UNIVERSIDAD NACIONAL DE LA PLATA
FACULTAD DE INFORMÁTICA

TRABAJO DE GRADO

MVC DESDE SMALLTALK A LA WEB: ANÁLISIS DE LA ARQUITECTURA QUE PRODUJO UN CAMBIO EN EL DISEÑO DE APLICACIONES

MARÍA ALEJANDRA GOS

Director: Dr. Gustavo Rossi

La Plata 2004

MVC DESDE SMALLTALK A LA WEB:

ANÁLISIS DE LA ARQUITECTURA QUE PRODUJO UN CAMBIO EN EL DISEÑO DE APLICACIONES

Agradecimientos

- ✓ Agradezco en general a todas las personas cuyo nombre no aparece aquí, especialmente a mis amigos expertos en desarrollo de aplicaciones que me ayudaron a evacuar todas las dudas que fueron surgiendo durante el desarrollo de este trabajo apoyándome siempre para que siga adelante.

- ✓ A Gustavo, mi director, por guiarme durante este trayecto y por brindarme siempre su indispensable experiencia para la realización de esta tesis.

- ✓ A Arturo, por responder a mis numerosas preguntas sobre programación orientada a objetos y por asistirme con su experiencia en trabajos de investigación en la realización de éste, mi primer trabajo en el área.

- ✓ A Juan Pablo, mi amigo de siempre, por colaborar conmigo en incontables ocasiones, en cualquier horario y todos los días, transmitiéndome su experiencia laboral en cuestiones de implementación.

- ✓ A Dean, mi marido y compañero, por esperar a que finalice con mi carrera.

- ✓ A mi hermana Laura y a mi mamá Mercedes por brindarme el apoyo familiar y tranquilidad necesarios para estudiar.

- ✓ Especialmente a Alberto, mi papá, por hacerme sentir que puedo lograr éste y todos mis objetivos.

A mi papá,
con todo mi cariño.

