Objektorienterad modellering och diskreta strukturer / design

Designmönster

Lennart Andersson

Reviderad 2012–09–11

2012

Projektet Computer: Genomförande

Gruppen kommer att få e-post från sin handledare denna vecka med information om var och exakt när mötet äger rum.

Gruppen

1. formulerar svaren på åtta frågor
2. gör en design med klass- och sekvensdiagram
3. skickar in klassdiagram och källkod (24 timmar före mötet)
4. träffar sin handledare för designmöte (18-21 sept))
5. reviderar designen och implementerar
6. skickar in klassdiagram och källkod (30 september)

Merparten av projekttiden brukar behövas för designen.

Designmönster

- Command
- Composite
- Template Method
- Strategy
- Singleton
- Decorator
- Null Object
- Observer
- Factory Method
- Iterator
- Interpretor

Klipp och klistra-orienterad programmering

```
public class LoadMenuItem extends JMenuItem implements ActionListener {
    private final Gui gui;
    private String title;

    public LoadMenuItem(Gui gui, String title) {
        super(title);
        this.gui = gui;
        this.title = title;
        addActionListener(this);
    }

    public void actionPerformed(ActionEvent event) {
        try {
            FileDialog dialog = new FileDialog(gui, title, FileDialog.LOAD);
            dialog.setVisible(true);
            String file = dialog.getFile();
            String dir = dialog.getDirectory();
            dialog.dispose();
            String fullName = dir + file;
            gui.storage.load(new BufferedReader(new FileReader(fullName)));
        } catch (IOException ex) {
            gui.statusArea.setText(ex.toString());
        }
    }
}
```

Kopian

```
public class SaveMenuItem extends JMenuItem implements ActionListener {
    private final Gui gui;
    private String title;

    public SaveMenuItem(Gui gui, String title) {
        super(title);
        this.gui = gui;
        this.title = title;
        addActionListener(this);
    }

    public void actionPerformed(ActionEvent event) {
        try {
            FileDialog dialog = new FileDialog(gui, title, FileDialog.SAVE);
            dialog.setVisible(true);
            String file = dialog.getFile();
            String dir = dialog.getDirectory();
            dialog.dispose();
            String fullName = dir + file;
            gui.storage.save(new PrintStream(fullName));
        } catch (IOException ex) {
            gui.statusArea.setText(ex.toString());
        }
    }
}
```


Eliminera duplicerad kod: Försök 1

Med en flagga:

```
private boolean save;
...
if(save) {
    gui.storage.save(new PrintStream(fullName));
} else {
    gui.storage.load(new BufferedReader(
                         new FileReader(fullName)));
}
...
```

Vilken princip bryter vi mot?

Eliminera duplicerad kod: Template method

Bryt ut det som är gemensamt och placera det i en abstrakt superklass. Fyll hålen med anrop av abstrakta metoder som implementeras i subklasserna.

Klassdiagram!

De nya subklasserna

I LoadMenuItem

```
public action(Storage storage, String fullName) {
  storage.load(new BufferedReader(
                   new FileReader(fullName)));
}
```

I SaveMenuItem

```
public action(Storage storage, String fullName) {
  storage.save(new PrintStream(fullName));
}
```

Superklassen

I superklassen

protected abstract void
action(Storage storage, String fullName)
throws IOException;

och i hålet

action(gui.storage, fullName);

FileMenuItem

```
public abstract class FileMenuItem extends JMenuItem implements ActionListener {
    private final Gui gui;
    private String title;

    protected FileMenuItem(Gui gui, String title) {
        super(title);
        this.gui = gui;
        this.title = title;
        addActionListener(this);
    }

    protected abstract void action
            (Storage storage, String fullName) throws IOException;
    public void actionPerformed(ActionEvent event) {
        try {
            FileDialog dialog = new FileDialog(gui, title, FileDialog.??);
            dialog.setVisible(true);
            String file = dialog.getFile();
            String dir = dialog.getDirectory();
            dialog.dispose();
            String fullName = dir + file;
            action(gui.storage, fullName);
        } catch (IOException ex) {
            gui.statusArea.setText(ex.toString());
        }
    }
```


