Objektorienterad modellering och diskreta strukturer / design

Fler mönster och Paketdesign

Lennart Andersson

Reviderad 2011–09–12

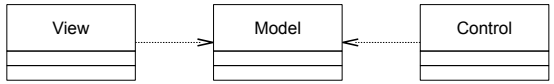
2011

Dagens program

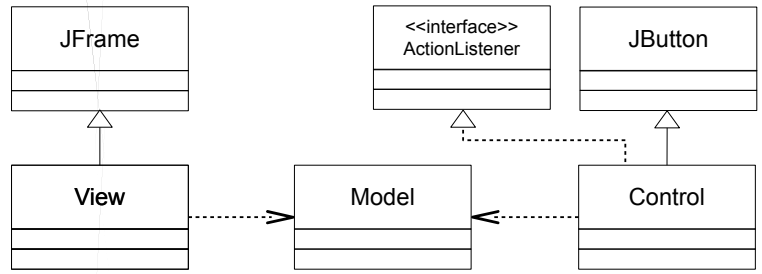
- Model–View–Control
- Observer
- Paketdesign
- Namngivning

Model/View/Control-arkitektur

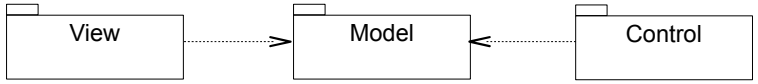
- Modellen beskriver systemets tillstånd.
- Vyn visar upp systemets tillstånd.
- Control förändrar systemets tillstånd.



Model/View/Control-implementering



Paketberoenden



Model/View/Control

Varför?

- Principen om enkelt ansvar.
- Integritet: Modellen behöver inte känna till vyn.
- Flera vyer av en modell.

Om Control saknar tillstånd slår vi ofta ihop vyn och control till ett grafiskt användargränssnitt, GUI.

Observer

- Observer-mönstret används för att separera "modellen" från användargränssnittet, "vyn".
- Vyn implementerar gränssnittet `Observer`.
- Modellen utvidgar klassen `Observable`.
- När modellens tillstånd förändras informeras alla observatörer om att det skett en förändring och vyerna uppdaterar sig genom att hämta information från modellen.
- Observer/Observable är ett *ramverk*; klasserna finns färdiga.

Observer

Finns i java.util

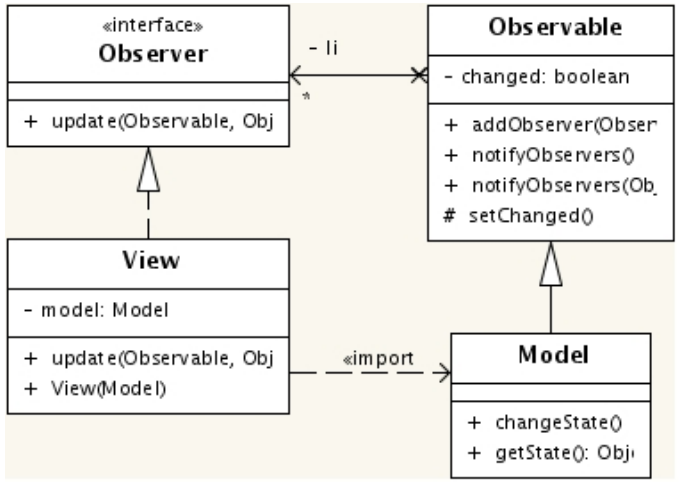
```
public interface Observer {
    public void update(Observable observable,
                       Object object);
}
```


Observable

Finns i java.util

```
public class Observable {
    public void addObserver(Observer observer)
    public void deleteObserver(Observer observer)
    protected void setChanged()
    public void notifyObservers(Object object)
    public void notifyObservers()
    // omissions
}
```

Observer–mönstret



Model

```
public class Model extends Observable {
  private int state;
  public void changeState() {
      state++;
      setChanged();
      notifyObservers();
  }
  public String state() {
    return String.valueOf(state);
  }
}
```


View

```
public class View extends JLabel implements Observer {
  private Model model;
  public View(Model model) {
    this.model = model;
    model.addObserver(this);
  }
  public void update(Observable observable, Object object)
      setText(model.state());
  }
}
```

Tydlig koppling till modellen.

java.util.Observable ...

```
public class Observable {
  private boolean changed = false;
  private Vector<Observer> list = new Vector<Observer>();
  public synchronized void addObserver(Observer o) {
    if (!list.contains(o)) {
      list.add(o);
    }
  }
  protected synchronized void setChanged() {
    changed = true;
  }
  . . .
}
```

