TDA550 – Objektorienterad programvaruutveckling IT, forts. kurs
Övning vecka 3

Pelle Evensen, Daniel Wetterbro

14 november 2011

Sammanfattning

Denna vecka ska vi titta på polymorfism, dynamisk kontra statisk bindning, implementationsarv, `equals` samt (mycket) lite om klassinvarianter.
Övningarna är graderade (något subjektivt) från lätt (*) till svår (***). Svårighetsgraden hos en övning har inte nödvändigtvis med lösningens storlek att göra.

1 Liskovs substitutionsprincip (LSP)

LSP säger att alla objekt av en subtyp **U** ska kunna fungera som substitut för objekt av vilken som helst av **U**:s supertyper.

1.1 Specialisering eller generalisering I? *

I fig. 1 ges källkoden för klasserna **IntegerClass** och **ObjectClass**. Då vi ser att de är så gott som identiska, hade vi kunnat låta den ena vara en subklass till den andra? Om så är fallet, spelar det någon roll om det är **IntegerClass** som är subklass till **ObjectClass** eller tvärtom?

```
class IntegerClass {
    public static Integer square(int x) {
        return new Integer(x * x);
    }
}

class ObjectClass {
    public static Object square(int x) {
        return new Integer(x * x);
    }
}
```

Figur 1: Klasserna **IntegerClass** och **ObjectClass**.

1.2 Specialisering eller generalisering II? *

I fig. 2 ges källkoden för klasserna **SuperClass** och **SubClass**. När vi använder klassen **SubClass** vill vi inte att man ska kunna komma åt värdet direkt utan bara dess kvadrat. Detta har vi försökt åstadkomma genom att deklarera metoden `getValue` i **SubClass** `private`.

Varför går det inte att kompilera klassen **SubClass**?

```
class SuperClass {
    private final int value;

    public SuperClass(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

class SubClass extends SuperClass {
    public SubClass(int x) {
        super(x);
    }

    private int getValue() {
        return super.getValue();
    }

    public int getSquare() {
        return getValue() * getValue();
    }
}
```

Figur 2: Klasserna **SuperClass** och **SubClass**.

2 Polymorfism och dynamisk typ **

2.1 Dynamisk kontra statisk bindning I

Klassen **Binding1** (fig. 4) innehåller metoder för utskrift av olika typer av objekt. Alla klasser i Java är subklasser till klassen **Object** som innehåller metoden `toString()`.

Metoden `printValue` är *överlagrad* och skiljer sig bara med avseende på typen hos den formella parametern. Metoden `toString` är däremot *överskuggad*. Givet definitionerna för klasserna **A**, **B** och **S** (fig. 3), vad kommer utskriften att bli när vi kör `Binding1.main`?

Vilka metoder i vilka klasser kommer att exekveras vid körningen av `Binding1.main` och framförallt; varför anropas just de metoderna?

```
class S { public String toString() { return "I am an S!"; } }

class A extends S { public String toString() { return "I am an A!"; } }

class B extends A { public String toString() { return "I am a B!"; } }
```

Figur 3: Klasserna **A**, **B** och **S**.

2.2 Dynamisk kontra statisk bindning II

Studera klassen **Binding2** (fig. 5). Antag att klasserna **A**, **B** och **S** är som tidigare givna i fig. 3. Vad blir utskriften när `Binding2.main` körs? Varför?

```
public class Binding1 {
    public static void printValue(A v) { System.out.println("This was an A: " + v.toString()); }

    public static void printValue(B v) { System.out.println("This was a B: " + v.toString()); }

    public static void printValue(S v) {
        System.out.print("That was an S");
        if (v.getClass() != S.class) {
            System.out.print(", subclassed by " + v.getClass() + ": ");
        } else {
            System.out.print(", not a strict subclass of S." + ": ");
        }
        System.out.println(v.toString());
    }

    public static void main(String[] args) {
        S spa = new S();
        A apa = new A();
        B bepa = new B();
        S apalt = apa;
        A bepalt = bepa;

        printValue(spa);
        printValue(apa);
        printValue(bepa);
        printValue(apalt);
        printValue(bepalt);
    }
}
```

Figur 4: Klassen **Binding1**.

```
public class Binding2 {
    public static void main(String[] args) {
        S[] objs1 = new S[] { new A(), new B(), new S() };
        for (S o : objs1) {
            Binding1.printValue(o);
        }

        A[] objs2 = new A[] { new A(), new B() };
        for (S o : objs2) {
            Binding1.printValue(o);
        }

        for (A o : objs2) {
            Binding1.printValue(o);
        }
    }
}
```

Figur 5: Klassen **Binding2**.

