Objektorienterad modellering och diskreta strukturer / design

Designprinciper

Lennart Andersson

Reviderad 2011–09–01

2011

QED

- Angelägen för utbildningen edaf10: +92/eda061: +73
- Överlag nöjd +45/+73
- Förståelseinriktad examination +50/+63

QED: önskvärda förbättringar

- Dela upp EDAF10 i två kurser
- Eliminera C++ i läroboken

QED: det bästa med kursen

- Kursen har förändrat min syn på programmering.
- Jag har lärt mig mer programmering än i någon annan kurs.
- Bra och meningsfulla projekt.

Att göra

- Bilda grupper för projekt och anmäla i SAM. Mingel i pausen.
- Välja övningsgrupp i SAM. Förbereda övningen.
- Göra eclipse-lab.

Agile?

- Agile Software Development ?
  Agile betyder vig, rörlig, kvick eller anpasslig.
- På svenska vanligen: lättrörlig

Lärobokens undertitel

- Principles,
- Patterns,
- and Practices

Utvecklingsmodellen

Extreme Programming, XP

- Berättelser (stories)
- Korta iterationer
- Testa först
- Parprogrammering
- Enkel design
- Ständig refaktorisering

Programutveckling i grupp, HT2 för D2, valbar för CEFPi

God praxis

- Designa ej på spekulation, dvs för saker som kanske inte kommer att användas.
- Gör om designen när den inte längre är bra. Refaktorisering.
- Optimera inte programmet förrän du konstaterat att det är nödvändigt.
- Producera ingen dokumentation om den inte behövs genast och är betydelsefull. (Martin)

Design smells

- Stelhet (Rigidity) — Designen är svår att modifiera
- Bräcklighet (Fragility) — Designen tål inte modifiering
- Orörlighet (Immobility) — Designen går inte att återanvända
- Seghet (Viscosity) — Det är svårare att göra snygga ändringar än "hack"
- Spekulativ design (Needless Complexity)
- Duplicerad kod (Needless Repetition) — Klipp & klistra–orienterad programmering
- Oklarhet (Opacity) — Obegriplig design

Exempel på usel design

USEL DESIGN

```
public class EnduroHandler { // or TotalComputation
...
startTime = enduro.getRaces()[i].
    getCompetitors()[j].getStartTime().getSeconds();

endTime = enduro.getRaces()[i].
    getCompetitors()[j].getEndTime().getSeconds();

enduro.getRaces()[i].getCompetitors()[j].
    setTotalTime(endTime - startTime);
...
}
```

USEL DESIGN

Fortfarande uselt

USEL DESIGN

```
public class  EnduroHandler {
...
result = enduro.getRaces()[i].
    getResults()[j];

startTime = result.getStartTime().getSeconds();

endTime =  result.getEndTime().getSeconds();

result.setTotalTime(endTime - startTime);
...
}
```

USEL DESIGN

Lokalitetsprincipen!

Mycket bättre:

```
public class Enduro {
  private ArrayList<Race> races;
  public void computeTotal() {
    for(Race race: races ) {
      race.computeTotal();
    }
  }
...
}
```

som delegerar arbetet till den klass som vet allt om Race:

Race

Mycket bättre:

```
public class Race {
  private ArrayList<Result> results;
  public void computeTotal() {
    for(Result result: results) {
      result.computeTotal();
    }
  }
...
}
```

som delegerar till den klass som vet allt om Result:

Result

Mycket bättre:

```
public class Result {
  private Competitor competitor;
  private Time startTime, endTime, totalTime;
  public void computeTotal() {
    totalTime = endTime.difference(startTime);
  }
...
}
```

som delegerar till experten på tidsberäkningar: Time

Designprinciper

- SRP Single Responsibilty Principle
- OCP Open/Closed Principle
- DIP Dependency Inversion Principle
- LSP Liskov Substitution Principle
- ISP Interface Segregation Principle

- ALP Lokalitetsprincipen

SRP — Enkelt ansvar

Single Responsibility Principle

A class should have only one reason to change. (Martin)

En klass med flera ansvarsområden ger bräcklighet.
En klass utan ansvar är onödig.

