El patrón de diseño Modelo-Vista-Controlador (MVC) y su implementación en Java Swing

Ernesto Bascón Pantoja

Desarrollador de software
Jalasoft
e-mail: ebascon@jalasoft.com

1. Introducción

Durante el proceso de enseñanza de programación, los docentes tienen que luchar contra el hábito natural que la mayoría de los estudiantes tienen de solucionar los problemas en largas, desordenadas, incomprensibles y enormes cantidades de código que mezclan algoritmos de procesamiento de datos, combinaciones de colores, mensajes al usuario, iconos animados, gráficos de barras y otras cosas más que hacen de ese código difícil de leer, de comprender y por tanto, de ser modificado, mantenido y extendido. El algoritmo 1 muestra esta mala forma de resolver un problema:

```
1   Mientras el usuario no termine la aplicación, hacer:
2       Solicitar el ingreso de la palabra a buscar
3       Buscar registros en la base de datos con esa palabra → Lista de Resultados
4       Si la lista de resultados está vacía:
5           Mostrar un mensaje de error
6       caso contrario
7           Recorrer la lista de resultados
8               Sacar cada dato de la lista de resultados → dato
9               Agregar el dato a una tabla en la pantalla
10          Mostrar la tabla en la pantalla
11          Mostrar la lista de resultados en un gráfico estadístico
12      Esperar que el usuario presione "CONTINUAR"
```

Algoritmo 1.

Como se puede ver, el algoritmo anterior mezcla acceso a bases de datos, introducción de datos, despliegue de resultados, procesos de búsqueda, salida de mensajes de error y gráficos estadísticos en un simple bloque de código y, aunque se ve exagerado, una enorme cantidad de aplicaciones hechas a medida y aplicaciones web desarrolladas

en nuestro medio están implementadas de esa manera. Si bien éste es un error causado por un mal hábito del programador, las herramientas de desarrollo rápido de aplicaciones (RAD[1]) como Delphi, Visual Basic y los lenguajes de *scripting* de desarrollo de aplicaciones web (ASP[2] , JSP[3] o PHP[4]) contribuyen a crear esa clase de aplicaciones muy fáciles de implementar, pero muy desorganizadas y difíciles de comprender y modificar. El patrón de diseño presentado en el presente artículo, plantea un método formal para separar los módulos de entrada, de procesamiento y de salida de datos en una aplicación y la forma de comunicación entre dichos módulos; también se verán las características, ventajas y desventajas de la implementación de este patrón de diseño en la librería de interfaz de usuario Java Swing.

2. El patrón de diseño Modelo-Vista-Controlador

Durante toda la década del setenta, SmallTalk y algunos otros lenguajes como Simula I, fueron construyendo gradualmente el paradigma de programación orientada a objetos y estableciendo conceptos tales como objetos, clases, encapsulación, herencia y polimorfismo [3]. Si bien dichos lenguajes no son usados actualmente para implementar aplicaciones comerciales, los conceptos que dejaron en el mundo del desarrollo de software están vigentes en la actualidad y son la base de lenguajes modernos como C++, Java o C#.

SmallTalk también fue el primer lenguaje de programación que permitió diseñar interfaces de usuario con múltiples "ventanas" desplegadas en una misma pantalla, concepto que después fue aplicado por GEMS, Macintosh, X11, Windows y otras interfaces gráficas de usuario modernas. El concepto central detrás de las librerías de interfaz de usuario provistas por SmallTalk está basado en el patrón de diseño MVC, creado por el profesor Trygve Reenskaug [4].

MVC es un patrón de diseño que considera dividir una aplicación en tres módulos claramente identificables y con funcionalidad bien definida: El Modelo, las Vistas y el Controlador.

2.1. El modelo

El modelo es un conjunto de clases que representan la información del mundo real que el sistema debe procesar, así por ejemplo un sistema de administración de datos climatológicos tendrá un modelo que representará la temperatura, la humedad ambiental, el estado del tiempo esperado, etc. sin tomar en cuenta ni la forma en la que esa información va a ser mostrada ni los mecanismos que hacen que esos datos estén dentro del modelo, es decir, sin tener relación con ninguna otra entidad dentro de la aplicación.

[1]Rapid Application Development, entornos de desarrollo integrados que proporcionan todos los mecanismos para construir rápidamente aplicaciones.