Tabla de contenidos

Abstract XII

Introducción XIII

Sistemas interactivos – GUI's XIII

Cómo diseñar una aplicación con GUI XIV

Un ejemplo de aplicación interactiva XV

Motivación y Objetivos XV

Contribuciones del trabajo XVI

Guía sobre los capítulos del trabajo XVII

1. Fundamentos de MVC 1

1.1. Contexto de MVC 1

1.2. Qué es Model – View – Controller 3

1.3. Estructura de MVC 3

1.4. Relación entre componentes de MVC 4

1.4.1. Pattern Observer 5

1.5. Funcionamiento de MVC 6

1.6. Consecuencias del uso de MVC 8

1.6.1 Múltiples vistas del mismo modelo 8

1.6.2 Vistas sincronizadas 8

1.6.3 Vistas y controladores conectables 8

1.6.4 Intercambio de 'look and feel' 9

1.7. Algunas desventajas del uso de MVC 9

1.7.1 Mayor Complejidad 9

1.7.2 Número excesivo de actualizaciones 9

1.7.3 Conexión fuerte entre vista y controlador 9

1.7.4 Acoplamiento estrecho de vistas y controladores a un modelo 10

1.7.5 Ineficiencia de acceso de datos en la vista 10

1.7.6 Cambio inevitable de vista y controlador al portar 10

1.7.7 Dificultad al usar MVC con herramientas modernas para construcción de interfaz de usuario 10

2. Uso de MVC en GUI's tradicionales 11
2.1 La implementación de MVC en VisualWorks 11
2.1.1 Clase Model: soporte de dependencias para MVC 12
2.1.2 La Vista de MVC en VisualWorks 14
2.1.2.1 Clase VisualPart 14
2.1.2.2 Clase ScheduledWindow: la ventana de la interfaz 16
2.1.2.3 Clases para composición de vistas: Wrapper y CompositePart 16
2.1.2.4 El método que dibuja a un componente vista 17
2.1.3 El Controlador en VisualWorks 18
2.1.3.1 Clase Controller 18
2.1.3.2 El flujo de control de los controladores 19
2.1.3.3 Algunas clases de la jerarquía Controller 20
2.1.3.4 Clases InputSensor y Cursor 21
2.1.4 Conexión entre los tres componentes de MVC 22
2.1.5 Vistas Customizadas 23
2.1.6 Mejora a la implementación de MVC en Smalltalk: el framework ValueModel de VisualWorks 23
2.1.6.1 El ApplicationModel: mediador entre el modelo y la interfaz 23
2.1.6.2 El ApplicationModel: controlador de la aplicación 24
2.1.6.3 ValueModels: conexión entre una vista y un aspecto del modelo 25
2.1.6.4 ValueModels: Dependencias entre objetos 26
2.1.6.5 ValueModels: los mensajes value y value: 27
2.1.6.6 Ejemplo de ValueModel con más comportamiento: ProtocolAdaptor 28
2.2 Análisis de Frameworks MVC 30
2.3 Características de MVC para el análisis de frameworks 31
2.3.1 Manejo de dependencias del modelo y propagación de cambios 31
2.3.2 Actualización inmediata de dependientes 31
2.3.3 Independencia de la vista con respecto al modelo 31
2.3.4 Separación entre Vista y Controlador: Independencia de la vista con respecto a su controlador 32
2.3.5 Independencia del Controlador con respecto al Modelo 33
2.3.6 Vistas anidadas, Controladores anidados 34
2.3.7 Separación total del modelo y sus vistas 35

2.3.8 Múltiples vistas para un mismo modelo y sincronización entre ellas 36
2.3.9 MVC a nivel componente 36
2.4 Las propiedades de MVC a nivel aplicación y a nivel componente 37
2.5 Análisis del framework VisualWorks sobre las características de MVC 38
2.5.1 Manejo de dependencias del modelo y propagación de cambios 38
2.5.2 Actualización inmediata de dependientes 39
2.5.3 Independencia de la vista con respecto al modelo 39
2.5.4 Separación entre Vista y Controlador: Independencia de la vista con respecto a su controlador 40
2.5.5 Independencia del Controlador con respecto al Modelo 41
2.5.6 Vistas anidadas, Controladores anidados 41
2.5.7 Separación total del modelo y sus vistas 42
2.5.8 Múltiples vistas para un mismo modelo y sincronización entre ellas 42
2.5.9 MVC a nivel componente 43
2.6 La implementación de MVC en Java Swing 43
2.6.1 El Modelo de un Componente Swing y el mecanismo de propagación de cambios 44
2.6.1.1 La propagación de cambios del modelo: orientada a eventos 46
2.6.1.2 Actualización de las vistas 47
2.6.1.3 El modelo encapsulado 47
2.6.2 Vista/Controlador: Look & Feel conectables 47
2.6.2.1 El UIDelegate: look&feel específico de un componente 48
2.6.2.2 UI Manager: control del Look&Feel actual 49
2.6.2.3 Conexión de un look&feel 49
2.6.2.4 Otras funciones de UIDelegate: layout de la interfaz y dibujo del componente . . 50
2.6.2.5 La ventana en Swing y la jerarquía de componentes 50
2.6.2.6 Manejo de Eventos de Entrada: otra implementación de Observer 51
2.6.2.7 La función de controlador del UIDelegate 53
2.7 Análisis del framework Swing sobre las características de MVC 54
2.7.1 Manejo de dependencias del modelo y propagación de cambios 54
2.7.2 Actualización inmediata de dependientes 55
2.7.3 Independencia de la vista con respecto al modelo 55
2.7.4 Separación entre Vista y Controlador: Independencia de la vista con respecto a su controlador 56

2.7.5 Independencia del Controlador con respecto al Modelo 57
2.7.6 Vistas anidadas, Controladores anidados 58
2.7.7 Separación total del modelo y sus vistas 58
2.7.8 Múltiples vistas para un mismo modelo y sincronización entre ellas 59
2.7.9 MVC a nivel componente 59
2.8 Resumen del análisis de propiedades MVC 59

3. Evolución del Diseño MVC 62
3.1 Ejemplo de aplicación de MVC tradicional 63
3.2 Un ejemplo de mayor complejidad: lógica de la aplicación 64
3.2.1 La Lógica de Negocios 67
3.2.2 El modelo desde una visión simplista: lógica de aplicación y lógica de negocios . . 67
3.3 Limitación del diseño MVC tradicional 68
3.4 Rediseño de MVC: controlador de la aplicación 69
3.5 Controlador de la aplicación: distintas responsabilidades 70