Time har bara ett ansvarsområde

```
public class Time implements Comparable<Time> {

   private int sec;

   public Time difference(Time other)
   public int compareTo(Time other)
   public String toString()
   public Time(String time)
}
```


Competitor har flera

USEL DESIGN

```
public class Competitor {

   private String firstName, secondName;
   private String idNumber;
   private int start, end, total;

   public String fullName()
   private boolean checkIdNumber()
   public void computeTotal()
}
```

USEL DESIGN

Delegera ansvaren!

```
public class Competitor {

   private Name name;
   private IdNumber idNumber;
   private Result result;

   public String fullName() {
     return name.toString();
   }

   public void computeTotal() {
     result.computeTotal();
   }
}
```

GenerateCode har massor av ansvar

```
public static void generateCode(Instruction c) {
switch (c.opCode) {
case 0: //MOV
    if (!(c.arg1 instanceof Current || c.arg1 instanceof Next) &&
        !(c.arg2 instanceof Current || c.arg2 instanceof Next)) {
        out.writeBytes("mov "  + code.convert(c.arg1) +
            " , " + code.convert(c.arg2) + "\n");
    } else if ((c.arg1 instanceof IntConst || c.arg1 instanceof BoolConst) &&
        c.arg2 instanceof Current) {
        out.writeBytes("mov "  + code.convert(c.arg1)+", "+
            code.saveToReg(c.arg1) +"\n");
        out.writeBytes("set "  + code.convert(c.arg2,staticlevel-1) +" , " +
            code.saveToReg(c.arg2) +"\n");
        out.writeBytes("st " + code.saveToReg(c.arg1) + " , " +
            "[" + code.saveToReg(c.arg2) + "]\n");
    } else if((c.arg1 instanceof Temp) && c.arg2 instanceof Current) {
        out.writeBytes("set "  + code.convert(c.arg2,staticlevel-1) +
            " , " + code.saveToReg(c.arg2) +"\n");
        out.writeBytes("st "  + code.saveToReg(c.arg1) + " , " +
            "[" + code.saveToReg(c.arg2) +"]\n");
    } else if(c.arg1 instanceof Current && c.arg2 instanceof Temp) {
        out.writeBytes("set "  + code.convert(c.arg2) + " , " +
            code.saveToReg(c.arg2) +"\n");
        out.writeBytes("ld "  + "["+ code.saveToReg(c.arg2) +"]" +
            " , " + code.saveToReg(c.arg1) +"\n");
    } else if(c.arg1 instanceof Current && c.arg2 instanceof Current) {
        if(!(c.arg1.toString().equals(c.arg2.toString()))) {
            out.writeBytes("set "  + code.convert(c.arg1,staticlevel-1) +
                " , "+ code.saveToReg(c.arg2) +"\n");
```


Vilka ansvarsområden?

Kodgenerering för

- MOV–instruktionen
- ...
- OR–instruktionen
- Current–operander
- Temp–operander
- ...

Aktivitet

Gör en bättre design!

OCP — Öppen/sluten-principen

Open/Closed Principle

> *Classes should be open for extension and closed for modification. (Meyer)*

Det bör vara möjligt att lägga till ny funktionalitet utan att modifiera existerande kod.

Dålig design: Circle, Square

```
public class Circle {
    int radius;
    int x, y;
    public Circle(int radius, int x, int y) {
        this.radius = radius;
        this.x = x;
        this.y = y;
    }
}
```

Square är analog.

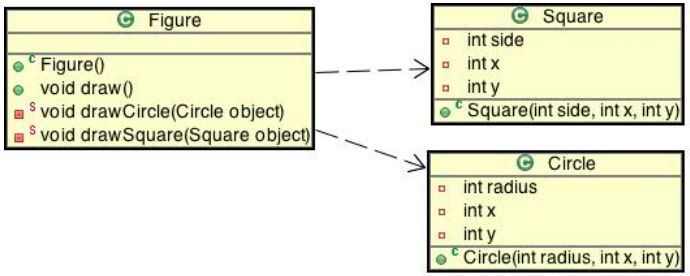
Dålig design: Figure

```
public class Figure extends ArrayList {
  public Figure() {
    add(new Square(4, 1, 1));
    add(new Circle(4, 2, 2));
  }
  public void draw() {
    for (Object object: this) {
      if (object instanceof Square) {
        drawSquare((Square) object);
      } else {
        drawCircle((Circle) object);
      }
    }
  }
  ...
}
```