[2]Active Server Pages, (Páginas de Servidor Activas), tecnología de Microsoft para desarrollo de aplicaciones web.

[3]Java Server Pages, arquitectura similar a ASP pero para plataformas Java.

[4]PHP: Arquitectura similar a ASP pero desarrollada para plataformas Linux.

El modelo desconoce la existencia de las vistas y del controlador. Ese enfoque suena interesante, pero en la práctica no es aplicable pues deben existir interfaces que permitan a los módulos comunicarse entre sí, por lo que SmallTalk sugiere que el modelo en realidad esté formado por dos submódulos: El modelo del dominio y el modelo de la aplicación. El presente artículo utiliza la propuesta de SmallTalk por ser la base de la implementación de la librería Swing que es vista más adelante.

2.1.1. Modelo del dominio

Se podría decir que el modelo del dominio (o el modelo propiamente dicho) es el conjunto de clases que un ingeniero de software modela al analizar el problema que desea resolver; así, pertenecerían al modelo del dominio: El cliente, la factura, la temperatura, la hora, etc. El modelo del dominio no debería tener relación con nada externo a la información que contiene.

2.1.2. Modelo de la aplicación

El modelo de la aplicación es un conjunto de clases que se relacionan con el modelo del dominio, que tienen conocimiento de las vistas y que implementan los mecanismos necesarios para notificar a éstas últimas sobre los cambios que se pudieren dar en el modelo del dominio. El modelo de la aplicación es llamado también *coordinador de la aplicación.*[4]

2.2. Las vistas

Las vistas son el conjunto de clases que se encargan de mostrar al usuario la información contenida en el modelo. Una vista está asociada a un modelo, pudiendo existir varias vistas asociadas al mismo modelo; así por ejemplo, se puede tener una vista mostrando la hora del sistema como un reloj analógico y otra vista mostrando la misma información como un reloj digital.

Una vista obtiene del modelo solamente la información que necesita para desplegar y se actualiza cada vez que el modelo del dominio cambia por medio de notificaciones generadas por el modelo de la aplicación.[2]

2.3. El controlador

El controlador es un objeto que se encarga de dirigir el flujo del control de la aplicación debido a mensajes externos, como datos introducidos por el usuario u opciones del menú seleccionadas por él. A partir de estos mensajes, el controlador se encarga de modificar el modelo o de abrir y cerrar vistas. El controlador tiene acceso al modelo y a las vistas, pero las vistas y el modelo no conocen de la existencia del controlador. [3]

La figura 1 muestra la relación entre los módulos de MVC:

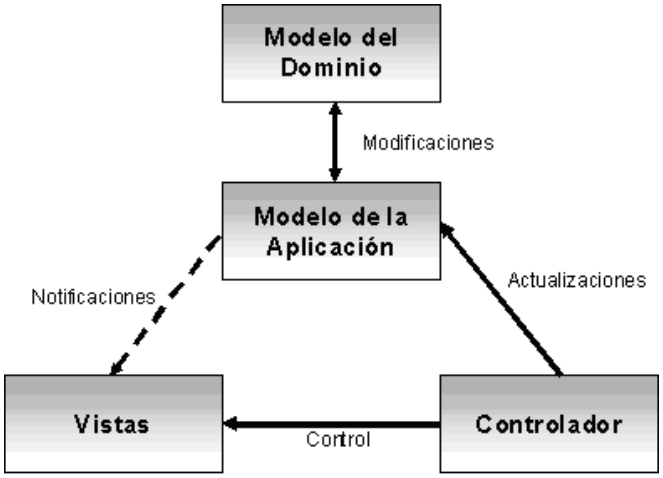

Figura 1: Relación entre los módulos del patrón MVC.

2.4. Un ejemplo

Tomemos como ejemplo una aplicación hecha para almacenar y procesar los datos de las elecciones municipales.

El modelo del dominio sería bastante simple: Un conjunto de votos, un conjunto de mesas y un conjunto de departamentos. Cada voto almacenaría la selección hecha por el votante y la mesa donde emitió su voto. Cada mesa contendría información sobre el lugar de votación y el departamento donde estaría ubicada.

El conjunto de vistas sobre el modelo también sería sencillo: Se podría obtener un gráfico estadístico de votos por departamento en barras, otro gráfico de votos por departamento mostrado como torta, el conjunto de votos totales en una tabla, el conjunto de votos totales en barras o en torta, etc. Como se puede ver, aunque todas las vistas estarían mostrando la información de diferente manera, todas estarían asociadas al mismo modelo del dominio.

