

DD2385

Programutvecklingsteknik

Några bilder till föreläsning 12

14/5 2014

Innehåll

- ▶ Testning med JUnit
- ▶ Refactoring
- ▶ Komposition
- ▶ Några designprinciper

JUnit

- ▶ Ramverk i Java för testning av Java-klasser
- ▶ Utvecklat av Gamma & Beck, första versionen kom 2002
- ▶ `@Test` = annotering för testmetod
- ▶ `@Suite.SuiteClasses(...)`
Klasser med testmetoder bygger upp testsviter enligt mönstret *Composite*
- ▶ Textgränssnitt eller grafiskt
- ▶ (Ganska) enkelt att använda
- ▶ Gratis på www.junit.org

Refactoring

- ▶ Förbättra programkoden
 - ▶ Minskad kodupprepning
 - ▶ Ökad flexibilitet
 - ▶ Tydligare
 - ▶ Effektivare
- ▶ "Utsidan" / gränssnitt mot användare ändras ej
- ▶ Funktionaliteten ändras ej
- ▶ Svårt
- ▶ Några enkla exempel visas

Refactoring - kodupprepning inom klass

```
class C {  
    void m1() {  
        <A> do1(); do2(); do3(); <B>  
    }  
    void m2() {  
        <C> do1(); do2(); do3(); <D>  
    }  
}
```

förbättras till

```
class C {  
    void doAll { do1(); do2(); do3(); }  
    void m1() { <A> doAll(); <B> }  
    void m2() { <C> doAll(); <D> }  
}
```

Refactoring - kodupprepning i olika klasser

```
class A {  
    void m1() {  
        <A> do1(); do2(); do3(); <B>  
    }  
}
```

```
class B {  
    void m2() {  
        <C> do1(); do2(); do3(); <D>  
    }  
}
```

förbättras genom arv eller delegering

Arv

```
class C {  
    void doAll { do1(); do2(); do3(); }  
}  
  
class A extends C {  
    void m1() { <A> doAll(); <B> }  
}  
  
class B extends C {  
    void m2() { <C> doAll(); <D> }  
}
```

Delegering

```
class Helper {  
    void doAll() {  
        do1(); do2(); do3();  
    }  
}  
  
class A {  
    Helper helper = new Helper();  
    void m1() { <A> helper.doAll(); <B> }  
    ..  
}  
  
class B {  
    Helper helper = new Helper();  
    void m2() { <B> helper.doAll(); <D> }  
    ..  
}
```

Kodupprepning igen

```
class A {  
    void m() {  
        <common 1>  
        <A-spec>  
        <common 2>  
    }  
}  
  
class B {  
    void m() {  
        <common 1>  
        <B-spec>  
        <common 2>  
    }  
}
```

Kodupprepning igen, forts.

```
class C {  
    void specm(){}; // tom eller abstrakt  
    void m() {  
        <common 1>  
        specm();  
        <common 2>  
    }  
}  
  
class A extends C { // Vilket designpattern  
    void specm() { // ar detta ?  
        <A-spec>  
    }  
}  
  
class B extends C {  
    void specm() {  
        <B-spec>  
    }  
}
```

Komposition

- ▶ Ärva från två klasser **går inte i Java**
- ▶ Hur får man funktionalitet från två klasser **A** och **B** i en klass?
- ▶ Alt 1: Gör ny klass som **ärver från den ena, skapar objekt av den andra**
- ▶ Alt 2: **Komposition**
Gör en ny klass där objekt av både **A** och **B** skapas
- ▶ Kan man få den nya klassen att **bete sig** som både **A** och **B** ???

```
interface AInterface {  
    void ma();  
}  
class A implements AInterface {  
    void ma() {...} // method body  
}  
  
interface BInterface {  
    void mb();  
}  
class B implements BInterface {  
    void mb() {...} // method body  
}  
  
class Composition implements AInterface, BInterface {  
    // next page ...  
}
```

```

class Composition implements Ainterface,
                               Binterface {
    A myA; B myB;

    C () {
        myA = new A();
        myB = new B();
    }

    void ma() {
        myA.ma();
    }

    void mb() {
        myB.mb();
    }
}