Dålig design: Figure

```
public class Figure {
  ...
  private static void drawCircle(Circle circle) {
    // TODO Implement method
  }
  private static void drawSquare(Square square) {
    // TODO Implement method
  }
}
```


Dålig design



Bra design: Circle

```
public class Circle implements Shape {
    private int radius;
    private int x, y;
    public void paint(Graphics graphics) {
        // TODO Auto-generated method stub
    }
}
```

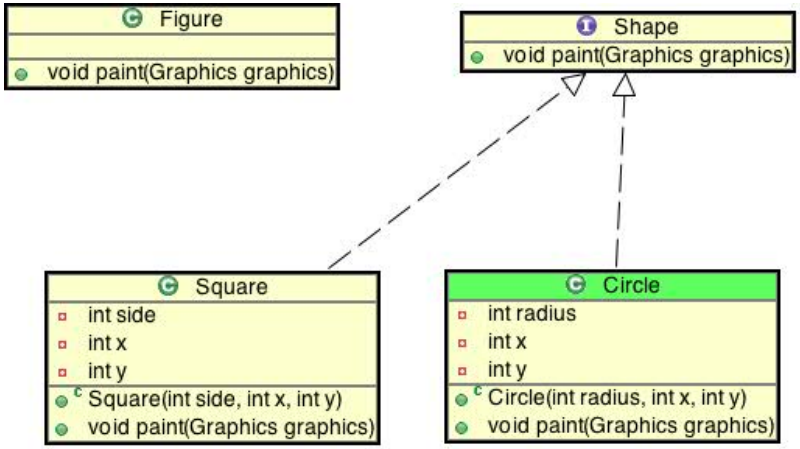

Bra design: Shape

```
public interface Shape {
    public void paint(Grahpics graphics);
}
```


Bra design: Figure

```
public class Figure extends ArrayList<Shape> {
  public void paint(Graphics graphics) {
    for (Shape shape : list) {
      shape.paint(graphics);
    }
  }
}
```


Bra design



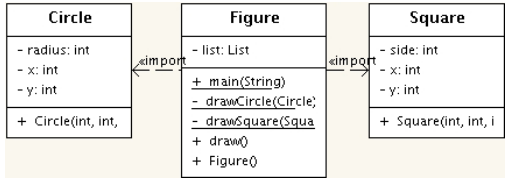
DIP — Bero på abstraktioner

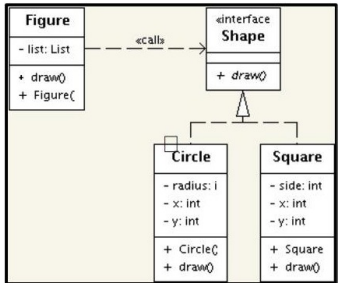
Dependency Inversion Principle

High level classes should not depend on low level classes; both should depend on abstractions.
Abstractions should not depend on details. Details should depend on abstractions.

UML-pilar skall gå mot gränssnitt och abstrakta klasser.

