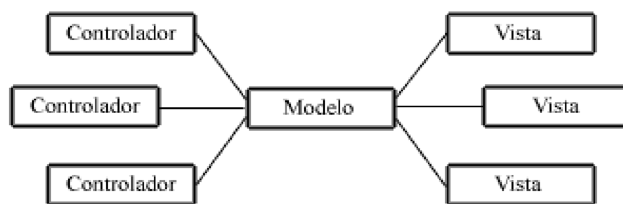
ARQUITECTURA Modelo/Vista/Controlador

La arquitectura MVC (*Model/View/Controller*) fue introducida como parte de la versión Smalltalk-80 del lenguaje de programación Smalltalk. Fue diseñada para reducir el esfuerzo de programación necesario en la implementación de sistemas múltiples y sincronizados de los mismos datos. Sus características principales son que el Modelo, las Vistas y los Controladores se tratan como entidades separadas; esto hace que cualquier cambio producido en el Modelo se refleje automáticamente en cada una de las Vistas.

Además del programa ejemplo que hemos presentado al principio y que posteriormente implementaremos, este modelo de arquitectura se puede emplear en sistemas de representación gráfica de datos, como se ha citado, o en sistemas CAD, en donde se presentan partes del diseño con diferente escala de aumento, en ventanas separadas.

En la figura siguiente, vemos la arquitectura MVC en su forma más general. Hay un Modelo, múltiples Controladores que manipulan ese Modelo, y hay varias Vistas de los datos del Modelo, que cambian cuando cambia el estado de ese Modelo.



Este modelo de arquitectura presenta varias ventajas:

- Hay una clara separación entre los componentes de un programa; lo cual nos permite implementarlos por separado
- Hay un API muy bien definido; cualquiera que use el API, podrá reemplazar el Modelo, la Vista o el Controlador, sin aparente dificultad.
- La conexión entre el Modelo y sus Vistas es dinámica; se produce en tiempo de ejecución, no en tiempo de compilación.

Al incorporar el modelo de arquitectura MVC a un diseño, las piezas de un programa se pueden construir por separado y luego unirlas en tiempo de ejecución. Si uno de los Componentes, posteriormente, se observa que funciona mal, puede reemplazarse sin que las otras piezas se vean afectadas. Este escenario contrasta con la aproximación monolítica típica de muchos programas Java. Todos tienen un *Frame* que contiene todos los elementos, un controlador de eventos, un montón de cálculos y la presentación del resultado. Ante esta perspectiva, hacer un cambio aquí no es nada trivial.

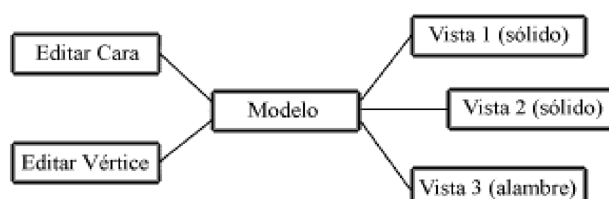
Definición de las partes

El *Modelo* es el objeto que representa los datos del programa. Maneja los datos y controla todas sus transformaciones. El Modelo no tiene conocimiento específico de los Controladores o de las Vistas, ni siquiera contiene referencias a ellos. Es el propio sistema el que tiene encomendada la responsabilidad de mantener enlaces entre el Modelo y sus Vistas, y notificar a las Vistas cuando cambia el Modelo.

La *Vista* es el objeto que maneja la presentación visual de los datos representados por el Modelo. Genera una representación visual del Modelo y muestra los datos al usuario. Interactúa con el Modelo a través de una referencia al propio Modelo.

El *Controlador* es el objeto que proporciona significado a las ordenes del usuario, actuando sobre los datos representados por el Modelo. Cuando se realiza algún cambio, entra en acción, bien sea por cambios en la información del Modelo o por alteraciones de la Vista. Interactúa con el Modelo a través de una referencia al propio Modelo.

Vamos a mostrar un ejemplo concreto. Consideremos como tal el sistema descrito en la introducción a este capítulo, una pieza geométrica en tres dimensiones, que representamos en la figura siguiente:



En este caso, la pieza central de la escena en tres dimensiones es el Modelo. El Modelo es una descripción matemática de los vértices y las caras que componen la escena. Los datos que describen cada vértice o cara pueden modificarse (quizás como resultado de una acción del usuario, o una distorsión de la escena, o un algoritmo de sombreado). Sin embargo, no tiene noción del punto de vista, método de presentación, perspectiva o fuente de luz. El Modelo es una representación pura de los elementos que componen la escena.

La porción del programa que transforma los datos dentro del Modelo en una presentación gráfica es la Vista. La Vista incorpora la visión del Modelo a la escena; es la representación gráfica de la escena desde un punto de vista determinado, bajo condiciones de iluminación determinadas.

