Objektorienterad modellering och diskreta strukturer / design

Designmönster

Lennart Andersson

Reviderad 2011–09–08

2011

Lokaler

Föreläsningen på torsdag äger rum i MH:B.

Övningarna på onsdagar kl 10-12 är i E3319.

Projektet Computer: Specifikation

```
public class Computer {
    public Computer(Memory memory)
    public void load(Program program)
    public void run()
}
```


Projektet Computer: Test

```
public static void main(String[] args) {
    Program factorial = new Factorial();
    System.out.println(factorial);
    Computer computer = new Computer(new LongMemory(64));
    computer.load(factorial);
    computer.run();
}
```

Projektet Computer: Data

```
public class Factorial extends Program {
    public Factorial() {
        Address n = new Address(0),
                fac = new Address(1);
        add(new Copy(new LongWord(5), n));
        add(new Copy(new LongWord(1), fac));
        add(new JumpEq(6, n, new LongWord(1)));
        add(new Mul(fac, n, fac));
        add(new Add(n, new LongWord(-1), n));
        add(new Jump(2));
        add(new Print(fac));
        add(new Halt());
    }
}
```


Projektet Computer: Genomförande

Gruppen

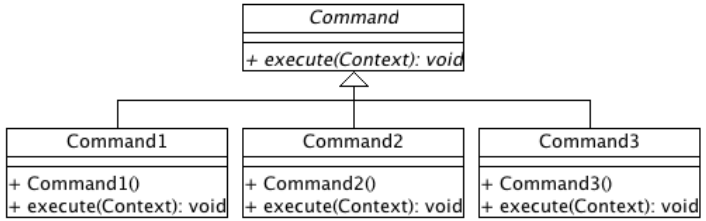
1. registrerar sig via Sam (senast 5 september deadline)
2. formulerar svaren på åtta frågor
3. gör en design med klass- och sekvensdiagram
4. skickar in klassdiagram och källkod (24 timmar före mötet)
5. träffar sin handledare för designmöte (13-15 sept))
6. reviderar designen och implementerar
7. skickar in klassdiagram och källkod (25 september)

Merparten av projekttiden brukar behövas för designen.

Designmönster

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

Command



Composite

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.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());
        }
```

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));
    }
}
```


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

Template-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 "* ";
    }
}
```

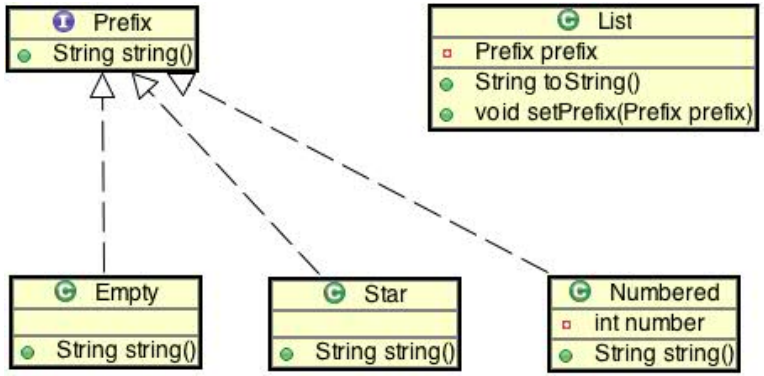
Allt kan definieras i Prefix

```
public interface Prefix {
    public String string();
    public final static Prefix star =
        new Prefix() {
            public String string() {
                return "* ";
            }
        };
    public final static Prefix empty =
        new Prefix() {
            public String string() {
                return "";
            }
        };
}
```

Svårläst — Undvik sådant!

Strategy

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;
    }
}
```



```
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.

TANSTAAFL

TANSTAAFL

Martin, sidan 319.

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

Ett designmönster är inte gratis. Det måste finnas ett 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 utan att skapa globalt tillgängliga objekt.

Decorator-mönstret

Skånetrafiken vill få en lista på alla köp av biljetter under oktober. Standardautomaten:

```
public StandardPayStation implements PayStation {
  private double sum;
  public double pay(TravelCard travelCard, int zones) {
    double price = travelCard.price(zones);
    sum += price;
    return return price;
  }
}


public interface PaySation {
  public double pay(TravelCard travelCard, int zones);
}
```


Decorator-mönstret

Den nya funktionaliteten:

```
public LoggingPayStation implements PayStation {
  private PayStation payStation;
  private List<String> log;
  public LoggingPayStation(PayStation payStation, List<Str
    this.payStation = payStation;
    this.log = log;
  }
  public double pay(TravelCard travelCard, int zones) {
    log.add(travelCard.toString() + zones);
    return return payStation.pay(travelCard, zones);
  }
}
```


Loggningen kan göras dynamiskt

```
PayStation standardPayStation = new StandardPayStation()
PayStation payStation = StandardPayStation;
```

I början av oktober lägger vi till loggningen:

```
payStation = new LoggingPayStation(standardPayStation);
```

I slutet av oktober tar vi bort den:

```
payStation = standardPaystation;
```

Tillståndet i standardPayStation bevaras.

Decorator-mönstret i Reader-klasserna

```
public class BufferedReader extends Reader {
    private Reader in;
// omissions
}

public abstract class FilterReader extends Reader {
    protected Reader in;
// omissions
}

public class PushbackReader extends FilterReader {
// omissions
}
```