Skolan för datavetenskap och kommunikation

Objektorienterad Programkonstruktion, DD1346

Tentamen 2014–05–23, kl. 9.00-12.00

Tillåtna hjälpmedel: Papper, penna och radergummi.

Notera: Frågorna i del I kan besvaras på för ändamålet lämnad plats i tentamenslydelsen. Frågorna i del II besvaras på separat papper. Behandla högst en uppgift per sida. Kom ihåg att skriva namn och personnummer på alla inlämnade blad. Skriv tydligt!

Betygsgränser: Betyg FX: $\geq$ 17p i del I
Betyg E: $\geq$ 20p i del I
Betyg D: $\geq$ 20p i del I **och** $\geq$ 5p i del II
Betyg C: $\geq$ 20p i del I **och** $\geq$ 10p i del II
Betyg B: $\geq$ 20p i del I **och** $\geq$ 15p i del II
Betyg A: $\geq$ 20p i del I **och** $\geq$ 20p i del II

Ansvarig: Christian Smith (ccs@kth.se)

Lycka till!

Del I - flervalsfrågor

1. I denna uppgift finns 6 kodexempel. För varje kodlistning, ange det designmönster som bäst beskriver det. Välj mönster från listan nedan. Varje korrekt angivet designmönster ger 1 p. *Notera att för att det ska bli överskådligt är koden inte nödvändigtvis komplett eller kompilerbar, utan illustrerar koncept.* (6 p)

MVC	**Singleton**	**Adapter**	**Proxy**	**Composite**	**Flyweight**	**Threadpool**
Lock	**Observer**	**Factory**	**Builder**	**Prototype**	**Facade**	**Socket**

a) Singleton

```
public class PatternA{

    private static PatternA varA = new PatternA();

    private PatternA(){}

    public static PatternA getA(){
        return varA;
    }
}
```

b) Prototype

```
public class PatternB implements Cloneable{

    private static PatternB varB = new PatternB();

    private PatternB(){
        doHeavyInitialization();
    }

    public static PatternB getB(){
        return varB.clone();
    }

}
```

c) Adapter

```
public class PatternC extends SomeClass{

    private OtherClass oc = new OtherClass();

    @Override
    public float getC(){
        return (float)oc.getDoubleC();
    }

}
```

d) Observer

```
public class PatternD{

    private int d = 0;
    private List<SomeType> myList = new LinkedList<SomeType>();

    public void incD(){
        d++;
        doSomething();
    }

    public void add(SomeType t){
        myList.add(t);
    }

    public void doSomething(){
        for(SomeType t : myList){
            t.someMethod(d);
        }
    }
}
```

e) Proxy

```
public class PatternE extends AbstractSuperClass{

    private AbstractSuperClass oc = new OtherClass();

    public void doStuff(){
        if(userHasAccessRights()){
           oc.doStuff();
        }
    }
}
```

f) Builder

```
public class PatternF{

    MyType1 myObject1;
    MyType2 myObject2;

    public PatternF(){
        myObject1 = null;
        myObject2 = null;
    }

    public void setObject1(Object o){
        myObject1 = new MyType1();
        myObject1.doSomeInitialStuff(o);
    }

    public void modifyObject1(Object o){
        myObject1.doSomeShallowStuff(o);
    }

    public void setObject2(Object o){
        myObject2 = new MyType2();
        myObject2.doSomeInitialStuff(o);
    }

    public void modifyObject2(Object o){
        myObject2.doSomeDeeperStuff(o);
    }
```

```
    public myType3 getType3(){
        myType3 myOutputObject = new myType3(this);
        return myOutputObject;
    }
}
```

2. Nedan följer 4 kodexempel. Vi antar att klasserna **A** och **B** och gränssnittet **Bint** ligger i separata filer, i underkatalogen `AB`. För varje rad i metoden `testMethod()` i klassen **X**, ange om den är korrekt eller om den ger upphov till kompileringsfel. Varje korrekt klassifierad rad ger 0.5 p (4 p).

```
package AB;
public class A{

    public int myInt = 1;
    private double myDbl = 1.2;
    protected Bint myBint = new B();

    public void doStuff(){
        myInt++;
    }

    private void doPrivStuff(){
        myDbl *= -1.0;
    }
}
-------------------------------------------------

package AB;
public class B extends A implements Bint{

    float myFlt = 1.4f;
    private String myStr = "Putte";

    public void doBStuff(){
        myFlt += 5;
    }

    public void doStuff(int a){
        myFlt += a;
    }
}
-------------------------------------------------
package AB;
public interface Bint{
    public void doBStuff();
}
```

```
import AB.*;
public class X extends A{

    float myFlt = 1.4f;
    private String myStr = "Hello World!";
    private B myB = new B();

    public void doStuff(){
        myInt += 5;
    }

    public void TestMethod(){

        myBint.doStuff();Fel
        super.myInt = (int)myFlt; Korrekt
        myDbl = 2.1;Fel
        myB.myStr = myStr; Fel
        myB.doStuff((int)myB.myDbl); Fel
        myB.myFlt = 0.4f; Fel
        myB.doBStuff(); Korrekt
        super.doPrivStuff(); Fel


    }
}
```

