Tentamen Programmering fortsättningskurs DIT950

Joachim von Hacht

Datum: 2013-03-13

Tid: 14.00-18.00

Hjälpmedel: Engelskt-Valfritt språk lexikon

Betygsgränser:

- U: -23
- G: 24-43
- VG: 44-60 (max 60)

Lärare: Någon (besöker ca 15.30 och 17.00), 0707/311066

Granskning: Tentan kan granskas på studieexpeditionen. Vi ev. åsikter om rättningen epost mig och ange noggrant vad du anser är fel så återkommer jag.

Instruktioner:

- För full poäng krävs ett läsbart, begripligt och heltäckande svar. Generellt 1p för varje relevant aspekt av problemet, konkreta kodexempel kan ge mer.
- Det räcker med enbart relevanta kodavsnitt, övrig kod ersätts med "..." (aldrig import, hjälpklasser såsom Main, main-metod, etc....)
- Oprecisa eller alltför generella (vaga) svar ger inga poäng. Konkretisera och/eller ge exempel. Det är aldrig någon risk att vara övertydlig!

LYCKA TILL...

1. Vad avses med (ingen uttömmande förklaring krävs); 4p

   a) Statiskt attribut.

   b) Overload.

   c) Inre klass.

   d) Värde och referens semantik (by value, by reference).

2. Givet klasserna och gränssnitten nedan (ev. vänd sida). 10p

   a) Rita ett UML-diagram över klasserna och grässnittet.

   b) Ange för varje kommentar a)-j) i koden nedan om kodstycket är korrekt eller ej. Om det ej är korrekt specificera om det är compile-time eller runtime-fel. Om det är korrekt, vad skrivs ut?

```
// a)
IA ia = new C();
IB ib = ia;
ia.doA();
// b)
IA ia = new C();
IB ib = (IB) ia;
ib.doB();
// c)
IA ia = new B();
IB ib = (IB) ia;
ia.doA();
// d)
IB ib = new C();
C c = (C) ib;
IA ia = c;
ia.doA();
// e)
B b = new C();
b.doA();
// f)
C c = (C) new B();
c.doC();
// g)
A a = new B();
C c = (C) a;
c.doA();
// h)
A a = new D();
a.doA();
```

```
// i)
A a = new B();
B b = (B)a;
b.doIt(1.0);
// j)
A a = new C();
D d = a.doIt();
d.doC();
// ------ Types -------------------
public interface IA {public void doA();}
public interface IB {public void doB();}

public class A implements IA {
    public void doA() {System.out.println("A doA");}
    public C doIt(){return new C();};
}
public class B extends A {

    public void doA() {System.out.println("B doA");}
    public void doIt(int i){ i++;};
}
public class C extends A implements IB {

    public void doC() {System.out.println("C doC");}
    public void doB() {System.out.println("C doB");}
    public D doIt(){return new D();};
}
public class D extends C {
    public void doA() {System.out.println("D doA");}
}
```

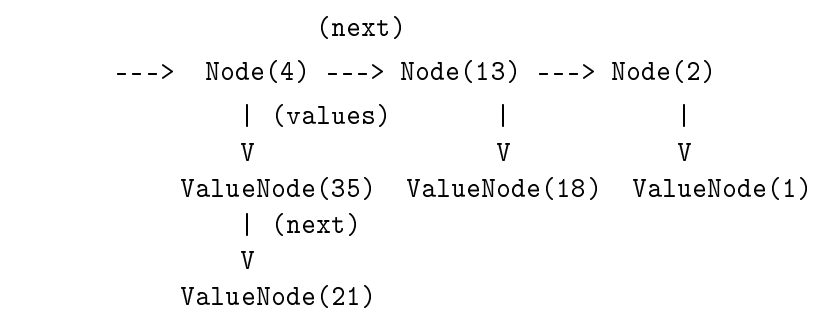
3. Vad skrivs ut i main nedan. Motivera i detalj! Verkligen i detalj! 4p

```
// I main
A a = new B();
System.out.println(a.doIt());
a.doOther();


 public class A {
    protected int i = 1;
    public int doIt() { return this.i; }
    public void doOther() {
```

```
        System.out.println(this.i);
        System.out.println(this.doIt());
    }
}
public class B extends A {
    protected int i = 2;
    public int doIt() {
        return this.i;
    }
}
```