Ena subklassen

```
public class LoadMenuItem extends FileMenuItem {
    private String title;

    public LoadMenuItem(Gui gui, String title) {
        super(gui, title);
    }

    protected void action(Storage storage, String fullName)
                     throws IOException {
        storage.load(new BufferedReader(new FileReader(fullName));
    }
}
```


Aktivitet

Använd *Template method*-mönstret för att hantera `FileDialog.SAVE` och `FileDialog.LOAD`.

```
public abstract class FileMenuItem extends JMenuItem implements ActionListener {
    private final Gui gui;
    private String title;

    protected abstract void action
            (Storage storage, String fullName) throws IOException;
    public void actionPerformed(ActionEvent event) {
        try {
            FileDialog dialog = new FileDialog(gui, title, FileDialog.SAVE);
            dialog.setVisible(true);
            String file = dialog.getFile();
            String dir = dialog.getDirectory();
            dialog.dispose();
            String fullName = dir + file;
            action(gui.storage, fullName);
        } catch (IOException ex) {
            gui.statusArea.setText(ex.toString());
        }
    }
}
```

Template method

`Add`- och `Mul`-klasserna i `Expr` innehåller duplicerad kod. Använd *Template method*-mönstret för att eliminera den!

Add

```
public class Add implements Expr {
    private Expr expr1, expr2;
    public Add(Expr expr1, Expr expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public int value() {
        return expr1.value() + expr2.value();
    }
}
```


Mul

```
public class Mul implements Expr {
    private Expr expr1, expr2;
    public Mul(Expr expr1, Expr expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public int value() {
        return expr1.value() * expr2.value();
    }
}
```


BinExpr – Gemensam superklass

```
public abstract class BinExpr implements Expr {
    private Expr expr1, expr2;
    protected BinExpr(Expr expr1, Expr expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    protected abstract int op(int int1, int int2);
    public int value() {
        return op(expr1.value(), expr2.value());
    }
}
```

Add

```
public class Add extends BinExpr {
    public Add(Expr expr1, Expr expr2) {
        super(expr1, expr2);
    }
    protected int op(int int1, int int2) {
        return int1 + int2;
    }
}
```


Template method

Template method-mönstret är olämpligt att använda när det är mer än funktionalitet som kan variera.

Om det i Expr-klasserna finns 5 olika operationer och 4 typer att räkna med (int, long, float, double) så blir det 4*5=20 subklasser.

Använd *Strategy*-mönstret (kommer strax) för tillkommande funktionaliteter.

Strategy

Strategimönstret användas när man vill kunna ändra hur en operation utförs under exekveringen av programmet; man byter strategi.

Ett exempel visar hur man definierar strategier för att skriva ut listor med olika prefix.

Strategy

Först själva listan:

```
public class List<T> extends ArrayList<T> {
    private Prefix prefix = new Empty();

    public void setPrefix(Prefix prefix) {
        this.prefix = prefix;
    }

    public String toString() {
        StringBuilder builder = new StringBuilder();
        for (T t : this) {
            builder.append(prefix.string());
            builder.append(t).append('\n');
        }
        return builder.toString();
    }
}
```

Strategin

```
public interface Prefix {
    public String string();
}
```


Strategiklassserna

```
public class Star implements Prefix {
    public String string() {
        return "* ";
    }
}
```

```
public class Numbered implements Prefix {
    private int number;
    public String string() {
        number++;
        return String.valueOf(number) + " ";
    }
}
```


Star-strategin

Alla `Star`-objekt är likadana och kan inte modifieras. Det behövs bara ett.

```
public class Star implements Prefix {
    private class Star() {}
    public final static Prefix star = new Star();
    public String string() {
        return "* ";
    }
}
```


Aktivitet

Rita ett klassdiagram!