4. Introducción a MVC en la Web 71
4.1 Conceptos básicos relacionados con la Web 71
4.2 ¿Por qué MVC en la Web? 73
4.3 Factores inherentes a la Web que influyeron sobre el MVC original 73
4.3.1 La conexión sin estado entre el cliente y el servidor 73
4.3.2 El lugar físico donde reside la "vista" de la aplicación web es distinto del que contiene al resto de la aplicación 74
4.3.3 La diferencia entre la tecnología usada para la vista, y la usada para los otros dos componentes de MVC 74
4.4 Consecuencias de los factores anteriores sobre MVC 75
4.4.1 Sobre la notificación de cambios del modelo 75
4.4.2 Sobre las vistas y la actualización en respuesta a la notificación de cambios 75
4.4.3 Sobre los eventos generados desde la interfaz 76
4.4.4 Sobre las responsabilidades del controlador 76
4.4.5 Sobre los componentes que forman una vista y el MVC a nivel componente 78
4.5 Soluciones a los "problemas" Web 78
4.5.1 Mantener la conexión entre cliente y servidor: la sesión 78

4.5.2 Diseñadores vs. Programadores: Push MVC y Pull MVC 79
4.5.2.1 Pull MVC: similitud al MVC tradicional 79
4.5.2.2 Push MVC: la Vista independiente del modelo 81

5. Implementaciones de MVC en la Web 83
5.1 Redefinición de propiedades de MVC en el contexto de la Web 84
5.1.1 Manejo de dependencias del modelo y propagación de cambios (no aplicable) . . 85
5.1.2 Actualización inmediata de dependientes (no aplicable) 85
5.1.3 Independencia de la vista con respecto al modelo 85
5.1.4 Separación entre Vista y Controlador: Independencia de la vista con respecto a su controlador 85
5.1.5 Independencia del Controlador con respecto al Modelo 86
5.1.6 Separación total del modelo y sus vistas (no aplicable) 86
5.1.7 Múltiples vistas para un mismo modelo y sincronización entre ellas (no aplicable) 86
5.1.8 Representación del componente Vista en el servidor 87
5.1.9 Vistas anidadas, controladores anidados 87
5.1.10 MVC a nivel componente en la web 87
5.1.11 Controlador de la aplicación 88
5.1.12 Reuso de la estructura de las vistas 88
5.1.13 Posibilidad de portar interfaces basadas en HTML a otras plataformas 89
5.1.14 Reusabilidad de los componentes visuales 90
5.1.15 Posibilidad de que una sola interfaz se adapte a quien accede a la aplicación . . 90
5.2 Análisis de implementaciones de MVC en la Web 90
5.2.1 Java Server Pages y Java Servlets: tecnologías para el desarrollo de aplicaciones . 90
5.2.1.1 Servlet monolítico: vista y controlador acoplados 91
5.2.1.2 JSP Model 1: vista y controlador acoplados 92
5.2.1.3 JSP Model 2: MVC para la web 93
5.2.1.4 Custom tags: no más código Java en la Vista JSP 95
5.2.1.5 JSP- Servlets y las propiedades de MVC 95
5.2.2 Struts framework: Controlador para aplicaciones Web 97
5.2.2.1 Componentes del framework 97
5.2.2.2 El rol de Controlador: ActionServlet y Actions 97
5.2.2.3 Un modelo para Struts: JavaBeans 98

