TDA550 – Objektorienterad programvaruutveckling IT, forts. kurs
Övning vecka 4

Pelle Evensen *

16 oktober 2012

Sammanfattning

Denna vecka ska vi titta på gränssnitten **Comparable** och **Comparator** samt exceptions, strategy-designmönstret samt icke-muterbarhet kontra muterbarhet. Vi gör också små återbesök på temat "programming by contract".
Övningarna är graderade (något subjektivt) från lätt (*) till svår (***). Svårighetsgraden hos en övning har inte nödvändigtvis med lösningens storlek att göra.

1 Comparator & Comparable

Vi kan för en viss klass göra så att den får en *total ordning* genom att låta den implementera gränssnittet **Comparable**. I de fall då vi vill tillhandahålla mer än en ordning för en klass använder vi oss av gränssnittet **Comparator**. Detta kan vi så klart också använda för att ordna objekt av en klass som inte redan implementerar **Comparable**.

*Exceptionsövningar ursprungligen lånade av Bror Bjerner

1.1 Implementation av Comparable *

I fig. 1 ges klassen **Car**. Vi vill göra så att klassen vid sortering sorteras i stigande ordning jämförandes följande variabler;

1. modelYear
2. manufacturer
3. modelName

Om två bilar är av samma årsmodell så jämför man också tillverkare, om tillverkare också är lika, jämför också modellnamn.

Skriv om `compareTo` i **Car** så att den får denna totala ordning.

```
public final class Car implements Comparable<Car> {
    private final String manufacturer;
    private final String modelName;
    private final int modelYear;
    private final String serialNumber;

    // Accessors go here.

    public String toString() {
        return getClass().toString() + " manufacturer: " + this.manufacturer+
        " modelName: " + this.modelName +
        " modelYear: " + this.modelYear +
        " serialNumber: " + this.serialNumber;
    }

    public Car(String manufacturer, String modelName, int modelYear,
           String serialNumber) {
        this.manufacturer = manufacturer;
        this.modelName = modelName;
        this.modelYear = modelYear;
        this.serialNumber = serialNumber;
    }

    public int compareTo(Car c) {
        // Your code here.
        throw new UnsupportedOperationException("compareTo missing! Help!");
    }

    public boolean equals(Object o) {
        // Your code here.
        throw new UnsupportedOperationException("equals missing! Help!");
    }
}
```

Figur 1: Klassen **Car**.

1.2 Rimliga förvillkor för en mer flexibel Comparator ***

Det kan tänkas att man inte kan förutspå alla totala ordningar som kan vara av intresse vid en senare tidpunkt.

Ett vanligt fall är då vi i en applikation visar en tabell och man genom att klicka på kolumnernas rubriker bestämmer om posterna ska sorteras i stigande eller fallande ordning med avseende på värdet i kolumnen. Givet att vi har n kolumner så kan dessa sorteras (med ett av alternative stigande *eller* fallande) på $n!$ olika sätt.

Om vi dessutom lägger till egenskapen att vi ska kunna ha både stigande och fallande ordning för varje fält individuellt så får vi $2^n n!$ möjliga ordningar. Även för små n blir detta snabbt ohanterligt. Vi skulle t.ex. behöva *384 olika komparatorer* för att få alla totala ordningar av 4 fält ($2^4 4! = 16 \times 24 = 384$).

```
public enum Field {
    MANUFACTURER, MODELNAME, MODELYEAR, SERIAL
}
```

```
public enum Order {
    ASCENDING, DESCENDING
}
```

Figur 2: Enum-typerna **Field** och **Order**.

```
/**
 * A comparator that can provide all total orderings for a Car-object.
 * @inv Describe reasonable class invariants.
 */
class CarComparator implements Comparator<Car> {
    private final Field[] f;
    private final Order[] o;

    /**
     * @pre Describe reasonable preconditions.
     */
    public CarComparator(Field[] f, Order[] o) {
        // Your implementation.
    }

    public int compare(Car c1, Car c2) {
        // Your implementation.
        throw new UnsupportedOperationException("compare missing! Help!");
    }
}
```

Figur 3: Klassen **CarComparator**.

1.2.1 Förvillkor och kontrakt?

Klassen **CarComparator** (fig. 3) är ett skelett för att kunna tillhandahålla alla möjliga komparatorer för att ordna objekt av typ **Car**.

Vilka förvillkor (preconditions) för konstruktorn är rimliga? Har t.ex. alla par av fält och ordning en vettig tolkning? Ger alla unika val av t.ex. par och tripplar distinkta totala ordningar?

