Objektorienterad modellering och diskreta strukturer / design

Fler mönster och Paketdesign

Lennart Andersson

Reviderad 2012–09–17

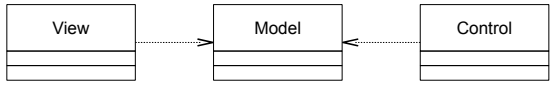
2012

Dagens program

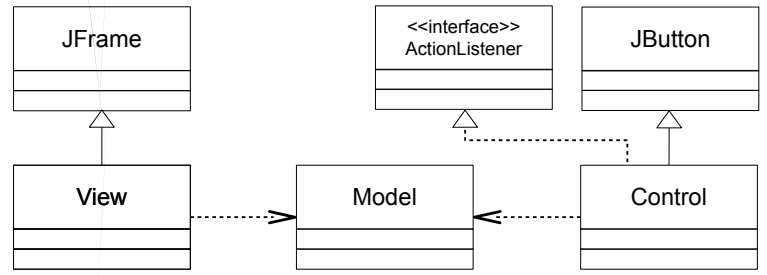
- Model–View–Control
- Observer
- Paketdesign
- Namngivning

Model/View/Control-arkitektur

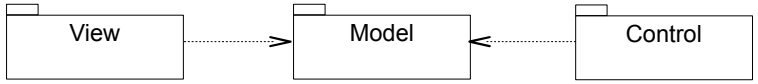
- Modellen beskriver systemets tillstånd.
- Vyn visar upp systemets tillstånd.
- Control förändrar systemets tillstånd.



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);
}
```

This method is called whenever the observed object is changed. An application calls an Observable object's notifyObservers method to have all the object's observers notified of the change.

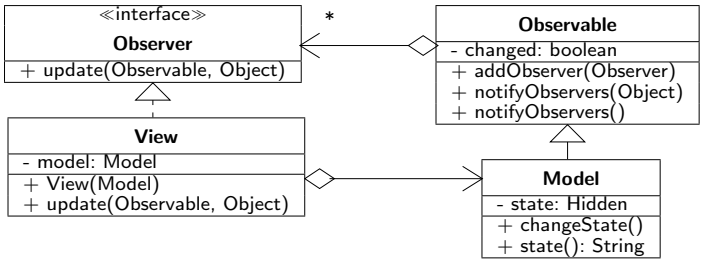
Observable

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
}
```

This class represents an observable object, or data"in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed.

Observer–mönstret



Metoden `state()` i modellen är en sorts getter. Det är bra om denna inte avslöjar hur tillståndet representeras.

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.

Extern ihopkoppling

Kopplingen mellan vy och modell kan också göras där vy och modell skapas:

```
Model model = new Model();
View view = new View();
model.addObserver(view);
```

```
public class View extends JLabel implements Observer {

  public void update(Observable observable, Object object) {
      setText(observable.toString());
  }
}
```

java.util.Observable implementering ...

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


... java.util.Observable

Något förenklad:

```
public class Observable {
  private boolean changed = false;
  private Vector<Observer> vector = new Vector<Observer>();
  public void notifyObservers() {
    notifyObservers(null);
  }
  public synchronized void notifyObservers(Object arg) {
    if (!changed) {
      return;
    }
    changed = false;
    for (Observer observer: vector) {
      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-parametrarna

```
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.

Fråga Hur kan modellen veta vad vyn vill ha?

Svar Det kan den inte veta.

Fråga Det kan finnas flera observatörer som vill veta olika saker. Hur skall informationen förpackas?

Svar Det är svårt att veta om man inte känner till alla observatörer som kan vara aktuella. Det finns ingen naturlig plats att dokumentera förpackningen.

Designprincip för Observer.update

Undvik att använda parametrarna.

LSP — Substitutionsprincipen

Liskov Substitution Principle

Subtypes must be substitutable for their base types.

När man konstruerar en subklass till en superklass så skall man göra det så att alla metoder som har superklassen som parametertyp även fungerar när argumentet är en instans av subklassen.

Brott mot LSP

```
public class IntPair {
  private int i1,i2;
  public boolean equals(Object object) {
     IntPair other = (IntPair) object;
     return i1==other.i1 && i2==other.i2;
  }
}
```

IntPair är en subklass till Object. Den bryter mot LSP på två sätt:

- Metoden kastar ett undantag om `object` har fel typ. Den borde returnera `false` när detta händer.
- Klassen fungerar ej som nyckel i en hashtabell. Lika objekt skall ha samma hashkod.

Reparation av IntPair

```
public final class IntPair {
  private int i1,i2;
  public int hashCode() {
      return i1 + 13 * i2;
  }
  public boolean equals(Object object) {
     if(object instanceof IntPair) {
         IntPair other = (IntPair) object;
         return i1==other.i1 && i2==other.i2;
     }
     return false;
  }
}
```


Implementering av `equals`

Om klassen inte är `final` är det mer komplicerat att göra det korrekt, men Eclipse kan göra det åt oss.

```
public class Pair {
    private A a1, a2;

    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((a1 == null) ? 0 : a1.hashCode());
        result = prime * result + ((a2 == null) ? 0 : a2.hashCode());
        return result;
    }
```


Implementering av `equals`

```
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Pair other = (Pair) obj;
    if (a1 == null) {
        if (other.a1 != null)
            return false;
    } else if (!a1.equals(other.a1))
        return false;
    if (a2 == null) {
        if (other.a2 != null)
            return false;
    } else if (!a2.equals(other.a2))
        return false;
    return true;
}
```

Designprinciper för paket

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

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: Mjukvarupaketet i Computer. Om man behöver en instruktion så behövs också de andra och Program-klassen.

Motsvarighet: Principen om enkelt ansvar.

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.

Exempel: När hårdvarugruppen i det stora Computer-projektet kommer med en ny version av hårdvaran så kan simuleringsgruppen själv bestämma när den skall uppgardera hårdvaran och kan göra det utan att titta på implementeringen.

Motsvarighet: Lokalitets- och integritetsprinciperna

Common closure principle

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

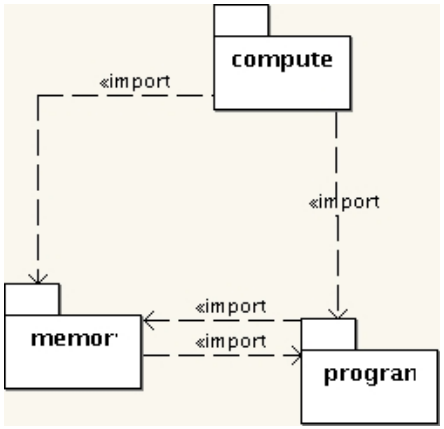
Exempel: När vi vill introducera VeryLongWord i Computer-projektet skall man bara behöva ändra i hårdvarupaketet.

Motsvarighet: Öppen/sluten-principen.

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 *stabilitet*

$$S = \frac{C_a}{C_e + C_a}$$

- 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. a står för *afferent* (inåtriktad).

- Om $S = 1$ för alla paket i systemet så är det inget system.
- Paketets *instabilitet*

$$I = 1 - S = \frac{C_e}{C_e + C_a}$$

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;
    }
}
```

Vad handlar det om?

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.isComplete()) {
                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 isComplete() {
        return size() >= MAXSIZE;
    }
}
```


Dokumentation av MyList

De reviderade klasserna behöver knappast någon särskild dokumentation. MyList gör det:

- 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.