El Controlador sabe que puede hacer el Modelo e implementa el interface de usuario que permite iniciar la acción. En este ejemplo, un panel de datos de entrada es lo único que se necesita, para permitir añadir, modificar o borrar vértices o caras de la figura.

Observador y Observable

El lenguaje de programación Java proporciona soporte para la arquitectura MVC mediante dos clases:

- **Observer**: Es cualquier objeto que desee ser notificado cuando el estado de otro objeto sea alterado
- **Observable**: Es cualquier objeto cuyo estado puede representar interés y sobre el cual otro objeto ha demostrado ese interés

Estas dos clases se pueden utilizar para muchas más cosas que la implementación de la arquitectura MVC. Serán útiles en cualquier sistema en que se necesite que algunos objetos sean notificados cuando ocurran cambios en otros objetos.

El Modelo es un subtipo de **Observable** y la Vista es un subtipo de **Observer**. Estas dos clases manejan adecuadamente la función de notificación de cambios que necesita la arquitectura MVC. Proporcionan el mecanismo por el cual las Vistas pueden ser notificadas automáticamente de los cambios producidos en el Modelo. Referencias al objeto Modelo tanto en el Controlador como en la Vista permiten acceder a los datos de ese objeto Modelo.

Funciones Observer y Observable

Vamos a enumerar las funciones que intervienen en el control de Observador y Observable:

Observer

```
public void update( Observable obs,Object obj )
```

Llamada cuando se produce un cambio en el estado del objeto Observable

Observable

```
public void addObserver( Observer obs )
```

Añade un observador a la lista interna de observadores

```
public void deleteObserver( Observer obs )
```

Borra un observador de la lista interna de observadores

```
public void deleteObservers()
```

Borra todos los observadores de la lista interna

```
public int countObserver()
```

Devuelve el número de observadores en la lista interna

```
protected void setChanged()
```

Levanta el flag interno que indica que el Observable ha cambiado de estado

```
protected void clearChanged()
```

Baja el flag interno que indica que el Observable ha cambiado de estado

```
protected boolean hasChanged()
```

Devuelve un valor booleano indicando si el Observable ha cambiado de estado

```
public void notifyObservers()
```

Comprueba el flag interno para ver si el Observable ha cambiado de estado y lo notifica a todos los observadores

```
public void notifyObservers( Object obj )
```

Comprueba el flag interno para ver si el Observable ha cambiado de estado y lo notifica a todos los observadores. Les pasa el objeto especificado en la llamada para que lo usen los observadores en su método *notify()*.

Cómo utilizar Observer y Observable

Vamos a describir en los siguientes apartados, como crear una nueva clase **Observable** y una nueva clase **Observer** y como utilizar las dos conjuntamente.

Extender un Observable

Una nueva clase de objetos observables se crea extendiendo la clase **Observable**. Como la clase **Observable** ya implementa todos los métodos necesarios para proporcionar el funcionamiento de tipo Observador/Observable, la clase derivada solamente necesita proporcionar algún tipo de mecanismo que lo ajuste a su funcionamiento particular y proporcionar acceso al estado interno del objeto Observable.

En la clase **ValorObservable** que mostramos a continuación, el estado interno del Modelo es capturado en el entero `n`. A este valor se accede (y más importante todavía, se modifica) solamente a través de sus métodos públicos. Si el valor cambia, el objeto invoca a su propio método *setChanged()* para indicar que el estado del Modelo ha cambiado. Luego, invoca a su propio método *notifyObservers()* para actualizar a todos los observadores registrados.

```
import java.util.Observable;

 public class ValorObservable extends Observable {
     private int nValor = 0;

     // Constructor al que indicamos el valor en que comenzamos y los
     // limites inferior y superior que no deben sobrepasarse
     public ValorObservable( int nValor,int nInferior,int nSuperior ) {
        this.nValor = nValor;
        }

     // Fija el valor que le pasamos y notifica a los observadores que
     // estan pendientes del cambio de estado de los objetos de esta
     // clase, que su etado se ha visto alterado
     public void setValor(int nValor) {
        this.nValor = nValor;

        setChanged();
        notifyObservers();
        }

     // Devuelve el valor actual que tiene el objeto
     public int getValor() {
        return( nValor );
        }
     }
```

Implementar un Observador

Una nueva clase de objetos que observe los cambios en el estado de otro objeto se puede crear implementando la interface **Observer**. Esta interface necesita un método *update()* que se debe

proporcionar en la nueva clase. Este método será llamado siempre que el Observable cambie de estado, que anuncia este cambio llamando a su método *notifyObservers()*. El observador entonces, debería interrogar al objeto Observable para determinar su nuevo estado; y, en el caso de la arquitectura MVC, ajustar su Vista adecuadamente.