El controlador se encargaría de mostrar las vistas que el usuario desearía ver y de permitir al usuario introducir información de votos. Si el usuario desearía ver una vista, el controlador crearía la vista solicitada, esta vista obtendría la información necesaria del modelo y la desplegaría. Si el usuario aumentaría información de votos al sistema, el controlador se encargaría de actualizar la información contenida en el modelo del dominio que, al ser modificado, anunciaría al modelo de la aplicación la existencia de cambios y éste notificaría a todas sus vistas asociadas para que se actualicen [2]. De esta manera, las vistas estarían siempre actualizadas mostrando exactamente la misma información contenida en el modelo.

2.5. Ventajas

Desarrollar una aplicación siguiendo este patrón de diseño tiene muchas ventajas:

- La aplicación está implementada modularmente.
- Sus vistas muestran información actualizada siempre.
- El programador no debe preocuparse de solicitar que las vistas se actualicen, ya que este proceso es realizado automáticamente por el modelo de la aplicación.
- Si se desea hacer una modificación al modelo del dominio, como aumentar métodos o datos contenidos, sólo debe modificarse el modelo y las interfaces del mismo con las vistas, no todo el mecanismo de comunicación y de actualización entre modelos.
- Las modificaciones a las vistas no afectan en absoluto a los otros módulos de la aplicación.
- MVC es bastante utilizado en la actualidad en marcos de aplicación orientados a objeto desarrollados para construir aplicaciones de gran tamaño; Java Swing, Apache Struts, Microsoft ASP.NET, las transformaciones XSL o incluso los documentos LaTeX siguen este patrón de diseño.
- MVC está demostrando ser un patrón de diseño bien elaborado pues las aplicaciones que lo implementan presentan una extensibilidad y una mantenibilidad únicas comparadas con otras aplicaciones basadas en otros patrones.

2.6. Desventajas

- El tiempo de desarrollo de una aplicación que implementa el patrón de diseño MVC es mayor, al menos en la primera etapa, que el tiempo de desarrollo de una aplicación que no lo implementa, ya que MVC requiere que el programador implemente una mayor cantidad de clases que en un entorno de desarrollo común no son necesarias. Sin embargo, esta desventaja es muy relativa ya que posteriormente, en la etapa de mantenimiento de la aplicación, una aplicación MVC es muchísimo más mantenible, extensible y modificable que una aplicación que no lo implementa.
- MVC requiere la existencia de una arquitectura inicial sobre la que se deben construir clases e interfaces para modificar y comunicar los módulos de una aplicación. Esta arquitectura inicial debe incluir, por lo menos: un mecanismo de eventos para poder proporcionar las notificaciones que genera el modelo de aplicación; una clase Modelo, otra clase Vista y una clase Controlador genéricas que realicen todas las tareas de comunicación, notificación y actualización que serán luego transparentes para el desarrollo de la aplicación.
- MVC es un patrón de diseño orientado a objetos por lo que su implementación es sumamente costosa y difícil en lenguajes que no siguen este paradigma.

3. MVC en Java Swing

Java es un lenguaje orientado a objetos desarrollado por Sun Microsystems que tiene como características fundamentales su portabilidad a un gran número de plataformas (Java es en sí una plataforma), su simplicidad y su extenso conjunto de librerías.

La librería de interfaces gráficas de usuario que viene en el SDK[5] de Java se llama Java Swing y tiene estas características sobresalientes:

- Implementa bastantes componentes visuales (botones, campos de texto, tablas, barras de menús, árboles, etc.).
- Los componentes provistos por Java Swing son independientes de la plataforma donde se ejecuta la aplicación (contrario a la idea de AWT[6] , otra librería de interfaces de usuario de Java, que utiliza componentes nativos de la interfaz de usuario del sistema operativo donde se ejecuta la aplicación) y por tanto, la portabilidad de las aplicaciones desarrolladas con Swing está garantizada.
- Permite "conectar" y "desconectar" estilos de interfaz de usuario (llamados "look and feels") que modifican la forma en que se muestra y se comporta toda la interfaz de usuario, así, la misma aplicación puede verse como una aplicación Windows o como una aplicación Motif[7] simplemente conmutando el *look and feel* en tiempo de ejecución.
- Cumple con el diseño de JavaBeans que hace que los componentes de esta librería puedan ser utilizados en entornos de desarrollo integrados (IDEs) como JBuilder, Sun Forte for Java o Eclipse.
- Su arquitectura está profundamente basada en MVC, lo que proporciona un alto grado de extensibilidad y de personalización de los componentes de la librería.

