Tentamen i Objektorienterad modellering och design

Vid bedömningen kommer hänsyn att tas till lösningens kvalitet. UML-diagram skall ritas i enlighet med UML-häftet. Man får förutsätta att det finns standardkonstruerare i alla klasser. De behöver ej redovisas i lösningar.

Hjälpmedel: Martin: Agile Software Development
Andersson: UML–syntax
Holm: Java snabbreferens

---

1 Denna uppgift innehåller uppgifter med påståenden och anledningar. För varje uppgift 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.

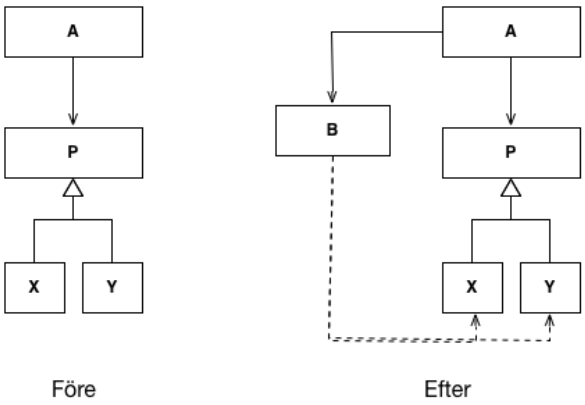
C Påståendet är ett korrekt uttalande, men anledningen är falsk.

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

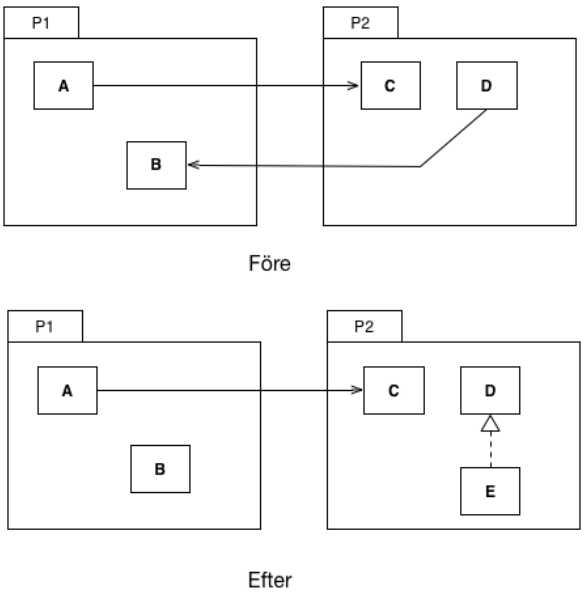
E Både påståendet och anledningen är falska.

Det går bra att svara direkt i formuläret. **Glöm då inte att lämna in formuläret tillsammans med övriga lösningar.** (4p)

	Påstående	**Anledning**	**Svar A,B,C, D,E**
T1	Strategy är ett bra mönster då man vill kunna ändra beteendet hos ett objekt under exekvering	Det är lätt att skapa en ny strategi genom att skapa en klass som implementerar strategi-gränssnittet	
T2	Figur 1 visar hur OCP-principen tillämpats.	Figur 1 visar ett exempel på Factory-mönstret	
T3	Figur 2 nedan visar hur ett cirkulärt beroende mellan två paket har tagits bort	DIP kan användas för att ta bort cirkulärt beroende mellan två paket	
T4	Ett stabilt paket är ett paket som inte så många andra paket har beroende till	Ett paket som inte så många andra paket beror av, kan man ändra i utan att ändringarna får så stor spridning	

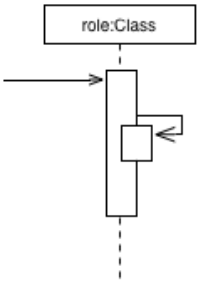


Figur 1: Figur till T2



Figur 2: Figur till T3

2 Vilket mönster illustrerar nedanstående sekvensdiagram bäst (av de mönster vi gått igenom i kursen)? (1p)



Figur 3: Sekvensdiagram

3 Ett klientprogram enligt nedan använder sig av en kommunikationskanal för att skicka och ta emot textsträngar.

```
public class Client {
    ComChannel myCC = new FastComChannel();

    private void send(String text) {
        myCC.send(text);
    }
    private String receive() {
        return myCC.receive();
    }
    public void startLogg(DB loggDB) {
        myCC = new LoggedComChannel(loggDB, myCC);
    }
    public void stopLogg() {
        if (myCC instanceof LoggedComChannel) {
            myCC = ((LoggedComChannel)myCC).getOrigCC();
        }
    }
}
public interface ComChannel {
    public void send(String text);
    public String receive();
}
public class FastComChannel implements ComChannel {

    public void send(String text) {
        // omissions
    }
    public String receive() {
        String text = null;
        // omissions
        return text;
    }
}
public class LoggedComChannel implements ComChannel {
    ComChannel origCC;
    DB loggDB;
    public LoggedComChannel(DB db, ComChannel origCC) {
        this.origCC = origCC;
        this.loggDB = db;
    }
    public ComChannel getOrigCC() {
        return origCC;
    }
    public void send(String text) {
        loggDB.appendSent(text);
        origCC.send(text);
    }
    public String receive() {
        String text;
        text = origCC.receive();
        loggDB.appendReceived(text);
        return text;
    }
}
```

a. De har gjort sin design sådan att de enkelt kan slå på och av loggning av trafiken. Vad heter det mönster de har använt för att uppnå detta?

b. Rita ett klassdiagram för koden, där du får med all information du känner till.