3. För varje påstående om arv i Java[1], ange om det är sant eller falskt. 5 korrekta svar ger 4p, 4 korrekta svar ger 2p, 3 korrekta svar ger 1p, 2 eller färre korrekta svar ger 0 p (4 p)

   a) **Abstrakta klasser** kan ärva från **gränssnitt**. falskt
   b) Ett **gränssnitt** kan ärva från flera andra **gränssnitt**. sant
   c) Det går inte att ärva från en klass deklarerad med nyckelordet `static`. falskt
   d) Det går att göra klasser som man varken kan instansiera eller ärva från. sant
   e) Man kan skapa **abstrakta klasser** som inte ärver från `Object`. falskt

4. För varje påstående om nästade klasser, ange om det är sant eller falskt. 4 korrekta svar ger 2p, 3 korrekta svar ger 1p, 2 eller färre korrekta svar ger 0 p (2 p)

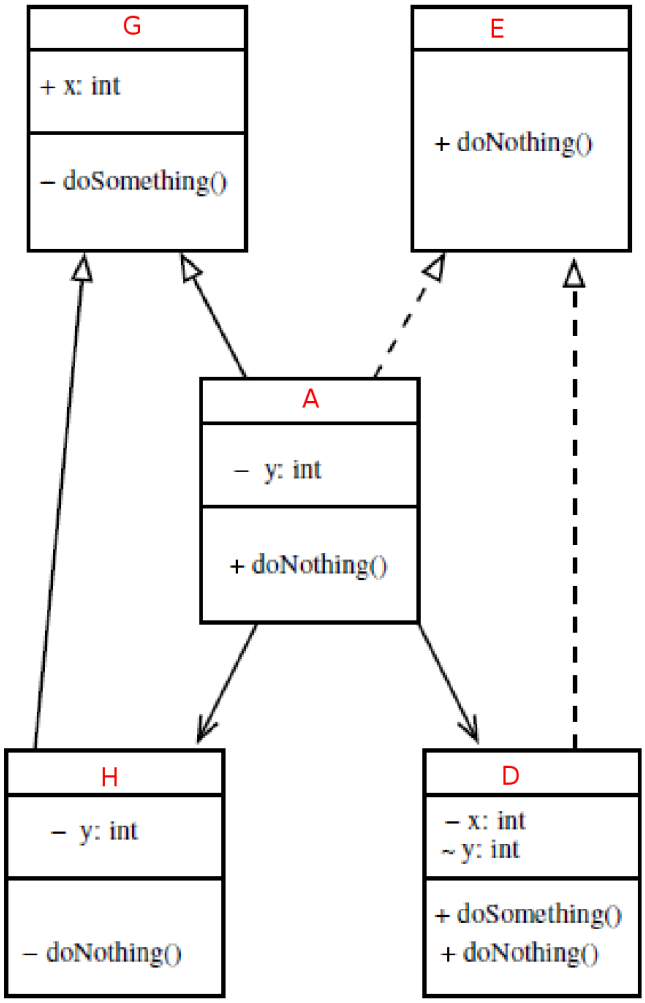
   a) En **nästad klass** har tillgång till alla fält och metoder i den omgivande klassen, inklusive de som deklarerats som `private`. sant
   b) En **anonym subklass** i en klass **B** kan instansieras från en annan klass **A** om både **A** och **B** ligger i samma paket. falskt
   c) **Statiska nästade klasser** kräver en instans av den omgivande klassen för att instansieras. falskt
   d) En **nästad klass** ärver alltid från den omgivande klassen. falskt

5. För varje påstående om designmönster nedan, ange om det är sant eller falskt. 5 korrekta svar ger 4p, 4 korrekta svar ger 2p, 3 korrekta svar ger 1p, 2 eller färre korrekta svar ger 0p (4 p)

   a) Det främsta syftet med `Singleton` är att minska minnesanvändning. falskt
   b) `Threadpool` kan användas för att minska overhead i parallella program. sant
   c) **Thread Interference** är ett problem som man ofta kan undvika med `Lock`. sant
   d) Ett vanligt mål med att använda `Observer` är att åstadkomma lösare koppling mellan olika klasser. sant
   e) En `Adapter` är ett specialfall av `Proxy`. falskt