En la clase **ObservadorDeTexto**, que muestra el código siguiente, el método *notify()* primero realiza una comprobación para asegurarse de que el Observable que ha anunciado un cambio es el Observable que él esta observando. Si lo es, entonces lee su estado e imprime el nuevo valor.

```
import java.util.Observer;
 import java.util.Observable;

 public class TextoObservador extends Frame implements Observer {
     private ValorObservable vo = null;

     public TextoObservador( ValorObservable vo ) {
         this.vo = vo;
         }

     public void update( Observable obs,Object obj ) {
         if( obs == vo )
             tf.setText( String.valueOf( vo.getValor() ) );
         }
     }
```

Usando Observador y Observable

Un programa indica a un objeto Observable que hay un objeto observador que debe ser notificado cuando se produzca un cambio en su estado, llamando al método *addObserver()* del objeto Observable. Este método añade el Observador a la lista de observadores que el objeto Observable ha de notificar cuando su estado se altere.

En el ejemplo siguiente, en donde mostramos la clase **ControlValor**, ControlValor.java, vemos como se usa el método *addObserver()* para añadir una instancia de la clase **TextoObservador** a la lista que mantiene la clase **ValorObservable**.

```
public class ControlValor {

     // Constructor de la clase que nos permite crear los objetos de
     // observador y observable
     public ControlValor() {
         ValorObservable vo = new ValorObservable( 100,0,500 );
         TextoObservador to = new TextoObservador( vo );

         vo.addObserver( to );
         }

     public static void main( String args[] ) {
         ControlValor m = new ControlValor();
         }
     }
```

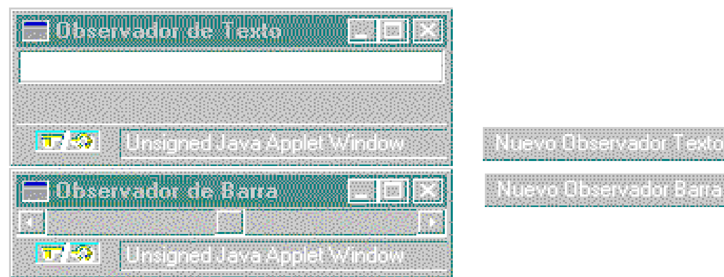
En la siguiente secuencia, vamos a describir como se realiza la interacción entre un Observador y un objeto Observable, durante la ejecución de un programa:

1. En primer lugar el usuario manipula un elemento del interface de usuario representado por el Controlador. Este Controlador realiza un cambio en el Modelo a través de uno de sus métodos públicos de acceso; en nuestro caso, llama a *setValue()*.
2. El método público de acceso modifica el dato privado, ajusta el estado interno del Modelo y llama al método *setChanged()* para indicar que su estado ha cambiado. Luego llama al método *notifyObservers()* para notificar a los observadores que su estado no es el mismo. La llamada a este método puede realizarse en cualquier lugar, incluso desde un bucle de actualización que se esté ejecutando en otro thread.
3. Se llama a los métodos *update()* de cada Observador, indicando que hay un cambio en el estado del objeto que estaban observando. El Observador accede entonces a los datos del Modelo a través del método público del Observable y actualiza las Vistas.

Ejemplo de aplicación MVC

En el ejemplo siguiente, vemos como colaboran juntos Observador y Observable en la arquitectura MVC:

Tu navegador no entiende la marca <APPLET>. La imagen siguiente es la reproducción de la apariencia del applet en pantalla:



El Modelo de este ejemplo es muy simple. Su estado interno consta de un valor entero. Este valor, o estado, es manipulado exclusivamente a través de métodos públicos de acceso. El código del modelo se encuentra implementado en ValorObservable.java.

Inicialmente, hemos escrito una clase simple de Vista/Controlador. La clase combina las características de una Vista (presenta el valor que corresponde al estado actual del Modelo) y un Controlador (permite al usuario introducir un nuevo valor para alterar el estado del Modelo). El código se encuentra en el fichero TextoObservador.java. Podemos crear instancias de esta vista pulsando el botón superior que aparece en el applet.

A través de este diseño utilizando la arquitectura MVC (en lugar de colocar el código para que el Modelo, la Vista y el Controlador de texto en una clase monolítica), el sistema puede ser fácilmente rediseñado para manejar otra Vista y otro Controlador. En este caso, hemos visto una clase Vista/Controlador con una barra de desplazamiento. La posición del marcador en la barra representa el valor actual que corresponde con el estado del Modelo y puede ser alterado a través de movimientos del marcador sobre la barra por acción del usuario. El código de esta clase se encuentra en BarraObservador.java. Se pueden crear instancias de esta clase pulsando el botón inferior del applet de esta página.