Lunds Tekniska Högskola
Datavetenskap
Mathias Haage, Emilie Engström, Roger Henriksson

Tentamen i Objektorienterad modellering och design

Vid bedömningen kommer hänsyn att tas till lösningens kvalitet. UML-diagram skall ritas i enlighet med UML-häftet. Man får förutsätta att det finns standardkonstruerare i alla klasser. De behöver ej redovisas i lösningar.

Hjälpmedel: Martin: Agile Software Development
Andersson: UML–syntax
Holm: Java snabbreferens

---

1 I programspråket LISP är ett *uttryck* antingen en *atom* eller en *elista*. En atom kan vara ett tal eller en sträng. En elista är antingen tom och saknar innehåll eller består av ett *huvud* som är ett uttryck och en *svans* som är en elista. Ett uttryck har en storlek som utgörs av det totala antalet atomer i uttrycket.

a. Rita ett klassdiagram för uttryck enligt beskrivningen ovan med arv, associationer med multipliciteter och riktningar, attribut och en metod `size()` som returnerar storleken på ett uttryck (antalet ingående atomer).

Din design ska vara sådan att det är möjligt att implementera metoden `size()` utan att använda sig av `if`-, `for`- eller `while`-satser. Det ska heller inte behöva förekomma några referenser med värdet `null` i en implementation av designen. För att åstadkomma detta kan ett särskilt designmönster som förekommer i kursen användas.

b. Vad brukar man kalla designmönstret som det refereras till i föregående deluppgift?

c. Implementera metoden `size()` (i Java) på rätt plats(er). Se till att det tydligt framgår i vilka klasser `size()` måste vara implementerad och hur.

(5p)

2 Följande program bryter mot lokalitetsprincipen.

a. Åtgärda detta. Lösningen redovisas med Java-kod.

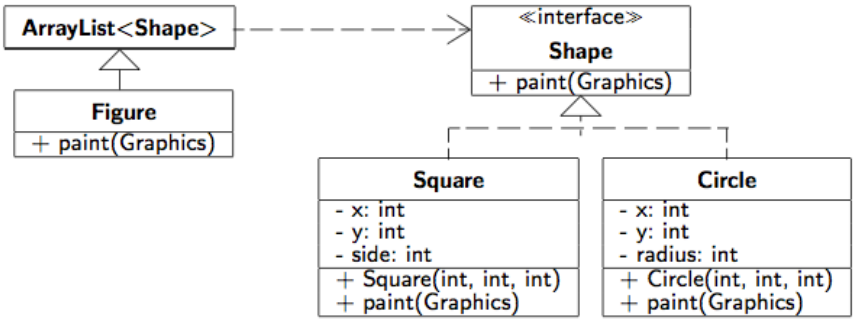
b. Motivera varför den nya lösningen i och med åtgärden är bättre än den gamla.

(3p)

```
public class Point {
  private int x,y;
  public int getX() {
    return x;
  }
  public void setX(int x) {
    this.x = x;
  }
  public int getY() {
    return y;
  }
  public void setY(int y) {
    this.y = y;
  }
}
```

```
public class Circle {
  private Point centre;
  private int radius;
  public Point getCentre() {
    return centre;
  }
  public int getRadius() {
    return radius;
  }
}
public class Drawing {
   private List<Circle> circles;
   //omissions
   private void moveCircle(Circle aC, int dx, int dy) {
     int x1 = aC.getCentre.getX();
     aC.getCentre.setX(x1 + dx);
     int y1 = aC.getCentre.getY();
     aC.getCentre.setY(y1 + dy);
   }
}
```

3 Nedan ett klassdiagram för en del av ett ritprogram där man kan hantera och rita ut kvadrater, cirklar och figurer bestående av kvadrater och cirklar. Designen behöver dock förbättras.



a. Klasserna `Circle` och `Square` har två identiska attribut `x` och `y`. Gör om designen så att attributen placeras i en gemensam superklass, lägg till metoden `void move(int dx, int dy)` i gränssnittet och implementera metoden på rätt plats(er).

