# DESIGN PATTERNS – Parte 3

## Model View Controller
## Observer
## GUI

### MODEL VIEW CONTROLLER

Questo pattern nasce per la gestione delle finestre, dove l'applicazione viene divisa in 3 gruppi, che per l'appunto sono:

- Model
- View
- Controller

Ognuno di essi ha uno scopo specifico che vedremo in maniera dettagliata..

*Controller:*
Rappresenta un oggetto artificioso che coordina le operazioni di sistema dell'applicazione. Questa classe può essere divisa in due specifici gruppi:

1) Caso d'uso: per ogni caso d'uso esiste uno specifico Controller, chiamato Handler.
2) Sessione: per ogni utente esiste uno specifico Controller, chiamato Session.

Il compito del Controller è quello di collegare le classi del Model al resto dell'applicazione, ovvero alla View.

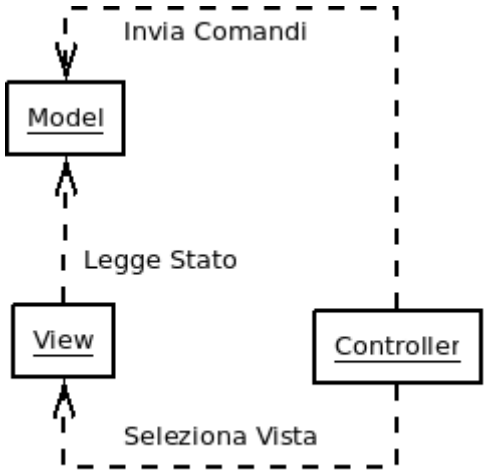
*Model:*
Rappresenta tutte le classi che hanno a che fare con il campo d'utilizzo e non con l'interazione dell'applicazione.

*View:*
È l'interfaccia grafica dell'applicazione. Nel caso delle applicazioni a riga di comando si riduce alle istruzioni di stampa a video.

Unendo questi tre elementi si ottiene il cosidetto MVC (Model View Controller), un design pattern che permette di potersi concentrare sull'area di maggior conoscenza. Per la precisione i tre componenti possono essere riassunti come:

- Modello (Model)
- Logica di presentazione (View)
- Logica di controllo (Controller)

Questo permette di rendere l'uno indipendente dall'altro, indi per cui è possibile un riuso del codice.




Ora però devo scoprire come fare a informare la View dei cambiamenti avvenuti nel Model senza creare una dipendenza tra Model e View?

*Soluzione:* il Model ha un metodo che chiamato dalla View avvia la modifica apportata. In questo modo si mantiene la dipendenza tra View e Model.

Esempio:

```
/* Classe del Model */
public class Model
{
     private MyComponent component;

     public void updateAction()
     {
     //update è un metodo proprio della classe MyComponent
          component.update(true);
     //Il metodo invalidate aggiorna la grafica del componente
     //Questo metodo è ereditato dalla classe JComponent.
          component.invalidate();
     }
}

/* Classe della View */
public class MyComponent extends JComponent
{
     private boolean state = false;

     public void update(boolean b) { state = b; }

     public void paintComponent(Graphics g){
          Graphics2D g2= (Graphics2D)g;
          g2.drawString("Stato del componente: "+state,50,100);
     }

}
```

Apparentemente non ci sono dipendenze forte, visto che JComponent è una classe della libreria Java, ma purtroppo questa dipendenza viene ripescata se dovessimo usare il Model per applicazioni di natura diversa (stand-alone/web). JComponent è disponibile solo per le applicazioni stand-alone e quindi inutilizzabile per le applicazioni web.