3.1. Arquitectura de Java Swing

Si bien Swing se basa en el patrón de diseño MVC, presenta algunas diferencias en su implementación: La diferencia más notoria es que el controlador y la vista están implementados como un único elemento denominado "delegado de interfaz de usuario" ("UI delegate"). Cada componente está asociado a un conjunto de datos (modelo del dominio) a través de un modelo de aplicación y del delegado de interfaz. Este delegado ha sido diseñado debido a la incapacidad de programar un controlador genérico con conocimiento de la vista a la que hubiera podido controlar, por tanto, cada componente tiene asociado un delegado de interfaz que a su vez, está asociado a un modelo de aplicación que proporciona toda la información que el delegado requiere y que notifica

[5] SDK: Software Development Kit, en el caso de Java es el conjunto de clases provistas por Sun Microsystems para desarrollar aplicaciones.

[6] AWT: Abstract Windowing Toolkit, primera librería de interfaz de usuario de Java, Swing utiliza un conjunto muy pequeño de las clases de AWT como base de su arquitectura.

[7] Motif, estilo de interfaz de usuario implementado en CDE (Common Desktop Environment), una interfaz de usuario bastante difundida en entornos UNIX.

al componente si algún cambio se ha dado en el modelo del dominio [5]; por tanto, Java Swing implementa el modelo MVC en una implementación por componente.

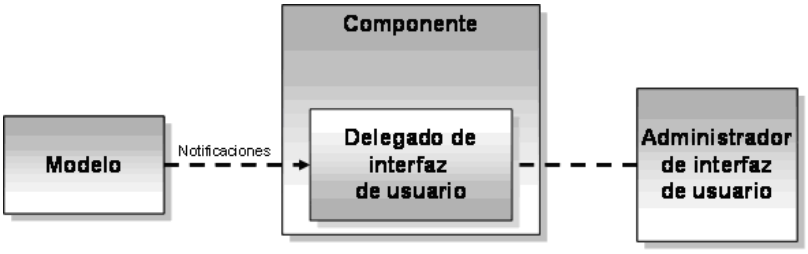

Figura 2: Relación entre los módulos de Java Swing.

En la figura 2 se observa también la existencia de un elemento denominado "administrador de interfaz de usuario" ("UI manager"); este elemento permite conectar un *look and feel* específico a cada componente visualizado y almacena también información general a la apariencia de los componentes en su visualización (color de fondo por defecto, tipo de letra por defecto, tipos de bordes, etc.). Todos los componentes de Swing están manejados por el mismo administrador de interfaz de usuario.

Los modelos implementados en Swing están clasificados en dos categorías: Modelos de estado de interfaz de usuario y modelos de datos. Los modelos de estado de interfaz definen el estado visual de un componente, como por ejemplo, el estado de un botón está presionado o información de los ítems seleccionados en determinada lista. Los modelos de datos representan información que está contextualizada en la aplicación. Estos modelos de datos sirven como puente de comunicación entre el modelo del dominio y el delegado de interfaz de un componente.

El cuadro 1 muestra la relación entre los componentes de Java Swing, sus delegados de interfaz y el modelo que tienen asociado [1].

3.2. Modelos y componentes

Como se pudo ver hasta el momento, cada tipo de componente tiene asociado un tipo particular de modelo de aplicación. Swing provee estos tipos de modelo como interfaces (declaraciones de métodos sin implementaciones, por ejemplo el `TableModel`), además proporciona modelos por defecto (`DefaultTableModel`) que proporcionan funcionalidad básica y lista para usarse. La ventaja de que los modelos de aplicación sean implementados como interfaces, es que el desarrollador puede implementar un modelo de aplicación adecuado al modelo del dominio que tiene, sin tener que reestructurar sus datos según requerimientos de Swing.