[1] Vi avser Java version 6 eller 7, som har använts i kursen, och inte den i mars 2014 släppta version 8

6. Fälten för namnen på delarna i nedanstående klassdiagram är tomma. Fyll i fälten med rätt namn (A–J) från klass- och gränssnittsdefinitionerna som följer från nästa sida. Vissa definitioner kan beskriva klasser/gränssnitt som inte fungerar eller inte finns med i UML-diagrammet. Det finns totalt 10 definitioner, från vilka 5 namn skall väljas ut. Varje rätt ifyllt namnfält ger 1 p. (5 p)



```
public class A extends G implements E{

    private int y;

    public void doNothing(){
        D myD = new D();
        H myH = new H();
    }
}

public class B extends A implements E{

    private int x;
    protected int y;

    public void doSomething(){
    }

    private void doNothing(){
    }
}

public class C {

    private int x;

    private void doSomething(){
    }
}

public class D implements E{

    private int x;
    int y;

    public void doSomething(){
    }

    public void doNothing(){
    }
}
```

```
public interface E {

    public void doNothing();

}

public interface F extends E {

    private int x;
    int y;

    public void doSomething();

    private void doNothing();
}

public class G {

    public int x;

    private void doSomething(){
    }
}

public class H extends G {

    private int y;

    private void doNothing(){
    }
}

public class I extends C implements E{

    int y;

    public void doNothing(){
        H myH = new H();
        F myF = new F();
    }
}
```

```
public interface J extends C implements A{

    private int y;

    public void doNothing(){
    }
}
```

Del II - fördjupningsfrågor

Följande uppgifter besvaras på separat papper.

7. a) Förklara funktionen av nyckelordet `static` i Java. (2 p)
   b) Förklara funktionen av nyckelordet `final` i Java. (2 p)

8. Förklara vad en **profilerare** är, och hur och varför man använder den. (2 p)

9. Det har visat sig att i den version av Java som finns på ditt företags nya serie av smarta vitvaror så saknas en massa funktionalitet, och av licensskäl kan ni inte använda existerande externa bibliotek. Du får i uppgift att skriva ett bibliotek för linjär algebra. Ditt bibliotek ska stödja vektorer och matriser av godtycklig storlek, samt ett stort antal vanliga operationer, som t.ex addition, multiplikation, inverser, pseudoinverser, egenvärdes- och singulärvärdesberäkningar, transponering, kryssprodukter, homogena transformer, och så vidare.

   Din uppdragsgivare säger att beräkningsprestanda inte är viktigt, eftersom man räknar med att kunna licensiera ett effektivt bibliotek till generation två av produktserien när de mer krävande funktionerna skall lanseras, men man vet ännu inte vilket bibliotek det kommer att bli. Man behöver dock ett fungerande bibliotek för stunden, för att funktionaliteten i generation ett ska bli komplett.

   Beskriv hur du tänker när du utformar ditt bibliotek. Hur representerar du datatyperna? Hur gör du med klasshierarkier? Vilka designmönster kan vara tillämpliga, och hur? Vad bör du tänka på inför framtiden? (7 p)

10. Antag att man har ett stort och beräkningsintensivt problem som lämpar sig väl för parallellisering. Beskriv hur man kan göra om man vill lägga ut problemet på flera olika datorer och vilka designmönster som kan vara tillämpliga. (3 p)

11. Nedan följer fem olika designmönster. Välj ut tre olika kombinationer av två mönster ur denna lista, och beskriv hur dessa mönsterpar kan kombineras med varandra, och varför man skulle vilja göra det. Notera att det ska handla om en direkt kombination av mönster, inte bara en applikation som råkar innehålla båda mönstren. (6 p)

    Observer MVC Facade Composite Iterator

12. Din vän, som håller på att lära sig Java, har skrivit följande rad i ett program:

    ```
    List<Comparable> myList = new List<Comparable>();
    ```

    Din vän vill koda så generellt som möjligt, och åstadkomma lös koppling genom att inte använda några explicita typer om det inte är absolut nödvändigt, men just här gick det som synes fel. Ge ett svar som förklarar för vännen varför raden inte fungerar, och hur man i stället skulle kunna göra för att åstadkomma en fungerande lösning som gör det hen vill, men ändå inte behöva ange mer explicita typer. (3 p)