Dependency Inversion



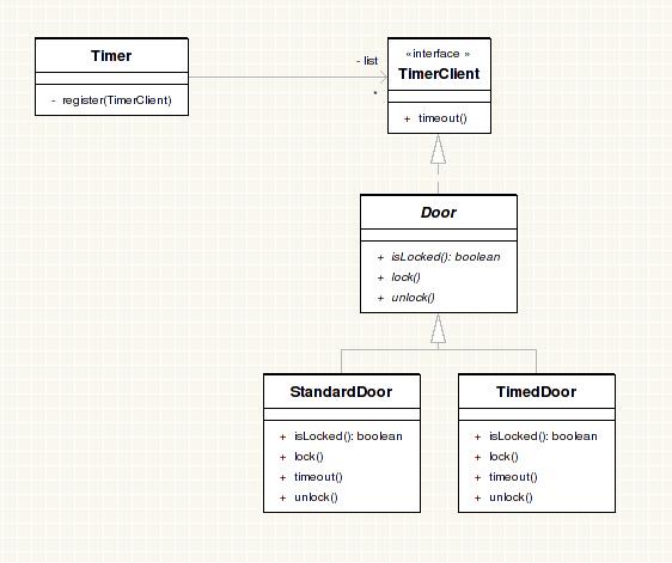


ISP — Segregation

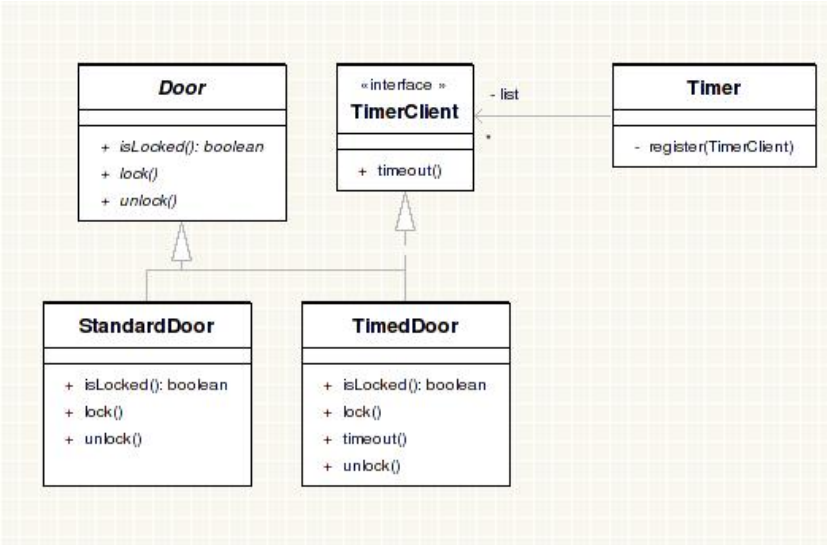
Interface Segregation Principle

Classes should not be forced to depend on methods that they do not use.

Brott mot ISP



Enligt ISP



LSP — Substitutionsprincipen

Liskov Substitution Principle

Subtypes must be substitutable for their base types.

Subklasser skall implementeras så att de hanteras korrekt även när "användaren" bara känner till en superklass.

Brott mot LSP

```
public class IntPair {
  private int i1,i2;
  public boolean equals(Object object) {
     IntPair other = (IntPair) object;
     return i1==other.i1 && i2==other.i2;
  }
}
```

Klassen bryter mot LSP på två sätt:

- Metoden kastar ett undantag om `object` har fel typ. Den borde returnera `false` när detta händer.
- Klassen fungerar ej som nyckel i en hashtabell. Lika objekt skall ha samma hashkod.

Brott mot LSP – 2

```
class Rectangle {
  int width, height;
  int x, y;
  int getWidth() {
    return width;
  }
  void setWidth() {
    this.width = width;
  }
  void setHeight() {
    this.height = height;
  }
  int area() {
    return height*width;
  }
}
```

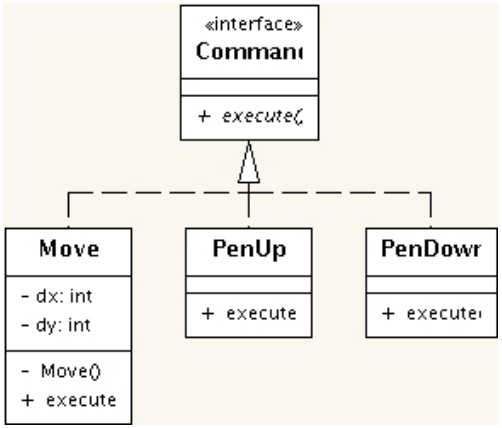
A Square **is a** Rectangle

```
class Square extends Rectangle {
  Square(int side) {
    super(side, side);
  }
  void setWidth(int side) {  // identical setHeight
    width = side;
    height = side;
  }
}

void testRectangle(Rectangle rectangle) {
  rectangle.setWidth(2);
  rectangle.setHeight(3);
  assertEqual(6, rectangle.area());
}

testRectangle(new Square(1))  // reports error
```


Designmönster

- Command
- Composite

Command



Command användning

```
List<Command> commandList = new ArrayList<Command>();
commandList.add(new PenDown());
commandList.add(new Move(1, 0));
commandList.add(new Move(0, 1));
commandList.add(new Move(-1, 0));
commandList.add(new Move(0, -1));
commandList.add(new PenUp());

for (Command command : commandList) {
    command.execute();
}
```

Command med undo

```
public interface Command {
    public void execute();
    public void undo();
}
```


Command fördelar

- Man delegerar arbetet till klasser med enkelt ansvar.
- Separation i tiden mellan konstruktion och exkvering av kommandon
- Separation i rummet mellan konstruktion och exekvering av kommandon
- Historielista
- Undo