Veamos todo esto con un ejemplo: Se desea hacer un programa que muestre en una tabla una lista de personas, para lo cual se implementa primeramente una clase `Persona` que tenga nombre, apellido, carnet de identidad y teléfono como atributos:

Componente	*Delegado de interfaz* **(controlador, vista)**	*Modelo de aplicación*	*Tipo*
JButton	ButtonUIB	ButtonModel	Estado
JToggleButton	ButtonUI	ButtonModel	Estado
JCheckBox	ButtonUI	ButtonModel	Estado
JRadioButton	ButtonUI	ButtonModel	Estado
JMenu	MenuBarUI	SingleSelectionModel	Estado
JMenuItem	MenuItemUI	ButtonModel	Estado
JComboBox	ComboBoxUI	ComboBoxModel	Datos
JProgressBar	ProgressBarUI	BoundedRangeModel	Dat./Est.
JScrollBar	ScrollBarUI	BoundedRangeModel	Dat./Est.
JSlider	SliderUI	BoundedRangeModel	Dat./Est.
JTabbedPane	TabbedPaneUI	SingleSelectionModel	Estado
JList	ListUI	ListModel	Datos
JTable	TableUI	TableModel	Datos
JTree	TreeUI	TreeModel	Datos
JTextArea	TextUI	Document	Datos
JTextField	TextUI	Document	Datos

Cuadro 1: Relación entre los componentes de Java Swing, sus delegados de interfaz y sus modelos.

```
class Persona
{
   protected String nombre;
   protected String apellido;
   protected String carnet;
   protected String telefono;

   public Persona(String aNombre, String aApellido, String aCarnet,
                  String aTelefono)
   {
      nombre = aNombre;
      apellido = aApellido;
      carnet = aCarnet;
      telefono = aTelefono;
   }

   public String getNombre()
   {
      return nombre;
   }

   public String getApellido()
   {
```

```
      return apellido;
   }

   public String getCarnet()
   {
      return carnet;
   }

   public String getTelefono()
   {
      return telefono;
   }
}
```

Segmento de código 1: Implementación de la clase Persona.

Luego se construye una clase denominada `ListaDePersonas` que implementa los métodos para agregar y acceder a las personas a la lista:

```
class ListaDePersonas
{
   protected Vector vectorDePersonas;

   public ListaDePersonas()
   {
      vectorDePersonas = new Vector();
   }

   public void agregarPersona(Persona aPersona)
   {
      vectorDePersonas.add(aPersona);
   }

   public int getNumeroDePersonas()
   {
      return vectorDePersonas.size();
   }

   public Persona getPersona(int aIndex)
   {
      return (Persona) vectorDePersonas.elementAt(aIndex);
   }
}
```

Segmento de código 2: Implementación de la clase ListaDePersona.

Las clases `Persona` y `ListaDePersonas` se constituyen en el modelo del dominio de la aplicación de ejemplo. Para que este modelo pueda ser mostrado por un elemento de interfaz, debe construirse un modelo de aplicación, en este caso, la clase `PersonasTableModel` que se basa en la clase `AbstractTableModel` que define la funcionalidad básica del modelo de la tabla:

```
class PersonasTableModel extends AbstractTableModel
{
   protected ListaDePersonas listaDePersonas;

   PersonasTableModel(ListaDePersonas aListaDePersonas)
   {
      listaDePersonas = aListaDePersonas;
   }

   public int getRowCount()
   {
      return listaDePersonas.getNumeroDePersonas();
   }

   public int getColumnCount()
   {
      return 4; //hay cuatro atributos en la clase Persona
   }

   public Object getValueAt(int aRow, int aCol)
   {
      Persona persona = listaDePersonas.getPersona(aRow);
      switch (aCol)
      {
         case 0: return persona.getNombre();
         case 1: return persona.getApellido();
         case 2: return persona.getCarnet();
         case 3: return persona.getTelefono();
      }
      return ‘‘’’;
   }

   public String getColumnName(int aCol)
   {
      switch (aCol)
      {
         case 0: return ‘‘Nombre’’;
         case 1: return ‘‘Apellido’’;
         case 2: return ‘‘C.I.’’;
         case 3: return ‘‘Telefono’’;
```

```
      }
      return ‘‘’’;
   }
}
```

Segmento de código 3: Implementación de la clase PersonasTableModel.

Los métodos implementados en la clase `PersonasTableModel`, son métodos definidos en la clase `AbstractTableModel` que se encargan de proporcionar toda la información que el delegado de la tabla necesita para mostrar esa información. Como se puede ver en el ejemplo, la clase `PersonasTableModel` sirve como un puente entre el modelo del dominio y el delegado de interfaz de usuario y permite que el componente se adapte a la estructura de datos manejada.