c. Rita ett sekvensdiagram för följande kodrader :

```
myApp.send("Jag");
myApp.startLogg(myDB);
myApp.send("kan");
myApp.stopLogg();
myApp.send("detta");
```

Du kan förutsätta att myApp och myDB är deklarerade enligt nedan och sedan skapade och initialiserade på rätt sätt.

```
Application myApp;
DB myDB;
```

(5p)

4 Vi vill ha en klass `Stock` och en klass `Portfolio`, enligt beskrivningen nedan.

- Klassen `Stock` skall hålla reda på värdet av en given aktie, dess 'update'-metod kommer att anropas av någon annan så snart aktiens värde ändras. Varje gång aktiens värde ändras skall den meddela sina observatörer att den ändrats (`Stock` skall vara en `Observable`).
- Klassen `Portfolio` skall hantera en aktieportfölj (dvs ett antal aktier), och hålla reda på hur värdet på portföljen har utvecklats sedan vi lade till aktierna (utvecklingen, *current yield*, är summan av aktuella kurser delat med summan av inköpskurserna).

  `Portfolio` skall vara en `Observable`, och meddela sina observatörer varje gång värdet på en ingående aktie ändras på ett sådant sätt att portföljens utveckling understiger en given gräns (*yield limit* i specifikationen nedan).

Klasserna skall ha följande specifikationer:

```
class Stock extends Observable {

   public void update(double value) // ... din kod ...
   public double value()            // ... din kod ...
}

class Portfolio ...din kod... {

   public Portfolio(double yieldLimit) // ... din kod ...
   public void add(Stock stock)        // ... din kod ...
   public double currentYield()        // ... din kod ...
}
```

a. Implementera `Stock` och `Portfolio` enligt beskrivningen ovan.

b. `Portfolio` har nu två uppgifter, både att räkna samman, och att varna – flytta ansvaret för att varna till en ny klass (observera att detta innebär att `Portfolio` nu bör meddela när dess värde ändrats, oavsett om dess värde är för lågt eller ej). Redovisa med Javakod för den nya klassen och förändringarna i Portfolio-klassen.

(5p)

5 Vi har ett interface `Drawing` (se nedan), och vill kunna rita olika slags figurer i ett `Drawing`-objekt.

```
interface Drawing {

   void clear();
   void useForegroundColor();
   void useBackgroundColor();
   void moveTo(int x, int y);
   void circle(int radius);
   void rectangle(int width, int height);
}
```

I ett `Drawing`-objekt finns alltid en aktuell punkt, och vi flyttar den med `moveTo`. Metoden `circle` ritar en cirkel med given radie runt den aktuella punkten, `rectangle` ritar en rektangel med given storlek med den aktuella punkten *som mittpunkt*.

För att rita vill vi använda *Command Pattern*, och vi vill både kunna rita och ångra figurer (för att 'ångra' en figur ritar vi den i bakgrundsfärg). Så vi har följande interface:

```
interface DrawCommand {

   void draw(Drawing d);
   void undo();
}
```

a. Implementera klasserna `DrawCircle` och `DrawSquare` som ritar (och ångrar) cirklar och kvadrater. Man skall kunna skapa cirkel- och kvadratritningsobjekt genom att skriva:

```
void example(Drawing d) {
    DrawCommand c = new DrawCircle(0, 0, 1);
    DrawCommand s = new DrawSquare(0, 0, 2);
    c.draw(d);
    // ... woops, take back:
    c.undo();
}
```

Parametrarna till konstruktorerna i `DrawCircle` och `DrawSquare` är i tur och ordning x- och y-koordinaterna för mittpunkten, och radie eller sidlängd.

Använd lämpliga principer och mönster från kursen när du skriver din kod.

b. Rita ett fullständigt klassdiagram med klasserna i din lösning. Attribut behöver dock inte vara med.

(5p)

6 Vi vill nu koppla våra ritklasser i föregående problem till ett GUI, och vill ha ett antal `Action`-klasser, som beskriver användarens interaktion med programmet:

```
interface Action {
   void execute(Drawing d, Stack<DrawCommand> history);
}
```

Här är `history` en stack med alla de ritkommandon som har utförts (vi behöver dem om vi skall kunna ångra dem).

Klassen `Stack` har specifikationen

```
class Stack<E> {

   public boolean isEmpty();
   public void push(E value);
   public E pop();
}
```

Vi kommer i uppgiften att hantera tre slags `Action`-klasser:

- `DrawAction`: ritar en figur – man skickar med ett `DrawCommand` när man skapar en `DrawAction`, så konstrueraren har rubriken:

  ```
  public DrawAction(DrawCommand cmd)
  ```

- `Undo`: ångrar den senaste utritade figuren. Man skall kunna ångra hur många utritade figurer som helst, så efter att vi ångrat en figur räknas den figur vi ritade precis före den borttagna som senaste figur.
- `Exit`: avslutar programmet (med `System.exit(0)`).

Vi har även ett interface `GUI`, med en metod `next()` som ligger och tolkar det användaren gör, och skickar tillbaka nästa 'action':

```
interface GUI {

   Action next();
}
```

En kollega kommer att skriva en `GUI`-klass, och hon känner till hur våra `Action`-objekt skapas.

a. Implementera de tre `Action`-subklasserna ovan.

b. Skriv en metod med rubriken:

```
void run(GUI gui, Drawing d) {
    // ... här skall du fylla i din kod ...
}
```

Den skall hämta kommandon från GUI och uppdatera ritningen enligt användarens önskemål.

(5p)