Composite

```
public class Macro implements Command {
    private List<Command> commandlist =
                          new ArrayList<Command>();
    public void add(Command command) {
        commandList.add(command);
    }

    public void execute() {
        for (Command command : commandList) {
        command.execute();
    }
}
```


Composite – enklare

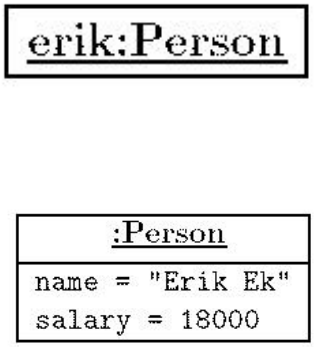
```
public class Macro extends ArrayList<Command>
                   implements Command {
    public void execute() {
        for (Command command : this) {
            command.execute();
        }
    }
}
```

men med sämre integritet.

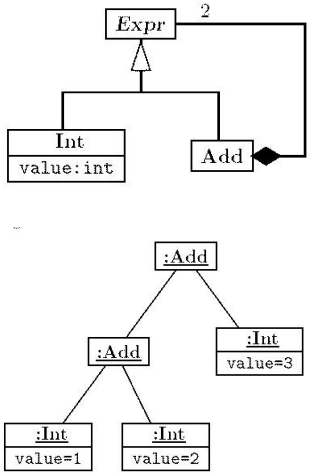
Composite användning

```
Macro macro = new Macro();
macro.add(new PenDown());
macro.add(new Move(2, 0));
macro.add(new Move(0, 2));
macro.add(new Move(-2, 0));
macro.add(new Move(0, -2));
macro.add(new PenUp());

macro.execute();
new Move(1, 1).execute();
macro.execute();
```


Objektdiagram

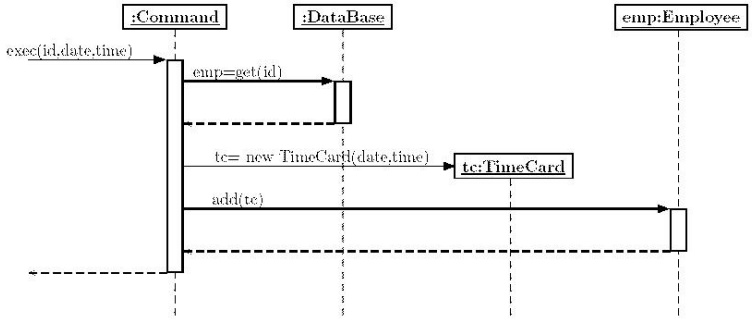


Objektdiagram

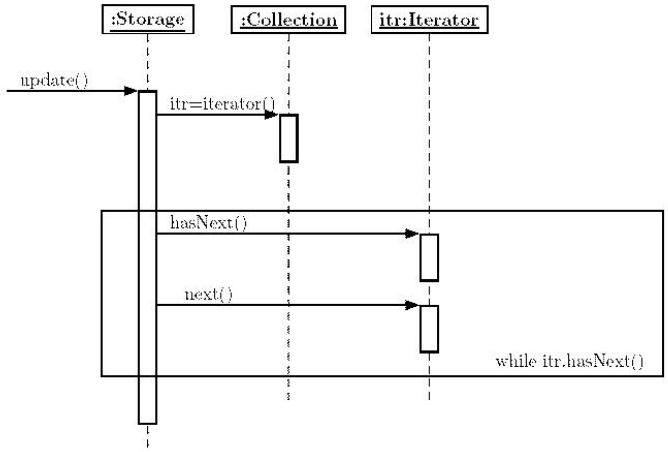


$(1 + 2) + 3$

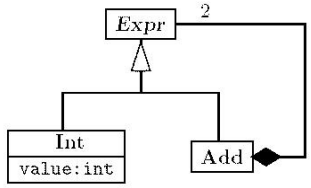
Sekvensdiagram



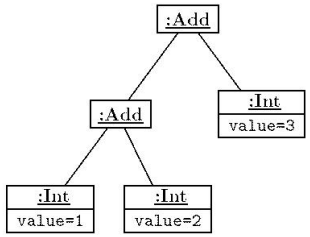
Sekvensdiagram



Sekvensdiagram



$(1 + 2) + 3$



Beräkna $(1 + 2) + 3$

Sekvensdiagram