Una vez definidas estas clases, se procede a colocar el componente `JTable` (asociado al modelo definido anteriormente) en una ventana:

```
import javax.swing.*;
public class PersonasTableFrame extends JFrame
{
   protected JTable table;
   protected ListaDePersonas listaDePersonas;

   public MVCTestFrame()
   {
      setSize(400, 300);
      setTitle(‘‘PersonasTableModel’’);

      initData();
      initUI();

      setVisible(true);
   }

   protected void initData()
   {
      listaDePersonas = new ListaDePersonas();

      //Datos ejemplo
      listaDePersonas.agregarPersona(new Persona(‘‘Julio’’, ‘‘Pérez’’,
      ‘‘3949345 PT’’,‘‘4295453’’));
      listaDePersonas.agregarPersona(new Persona(‘‘Alfredo’’, ‘‘López’’,
      ‘‘3554352 LP’’, ‘‘591-4-4481353’’));
      listaDePersonas.agregarPersona(new Persona(‘‘Ricardo’’, ‘‘Loayza’’,
      ‘‘4002321 CB’’, ‘‘591-2-6227587’’));
      listaDePersonas.agregarPersona(new Persona(‘‘Elena’’,
```

```
      ‘‘Mendoza’’, ‘‘3443432 CB’’, ‘‘591-4-4443211’’));
   }

   protected void initUI()
   {
      setDefaultCloseOperation(EXIT_ON_CLOSE);

      getContentPane().setLayout(new BorderLayout(5, 5));

      PersonasTableModel tableModel =
      new PersonasTableModel(listaDePersonas);
      table = new JTable(tableModel);

      JScrollPane tableScrollPane =
      new JScrollPane(table);
      getContentPane().add(tableScrollPane, BorderLayout.CENTER);
   }
}
```

Segmento de código 4: Implementación de la ventana principal.

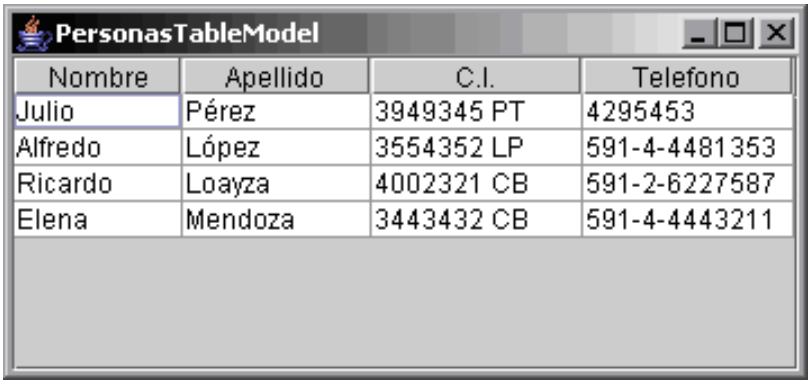

Figura 3: Ejecución del ejemplo descrito.

3.3. Notificaciones

A lo largo de todo este artículo hemos hablado de notificaciones hechas a las vistas debido a cambios ocurridos en el modelo. Java Swing maneja el paso de notificaciones por medio de un mecanismo de eventos y de “escuchadores” (“listeners”) de eventos.

Para que una vista se actualice debido al cambio de su modelo asociado, Swing provee dos tipos de notificaciones: Las notificaciones ligeras (“lightweight”) y las notificaciones de estado (“stateful”).

Las *notificaciones ligeras* envían a las vistas un mensaje de "modelo modificado". Es la vista la que debe solicitar al modelo más información sobre los cambios realizados. Este tipo de notificaciones es bastante til cuando la frecuencia de modificaciones al modelo es alta (como en el modelo del `JScrollPane` que se modifica cada que se arrastra el control del *scroll*) y es bastante usado en modelos que almacenan estado de interfaz. El cuadro 2 muestra los modelos, los *listeners* y los eventos que funcionan con esta clase de notificaciones:

Modelo	***Listener***	***Evento***
`BoundedRangeModel`	`ChangeListener`	`ChangeEvent`
`ButtonModel`	`ChangeListener`	`ChangeEvent`
`SingleSelectionModel`	`ChangeListener`	`ChangeEvent`

Cuadro 2: Modelos con sus respectivos Listeners y eventos para notificaciones ligeras.