*Alternativa:* spostare il “controllo” al Controller. Questo però porta a delle difficoltà, ovvero: complessità del codice, bassa coesione e funziona solo se il Controller conosce quando il Model cambia stato.
Il diagramma delle classi quindi non cambia, ma cambia il codice:

```
/* Classe del Controller */
public class Controller
{
     private MyComponent component;
     private Model model;

     public void doAction()
     {
          //update è un metodo proprio della classe Model
          model.update(true);
          component.setProperty(true);
          component.invalidate();
     }
}

/* Classe del Model */
public class Model
{
     private boolean state = false;

     public void update(boolean b) { state = b; }
}

/* Classe della View */
public class MyComponent extends JComponent
{
     private boolean property;

     public void setProperty(boolean b) { property = b; }

     public void paintComponent(Graphics g)
     {
          Graphics2D g2= (Graphics2D)g;
          g2.drawString(“Stato del componente:”+property,50,100);
     }

}
```

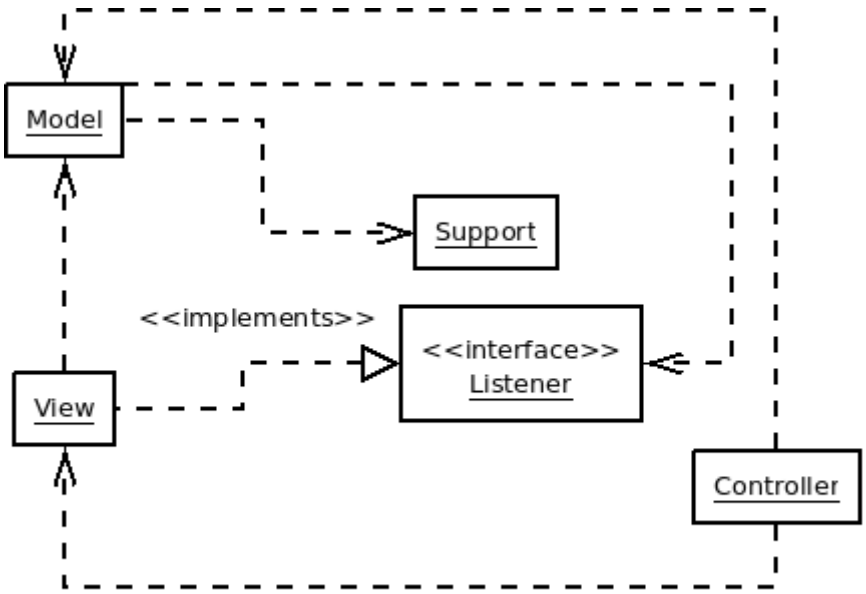
## OBSERVER

Il problema qui soprastante può essere migliorato attraverso il Pattern Observer. Questo pattern si basa sull'idea dei Subscriber e dei Publisher. Abbiamo più Subscriber che sono interessati ad un certo evento di un oggetto detto Publisher.

Questo è ottenibile in questa maniera:

1) Definisco delle interfacce Listener (identificheranno l'evento)
2) Subscriber implementerà l'interfaccia Listener
3) Publisher registrerà dinamicamente i Subscriber al relativo Lisener
4) Il Publisher avviserà i Subscriber quando accade l'evento
5) Varie classi (Support) che definiscono e implementano i Listener

Lo schema delle classi sarà quindi il seguente:



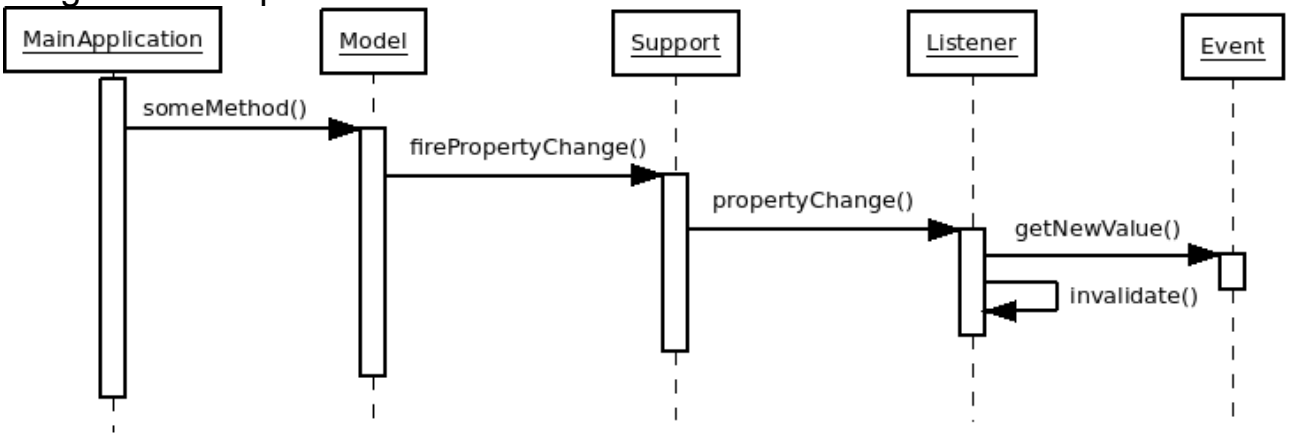

Esempio:

```
/* Classe del Model */
public class Model {
     private PropertyChangeSupport listeners = new
PropertyChangeSupport(this);
     private boolean state = false;

public void addPropertyChangeListener(PropertyChangeListener l) {
          listeners.addPropertyChangeListener(l);
     }

public void remPropertyChangeListener(PropertyChangeListener l) {
          listeners.removePropertyChangeListener(l);
     }

     public void update(boolean b) {
          boolean oldstate = state;
          state = b;
          listeners.firePropertyChange("state", state, oldstate);
     }
}
```

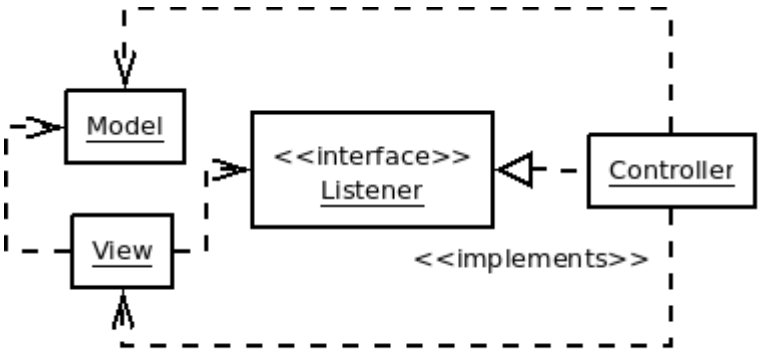
```
/*Classe della View */
public class MyComponent extends JComponent implements
PropertyChangeListener
{
     private boolean state;

     public void propertyChange(PropertyChangeEvent evt)
     {
          property = evt.getNewValue();
     }

     public void paintComponent(Graphics g)
     {
          Graphics2D g2= (Graphics2D)g;
          g2.drawString("Stato del componente: "+property,50,100);
     }

}

/*Classe del Controller*/
public class Controller
{
     private Model model;
     private MyComponent component;

     public void init() {
          //...
          model.addPropertyChangeListener(component);
          //...
     }
}
```

Nella classe Model, come detto, farà da Publisher, e quindi con il metodo addPropertyChangeListener riceverà i vari Subscriber.
La classe View sarà il Subscriber che attenderà l'evento per aggiornarsi.
Controller invece sarà la classe che all'avvio del programma assegnerà al Publisher i vari Subscriber.

Diagramma sequenziale:



I Listener sono abitualmente usati nelle interfacce grafiche come detto, si presupponga che si voglia che l'utente premuto un bottone chiami una specifica funzione, come si fa?
Innanzitutto si assegna al bottone il Listener specifico e poi il bottone stesso invocherà il metodo del Listener.
Avremmo quindi un diagramma e una GUI tipo questa:





Esempio:

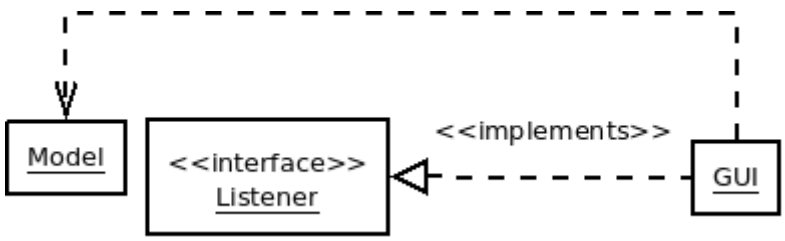
```
/* Classe della View */
public class NewJFrame extends JFrame {

     JButton ok = new JButton("ok");
     JLabel label1 = new JLabel("Dato 1");
     JLabel label2 = new JLabel("Dato 2");
     JTextField dato1 = new JTextField();
     JTextField dato2 = new JTextField();

     public NewJFrame(ActionListener controller){
          //...
          ok.addActionListener(controller);
     }

     public String getDato1() { dato1.getText(); }

     public String getDato2() { dato2.getText(); }

     public void setEsito1(boolean e) {
          if(!e) dato1.setText("Dato 1 errato:");
     }

     public void setEsito2(boolean e) {
          if(!e) dato2.setText("Dato 2 errato:");
     }

     //Eventuali altri metodi

}
```

```
/*Classe del Controller */
public class OkHandler implements ActionListener {
     NewJFrame frame;

     public OkHandler() {
          frame = NewJFrame(this);
     }

     public void actionPerformed(ActionEvent evt)
     {
          String dato1 = frame.getDato1();
          String dato2 = frame.getDato2();

          try { Model m = new Model(dato1, dato2); }
          catch(Dato1Exception de) { frame.setEsito1(false); }
          catch(Dato2Exception de) { frame.setEsito2(false); }

     }
}

/*Classe del Model*/
public class Model
{
     public Model(String d1, String d2)
          throws Dato1Exception, Dato2Exception
     { /* … */ }

     //Eventuali altri metodi
}
```

L'Handler gestisce il caso d'uso singolo del bottone Ok. Ma vediamo come avviene ciò: innanzitutto la finestra viene creata dall'Handler che si fa passare come riferimento alla finestra direttamente. Fatto questo si fa aggiungere come ascoltatore al bottone. Questo significa che al click del bottone l'Handler invocherà il metodo actionPerformed().
Questo metodo tenterà di creare un oggetto, nel caso fallisse modificherà le Label che segneranno che è avvenuto un errore.

## GUI

Generalmente il Controller e la View nelle applicazioni grafiche fanno parte della stessa classe, detta per l'appunto GUI.



Esempio (sempre con la GUI di prima):

```
/* Classe della GUI */
public class NewJFrame extends JFrame implements ActionListener {

     JButton ok = new JButton("ok");
     JLabel label1 = new JLabel("Dato 1");
     JLabel label2 = new JLabel("Dato 2");
     JTextField dato1 = new JTextField();
     JTextField dato2 = new JTextField();

     public NewJFrame(){
          //...
          ok.addActionListener(this);
     }

     public String getDato1() { dato1.getText(); }

     public String getDato2() { dato2.getText(); }

     public void setEsito1(boolean e) {
          if(!e) dato1.setText("Dato 1 errato:");
     }

     public void setEsito2(boolean e) {
          if(!e) dato2.setText("Dato 2 errato:");
     }

     public void actionPerformed(ActionEvent evt)
     {
          String dato1 = this.getDato1();
          String dato2 = this.getDato2();

          try { Model m = new Model(dato1, dato2); }
          catch(Dato1Exception de) { this.setEsito1(false); }
          catch(Dato2Exception de) { this.setEsito2(false); }

     }

     //Eventuali altri metodi

}
```

Le uniche modifiche da apportare sono alla classe NewJFrame che assorbe dentro a se i metodi della classe OkHandler.
Nel caso si volesserò gestire più Listener basterà fare un semplice if:

```
public void actionPerformed(ActionEvent evt){
     if(evt.getSource() == componente) { /*fai qualcosa*/ }
     else if( evt.getSource() == altroComp) { /*fai tutt'altro*/ }
     else { /* fai altro */ }
}
```