Tentamen i EDAF25

21 augusti 2017

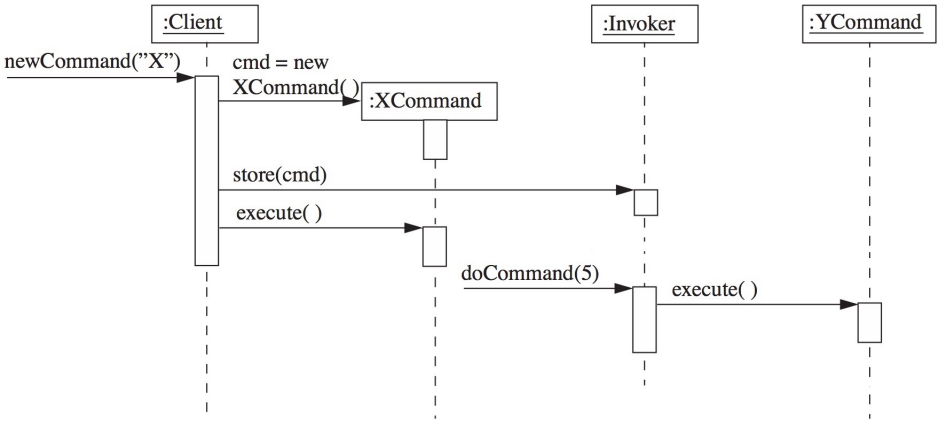
Skrivtid: 8-13

- Skriv bara på ena sidan av pappret – tentorna kommer att skannas 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.

Hjälpmedel:
- Koffman & Wolfgang: *Data Structures*
- Martin: *Agile Software Development*
- Andersson: UML–syntax
- Holm: Java snabbreferens

Uppgift 1

I ett program kan man ge tre kommandon: X, Y och Z. Ett kommando beskrivs av en klass som innehåller en metod execute( ) som utför kommandot. Kommandon lagras efter hand som de utförs, och man kan senare ge numret på ett lagrat kommando som skall utföras igen (1 motsvarar det senast utförda kommandot, 2 det näst senaste, osv). Detta beskrivs i sekvensdiagrammet i figur 1 (först lagringen och utförandet av ett X-kommando, sedan utförandet av det lagrade kommandot med nummer 5 som råkar vara ett Y-kommando i detta fall):



Figur 1: Sekvensdiagram. Om parametern till metoden newCommand är "Y" eller "Z" så skapas naturligtvis ett Y- eller Z-kommando i stället för det X-kommando som visas i diagrammet.

Skriv ett antal Javaklasser där all information som man kan utläsa ur den inledande beskrivningen och ur sekvensdiagrammet finns med! Använd designmönstret Command i din lösning. Du kan anta att parametern till metoden doCommand() alltid har ett giltigt värde (dvs motsvarar ett existerande tidigare kommando).

3 p

Uppgift 2

Följande klass agerar både modell, vy och kontroll (control).

```
public class Switch extends JButton implements ActionListener {
  private boolean on = false;

  public Switch() {
    super("OFF");
    addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    on = !on;
    setLabel(on ? "ON" : "OFF");
  }
}
```

Gör om designen så att MVC-mönstret används med tre separata klasser med användning av *Observer*-ramverket.

(a) Lösningen redovisas med Java-kod och skall visa hur och var följande metoder används.

```
public interface Observer {
  public void update(Observable observable, Object object);
}

public abstract class Observable {
        public void addObserver(Observer observer)
        protected void setChanged()
        public void notifyObservers()
}
```

(b) Rita ett fullständigt klassdiagram av lösningen där dina tre klasser finns med, samt klassen Observable och gränssnittet Observer.

