El patrón Modelo-Vista-Controlador

Tecnología de la Programación

Curso 2019-2020

Jesús Correas – jcorreas@ucm.es

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

(Basado en material creado por Yolanda García y los apuntes de la asignatura de Marco Antonio Gómez y Jorge Gómez)

Patrones de diseño

- Los patrones de diseño son **esquemas comunes de programación de sistemas de software**. Se empezaron a desarrollar en los años 80 y 90 en el ámbito de la programación orientada a objetos.
- Hay muchos casos en los que para resolver problemas muy diferentes se utilizan programas con estructuras muy parecidas.
- El objetivo es **reutilizar** en lo posible el esfuerzo de diseño de otros sistemas para facilitar la programación de nuevas aplicaciones.
- De hecho, ya hemos utilizado algunas técnicas:
  - Utilizamos **funciones y métodos** que se pueden utilizar en distintos sitios para no programar lo mismo cada vez.
  - La **herencia** nos permite programar determinados métodos una sola vez en la superclase y reutilizarlos en las subclases.
  - Las **bibliotecas** de clases (como el API de Java).
  - El uso de **genéricos** ayuda a programar clases reutilizables.
- Con los patrones podemos **reutilizar el diseño** de los sistemas de software.

El patrón modelo-vista-controlador

- Es un patrón de arquitectura de las aplicaciones de software.
- Se utiliza mucho en el desarrollo de aplicaciones visuales y aplicaciones web.
- Divide la aplicación en tres componentes:
  - El **modelo:** contiene los datos y funcionalidades del dominio de la aplicación.
  - La **vista:** es la representación gráfica de la aplicación. Puede ser gráfica o basada en texto.
  - el **controlador:** interpreta las acciones del usuario y las traduce en **operaciones sobre el modelo.**
- Esta estructura permite que cada componente pueda evolucionar por separado.
- En una aplicación existe un solo modelo, pero puede haber **una o varias vistas y uno o varios controladores.**
- Existen varias **instanciaciones de este patrón de diseño.** Veremos una de ellas.

MVC: El modelo

- El modelo contiene **los datos y funcionalidades (operaciones) que se pueden realizar en la aplicación.**
- Puede recibir **consultas** sobre su estado.
- Puede recibir **solicitudes para cambiar su estado** (realizar operaciones sobre su estado).

- El modelo **debe ser independiente de la vista y del controlador:** *no debe ver* las clases de los otros dos componentes.
  - Esto garantiza que se puede cambiar la vista y el controlador sin afectar al modelo.
  - Aun así, debe **notificar** a las vistas las modificaciones producidas en el modelo. Para ello, se utilizará otro patrón de diseño: el **patrón Observer.**
- El **controlador** debe tener acceso a las operaciones sobre el modelo.

MVC: Las vistas

- Son **representaciones visuales del modelo.**
- Pueden estar formadas por componentes gráficos, pero no necesariamente.
- Las vistas pueden ser una **representación parcial del modelo:** destacar algunos aspectos y ocultar otros.

- Las vistas **no deben ver las clases del modelo:** es el modelo el que *notifica* a las vistas los cambios que se producen en su estado.
  - En algunas variantes de MVC se permite que la vista tenga **acceso de consulta** del modelo, sin posibilidad de hacer cambios.
- Las vistas **sí deben tener acceso al controlador.** Así pueden indicar acciones del usuario que modifiquen el modelo.
- Es posible tener **varias vistas simultáneamente** para un modelo.
- De hecho, las vistas se pueden crear posteriormente sin necesidad de modificar el modelo.

MVC: El controlador

- El controlador determina las modificaciones del modelo de acuerdo a interacciones con la vista.
- El controlador **recibe peticiones de la vista** y responde a ellas modificando el modelo.
- El controlador **ve el modelo y modifica su estado invocando a los métodos necesarios.**
- La relación del controlador con las vistas puede ser diferente según la variante de MVC que se utilice:
  - En algunas variantes el controlador contiene los **listener** de las vistas.
  - Nosotros consideraremos los *listener* **como parte de las vistas**, que deben llamar a métods del controlador.
  - De esta forma, un cambio en la vista no afecta al controlador.

Comunicación entre vista y controlador

- **Las vistas ven el controlador y le avisan de la acción realizada por el usuario.**
- Una implementación para realizar esto es que la la vista contenga **un atributo con una referencia al controlador.**
- Por ejemplo:

```
//Vista.java
public class Vista extends JFrame implements ObservadorModelo {
  private Controlador control;
  ...
  public Vista(Controlador c) {
    ... control = c; ...
}}

// Controlador.java
public class Controlador {
  ...
  public void calcular(String valor) { ... }
  ...
}
```

Comunicación entre vista y controlador

- Cuando se produce una acción del usuario, la vista **captura el evento** e invoca al controlador para realizar la acción.
- En algunas variantes de MVC, el propio *listener* de los eventos Swing está en el controlador. Nosotros utilizaremos los *listener* en la vista que invocan a métodos del controlador:

```
//Vista.java
public class Vista extends JFrame implements ObservadorModelo {
  private Controlador control;
  ...
  public Vista(Controlador c) {
    ... control = c;
    ...
    btnCalcular.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        control.calcular(txtNumero.getText());
      }});
  }
}
```