5.2.2.4 Java Server Pages y Taglibs de Struts para el rol de Vista 99
5.2.2.5 Interacción de los componentes del framework 100
5.2.2.6 Struts y las propiedades de MVC 100
5.2.3 WebActions framework 101
5.2.3.1 Input Controller: procesamiento de la entrada 102
5.2.3.2 Application Controller: capa de Lógica de la aplicación 102
5.2.3.3 Comunicación con el modelo: clase Action 103
5.2.3.4 Objetos Business: modelo de la aplicación 103
5.2.3.5 Vista de WebActions 104
5.2.3.6 Funcionamiento del framework: ejecución de una acción sobre la aplicación . . 104
5.2.3.7 WebActions y las propiedades de MVC 105
5.2.4 Java Server Faces framework 106
5.2.4.1 Componentes del framework 107
5.2.4.2 Tags JSF 108
5.2.4.3 El modelo de una aplicación JSF 109
5.2.4.4 Funcionamiento del framework 109
5.2.4.5 JavaServer Faces y las propiedades de MVC 110
5.2.5 Una implementación de Push MVC: tecnología Enhydra XMLC 112
5.2.5.1 XMLC: generador de plantillas 113
5.2.5.2 Composición y funcionamiento de XMLC 113
5.2.5.3 Document Object Model 114
5.2.5.4 XMLC como tecnología para la vista de la aplicación 116
5.2.5.5 XMLC y las propiedades de MVC 116
5.2.6 ASP.NET WebForm Pages Framework 118
5.2.6.1 Las partes de un WebForm 118
5.2.6.2 Contribución del framework: interfaces web elaboradas 119
5.2.6.3 Componentes y programación de la Vista: Server Controls, Eventos y Manejadores de Eventos 119
5.2.6.4 Funcionamiento del framework: ciclo de vida de un WebForm 121
5.2.6.4 ASP.NET WebForms y las propiedades de MVC 122
5.3 Conclusiones del análisis de propiedades MVC en la Web 124
5.3.1 Independencia de la vista con respecto al modelo 124
5.3.1.1 Otra forma de obtener independencia: separación de contenido y formato en una

vista 125
5.3.2 Separación entre Vista y Controlador: Independencia de la vista con respecto a su controlador 125
5.3.3 Independencia del Controlador con respecto al Modelo 127
5.3.4 Representación del componente Vista en el servidor 127
5.3.5 Vistas anidadas, controladores anidados 128
5.3.6 MVC a nivel componente en la web 128
5.3.7 Controlador de la aplicación 129
5.3.8 Reuso de la estructura de las vistas 129
5.3.9 Posibilidad de portar interfaces basadas en HTML a otras plataformas 130
5.3.10 Reusabilidad de los componentes visuales 130
5.3.11 Posibilidad de que una sola interfaz se adapte a quien accede a la aplicación . . 130

6. Conclusiones 131
6.1 Sobre las propiedades de MVC 131
6.1.1 Push-Pull MVC frameworks: efectos sobre las propiedades de MVC 131
6.1.2 Las propiedades de la vista Web 133
6.2 Las desventajas del MVC original en el contexto Web 133
6.2.1 Mayor complejidad cuando la aplicación es sencilla 133
6.2.2 Número excesivo de actualizaciones 134
6.2.3 Conexión fuerte entre vista y controlador 134
6.2.4 Acoplamiento estrecho de vistas y controladores a un modelo 134
6.2.5 Ineficiencia de acceso de datos en la vista 134
6.2.6 Cambio inevitable de vista y controlador al portar 135
6.2.7 Dificultad al utilizar MVC con herramientas modernas para construcción de interfaces de usuario 135
6.3 Conclusiones finales 136
6.4 Trabajo Futuro 136
Glosario 138
Bibliografía y documentación consultada 142
Anexo Patrones de diseño 144

Abstract

Esta tesis presenta un estudio elaborado de la arquitectura de software Model-View-Controller que hoy en día es, la más utilizada en el diseño de aplicaciones que requieren interacción con usuarios. El análisis se apoya sobre un conjunto definido de características deseables en una aplicación de software construida con dicha arquitectura. Estas propiedades son luego proyectadas uniformemente sobre distintos frameworks de última generación que implementan Model-View-Controller. Al mismo tiempo, la tesis contiene un estudio de la evolución de MVC, de sus componentes y conceptos, para lo cual ha sido necesario exponer y explicar los obstáculos y problemas que surgieron con la llegada de aplicaciones interactivas de mayor complejidad. En particular el trabajo se aboca al uso de MVC en la Web. Por último, el análisis intenta transmitir la importancia que esta herramienta a cobrado desde sus comienzos hasta la actualidad, por qué su utilización es ventajosa pero no siempre sencilla, y que MVC trae consigo un conjunto de conceptos y prácticas que impactan considerablemente sobre el diseño de una aplicación, sobre su grado de reusabilidad y flexibilidad posteriores y sobre las partes que la componen.