Lösningen redovisas med ett klassdiagram och Javakod för alla klasser (utom metoden `paint`). Se till att det tydligt framgår i vilka klasser metoden move måste finnas och hur den ser ut i respektive klass.

b. Tyvärr kan resten av ritprogrammet fortfarande inte hantera figurer, kvadrater och cirklar på ett enhetligt sätt. Man vill att en figur ska kunna innehålla andra figurer likväl som andra former ("shapes"). Använd *Composite*-mönstret och förbättra designen i enlighet med detta önskemål. Implementera även metoden `public void add(Shape aShape)` i klassen `Figure`. Lösningen redovisas med Javakod (de ändringar du måste göra från föregående uppgift räcker). Inkludera dock metoderna `move(int,int)` och `paint(Graphics)` där `Graphics` innehåller

```
public interface Graphics {
  public void drawRectangle(int x, int y, int width, int height);
  public void drawCircle(int x, int y, int radius);
}
```

c. I en annan del av ritprogrammet har vi klassen `Drawing` där vi hittar följande kod:

```
public class Drawing {
      //omissions
      Figure f1 = new Figure();
      Shape s1 = new Square(10,10,30);
      Shape c1 = new Circle(40, 40, 15);
      f1.add(s1);
      f1.add(c1);
      f1.paint(gr);
      //omissions
}
```

Rita ett sekvensdiagram för exekvering av raden "`f1.paint(gr)`". Utgå ifrån att designen uppdaterats i enlighet med uppgift b ovan.

d. Klassen `Drawing` är fortfarande beroende på Figure, Square och Circle då den behöver skapa objekt av dessa typer. I en situation där man i stället till exempel skulle läsa in uppgifter om vilka typer av figurer som ska skapas från en fil kan detta vara onödigt. Ta därför bort detta beroende med hjälp av Factory-mönstret och låt en sträng-parameter bestämma vilken objekttyp som ska skapas och vilka parametrar objektet ska ha. För att skapa en cirkel i positionen (30,40) och med radien 20 skulle till exempel följande sträng kunna användas: "`Circle 30 40 20`"

Ledning: Om du skriver `String[] tokens = "Circle 30 40 20".split("\\s");` kommer vektorn `tokens` att ges innehållet {`"Circle","30","40","20"`}[1].

Lösningen redovisas med Javakod för de nya klass(er) och gränssnitt som behövs, samt de modifieringar och tillägg som behöver göras i `Drawing`. (Gör en direkt motsvarighet till den existerande klassen `Drawing` – det är *inte* meningen att du ska läsa in uppgifter från fil).

(12p)

4 Ett program för kalkylark innehåller paketen `expr` och `sheet`. I paketet `expr` finns bland annat

```
public interface Expr {
  public double value(Sheet sheet);
}
public class AddressExpr implements Expr {
  private int address;
  public double value(Sheet sheet) {
    return sheet.value(address)
  }
}
```

och i paketet `sheet` finns

```
public class Sheet {
  private Expr[] array;
  public double value(int address) {
    return array[address].value(this)
  }
  // omissions
}
```

a. Paketen är cykliskt beroende. Rita ett fullständigt klassdiagram för ovanstående kod. Dvs, visa paket, klasser, beroenden, attribut, metoder, synlighet, ... Markera i diagrammet det cykliska beroendet mellan paketen.

b. Förklara varför det är dålig design att ha cykliska beroenden mellan paketen som i exemplet ovan.

c. Eliminera det cykliska beroendet (utan att flytta klasser eller kod mellan paketen) så att paketet `expr` inte längre är beroende av paketet `sheet`. Det är tillåtet att modifiera och lägga till klasser och gränssnitt, men inte att tillfoga nya paket. Lösningen redovisas både med fullständigt klassdiagram och Java-kod.

(6p)

[1] Argumentet till operationen `split()` är ett s.k. reguljärt uttryck som beskriver vilka tecken som ska betraktas som avskiljare mellan de olika delarna av strängen – i detta fall mellanslag.