4. Vi vill skapa en länkad map-liknande struktur enligt följande (pilar är referenser, namn på referenser inom parentes). 6p



Skapa en klass som kan representera strukturen och som använder klasserna nedan. Låt klassen implementera en metod void insert (int key, int value) som lägger till värdet "value" för given nyckel "key". Nycklar måste vara unika (inga dubbletter). Ni behöver inte skriva set/get-metoder använd attributen rakt av.

```
public class Node {
    public final int key;
    public Node next;
    public ValueNode values;
    public Node(int key) {
        this.key = key;
    }
}
public class ValueNode {
    public final int value;
    public ValueNode next;
    public ValueNode(int value) {
        this.value = value;
    }
```

```
}
```

5. Redogör får 2 olika designprinciper. Förklara innebörden, varför man bör tillämpa dem? Ge kodexempel på hur principerna konkret tillämpas. 4p

6. I kursen har vi talat mycket om tillstånd (state). 10p

   a) Vad avses med tillstånd?

   b) Vilka problem är förknippade med tillstånd? Ge några exempel. Illustrera gärna.

   c) Ge exempel på hur vi försöker hantera problemen. Motivera för- och nackdelar med exemplen. För varje exempel måste du ge ett kort konkret kodexempel hur det ser ut i Java. Ta bara med relevant kod.

7. Redogör för 2 olika designmönster förutom Singleton. 6p

   a) Vad är syftet med mönstren?

   b) Visa i UML hur de ser ut (skriv förklaringar vid behov)

   c) Visa översiktliga implementationer i Java.

8. Implementationsarv kan leda till problem (vi inkludera även Java specifika problem) 8p

   a) Ange minst 2 problem. Ge konkreta kodexempel hur dessa uppkommer (kan uppkomma).

   b) Ange tänkbara lösningar (eller workarounds) för problemen i a). Ge konkreta kodexempel!

9. Klasserna nedan kan användas för att bygga ett generiskt träd. Vi vill kunna utföra olika operationer på trädet d.v.s. vi kan inte koda in något fixt beteende i trädet. Istället anpassar vi klasserna i trädet så att vi kan använda designmönstret Visitor (alla objekt i trädet har en accept-method). Vi vill dessutom åstadkomma en matchning på elementen (klasserna) i trädet eftersom man eventuellt har olika beteende i olika klasser. 8p

   a) Visa hur man med den givna Tree-klassen nedan bygger upp ett träd med heltal (skriv koden som behövs rakt av).

   b) Skapa en konkret Visitor (klass som implementerar IVisitor). Visitor:n skall summera all värden i trädet d.v.s man skall få ett heltal som slutresultat. Visa hur Visitor:n används för trädet ovan (anropet). Ingen felhantering krävs (null värden).

```
public abstract class Tree<T> {
    public abstract <R> R accept(IVisitor<T, R> v);
    public static <T> Branch<T> branch(final Tree<T> l, final Tree<T> r) {
        return new Branch<T>() {
          { left = l;right = r; }
          @Override
          public <R> R accept(IVisitor<T, R> v) {
            return v.visit(this);
          }
        };
    }
    public static <T> Leaf<T> leaf(final T d) {
        return new Leaf<T>() {
          { data = d; }
          @Override
          public <R> R accept(IVisitor<T, R> v) {
            return v.visit(this);
          }
        };
    }
}
public abstract class Branch<T> extends Tree<T> implements IVisible<T> {
    protected Tree<T> left;
    protected Tree<T> right;
    public Tree<T> getLeft(){
        return left;
    }
    public Tree<T> getRight(){
        return right;
    }
}
public abstract class Leaf<T> extends Tree<T> implements IVisible<T>{
    protected T data;
    public T getData(){
        return data;
    }
}
public interface IVisible<T> {
    public <R> R accept(IVisitor<T, R> v);
```

```
}
public interface IVisitor<T, R> {
    public R visit( Leaf<T> l);
    public R visit( Branch<T> b);
}
```