Tips: Jämför storleken på definitionsmängden för konstruktorn med antalet totala ordningar för 2 fält. Definitionsmängdens storlek är antalet element i den cartesiska produkten, alltså antalet möjliga distinkta tupler för $parameter_1$ gånger antalet möjliga indata för $parameter_2$, et.c. Om vi t.ex. har en metod `method(int a, int b)` så är definitionsmängdens storlek $2^{32} \times 2^{32} = 2^{64}$ – en `int` i Java har ju 32 bitar. Två `int` efter varandra blir då 64 bitar som tillsammans kan anta 2^{64} olika värden.

Den cartesiska produkten för *ett* jämförelsekriterium är alltså i detta fall antalet datafält (tillverkare, modellnamn, årsmodell och serienummer) gånger antalet ordningar för dessa fält (stigande, fallande). Om vi kallar storleken på denna mängd p så blir definitionsmängdens storlek för n jämförelsekriteria p^n.

Det bör gå att ställa upp åtminstone sju separata förvillkor för f och o tillsammans.

För att skapa en någorlunda flexibel, om än kanske inte så snabb, lösning på det allmäna fallet passar decorator-designmönstret utmärkt. Om detta ännu inte gåtts igenom på kursen kan man senare återkomma hit och fundera på hur en sådan lösning kan se ut.

2 Strategi-designmönstret

Strategier är en lämplig taktik att ta till då bara delar av en implementation varierar. Gränssnittet **Comparator** är ett exempel inom Javas standard-API som är en ren strategi. I en typisk sorteringsalgoritm är det lämpligt att bryta ut själva jämförelsen av elementen från resten av algoritmen. Som vi sett i problemet med **CarComparator** ovan så kan jämförelsekriterierna också förändras under körning.[1].

2.1 Numerisk integrering *

Metoden `integrate` i klassen **Integrator** (fig. 4, sid. 5) gör en numerisk skattning av integralen $\int_a^b f_k(x)\,dx$ där $f_k(x)$ är någon av funktionerna $f_1(x) = \log(\log(x))$, $f_2(x) = x^x$, $f_3(x) = \sin(\sin(x))$.

Ett allvarligt problem med denna klass är att vi måste förändra den varje gång vi numeriskt vill integrera en ny funktion, trots att metoden för själva integrationen inte förändras. Vi kan därmed inte skapa nya funktioner under körning.

Skriv om klassen **Integrator** så att den istället för fasta funktioner tar en strategi som parameter till metoden `integrate`. Du behöver bara skapa ett korrekt

[1] Se gärna Item 21 i [Blo08]

gränssnitt (interface) samt förändra metoden `integrate` på lämpligt vis.

Tips: Enum-typen **Function** ska inte finnas med.

```
public class Integrator {
    public enum Function {
        FUNCTION1, FUNCTION2, FUNCTION3
    };

    public static double integrate(Function f, double a, double b,
            int steps) {
        double sum = 0.0;
        double previous = fun(f, a);
        double delta = (b - a) / steps;

        for (int i = 1; i <= steps; i++) {
            double x = a + (b - a) * i / steps;
            double current = fun(f, x);

            // Compute the integral through Simpson's 3/8 rule.
            sum += delta / 8 * (previous + current + 3 *
                        (fun(f, (2 * (x - delta) + x) / 3.0) +
                            fun(f, (3 * x - delta) / 3.0)));
            previous = current;
        }

        return sum;
    }

    private static double fun(Function f, double x) {
        switch (f) {
            case FUNCTION1: { return Math.log(Math.log(x)); }
            case FUNCTION2: { return Math.pow(x, x); }
            case FUNCTION3: { return Math.sin(Math.sin(x)); }
            default:
                throw new IllegalArgumentException("Illegal function!");
        }
    }
}
```

Figur 4: Klassen **Integrator**.

3 Exceptions

3.1 Propagering av fel uppför anropsstacken **

En fördel med exceptions är möjligheten att propagera felrapportering uppåt i anropsstacken. Betrakta **PseudoKod1** (sid. 6). Antag att metoden `readFile()` är den fjärde metoden som anropas i en serie av nästlade anrop från vår main-metod: `metod1()` anropar `metod2()`, som anropar `metod3()`, vilken slutligen anropar `readFile()`.

Antag vidare att `metod1()` i **PseudoKod1** är den enda metod som är intressant för hantering av något fel som uppstått i `readFile()`.

3.1.1 Om vi inte haft exceptions?

Gör en implementation i pseudo-kod av dessa metoder utan att använda exceptions, d.v.s. komplettera **PseudoKod2** (sid. 6) med metoderna `metod2` och `metod3`.

Antag att `readFile()` rapporterar ett fel på samma sätt som `metod2()` gör i **PseudoKod2**.

3.1.2 Exceptions räddar dagen?

Hur kan nu koden i **PseudoKod2** skrivas om med användning av exceptions? Antag att vi kan använda oss av en exception-typ **ReadFileFailedException**.