Comunicación entre controlador y modelo

- **El controlador ve el modelo e invoca sus métodos para modificar su estado y obtener resultados.**
- Como en el caso anterior, el controlador puede tener un **atributo con una referencia al modelo.**
- Por ejemplo:

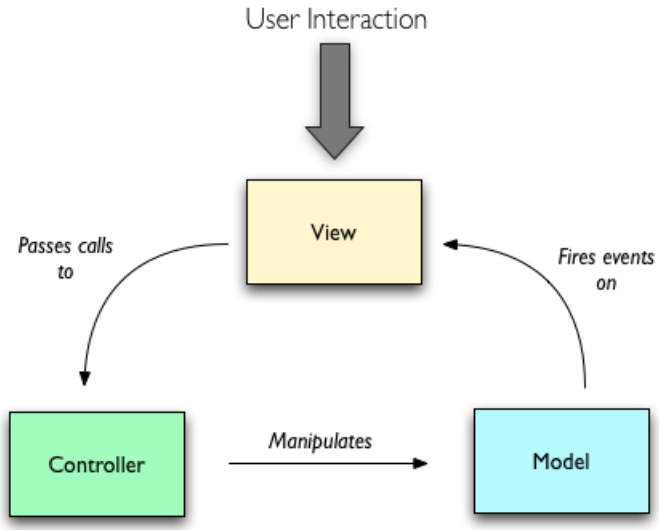
```
// Controlador.java
public class Controlador {
  Modelo modelo;
  public Controlador(Modelo m) { modelo = m; ... }

  public void calcular(String valor) {
    double num = ...
    modelo.calcular(num); ...
  }}

// Modelo.java
public class Modelo {
  public void calcular(Double num) { ... }
  ...
}
```

Comunicación entre modelo y vistas

- En la variante de MVC que vamos a utilizar aquí, la vista **no debe ver directamente el modelo:**
  - Así se reduce el **acoplamiento** entre los dos componentes, haciéndolos **independientes.**
  - La vista invoca operaciones del modelo a través del **controlador.**
  - El modelo **notifica** cambios en su estado a la vista a través de **observadores.**



- Para que el modelo pueda notificar a la vista los cambios se utiliza el **patrón Observer.**

(Imagen tomada de `http://blog.nodejitsu.com/scaling-isomorphic-javascript-code/`)

Comunicación entre modelo y vista – Patrón *Observer*

- En este patrón de diseño, las vistas **se registran como observadores del modelo** a través de métodos del controlador.
- El modelo ofrece un **interfaz** para que las vistas implementen los métodos con los que el modelo va a notificar los cambios.

```
//Vista.java
public class Vista extends JFrame implements ObservadorModelo {
  private Controlador control;
  ...
  public Vista(Controlador c) {
    ... control = c; ...
    control.addObservador(this);
  }

  // implementación del interfaz ObservadorModelo
  @Override public void actualizar(double dato) {
    ...
  }
}
```

- El controlador debe implementar el método `addObservador()`, que a

Comunicación entre modelo y vista – Patrón *Observer*

- El modelo guarda los observadores en una lista:

```
// Modelo.java
public class Modelo {
  ArrayList<ObservadorModelo> observadores;
  ...
  public void addObservador(ObservadorModelo ob) {
    observadores.add(ob);
  }
  public void removeObservador(ObservadorModelo ob) {
    observadores.remove(ob);
  }
  ...
```

- cuando el modelo cambia su estado lo notifica a los observadores:

```
  ...
  public void calcular(Double num) {
    ... double resultado = ...;
    for (ObservadorModelo ob : observadores) {
      ob.actualizar(resultado);
    }
```

Comunicación entre modelo y vista – Patrón *Observer*

- El modelo utiliza una lista para guardar los observadores, porque lo habitual es que el interfaz de usuario contenga **distintos observadores del modelo.**
- Por ejemplo, en la ventana de un juego de tablero, el panel que representa el **tablero** puede ser un observador, y el **marcador de puntuaciones** otro observador distinto.
- Además, MVC independiza el modelo del resto de la aplicación: permite utilizar **distintas vistas sin modificar el modelo.**
  - Una vista basada en ventanas, con diversos observadores por ejemplo.
  - Otra vista basada en texto.

Comunicación entre modelo y vista – Patrón *Observer*

- El interfaz `ObservadorModelo` que hemos utilizado en el ejemplo solo notifica un cambio mediante una declaración de método:

```
public interface ObservadorModelo {
  public void actualizar(double dato);
}
```

- Sin embargo, el mismo observador puede contener varios métodos para notificar distinto tipo de información a los observadores.
- Por ejemplo, se puede utilizar un método para notificar que ha ocurrido un error o excepción:

```
public interface ObservadorModelo {
  public void actualizar(double dato);
  public void notificarError(MiError err);
}
```

Varios controladores. Puesta en marcha de MVC

- En una aplicación que tiene varias vistas distintas (por ejemplo, interfaz de ventanas e interfaz de consola) puede ser necesario **implementar varios controladores.**
- En este caso lo más adecuado es declarar un **interfaz** que implementen ambos controladores, de forma que las llamadas desde las distintas vistas sean comunes.

- Por último, **para poner en marcha una arquitectura MVC** se suele utilizar la siguiente pauta de creación:

```
Modelo modelo = new Modelo();
Controlador control = new Controlador(modelo);
Vista vista = new Vista(control);
// la vista se añade a sí misma normalmente, si no:
//modelo.addObservador(vista);
```