Anexo C. Patrones de diseño de software

Patrón *Observer*

El patrón *Observer* establece una dependencia de cardinalidad "uno a muchos" entre objetos, de tal manera que cuando uno de ellos cambie de estado, todos los demás que dependan de éste primero sean notificados y actualizados automáticamente [Freeman, 2004]. Al objeto que notifica a los demás de su cambio de estado se le llama *subject*, y a los objetos que se actualizan al ser notificados de los cambios del *subject* se les llama *observers*.

El mecanismo de control de eventos en Java está basado en este patrón de diseño. Los componentes gráficos de interacción con el usuario como campos de texto, botones, menús y demás componentes AWT y *Swing* juegan el papel de *subjects*, mientras que las clases que implementan las interfaces de control de eventos juegan el papel de *observers*.

Como se puede notar en el ejemplo anterior, la interdependencia entre *subjects* y *observers* se minimiza con el uso de interfaces. Los objetos *subject* (los componentes *Swing* en el ejemplo anterior) no necesitan saber el tipo de objeto de los *observers*, sólo necesitan saber que implementan una determinada interfaz (las interfaces *Listener* en el ejemplo anterior) y llamar a sus métodos para actualizar automáticamente el estado de los *observers*. Esta característica de mínima interdependencia entre objetos constituye un principio de diseño de software llamado *loose coupling* [Freeman, 2004].

Al diseñar software basado en el patrón *observer*, se debe de tomar en cuenta que un mismo objeto puede llegar a tener los roles de *subject* y *observer* simultáneamente, pudiendo formarse grafos de relaciones entre objetos. Debe evitarse la formación de ciclos en estos grafos. La figura C-1 muestra el diagrama de clases de este patrón de diseño.

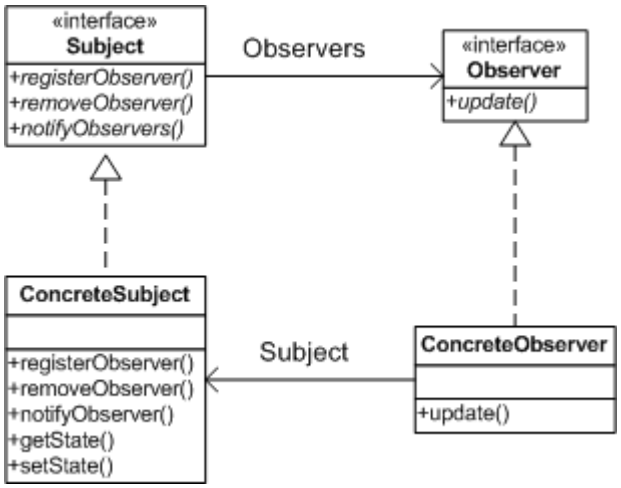

Figura C-1. Patrón de diseño *observer* [Freeman, 2004]

Patrón *Model View Controller*

El patrón *Model View Controller* es un patrón de diseño compuesto, esto quiere decir que sus componentes se basan en un conjunto de patrones que trabajan de maneja conjunta bajo un mismo diseño [Freeman, 2004]. Este patrón es comúnmente usado en el diseño de sistemas que tienen una interfaz gráfica de usuario y una lógica aplicativa complejas, y es de gran ayuda en el mantenimiento de la consistencia del estado de los mismos. Los tres componentes principales de este patrón son los siguientes [Freeman, 2004]:

1. Modelo: almacena los datos, el estado y la lógica de la aplicación. Este componente debe proveer una interfaz para manipular y averiguar su estado. Desde el punto de vista del patrón *Observer*, el modelo juega el rol de *subject*, y cada vez que cambie de estado debe notificárselo a las vistas, que juegan el rol de *observers*.
2. Vista: proporciona una representación visual del modelo, y la interfaz gráfica de usuario. Generalmente este componente obtiene directamente del modelo la información que necesita mostrar al usuario a través de las notificaciones automáticas que recibe del cambio de estado del mismo.
3. Controlador: es el componente mediador entre modelo y vista, que responde y obtiene las entradas del usuario y averigua qué puede hacer el modelo con ellas.