3.2 Arv av exceptions *

Antag att du i Java har definierat en egen klass **NotANumberException** som en subklass till **Exception**. Vidare har du definierat ytterligare en klass **NotAPositiveNumberException** som en subklass till **NotANumberException**.

```
void metod1() {
    do something A;
    metod2();
    do something B;
}

void metod2() {
    do something C;
    metod3();
    do something D;
}

void metod3() {
    do something E;
    readFile();
    do something F;
}
```

Figur 5: Klassen **PseudoKod1**.

```
do something A;
error = metod2();
if (error)
  doErrorProcessing;
else
  do something B;
}
...
...
```

Figur 6: Klassen **PseudoKod2**.

3.2.1 Vad fångas I?

Kommer koden i **Catcher1** att fånga ett **NotAPositiveNumberException**? Varför/Varför inte?

3.2.2 Vad fångas II?

Kommer koden i **Catcher2** att fungera? Varför/Varför inte?

```
catch(NotANumberException nane) {
  System.out.println("Fångade ett " +
          "NotANumberException");
}
```

Figur 7: Klassen **Catcher1**.

```
try {
    ...
}
catch(NotANumberException nane) {
    System.out.println("Fångade " +
        "ett NotANumberException");
}
catch(NotAPositiveNumberException napne) {
    System.out.println("Fångade ett " +
        "NotAPositiveNumberException");
}
```

Figur 8: Klassen **Catcher2**.

4 (Icke-)muterbarhet

Betrakta klassen **Pair** (fig. 9, sid. 8). Avsikten med denna klass är att den ska kunna fungera då man behöver en ad-hoc sammansättning av data där det inte finns någon tydlig operation på just denna kombination av data. Denna klass tillåter sammansättning av just par.

Då Java inte har något inbyggt stöd för *tupler*[2] kan detta (för specialfall, t.ex. par, triplar, et.c.) simuleras på detta vis. Vi har inte pratat om *generics* på kursen ännu. Om du inte har en klar bild av vad `<A, B>` i klassdeklarationen betyder, strunta i det, man måste inte förstå det för denna uppgift.

Ett exempel på användning finns i (den något artificiella) klassen **MinAndSize** (fig. 10, sid. 9).

4.1 Publika instansvariabler? *

Klassen **Pair** har publika instansvariabler. Detta bryter rimligtvis mot god sed med avseende på inkapsling och synlighet. Ställer detta till det på något vis i **Pair**-klassen?

4.2 Vad gör `final`? **

Variablerna `first` och `second` är deklarerade `final`. Vad är den tänkta klassinvarianten (formulera informellt) och kan man garantera att den håller? Jämför gärna med uppgift 3 från vecka 3:s övning.

Referenser

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

[2] Se t.ex. `http://en.wikipedia.org/wiki/Tuple`

```java
/**
 * Immutable class for generic pairs.
 * @inv State your expected invariant(s) here.
 */
public final class Pair<A, B> {
    /**
     * The first element of the pair.
     */
    public final A first;

    /**
     * The second element of the pair.
     */
    public final B second;

    /**
     * Makes a new pair of type <A,B>.
     *
     * @param first    the first element of the pair.
     * @param second the second element of the pair.
     */
    public Pair(final A first, final B second) {
        this.first = first;
        this.second = second;
    }

    @Override
    public boolean equals(final Object o) {
        return o instanceof Pair<?, ?>
            && ((Pair<?, ?>) o).first.equals(this.first)
            && ((Pair<?, ?>) o).second.equals(this.second);
    }

    @Override
    public int hashCode() {
        return this.first.hashCode() * 13579 + this.second.hashCode() * 23456789;
    }

    @Override
    public String toString() {
        return "Pair: <" + this.first.toString() + "," + this.second.toString() + ">";
    }
}
```

Figur 9: Klassen **Pair**.

```
public class MinAndSize<E extends Comparable<E>> {
    /**
     * Returns a pair containing the number of elements and the minimum element
     * of an iterable collection.
     */
    public Pair<Integer, E> minAndSize(final Iterable<E> elements) {
        final Iterator<E> it = elements.iterator();
        if (!it.hasNext()) {
            throw new IllegalArgumentException("min and max undefined for empty sets.");
        }

        E min = it.next();
        int i = 0;

        while (it.hasNext()) {
            E e = it.next();
            if (e.compareTo(min) < 0) {
                min = e;
            }
            i++;
        }

        return new Pair<Integer, E>(Integer.valueOf(i), min);
    }

    public static void main(final String[] args) {
        MinAndSize<String> m = new MinAndSize<String>();
        Pair<Integer, String> p = m.minAndSize(Arrays.asList(args));
        System.out.println("Elements: " + p.first + " Min: " + p.second);
    }
}
```

Figur 10: Klassen **MinAndSize**.