En variant av Strategy

Normalt är strategin ett attribut i klassen, men man kan i stället skicka med den som ett argument till en metod som behöver den.

```
public String toString(Prefix prefix) {
    StringBuilder builder = new StringBuilder();
    for (T t : this) {
        builder.append(prefix.string());
        builder.append(t).append('\n');
    }
    return builder.toString();
}
```


Jojo-kort med två strategier

```
public class TravelCard {
    private Tariff tariff;

    public double price(int zones, Rebate rebate) {
        double amount = tariff.price(zones);
        return rebate.price(amount);
    }

    public void setTariff(Tariff tariff) {
        this.tariff = tariff;
    }
}
```

`double` **bör ersättas med** `Money`

Två strategi-interface

```
public interface Tariff {
    public double price(int zones);
}

public interface Rebate {
    public double price(double price);
}
```


Två strategier

```
public class Skane implements Tariff {
    public double price(int zones) {
        switch (zones) {
        case 1:
            return 17.0;
        default:
            return 7.0 * zones + 7.0;
        }
    }
}

public class Family implements Rebate {
    public double price(double price) {
        return 1.5 * price;
    }
}
```

Strategierna har inget tillstånd

```
public final static Tariff JOJO = new Skane();
public final static Rebate DUO = new Family();

...

        TravelCard jojo = new TravelCard();
        jojo.setTariff(SKANE);
        double price = jojo.price(3, DUO);
```


Template method eller Strategy?

Båda mönstren kan användas för att eliminera duplicerad kod.

Använd Template method när funktionaliteten skall vara den samma under objektets hela livstid.

Använd Strategy när funktionaliteten skall kunna förändras under livstiden eller när det finns mer än en funktionalitet som kan variera.

Flera variabla funktionaliteter

```
public interface Expr {
    public Number value();
}

public class Floating implements Expr {
    private float value;

    public Floating(float value) {
        this.value = value;
    }

    public Number value() {
        return value;
    }
}
```


Flera variabla funktionaliteter

```
public class IntAdd extends BinOp {
    public Number value() {
        return (Integer) expr1.value()
                + (Integer) expr2.value();
    }
}

public class FloatAdd extends BinOp {
    public Number value() {
        return (Float) expr1.value()
                + (Float) expr2.value();
    }
}
```

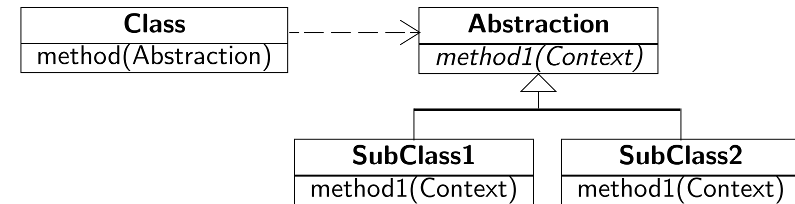
Flera variabla funktionaliteter

- 4 typer: int, long, float, double
- 4 binära opereatorer: +, -, *, /
- 4 basfall + 4 operationer $*$ 4 kombinationer $=$ 20 subklasser är inte bra.

Lösning? Jämför Computer-projektet.

Strategimönstret används överallt

Strategistrukturen återkommer ofta i sammanhang där namnet *strategi* kan vara konstigt.



TANSTAAFL

TANSTAAFL

Martin, sidan 319.

There ain't no such thing as a free lunch.

Att införa ett designmönster är inte gratis. Det måste finnas ett bra skäl att använda det. Man använder inte strategimönstret om det inte finns minst två strategier.

Singleton

- Hackerns favoritmönster.
- Syfte: att det bara skall kunna skapas en instans av klassen.

Konventionell Singleton

```
public class Singleton {
    private static Singleton instance;
    // attributes omitted
    private Singleton() {
        // omissions
    }
    public static Singleton instance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
    // other methods omitted
}
```

Singleton

- Bieffekt av den konventionella implementeringen: instansen blir ett globalt åtkomligt objekt.
- Globala objekt är bekväma att använda men innebär i regel dålig design.
- Konventionell Singleton får inte användas i kursen för klasser som har ett tillstånd!

Singleton

På nästa övning skall vi implementera Singleton-mönstret utan att skapa globalt tillgängliga objekt.

Misstaget som kostat 1 000 000 000 $

The billion dollar mistake

Tony Hoare

Tony Hoare, Inventor of QuickSort, Turing Award Winner

Sir Charles Antony Richard Hoare (Tony Hoare or C.A.R. Hoare, born January 11, 1934) is a British computer scientist, probably best known for the development in 1960 of Quicksort (or Hoaresort), one of the world's most widely used sorting algorithms.