La figura C-2 que se presenta a continuación muestra la interacción entre estos tres componentes.

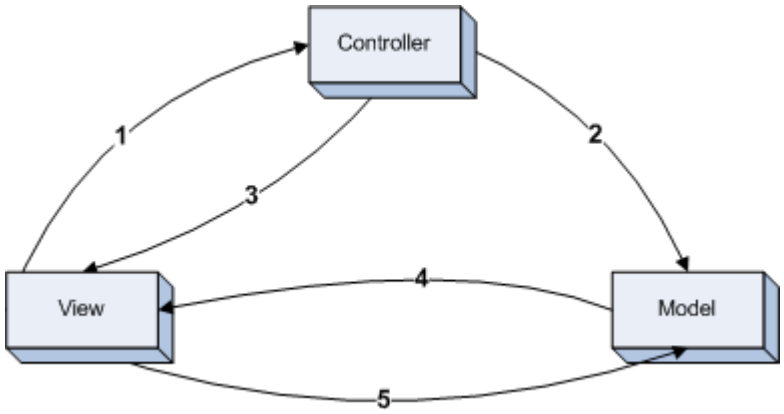

Figura C-2. Interacción entre *Model*, *View* y *Controller* [Freeman, 2004]

A continuación se describen de manera general cada uno de los pasos de esta interacción.

1. El usuario interactúa con la vista, y ésta notifica al controlador lo sucedido.
2. El controlador determina cómo manipular o cambiar el estado del modelo en base a la acción realizada por el usuario.
3. Opcionalmente, el controlador también puede actualizar el estado de la vista en respuesta a determinadas acciones del usuario. Ejemplos de estas situaciones son la habilitación o inhabilitación de componentes de interacción.
4. El modelo notifica a la vista de su cambio de estado.
5. La vista obtiene el estado del modelo, averigua cómo cambió su estado y actualiza su apariencia de acuerdo al cambio ocurrido.

Finalmente es importante notar que la relación entre modelo y vistas es originalmente de uno a muchos. Es decir, un mismo modelo puede tener asociadas varias vistas. En el diseño del software desarrollado para este proyecto, se limitó la cardinalidad a una modalidad de uno a uno, a menos que se haga uso de la herencia o de la composición para compensar esta situación.

Patrón *Framework*

Un *framework* es un conjunto de clases e interfaces que estructuran el mecanismo esencial de un dominio o problemática particular [Horstmann, 2006]. Generalmente las clases e interfaces de un *framework* están basadas en uno o varios patrones de diseño de software.

Un ejemplo de *framework* es la paquetería de Java para crear interfaces gráficas de usuario. Java provee el mecanismo esencial del funcionamiento de varios componentes gráficos de interacción como ventanas, paneles, diálogos, menús, botones, campos de texto, etc. La creación de GUI's personalizadas se debe hacer en base a las clases de esta paquetería y de acuerdo a las finalidades que sean requeridas.

Un *framework* tiene dos características de diseño importantes [Horstmann, 2006]:

- Especialización por medio de herencia: de acuerdo con esta característica, el uso de un *framework* se basa en la creación de subclases o en la implementación de las interfaces contenidas en el *framework*. En el ejemplo anterior, para hacer un panel, diálogo o ventana personalizada, se debe crear una subclase de la clase JPanel, JDialog o JFrame respectivamente. A estas subclases e implementaciones se les llama clases de aplicaciones específicas.
- Inversión de control: de acuerdo con esta característica, son las clases del *framework*, y no las clases de las aplicaciones específicas, las que controlan el flujo de la ejecución de la aplicación. En el ejemplo anterior, las clases de los *frameworks* AWT y *Swing* son las que implementan el código de acciones como mostrar u ocultar ventanas y diálogos, desplegar menús, habilitar o deshabilitar componentes, etc.

En el sistema desarrollado para este proyecto se crearon y usaron varios *frameworks*. Todos ellos están contenidos en el paquete *framework* del mismo. El más importante es el de los *managers*, que permite crear componentes que incluyen una vista, un modelo y un controlador que se relacionan entre sí con una cardinalidad inicial de uno a uno, y cuyas relaciones básicas son manejadas por las clases de este *framework*.