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

a) Rita ett klassdiagram som beskriver designmönstret composite.

(2 p)

b) Om man vill använda en slumptalsgenerator i en applikation och vill vara säker på att det bara finns en enda instans av generatorn kan man använda designmönstret singleton. Implementera klassen `SingletonRandom` enligt denna princip. Utnyttja `java.util.Random` i lösningen. Alla metoder i `Random` skall kunna anropas även i din nya klass.

(3 p)

Uppgift 2

I ett program för att simulera kast med en sexsidig tärning (*eng. die*) finns följande klass: [1]

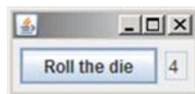
```
public class Die {
    private int value;
    private static Random random = new Random();

    public void roll() {
        value = 1 + random.nextInt(6);
    }
}
```

Eftersom man vill kunna se och "kasta" tärningen finns också ett grafiskt gränssnitt:

```
public class Gui extends JFrame {
    private JTextField textField;
    public Gui() {
        setLayout(new FlowLayout());
        JButton button = new JButton("Roll the die");
        add(button);
        textField = new JTextField(2);
        textField.setEditable(false);
        add(textField);
        pack();
        setVisible(true);
    }
}
```

Tanken är att fönstret skall se ut så här:



Ett tryck på knappen skall kasta tärningen och det nya värdet visas i textfältet. Tyvärr fungerar det inte (ännu) eftersom klasserna inte samarbetar. Modifiera koden ovan så att de två klasserna kommunicerar via designmönstret observer. Skriv också en `main`-metod som skapar och kopplar ihop objekt på lämpligt sätt till ett fungerande program. När det startas och fönstret visas skall tärningen vara kastad en gång och värdet visas, alltså innan det första knapptrycket. Inga ytterligare metoder får läggas till i `Die`. (6 p)

---

[1] Man skulle naturligtvis mycket väl kunna använda klassen `SingletonRandom` från uppgift 1 i klassen `Die`, men det saknar relevans för just denna uppgift.

Uppgift 3

En `SkipIterator` är en iterator som förutom de vanliga metoderna `hasNext`, `next` och `remove`, dessutom har metoden

```
public E skip(int n)
```

Ett exempel får visa hur det fungerar:

```
ArrayList<Integer> l = new ArrayList<>();
for ( int i = 1; i <= 10; i++ )
   l.add(i);

SkipIterator<Integer> skipit = new SkipIterator<>(l.iterator());
while ( skipit.hasNext() )
    System.out.print(skipit.next() + " ");

skipit = new SkipIterator<>(l.iterator());
while ( skipit.hasNext() )
    System.out.print(skipit.skip(2) + " ");
```

Den första utskriften blir `1 2 3 4 5 6 7 8 9 10` och den andra `3 6 9 10`. Anropet `skip(`*n*`)` hoppar alltså över *n* element och returnerar nästa element. Om det inte finns så många element returneras det sista elementet. Om iteratorn redan passerat sista elementet är beteendet samma som för metoden `next`, d.v.s. `NoSuchElementException` kastas. Anropet `skip(0)` är ekvivalent med anropet `next()`. Konstruera klassen `SkipIterator` genom att tillämpa designmönstret decorator. Gränssnittet `Iterator` definieras

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Tips: Om `Cls` skall implementera `Int<E>` skriver man `Cls<E> implements Int<E>`.

(10 p)

Uppgift 4

I den här uppgiften skall du konstruera ett Java-program, `Search`, som låter användaren söka efter alla Java-filer med ett visst textinnehåll i en angiven filkatalog. För de Java-filer som innehåller den sökta texten skall först filnamnet skrivas ut och därefter alla rader som innehåller den sökta texten. Ex. Om filerna i katalogen `test` är

A.java:
```
public interface A {
    void acc();
    long get();
}
```

B.java:
```
public class B implements A {
    private long sum = 0;
    private long term = 1;
    public void acc() {
        sum += term; term++;
    }
    public long get() {
        return sum;
    }
}
```

C.java:
```
public class C implements A {
    private long prod = 1;
    private long factor = 1;
    public void acc() {
        prod *= factor; factor++;
    }
    public long get() { return prod; }
}
```

D.java:
```
public class D implements A {
    protected A a;
    public D(A a) { this.a = a; }
    public void acc() { ... }
    public long get() { return ... }
}
```

och programmet körs med kommandot `java Search test private` så skall utskriften bli:

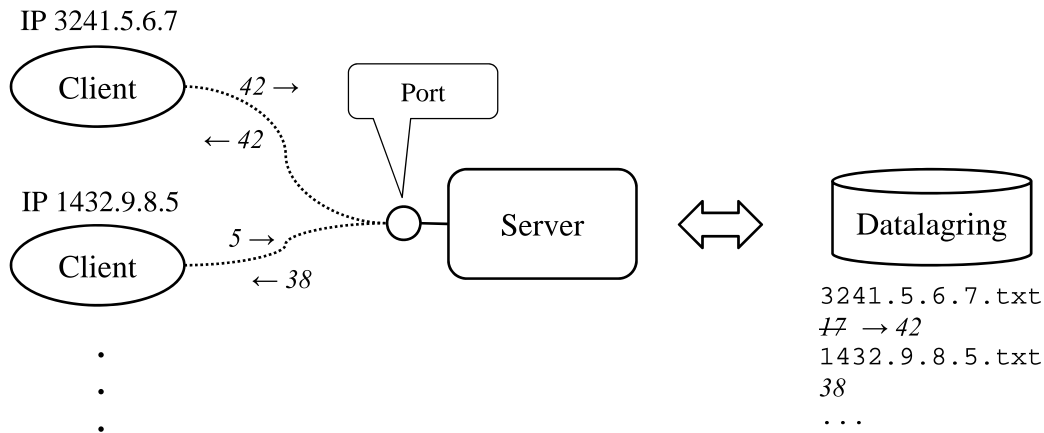
```
B.java:
    private long sum = 0;
    private long term = 1;
C.java:
    private long prod = 1;
    private long factor = 1;
```

Filkatalogens namn och den sökta texten skall alltså kunna ges som programargument och tas emot via argumentvektorn i `main`-metoden. Inför gärna privata hjälpmetoder. *Tips*: Använd standardklassen `File`.

(12 p)

Uppgift 5

I nedanstående system sänder klienter heltal till en server. Servern lagrar för varje klient det största värdet den tagit emot från klienten i en textfil som namnges efter klientens IP-adress. Varje gång ett värde tagits emot uppdateras maxvärdet i filen. Därefter sänds maxvärdet åter till klienten.



I uppgiften skall servern konstrueras. Kommunikationen skall baseras på Javas stöd för client/server-kommunikation. Varje klientanslutning skall hanteras i en egen tråd.
Dela in koden i trådklassen i flera privata hjälpmetoder. *Tips*: Använd klassen `File`.

(16 p)

Uppgift 6

Javas reflektionsmekanism ger som bekant möjligheter till dynamisk klassladdning. Till exempel kan man använda reflektion för att skapa menyer i ett GUI på ett flexibelt sätt. Till menyalternativ registreras lyssnare som vanligen delegerar åtgärden som skall utföras vidare till ett metodanrop. Vi beskriver här sådana åtgärder med gränssnittet:

```
public interface MenuAction {
    void fire();
}
```

Ex. Om klassen `OpenAction` implementerar `MenuAction` och `action` är en instans av `OpenAction` kan en lyssnare skrivas som `e -> action.fire()`. Om vi laddar sådana Action-klasser dynamiskt kan menyn byggas med utgångspunkt från befintliga klasser i programmets omgivning och menyn behöver inte hårdkodas i programmet. Även menyns och menyalternativens namn bör hanteras flexibelt, t.ex. för att lätt kunna språkanpassa programmet. Därför lagrar vi följande information i konfigurationsfiler, här två exempel (med ytterligare två Action-klasser):

menuconfig_EN.txt:
```
File
Open:OpenAction
Save:SaveAction
Exit:ExitAction
```

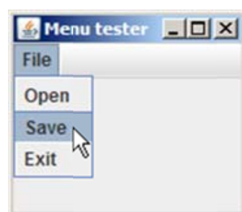
menuconfig_SV.txt:
```
Filmeny
Öppna:OpenAction
Spara:SaveAction
Avsluta:ExitAction
```

På första raden i en konfigurationsfil finns menyns namn. Därefter kommer för varje menyalternativ dess namn, samt namnet på Action-klassen som skall hantera händelsen när alternativet väljs. Dessa klasser skall laddas dynamiskt och användas för att hantera menyhändelserna. För att det skall fungera måste förstås Action-klasserna vara kompilerade så att motsvarande klassfiler existerar. Uppgiften är nu att konstruera metoden

```
public static JMenu createMenu(String configFile) throws IOException
```

Metoden skall ta namnet på en konfigurationsfil av ovanstående typ som parameter. Metoden skall läsa informationen i konfigurationsfilen, skapa en meny, ladda angivna klasser, samt använda dessa för att skapa lyssnare. För att få bättre struktur på koden får du använda privata hjälpmetoder.

Ex. En meny skapad från `menuconfig_EN.txt` ser ut så här när den visas:



När `Save` väljs anropas `fire` i klassen `SaveAction`.

(11 p)