TENTAMEN: Objektorienterade applikationer

Läs detta!

- *Uppgifterna är inte avsiktligt ordnade efter svårighetsgrad.*
- Börja varje uppgift på ett nytt blad.
- Skriv din tentamenskod på varje blad (så att vi inte slarvar bort dem).
- **Skriv rent dina svar. Oläsliga svar r ä t t a s e j!**
- Programmen skall skrivas i Java 5, eller senare version, vara indenterade, renskrivna och i övrigt vara utformade enligt de principer som lärts ut i kursen.
- Onödigt komplicerade lösningar ger poängavdrag.
- Programkod som finns i tentamenstesen behöver ej upprepas.
- Givna deklarationer, parameterlistor, etc. får ej ändras, såvida inte annat sägs i uppgiften.
- Läs igenom tentamenstesen och förbered ev. frågor.

I en uppgift som består av flera delar får du använda dig av funktioner klasser etc. från tidigare deluppgifter, även om du inte löst dessa.

Lycka till!

Uppgift 1

Gränssnittet `Die` beskriver tärningar. Man kastar tärningen med `roll` och läser av värdet med `getValue`:

```
public interface Die {
    void roll();
    int getValue();
}
```

Gränssnittet implementeras av klassen `StandardDie` som ger en sexsidig tärning med statistiskt jämn fördelning av utfallen. Exakt hur denna klass är konstruerad saknar relevans för uppgiften så vi lämnar dess interna detaljer därhän.

a)
Konstruera klassen `NoRepeatDie` som ger en tärning som aldrig ger samma utfall två gånger i följd. Lösningen skall utformas i enlighet med designmönstret *Decorator*. Nedanstående exempel visar hur de tre inblandade klasserna skall kunna användas:

```
Die nrd = new NoRepeatDie(new StandardDie());
int noOfDuplicates = 0;
int last = nrd.getValue();
for ( int i = 0; i < 1000000000; i++ ) {
    nrd.roll();
    if ( nrd.getValue() == last )
        noOfDuplicates++;
    last = nrd.getValue();
}
System.out.println("Number of duplicates: " + noOfDuplicates);
```

Om uppgiften är korrekt löst så skall utskriften bli

```
Number of duplicates: 0
```

när ovanstående kod exekveras.

(4 p)

b)
Rita ett klassdiagram som beskriver designmönstret *Decorator*.

(2 p)

Uppgift 2

Konfigurationsdata för program består ofta i nyckel-värdepar och lagras i CSV-filer (Colon Separated Values). Ex. Textfilen `config.txt` lagrar information om ett bildelement:

```
color:blue
height:100
width:200
```

Nycklarna är unika och internt är det lämpligt att lagra sådan information i en tabell (map).

a)
Konstruera en klass för att läsa och skriva konfigurationsdata från(till) en CSV-fil:

```
public class ConfigIO {
   public static Map<String,String> load(String fileName)
   throws IOException { // ToDo }

   public static void save(Map<String,String> table,String fileName)
   throws IOException { // ToDo }
}
```

Du får anta att nycklarna i CSV-filens vänstra kolumn är unika.

(6 p)

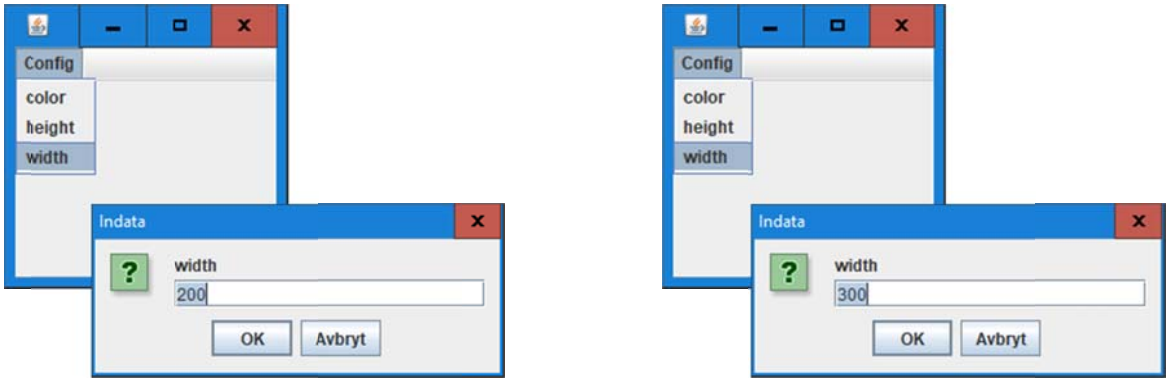
b)
Konstruera en klass som ger en intern representation av en CSV-fil. Data skall läsas och skrivas m.h.a. klassen i a). Filen skall läsas in till tabellen i klassens konstruktor och skrivas över så snart ett värde ändras i tabellen. Metoden `getKeys` returnerar nycklarna och används i en senare uppgift.

```
public class ConfigData {
   ...
   public ConfigData() throws IOException { // ToDO }

   public Set<String> getKeys() { // ToDO }

   public String getValue(String key) { // ToDO }

   public void setValue(String key,String value) throws IOException
   { // ToDO }
}
```