El segmento de código 5 muestra cómo el método `stateChanged` procesa notificaciones cuando el modelo de un botón es modificado:

```
protected void initUI()
{
   boton = new JButton(‘‘Evento’’);
   boton.setBounds(10, 10, 75, 25);
   ButtonModel modelo = boton.getModel();
   modelo.addChangeListener(this); //Listener
   getContentPane().add(boton);
}

//método encargado de escuchar la notificación
public void stateChanged(ChangeEvent aEvent)
{
   contador++;
   mensaje.setText(‘‘Modelo cambiado:’’ + contador + ‘‘veces’’);
}
```

Segmento de código 5: Ejemplo de procesamiento de cambio de estado de modelo.

Las *notificaciones de estado* proporcionan información mucho más precisa sobre el cambio que se ha dado en el modelo. El cuadro 3 muestra los modelos, los listeners y los eventos que funcionan con esta clase de notificaciones:

En el segmento de código 6 se ve cómo el método `insertUpdate` se encarga de procesar las notificaciones de estado cuando se inserta texto en el documento de un campo de texto:

```
protected void initUI()
{
```

*M*odelo	*L*istener	*E*vento
ListModel	ListDataListener	ListDataEvent
ListSelectionModel	ListSelectionListener	ListSelectionEvent
ComboBoxModel	ListDataListener	ListDataListener
TreeModel	TreeModelListener	TreeModelEvent
TreeSelectionModel	TreeSelectionListener	TreeSelectionEvent
TableModel	TableModelListener	TableModelEvent
Document	DocumentListener	DocumentEvent

Cuadro 3: Modelos con sus respectivos listeners y eventos para notificaciones de estado.

```
   textField = new JTextField();
   textField.setBounds(10, 10, 175, 22);
   textField.getDocument().addDocumentListener(
      new DocumentListener() //listener
      {
         public void insertUpdate(DocumentEvent aEvent) //evento
         {
            contador++;
            mensaje.setText(‘‘Modelo cambiado: ’’ + contador + ‘‘ veces’’);
         }

         public void changedUpdate(DocumentEvent aEvent) { }

         public void removeUpdate(DocumentEvent aEvent) { }
      });

   getContentPane().add(textField);
}
```

Segmento de código 6: Ejemplo de procesamiento de notificación de cambio de estado.

3.4. Ventajas

La implementación de Swing del patrón de diseño MVC presenta muchas ventajas:

- Permite la creación de interfaces de usuario de una manera sencilla y rápida, permitiendo el manejo del patrón MVC pero ocultando los detalles de su implementación.
- El mecanismo de eventos de Java se adapta perfectamente al mecanismo de notificaciones de MVC.
- Al estar los modelos separados de la vista, las posibilidades de extensión de la librería y de personalización de componentes ya existentes son enormes.

- Permite al usuario crear sus propias estructuras de datos y adaptar la interfaz de usuario a ellas y no a la inversa, como sucede con librerías ya implementadas.

3.5. Desventajas

- La principal desventaja de esta implementación es que, aunque la arquitectura de Java Swing lo permite, la personalización y extensión de los modelos de algunos componentes (especialmente tablas y árboles), es bastante dificultosa.
- Los modelos de estado y los modelos de datos de Java Swing no están claramente diferenciados en la implementación y, si bien el modelo de datos es la implementación del modelo de la aplicación, el modelo de estado está muy alejado de la definición de modelo que MVC introdujo.
- El controlador definido por MVC es casi inexistente en Java Swing y su trabajo puede confundirse con el del administrador de interfaz de usuario.

Referencias

[1] Java Swing architecture.
`http://www.cs.unc.edu/Courses/wwwp-s99/members/walkera/swing/presentSwingII.htm`.

[2] Joseph Bergin. Building Graphical User Interfaces with the MVC pattern. `http://csis.pace.edu/ bergin/mvc/mvcgui.html`.

[3] Steve Burbeck. Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC).
`http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html`.

[4] John Deacon. Model-View-Controller (MVC) Architecture.
`http://www.jdl.co.uk/briefings/mvc.pdf`.

[5] Amy Fowler. A Swing architecture overview.
`http://java.sun.com/products/jfc/tsc/articles/architecture/`.

[6] Sun Microsystems. Java^{TM} 2 Platform, Standard Edition, v 1.4.2, API Specification.
`http://java.sun.com/products/jdk1.4/doc`.