He also developed Hoare logic for verifying program correctness, and the formal language Communicating Sequential Processes (CSP) used to specify the interactions of concurrent processes (including the Dining philosophers problem) and the inspiration for the Occam programming language.

Abstract:

Presentation: "Null References: The Billion Dollar Mistake"

Track: Historically bad ideas

Time: Friday 13:00 - 14:00

Location: Abbey Room

Abstract: I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. In recent years, a number of program analysers like PREfix and PREfast in Microsoft have been used to check references, and give warnings if there is a risk they may be non-null. More recent programming languages like Spec# have introduced declarations for non-null references. This is the solution, which I rejected in 1965.

Gå på konferensen!

www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

Null Object

- Introduktionen av `null` i objektorienterade språk var ett fundamentalt misstag.
- `null` modellerar något som inte finns. Är inte det befängt?
- Länkade listor och träd avslutas ofta med `null`.
- `null` är inget objekt; att använda `null` för att representera "ingenting" är ett exempel på icke-objektorienterad programmering.

Konventionell länkad lista

```
public class Node {
    private Object object;
    private Node next;
}

public class List {
    private Node first;
    public int length() {
        int length = 0;
        Node node = first;
        while (node != null) {
            length++;
            node = node.next();
        return length;
    }
}
```


Null object

- Detta är en maskerad form av `instanceof`.
- next() är en getter.
- Node saknar intelligens.
- Programkoden blir enklare om man istället använder ett riktigt objekt. Detta är ett exempel på mönstret *Null Object*.

Objektorienterad beskrivning

En *lista* är antingen *tom* eller består av en *nod* med ett *element* som är ett Object och en *svans* som är en *lista*.

Objektorienterad representation

```
public interface List {
    public int length();
}
public class Empty implements List {
    public int length() {
        return 0;
    }
}

public class Node implements List{
    private Object object;
    private List tail;
    public int length() {
        return 1 + tail.length();
    }
}
```

Objektorienterad listimplementering

- Man ersätter iteration med rekursion; det gör programmet logiskt enklare; testet av det logiska villkoret i `while`-satsen försvinner.
- `ArrayList` är nästan alltid bättre än länkade listor.
- I nästan alla andra sammanhang är det bra design att representera något som är tomt med ett riktigt objekt.

ISP — Segregation

Interface Segregation Principle

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

java.util.Collection

```
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[ ] toArray();
   <T> T[ ] toArray(T[ ] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    void clear();
    boolean equals(Object o);
    int hashCode();
}
```


Collection javadoc

The "destructive" methods contained in this interface, that is, the methods that modify the collection on which they operate, are specified to throw UnsupportedOperationException if this collection does not support the operation. If this is the case, these methods may, but are not required to, throw an UnsupportedOperationException if the invocation would have no effect on the collection. For example, invoking the addAll(Collection) method on an unmodifiable collection may, but is not required to, throw the exception if the collection to be added is empty.

[Josh Bloch, Neal Gafter]

Collection bryter mot ISP

Alla som implementerar gränssnittet måste ha metoder som inte behöver fungera.

Hur borde det se ut?

Hur borde det se ut?

```
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[ ] toArray();
   <T> T[ ] toArray(T[ ] a);
    boolean containsAll(Collection<?> c);
    boolean equals(Object o);
    int hashCode();
}
```


Hur borde det se ut?

```
public interface ImmutableCollection<E>
    extends Collection<E> {}

public interface MutableCollection<E> extends Collection<E> {
    boolean add(E e);
    boolean remove(Object o);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    void clear();
}
```

En annan bra design finns i programspråket Scala.
Länk: www.scala-lang.org.

Vad tycker Josh och Neal?

Jag tror att de håller med. De har skrivit en bok som visar att Java har många brister och konstigheter.

Joshua Bloch, Neal Gafter: *Java Puzzlers — Traps, pitfalls, and corner cases. Addison-Wesley*, 2005, ISBN 0-321-33678-X.

Länk: www.javapuzzlers.com

Vad händer?

```
public class Elimentary {
  public static void main(String arg[ ]) {
    System.out.println(12345 + 5432l);
  }
}
```

1. Kompileringsfel.
2. Skriver 66666.
3. Skriver något annat.