```

Komposition

- ▶ Om **AInterface** specificerar **A:s** beteende och
- ▶ om **BInterface** specificerar **B:s** beteende
- ▶ så kommer **Composition** – objekt att bete sig som både **A** och **B**

Några designprinciper

- ▶ SRP – Single Responsibility Principle
- ▶ OCP – Open-Closed Principle
- ▶ DIP – Dependency Inversion Principle
- ▶ ISP – Interface Segregation Principle
- ▶ LSP – Liskov Substitution Principle

SRP Single Responsibility Principle

Lätt att säga men svårt att göra

- ▶ A class has a single responsibility: It does it all, does it well and does it only **[Bertrand Meyer]**
- ▶ Kan klassen beskrivas med högst 25 ord, utan "och" eller "eller" ?
- ▶ Olika ansvarsområden karakteriseras av olika skäl till förändring. En klass ska ha endast en "förändringsaxel"

OCP Open–Closed Principle

"A class should be closed for modification but open for extension"

- ▶ Klientklass använder interface.
Implementerande klasser står för utvidgningarna.
- ▶ Basklass definierar metoder.
Subklass utvidgar genom att
 - ▶ definiera om metoder
 - ▶ definiera nya metoder

DIP Dependency – Inversion Principle

- ▶ High-level modules should not depend on low-level modules. Both should depend on abstractions.
- ▶ Abstraction should not depend on detail. Details should depend on abstractions.

Exempel på DIP

```
void regulate
(double minTemp, double maxTemp) {
    for (;;) {
        while (in(THERMO) > minTemp)
            wait(1);
        out(HEATER, ENGAGE);
        while (in(THERMO) < maxTemp)
            wait(1);
        out(HEATER, DISENGAGE);
    }
}
```

Termostaten beror av lågnivåmodulerna termometer och värmare

Bättre struktur

```
void regulate
(Thermometer t, Heater h,
 double minTemp, double maxTemp) {
    for (;;) {
        while (t.read() > minTemp)
            wait(1);
        h.engage();
        while (t.read() < maxTemp)
            wait(1);
        h.disengage();
    }
}
```

Här beror högnivåmodulen (regulate) och lågnivåmodulerna (termometer, värmare) på abstraktioner.

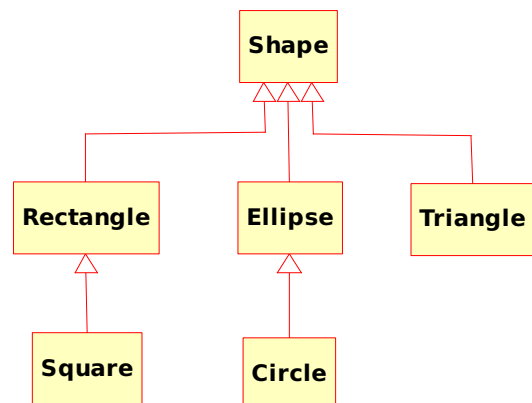
ISP Interface – Segregation Principle

Clients should not be forced to depend on methods they do not use

LSP Liskov Substitution Principle

- ▶ "Subtypes must be substitutable for their base types"
- ▶ Ett objekt av en subklass ska kunna användas där objekt av basklassen/superklassen används

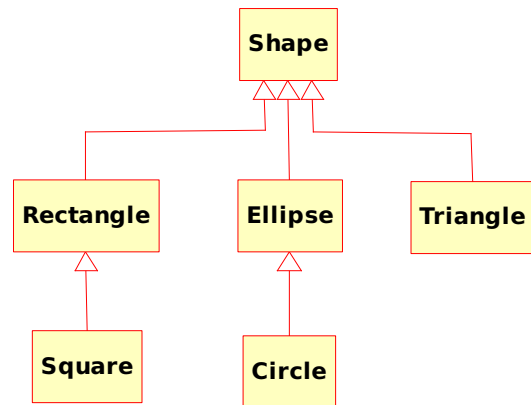
LSP motiveras med ett exempel



```

public class Rectangle extends Shape {
    private int width, height;
    public void setWidth (int w) {
        width = w;
    }
    public void setHeight (int h) {
        height = h;
    }
    public int getWidth () {
        return width;
    }
    public int getHeight () {
        return height;
    }
    public int area () {
        return width*height;
    }
}