This thesis presents an elaborate investigation of Model-View-Controller software architecture. This architecture is the most popular in the design of applications requiring user interaction. The analysis is based on a set of features which are expected from a software application that follows the architecture being mentioned. These properties are then studied uniformly in several latest-generation frameworks implementing Model-View-Controller. In addition, this thesis includes a study of MVC evolution, its components and concepts. For this purpose, it has been necessary to list and explain a group of impediments and constraints emerging with complex interactive applications. In particular, this project is focus on the use of MVC in the Web context. At last, one of the goals of this analysis is to transmit the value this tool has gained since its old days until the present, let the reader understand why making use of MVC results in a large number of advantages though it is not always the easiest way to build an application, and to make clear MVC brings concepts and practices which considerably affect the design of an application, its later reusability and flexibility and the parts it consists of.

Introducción

La forma en que se construye software ha ido cambiando velozmente. La elección de la estructura de datos y algoritmo adecuados era la clave para el correcto funcionamiento de un sistema se software. Sin embargo, en la actualidad, la existencia de librerías estándar y componentes facilitan el desarrollo del software y el manejo de las estructuras de datos y algoritmos para los programadores. Como resultado, el tiempo que se invertía en decisiones sobre el mejor algoritmo para resolver un problema, ahora puede dedicarse al diseño de la estructura entera de un sistema, a la definición de sus componentes y la relación entre ellos, lo que en conjunto se denomina arquitectura del software. Esta tarea es compleja pero clave para un buen desarrollo y posterior mantenimiento de las aplicaciones de software de hoy en día.

Una arquitectura especifica la estructura fundamental de una aplicación, de manera que todas las tareas que son parte del desarrollo de la aplicación deberán realizarse respetando esta estructura. Ejemplos de tareas son el diseño de subsistemas, la comunicación y colaboración entre las partes del sistema, y su posterior extensión.

Cada arquitectura permite alcanzar objetivos específicos. Model-View-Controller estructura sistemas de software que presenten interacción con el usuario. Estos sistemas son los que comúnmente se denominan “sistemas interactivos”.

Sistemas interactivos – GUI’s

Los antiguos sistemas ofrecían interacción mediante el ingreso de líneas de comandos para ejecutar alguna acción en el programa, lo cual resultaba “eficiente” para los conocedores del lenguaje pero poco amigable para nuevos usuarios, los que se veían obligados a invertir tiempo en aprender cómo interactuar con la aplicación. La llegada de las interfaces gráficas de usuario, usualmente llamadas GUI[1]’s, representa un escalón importante en la evolución de estos sistemas. Una GUI es una interfaz de programa que se vale de las posibilidades gráficas de una computadora para facilitar la utilización de dicho programa. Estas interfaces permiten interactuar amigablemente con una aplicación y los usuarios no requieren de un período de aprendizaje para

[1] Del inglés Graphical User Interface.

empezar a comunicarse con la misma, ya que además, las GUI's mantienen la apariencia de los sistemas operativos actuales: ventanas, menues, iconos y botones, más interacción con el mouse y teclado.

A partir de este punto, las interfaces evolucionan para adaptarse y acercarse a las necesidades del usuario en lugar de que éste tenga que adecuarse a una aplicación en particular. A su vez, si llevamos este concepto a un nivel más alto, nos encontramos con que distintos usuarios desearían interactuar con la aplicación de una manera en particular. Entonces tendríamos que programar distintas interfaces para los distintos usuarios o tipos de usuarios. Al mismo tiempo, las GUI's son componentes de software con lógica propia, estructura y comportamiento. Sumado a que la GUI es el único punto de acceso al sistema que tiene el usuario, este componente se convierte en el más importante, como es de esperar, en el ámbito de las aplicaciones interactivas.