(8 p)

Uppgift 3

I den här uppgiften skall vi addera en konfigurationsmeny till ett fönster. När man väljer en viss parameter i menyn skall dess aktuella värde visas i en popup-dialog. Ändrar man värdet skall det nya värdet sparas. I följande scenario får användaren veta att `with` har värdet `200` och ändrar det till `300`. Nästa gång `with` väljs (bilden till höger) ser vi att det sparade värdet visas.



Konstruera det grafiska gränssnittet som visas ovan, inklusive fönstret. Menyelementen skall hämtas från `ConfigData` i uppgift 2b, och övrig datahantering skall också ske med den klassen. Om detta ger upphov till undantag skall orsaken visas i en popup-dialog. Du får alltså i denna uppgift anta att 2b är löst. *Tips:* Nedan finns ett tillägg till API.

(10 p)

`class javax.swing.JOptionPane`:

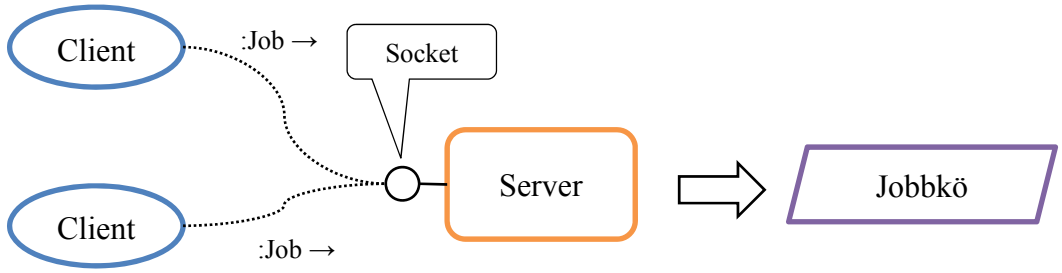
Denna metod saknas i vissa av de utdelade bilagorna:

```
static String showInputDialog(Object message,
    Object initialSelectionValue)
```

Shows a question-message dialog requesting input from the user, with the input value initialized to `initialSelectionValue`.

Uppgift 4

I nedanstående system sänder klienter jobb till en server. Servern lagrar varje mottaget jobb i en synkroniserad kö.



. . .

Uppgiften går ut på att konstruera klasser för klient och server. Javas stöd för client/server-kommunikation skall användas. Varje klientanslutning hanteras i en egen tråd i servern. Följande färdiga klasser skall användas i lösningen:

```
public class Job {
    ...
    public Job(String id,long time,String message) { ... }
    ...
    // Other contents omitted
}

public class SynchronizedQueue {
    ...
    public synchronized void put(Job j) { ... }
    public synchronized Job take() { ... }
}
```

a) Klientklassen skall ha en metod som skickar `Job`-objekt till en server.

```
public static void sendJob(String address,int port,Job job)
throws IOException
```

Implementera metoden (men inte resten av klassen). Kan `Job`-objekt skickas över en c/s-förbindelse? Om du anser att något saknas i klassdeklarationen så komplettera den.

(4 p)

b) Konstruera en serverklass som tar emot uppkopplingsförsök från klienter. För varje klient skall en tråd av typen `ClientHandler` skapas. Denna konstrueras i nästa deluppgift, men du får anta att den finns nu.

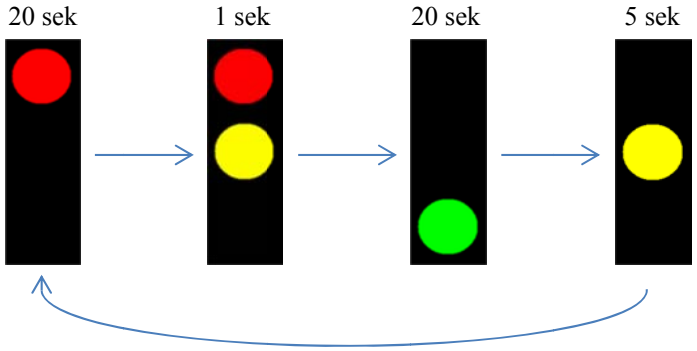
(5 p)

c) Konstruera trådklassen `ClientHandler` som oupphörligt tar emot `Job`-objekt via en socket. Varje mottaget objekt skall sparas i en kö av typen `SynchronizedQueue`. Låt trådklassens konstruktor ta socketen och kön som parametrar.

