6 C, Alla komponenter i en komposit måste implementera samma interface. Kompositmönstret låter dig sätta samman objekt i trädstrukturer som representerar hierarkier. Individuella objekt och sammansättningar behandlas på ett enhetligt sätt. Anledningen är inte ett korrekt uttalande utan beskrivningen gäller tillståndsmönstret snarare än kompositmönstret.

T7 B, båda påståenden är sanna. Objektdiagram är inte särskilt användbara för att visa en sekvens av händelser och syftet med ett objektdiagram är att visa relationer mellan en uppsättning objekt. Förklaringen ligger dock inte där utan i vilken typ av relationer som visas. I ett objektdiagram visas en statisk ögonblicksbild av systemet och linjerna representerar länkar/kopplingar mellan objekten vid ett tillfälle (snarare än metodanrop).

T8 C ISP säger att ett interface ska vara sådant att dess användare/klienter inte ska vara tvingade att bero på metoder som de inte använder. Anledningen är att man vill undvika onödiga beroenden. Om en klient beror på en klass som innehåller metoder som klienten inte använder, men som

andra klienter använder kommer denna klient ändå att påverkas av förändringar som dessa andra klienter påtvingar klassen. Ett sätt att undvika sådana beroenden/kopplingar är att separera interfacen.

ISP liknar SRP men tar modulanvändarens(klientens) perspektiv snarare än moduldesignerns. En klass kan ha ett tydligt ansvar även om den implementerar flera interface. Ett exempel är Sheet i XL-projektet. Användaren i Expression-paketet behövde bara de metoder som Environment-interfacet tillhandahöll alltså behövdes inget särskilt Sheet-interface. Att ersätta Environment-interfacet med ett sådant skulle bryta mot ISP men inte nödvändigtvis mot SRP som handlar om Sheet-klassens ansvar i det här fallet (som i sin tur utgör ett gränssnitt gentemot GUIt ...).

T9 A, Ett omuterbart objekt är ett objekt vars tillstånd inte kan förändras. Vill man ändra värde på ett attribut av en omuterbar typ måste man skapa ett nytt objekt.

T10 B, Klassen `SafeContainer` är muterbar eftersom attributet `object` är muterbart.

T11 E, LSP säger att metoder som använder referenser till basklasser måste kunna använda instanser av dess sub-klasser utan att känna till det. Således kan arv/generalisering om man inte är försiktig leda till brott mot LSP. Samtidigt är arv/generalisering ett verktyg för att undvika brott mot andra viktiga principer som t.ex. OCP och DRY. LSP säger således snarare något om *hur* arv ska tillämpas än att det ska undvikas.

T12 D, Kodsnutten visar ett exempel på hur observable implementerats och men funktionalteten för att notifiera observatörerna behöver inte implementeras i View utan Switch ärver den från klassen Observable.

T13 C, Tillståndsmönstret kan vara lämpligt att använda om ett objekts beteende beror på dess tillstånd och det därav behöver ändra betende dynamiskt. Tillståndsmönstret (State pattern) föreslår att tillståndsspecifika beteenden frikopplas från kontexten och kapslas in i flera tillståndsklasser. Varje möjligt tillstånd som kontextobjektet kan befinna sig i kan mappas till en separat tillståndsklass. Viket ger flexibilititet utifrån ett tillståndsperspektiv.

Samtidigt leder det till många små klasser uppdelade utifrån en abstrakt modell (tillståndsdiagram) som gör koden svårare att förstå och underhålla. Alltså behövs en avvägning mellan behovet av flexibilitet och behovet av begriplighet.

T14 C, Enligt MVC mönstret ska modellen (i det här fallet klassen `Switch`) inte bero på varken kontrollmodulen eller vymodulen. Däremot har View ett beroende till modellen `Switch`.

Lösningsförslag till uppgift 2

a) Stelhet innebär att det svårt att göra ändringar. En design är stel om en enda ändring innebär att en rad ändringar behöver göras i flera moduler. I exemplet kan man se varje metod som en modul. Här görs samma validering (kontrollen att beloppet överstiger 500) i alla metoder. För att lägga till eller ändra kriterier för validering behöver alla metoder skrivas om.

b) I exemplet har man missat att följa OCP med avseende på förändrade valideringskriterier, på samma sätt har man missat att fördela ansvar (SRP) mellan metoderna i klassen.

c) En utökning av valideringslogiken kommer att leda till duplicerad kod, brott mot DRY.

d) Lägg till en metod `isAccountValid()` som tar ansvar för valideringen.

Lösningsförslag till uppgift 3

a)

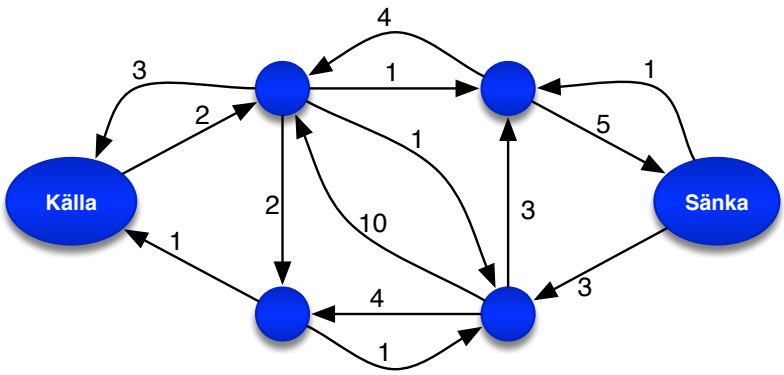
```
public abstract class BubbleSorter {
    protected int length =0;

    protected void doSort() {
        int n = length;
        for(int i=0; i < n; i++){
            for(int j=1; j < (n-i); j++){
                compareAndSwap(j-1)
            }
        }
    }
    private void compareAndSwap(int index){
        if(outOfOrder(index)){
            swap(index);
        }

    protected abstract void swap(int index);
    protected abstract boolean outOfOrder(index);
}
```

```
public class IntBubbleSorter extends BubbleSorter {
    private int[] array = null;
    public void sort(int[] intArray) {
        array = intArray;
        length = array.length;
        doSort();
    }

    protected void swap(int index){
        int temp = array[index];
        array[index] = array[index+1];
        array[index+1] = temp;
    }

    protected boolean outOfOrder(index){
        return (array[index] > array[index+1]);
    }

}
```

Exemplet är hämtat från Martin kap 14.

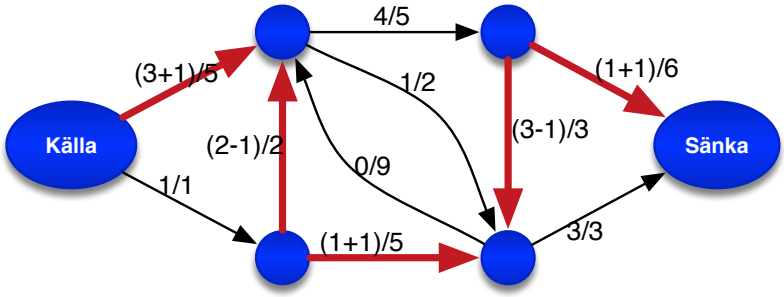
Lösningsförslag till uppgift 4

a)



(2p)

b)



(2p)