6 p

Uppgift 3

I ett klassbibliotek finns det en abstrakt klass för att representera ljudkortet i en dator (endast för uppgiften relevanta delar visas nedan):

```
public abstract class SoundCard {
    abstract String getVersion();
    abstract void playFile(String fileName);
    ...
}
```

Vidare finns i det givna klassbiblioteket en klass Hardware som kan användas för att få information om den installerade hårdvaran i den aktuella datorn. I denna finns bland annat en statisk operation för att skapa ett objekt som representerar ljudkortet i datorn. Beroende på vilket ljudkort som är installerat returneras ett objekt av en passande subklass till SoundCard eller null om inget ljudkort är installerat:

```
public class Hardware {
    SoundCard getSoundCard();
    ...
}
```

Vi vill använda detta bibliotek i ett program för att spela upp ljud om ett ljudkort är installerat. I annat fall kör programmet under tystnad. Ett utdrag ur ett sådan program skulle kunna se ut enligt följande kodexempel:

```
// Ask library hardware class for sound card reference or null if soundcard not installed.
SoundCard card = Hardware.getSoundCard();
...
// Print sound card version
if (card == null) {
    System.out.println("Sound card not available!");
} else {
    System.out.println(card.getVersion());
}
...
// Play sound file if sound card present
if (card != null) {
    card.playFile("mysoundfile.snd");
}
```

Som synes leder detta till att man behöver göra många kontroller för null i sitt program. Lös problemet med hjälp av NULL-mönstret. Visa din lösning genom att beskriva med javakod vilka *ändringar* du gör av kodexemplet ovan.

2 p

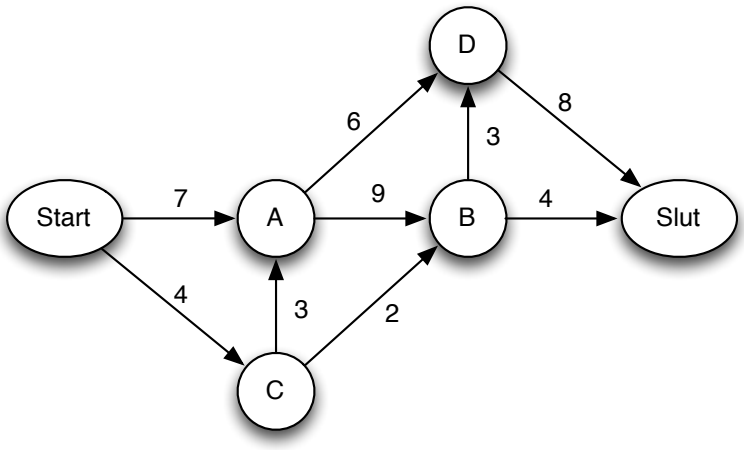
Uppgift 4

Grafen i figur 2 visar kapaciteten i ledningarna i ett flödesnätverk. *4 p*

(a) Ange det maximala flödet från start- till slutnod i nätverket.

(b) Namnge och beskriv kortfattat de övergripande stegen i algoritmen du använder.

(c) Visa också hur nätverket i figur 2 kan representeras med hjälp av en matris.



Figur 2: Flödesnätverk

Uppgift 5

Följande text är klippt från wikipedia där den används som ett exempel på när mönstret `Decorator` är lämpligt att använda:

> As an example, consider a window in a windowing system. To allow scrolling of the window's contents, one may wish to add scrollbars to it. Assume windows are represented by instances of the Window class, and assume this class has no functionality for adding scrollbars. In order to solve this problem, one could create a subclass ScrollingWindow that provides them.
>
> Now, assume one also desires the ability to add borders to windows. Again, the original Window class has no support. The ScrollingWindow subclass now poses a problem, because it has effectively created a new kind of window. If one wishes to add border support to many but not all windows, one must create subclasses WindowWithBorder and ScrollingWindowWithBorder etc. This problem gets worse with every new feature or window subtype to be added.

(a) Ovanstående problem kan som sagt lösas med hjälp av mönstret `Decorator` på ett bättre sätt vilket undviker den exponentiella ökningen av antalet nödvändiga klasser. Rita upp ett klassdiagram för `Decorator`-lösningen av problemet med möjlighet till `scrolling` och/eller `border`. Du behöver inte ta med några operationer eller attribut i klassdiagrammet – bara klasser och deras inbördes relationer.

(b) Rita ett objektdiagram för ett fönster dekorerat med både `scrolling` och `border`.

5 p

Uppgift 6

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.