... java.util.Observable

Något förenklad:

```
public class Observable {
  private boolean changed = false;
  private Vector<Observer> list = new Vector<Observer>();
  public void notifyObservers() {
    notifyObservers(null);
  }
  public synchronized void notifyObservers(Object arg) {
    if (!changed) {
      return;
    }
    changed = false;
    for (Observer observer: list) {
      observer.update(this, arg);
    }
  }
}
```


MVC med Observer

- En eller flera vyer registrerar sig som observatörer av modellen via `addObserver`.
- Ett kommando modifierar modellens tillstånd.
- Modellen informerar `Observable` att tillståndet förändrats via `setChanged`.
- Modellen begär att observatörerna informeras genom `notifyObservers`.
- notifyObservers i `Observable` informerar observatörerna via `update` om att modellen förändrats.
- Observatörerna hämtar modellens nya tillstånd och uppdaterar vyerna.

Observer, update-argument

```
public interface Observer {
  public void update(Observable observable, Object object);
}
```

När man anropar notifyObservers(object) i modellen kommer update(Observable, Object) att anropas i alla observatörer.

Vad skall man använda argumenten till?

update(observable, ...)

Observable kan användas för att komma åt modellen.

- Det är tydligare att ge observatören tillgång till modellen via observatörens konstruerare.
- Det kan vara praktiskt att anropa addObserver i observatörens konstruerare.

update(..., object)

Object kan användas för att skicka information från modellen till vyn.

- Hur kan modellen veta vad vyn vill ha?

  Det kan den inte veta.

- Det kan finnas flera observatörer som vill veta olika saker.

  Hur skall informationen förpackas?

  Det finns ingen naturlig plats att dokumentera förpackningen.

Designprincip för Observer.update

Undvik att använda parametrarna.

Designprinciper för paket

- Sammanhangsprinciper (cohesion)
  - Reuse-release equivalence principle
  - Common reuse principle
  - Common closure principle
- Stabilitetsprinciper
  - Acyclic dependencies principle
  - Stable dependencies principle
  - Stable abstractions principle

Reuse-release equivalence principle, REP

Återanvändning (reuse) innebär att det är någon annan som underhåller och ger ut (release) paketet. Den som använder paketet skall inte ändra på paketet och inte behöva titta på implementeringen och bestämmer själv när en ny utgåva skall integreras.
Principen säger något om storleken på paketet.

Common reuse principle

Klasserna i paketet är så nära relaterade att om man behöver en av dem så behövs i regel även de övriga.

Exempel: Instruktionerna i Computer.

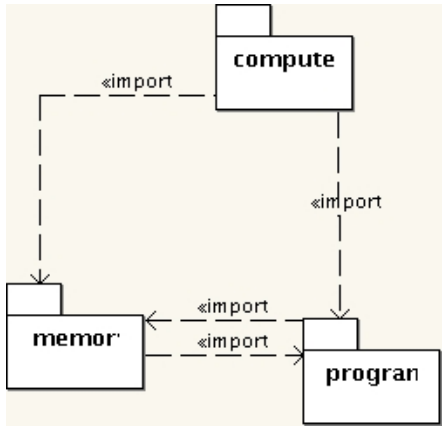
Common closure principle

Klasserna i ett paket skall vara stängda resp öppna tillsammans gentemot samma slags förändringar. Förändringar som påverkar en klass i paketet påverkar många andra klasser i paketet och inga klasser utanför paketet.

Stabilitetsprinciper

- Paket skall ej vara cykliskt beroende.
- Beroenden skall gå mot stabila paket.
- Stabila paket skall vara abstrakta.

Paketberoenden i Computer



Cykelproblem

Man kan inte bygga systemet "nerifrån".

Ett paket som ingår i en cykel går inte att återanvända utan att ta med hela cykeln och allt som cykelns paket beror på.

Med cykel

```
package a;
import b.B;
public class A {
  public void methodA(B b) {
    b.method();
  }
}
```

```
package b;
import a.A;
public class B {
  private A a;
  public void method() {
    a.methodA(this);
 }
}
```


Utan cykel

```
package a;

public interface AI {
  public void method();
}

public class A {
  public void methodA(AI ai){
    ai.method();
  }
}
```

```
package b;
import a.A;
public class B
       implements AI {
  private A a;
  public void method() {
    a.methodA(this);
 }
}
```


Stabilitetsmetrik

- Paketets *instabilitet*
$$I = \frac{C_e}{C_e + C_a}$$
- C_e antalet klasser i paketet som beror på klasser utanför paketet. *e* står för *efferent* (utåtriktad).
- C_a antalet klasser utanför paketet som beror på klasser i paketet. *e* står för *afferent* (inåtriktad).
- Om $I = 0$ för alla paket i systemet så är det inget system.