(6 p)

Uppgift 5

Signalväxlingarna i ett trafikljus sker i princip enligt följande cykliska schema (rött överst, gult i mitten, grönt nederst):



I denna uppgift skall vi simulera ett sådant trafikljus och följande kod är given. Signalklasserna används för att hålla reda på trafikljusets interna tillstånd:

```
public interface Signal {
    void goOn();
    void goOff();
    boolean isOn();
}
```

```
public class RedSignal
extends AbstractSignal {}

public class YellowSignal
extends AbstractSignal {}

public class GreenSignal
extends AbstractSignal {}
```

Klasserna är avsiktligt tomma.

```
public abstract class AbstractSignal
implements Signal {
    private boolean isOn = false;
    public void goOn() {
        if ( ! isOn ) {
            isOn = true;
        }
    }
    public void goOff() {
        if ( isOn ) {
            isOn = false;
        }
    }
    public boolean isOn() {
        return isOn;
    }
}
```

För det grafiska gränssnittet finns klasserna `Lamp` och `SignalGui`:

```
public class Lamp extends JLabel {
    private Icon lampIcon;
    public Lamp(Icon lampIcon) {
        this.lampIcon = lampIcon;
        switchOnOff(false);
    }
    public void switchOnOff(boolean turnOn) {
        if ( turnOn )
            setIcon(lampIcon);
        else
           setIcon(new ImageIcon("black light.png"));
    }
}
```

Forts. nästa sida

GUI

```
public class SignalGui extends JFrame {
    private Lamp redLamp;
    private Lamp yellowLamp;
    private Lamp greenLamp;
    public SignalGui() {
        setLayout(new GridLayout(3,1));
        redLamp = new Lamp(new ImageIcon("red light.png"));
        add(redLamp);
        yellowLamp = new Lamp(new ImageIcon("yellow light.png"));
        add(yellowLamp);
        greenLamp = new Lamp(new ImageIcon("green light.png"));
        add(greenLamp);
        pack();
        setVisible(true);
    }
}
```

a) Ange vilka tillägg som behöver göras ovan för att upprätta ett observatörsförhållande enligt designmönstret *Observer* mellan det grafiska gränssnittet och signalklasserna. Tänk efter vilken eller vilka signalklasser som berörs! Du behöver inte upprepa den givna koden.

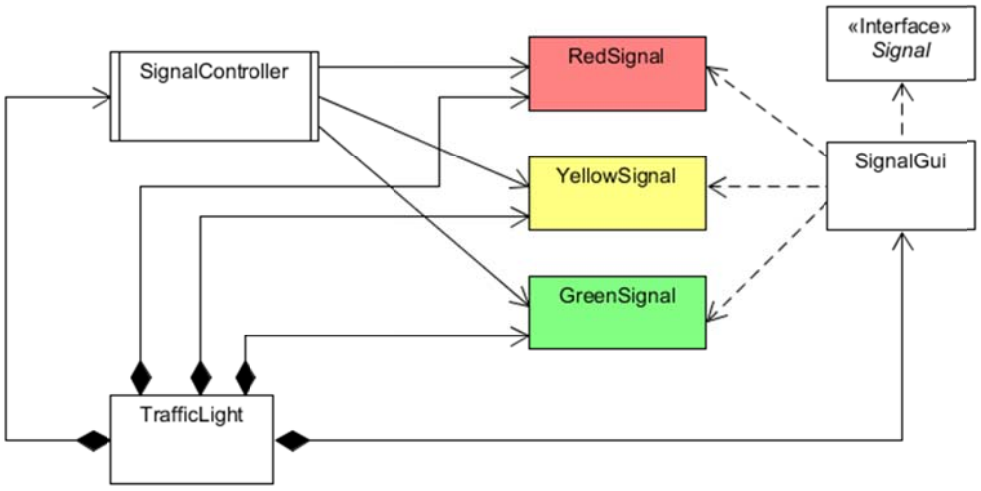
(3 p)

b) Konstruera klassen `SignalController`. Den skall styra tre signalobjekt enligt det cykliska schemat ovan. Signalcykeln skall starta med ett rött intervall. Klassen skall exekvera signalloopen i en egen tråd. Signalobjekten skall ges som argument till konstruktorn:

```
public SignalController(Signal greenSignal,
                        Signal yellowSignal,
                        Signal redSignal)
```

(7 p)

c) Konstruera klassen `TrafficLight` som implementerar ett trafikljus m.h.a. klasserna i a och b. Designmönstret *Observer* skall beaktas. Klassen skall ha en `main`-metod och starta tråden. Som ledning får du följande klassdiagram:



(5 p)