3 Klassinvarianter & aliasing ***

Klassen **Period** representerar ett datumintervall genom att innehålla ett startdatum och ett slutdatum. I kommentarerna till klassen visas dessutom vilket kontrakt implementationen av klassen måste uppfylla ("Programming by Contract"). En invariant är ett villkor som alltid måste vara uppfyllt, ett precondition utgör krav som måste vara uppfyllda innan en metod anropas och postcondition visar samband som gäller efter att metoden körts.

Några klassinvarianter för **Period** (vilka?) medför att klassen är *icke-muterbar*. Att klassen är icke-muterbar innebär att när ett objekt av klassen väl instantierats så kommer det under resten av sin livslängd ha samma värden som då det skapades.

Har programmeraren gjort sitt jobb eller finns det sätt att bryta mot klassens invarianter? Om så är fallet, hur kan vi åtgärda felen?

```
/**
 * @invariant getStart() is before or at the same time as getEnd()
 * @invariant getStart() consistently returns the same value after object creation
 * @invariant getEnd() consistently returns the same value after object creation
 */
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param start the beginning of the period
     * @param end the end of the period; must not precede start
     * @pre start <= end
     * @post The time span of the returned period is positive.
     * @throws IllegalArgumentException if start is after end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0) {
            throw new IllegalArgumentException(start + " after " + end);
        }
        this.start = start;
        this.end = end;
    }

    public Date getStart() {
        return start;
    }

    public Date getEnd() {
        return end;
    }
}
```

Figur 6: Klassen **Period**.

4 Likhet (`equals`)

För metoden `equals` förväntar vi oss typiskt att dessa egenskaper håller;

- *Reflexivitet:* `a.equals(a)`
- *Symmetri:* `a.equals(b)` $\implies$ `b.equals(a)`
- *Transitivitet:* `a.equals(b)` $\wedge$ `b.equals(c)` $\implies$ `a.equals(c)`

4.1 Relationsdrama[1] I ***

Betrakta klassen **CaseInsensitiveString** (fig. 8). Finns det några omständigheter under vilka de tre önskvärda relationerna inte gäller för klassen? Vad händer om man jämför med en **MyString**? Med en **CaseInsensitiveString**? Spelar ordningen någon roll?

Om det finns något problem, vad borde man gjort istället?

```
public class MyString {
    private final String s;

    public MyString(String s) {
        if (s == null) {
            throw new NullPointerException();
        }
        this.s = s;
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof MyString) {
            MyString os = (MyString) o;
            return s.equals(os.s);
        }
        return false;
    }

    public String asString() {    return s;  }
}
```

Figur 7: **MyString**.

```
public final class CaseInsensitiveString
        extends MyString {

    public CaseInsensitiveString(String s) {
        super(s);
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof MyString) {
            MyString os = (MyString) o;
            if (os instanceof
                    CaseInsensitiveString) {
                return asString()
                  .equalsIgnoreCase(os.asString());
            }
            return super.equals(o);
        }
        return false;
    }
}
```

Figur 8: **CaseInsensitiveString**.

[1] Lånad från [Blo08]

4.2 Relationsdrama II ***

Betrakta klasserna **Point** (fig. 9) och **NamedPoint** (fig. 10). Håller relationerna beskrivna i sektion 4 för dessa två klasser? Om inte, vilken eller vilka relationer håller inte? Om någon av relationerna inte håller, hur borde man implementerat `equals` istället?

```
public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Point)) {
            return false;
        } else {
            Point p = (Point) o;
            return p.x == x && p.y == y;
        }
    }
}
```

Figur 9: Klassen **Point**.

```
public class NamedPoint extends Point {
    private final String name;

    public NamedPoint(int x, int y,
            String name) {
        super(x, y);
        this.name = name;
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Point)) {
            return false;
        }
        // If o is a normal Point, do
        // a name-blind comparison
        if (!(o instanceof NamedPoint)) {
            return o.equals(this);
        }
        // o is a NamedPoint; do
        // a full comparison
        return super.equals(o)
            && ((NamedPoint) o).name == name;
    }
}
```

Figur 10: Klassen **NamedPoint**.

Referenser

[Blo08] Joshua Bloch. *Effective Java.* Addison-Wesley, 2nd edition, 2008.