Stabilitetsmetrik

- Paketets *Abstrakthet*

$$A = \frac{N_a}{N_c}$$

- N_a antalet abstrakta klasser (gränssnitt) i paketet.
- N_c antalet klasser i paketet.

A–I grafen

- $A = I = 1$ meningslösa zonen (*useless zone*)
- $A = I = 0$ problemzonen (*zone of pain*)
- $A = 0, I = 1$ och $A = 1, I = 0$ önskvärda zoner

Paketstrukturer

- **Komponenter** Paketen är komponenter i systemet. Exempel: `memory` och `program` i `Computer`.
- **Ramverk** Systemet byggs genom att utvidga klassser i andra paket. Exempel:
  - `javax.swing`, `java.awt`. Ett användargränssnitt konstrueras genom att utvidga och fylla i klasserna i ramverket.
  - Eclipse.
  - `Observer/Observable`

Ramverk

Ramverk är mycket svårare att bygga.

> *You have to build at least three frameworks and reject them before you can be reasonably confident that you have build the right archictecture for one domain.*

[Paraphrasing Rebecca Wirfs-Brock.]

Martin: Kapitel 30. Educational Testing Service.

Namngivning

Bra namngivning är fundamental för att göra program begripliga.

```
public class MyList {
    private List<int[]> theList;

    public List<int[]> getList() {
        List<int[]> list = new ArrayList<int[]>();
        for (int[] is : theList) {
            if (is[0] < 4) {
                list.add(is);
            }
        }
        return list;
    }
}
```


Det handlar om projektgrupper

I vilka grupper finns det plats för ytterligare studenter?

```
public class Groups {
    private List<Group> groups;

    public List<Group> incompleteGroups() {
        List<Group> list = new ArrayList<Group>();
        for (Group group : groups) {
            if (group.isIncomplete()) {
                list.add(group);
            }
        }
        return list;
    }
}
```


En grupp består av fyra studenter

```
public class Group extends ArrayList<Student> {
    private static final int MAXSIZE = 4;

    public boolean isIncomplete() {
        return size() < MAXSIZE;
    }
}
```


Dokumentation av MyList

- MyList innehåller en lista av grupper av studenter.
- En grupp representeras med en "heltalsarray" där det första elementet innehåller antalet studenter i gruppen och de övriga innehåller id-nummer för studenterna.
- Metoden get returnerar en lista med de grupper som har plats för ytterligare studenter.

Namngivning av klasser

- Ett klassnamn är normalt ett substantiv hämtat från uppdragsgivarens beskrivning eller den verklighet som programmet modellerar.
- Namnet inleds med en versal och följs av gemener. På engelska använder man versal i början på varje ord när namnet är sammansatt.

Namngivning av klasser

- Konstruera inte klasser som bäst beskrivs med namn som slutar på `Handler`, `Controller`, `Manager` och liknande. De indikerar ofta dålig design. `Builder` och `Parser` kan vara helt OK för en "fabrik".
- Undvik namn som `Item`, `Element`, `Value` och `Data` om det inte handlar om generiska typer.
- Undvik att inkludera representationstypen i namnet om typen bara förekommer i ett attribut.

```
class StringList extends ArrayList<String>
```

är OK, medan

```
class StringArray {
  private String string [];
  \\ omissions
}
```

inte är det.

Namngivning av gränssnitt

- Samma principer som för klassnamn.
- Namn på gränssnitt används oftare än namn på konkreta klasser; därför är det viktigare att dessa namn är bra.
- Undvik att indikera att det handlar om gränssnitt som i `ITree` `TreeInterface`. Hierarkien som i `List`, `AbstractList` och `ArrayList` är bra när alla nivåerna behövs.

Namngivning av metoder

- Metodnamn inleds med en gemen och följande ord i sammansatta namn med versal.
- Metoder som förändrar objektets tillstånd bör ha ett verb som namn.
- Metoder som inte förändrar objektets tillstånd bör ha ett substantiv som namn eller inledas med `is` om de implementerar ett predikat.
- Namn som inleds med `set` och `get` bör reserveras för de tillfällen då det handlar om att sätta och hämta attribut.

Namngivning av attribut, lokala variabler och parametrar

- Ett attribut kan ofta ha samma namn som sin typ, fast med inledande gemen.
- Om det finns flera attribut av samma typ som används på samma sätt kan man numrera dem: `Expr expr1, expr2;`
- När en variabel används som index i en "array" använder man gärna konventionella namn som `i` och `j`.
- För strängar använder man gärna namn som indikerar användningen: `String title;`
- Använd inte `the` som ett prefix i attributnamn. När det behövs är `this.` det självklara alternativet.