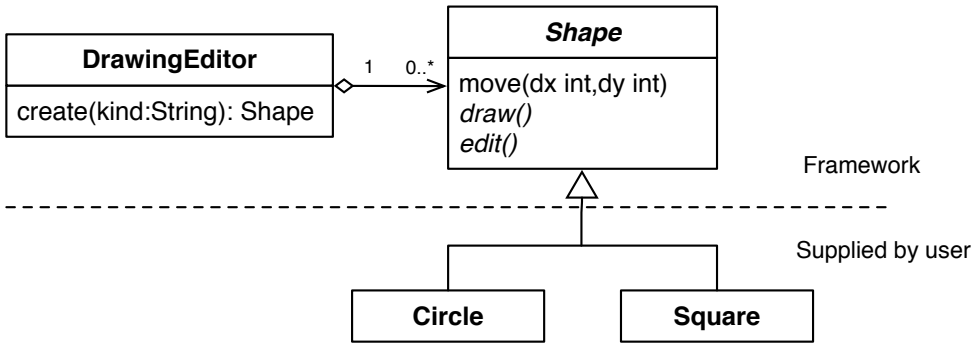
D Påståendet är felaktigt, men anledningen är ett korrekt uttalande.

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

	Påstående	**Anledning**
T1	Vid paketdesign bör man ha i åtanke att det vid återanvändning är användaren av paketet som kommer att underhålla och uppdatera paketet.	Användaren bestämmer själv när en ny utgåva av ett paket ska integreras i det egna systemet.
T2	Genom att tillämpa Mallmetod-mönstret (Template method) bryter man mot DRY principen.	I Mallmetodmönstret implementeras samma mallmetod i flera subklasser.
T3	Med hjälp av fabriksmetod-mönstret kan man lösa problemet med ramverket i figur 3 så att användaren kan lägga till sina egna specialiseringar av `Shape`.	Fabriksmetod-mönstret innebär att `DrawingEditor` delegerar instansieringen av de olika `Shape`-objekten till en abstrakt Fabriksklass.
T4	Designen av klassen `Figure` i figur 4 följer OCP mad avseende på olika typer av former.	Det är möjligt att lägga till komponenter av vilken typ som helst till Figure.
T5	Klassen Figure i figur 4 bryter mot SRP	Klassen `Object` som beskriver komponenterna i en `Figure` bryter mot SRP
T6	Med hjälp av strategimönstret skulle man i exemplet i figur 4 kunna öppna för förändringar i fler dimensioner, t.ex. olika färger eller filter. Dessa kan dessutom variera under livstiden för ett `Figure`-objekt.	Genom att tillämpa strategimönstret definierar man variationspunkter (familjer av algoritmer) och kapslar in olika implementationer av dem i separata utbytbara objekt.
T7	Klassen `Macro` i figur 5 visar ett exempel på hur kompositmönstret kan implementeras.	Klassen `Macro` kapslar in själva anropet av ett kommando. På så sätt kan man separera konstruktion (beslutet om vad som ska hända) och exekvering (själva händelsen) av anropet åt.
T8	Klassen `Macro` i figur 5 är muterbar	Klassen `ArrayList` är muterbar
T9	Arv bör undvikas om man vill ha kod som är lätt att återanvända.	Arv kan leda till brott mot LSP
T10	Tillståndsmönstret (State pattern) är en variant på strategimönstret.	Tillståndsmönstret används för att kapsla in varierande beteende för ett och samma objekt.

10 p



Figur 3: Ramverk DrawingEditor

```
/* Nedan beskrivs en klass Figure. En Figure är en lista med komponenter av typen Object.
 Figure har en metod draw() som ritar figuren genom att iterera över listan med
 komponenter och ritar dessa. */

public class Figure extends ArrayList<Object>{

    public void draw(){
        for (Object object: this){
            if (object instanceof Square){
                drawSquare((Square) object)
            } else if{
                drawCircle((Circle) object)
            }
        }
    }
}
```

Figur 4: Figure

```
public class Macro extends ArrayList<Command> implements Command{

    public void execute(){
        for (Command command : this){
            command.execute();
        }
    }
}
```

Figur 5: Macro