```



Square ärver från Rectangle, där **width == height**

```

public class Square extends Rectangle{
    public void setWidth (int w) {
        super.setWidth(w);
        super.setHeight(w);
    }
    public void setHeight (int h) {
        super.setHeight(h);
        super.setWidth(h);
    }
}

```

Lite slöaktigt att alla kvadrater har ett extra datafält!

Ett testprogram

```

class Test {
    static void test (Rectangle r) {
        r.setWidth(4);
        r.setHeight(5);
        System.out.println
            ("Area is " + r.area() +
             " should be 20");
    }

    public static void main (String[] a) {
        test (new Rectangle());
        test (new Square());
    }
}

```

Utskrift från Test blir

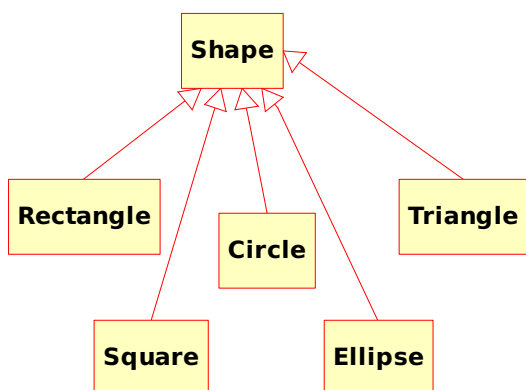
Area is 20 should be 20
 Area is 25 should be 20

LSP

Liskov Substitution Principle

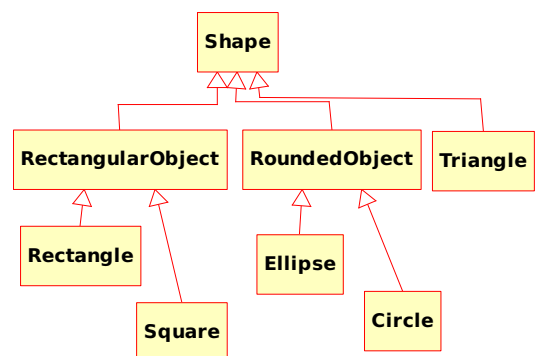
- ▶ "Subtypes must be substitutable for their base types"
- ▶ Ett objekt av en subclass ska kunna användas där objekt av basklassen/superklassen används

Square & Rectangle bryter mot LSP eftersom
Square inte kan ersätta Rectangle
i exemplet.



Vad betyder relationen "är en" ?

- ▶ En Square **är en** Rectangle
- ▶ Men en Square *uppför sig inte* som en Rectangle !!!
- ▶ En Rectangle har egenskapen att width och height kan ges värden oberoende av varandra.
- ▶ Bryt mot LSP endast vid signifikant vinst!
Exempel?



RectangularObject

```
abstract class RectangularObject extends Shape{  
    abstract int area ();  
    abstract int width();  
    abstract int height();  
}
```

Rectangle

```
class Rectangle extends RectangularObject{  
    int width, height;  
  
    void setWidth (int w) { width = w; }  
    void setHeight (int h) { height = h; }  
  
    int width() { return width; }  
    int height() { return height; }  
  
    int area () {  
        return width*height;  
    }  
}
```

Square

```
class Square extends RectangularObject {  
    int size;  
    void setSize (int s) {  
        size = s;  
    }  
    int width() {  
        return size;  
    }  
    int height() {  
        return size;  
    }  
    int area () {  
        return size*size;  
    }  
}
```