Cómo diseñar una aplicación con GUI

Al desarrollar un sistema que soportaba interacción con un solo tipo de usuario, el acceso a la información y a la funcionalidad básica de la aplicación se entremezclaban con la lógica específica de la interfaz tanto para la presentación como para el control. Sin embargo, esta aproximación es una solución a corto plazo ya que si se deseaba integrar otro tipo de usuario en el sistema, los programadores debían prácticamente reprogramar la aplicación completa para incluir el nuevo tipo de interfaz. El diseño no era flexible, ni se adaptaba a cambios y sus componentes no podían ser reutilizados.

Cuando especificamos la arquitectura de este tipo de sistemas, la intención entonces es mantener su funcionalidad básica independiente de la interfaz de usuario. La razón es que la funcionalidad suele permanecer estable a lo largo del tiempo de vida del sistema, mientras que la o las interfaces de usuario tienden a cambiar o adaptarse. Cuando el sistema soporta distintos tipos de interfaz o interfaces personalizadas, se necesita que los cambios a la interfaz no causen mayores efectos sobre la funcionalidad específica de la aplicación o sobre el modelo de datos.

Un ejemplo de aplicación interactiva

Consideremos un ejemplo muy simple de almacenamiento de información de CD's de música. Un comprador común desea ver ciertos datos de un CD como su título, intérprete, temas que incluye y precio. Por otro lado, un vendedor puede modificar el precio del CD y la cantidad de unidades disponibles del CD, para lo cual necesitará visualizar esta información. Al mismo tiempo, un manager de ventas desea ver cuántas unidades del CD han sido vendidas y para este fin necesita conocer información tal como el stock inicial y las unidades disponibles actualmente para el CD en cuestión, sin importarle ver los temas que el CD incluye.

El ejemplo es sencillo pero suficiente para comprender la necesidad de distintas formas de ver al mismo objeto: una interfaz para el comprador, otra para el manager y así siguiendo. Además, si un dato del CD es modificado en un determinado momento, el cambio debe reflejarse inmediatamente en las otras interfaces que estén desplegando información de dicho objeto. Incluso, no restringiéndonos al ámbito de las aplicaciones GUI tradicionales, otras formas de ver los mismos datos puede ser el caso de una interfaz para un browser web, una vista para un PDA, etc.

El ejemplo nos servirá a lo largo del trabajo para comprender con mayor facilidad los conceptos que analizaremos.

Motivación y Objetivos

Si bien existe documentación donde se define a MVC y se explican procedimientos para su efectiva utilización y, aunque abunda la información acerca de cómo los frameworks más nuevos se ajustan a MVC; no se cuenta con un documento que muestre la relación entre ambas ideas y que a su vez, no sea específico de una implementación en particular.

Este trabajo intenta reflejar cómo resulta la relación entre MVC y las aplicaciones en las que actualmente se lo utiliza. Con este fin, se identifica un grupo de propiedades de MVC y se analiza su implementación en distintos frameworks MVC. Para llevar a cabo el análisis nombrado precisamos de una clara descripción de los frameworks MVC, de una identificación de los roles

modelo-vista-controlador en los frameworks y de una evaluación de la implementación de dichas propiedades en los mismos. Una explicación de MVC en su forma tradicional no puede pasarse por alto y al mismo tiempo, no puede dejar de destacarse su utilización en frameworks para construir interfaces de escritorio, la cual es clave para comprender por qué MVC es la receta predominante en el diseño de aplicaciones GUI tradicionales.

Pasando al ambiente Web, y habiendo destacado las características inherentes a este contexto que impactan sobre MVC, se analizan las consecuencias de estos impactos para dar paso a un conjunto de nuevas propiedades que se esperan en un MVC Web. Estas últimas incluyen nuevos componentes, extensiones al diseño y mejoras al diseño tradicional para obtener mayor reusabilidad y flexibilidad en las aplicaciones web resultantes. Dado que en la actualidad las aplicaciones web se construyen con frameworks Web, tomaremos un conjunto de dichos frameworks y buscaremos estas propiedades en ellos, por dos razones: si un framework incluye estos conceptos, las aplicaciones construidas con él también los contendrán, y segundo, mediante la identificación de estas propiedades en distintos frameworks podremos concluir cuál es la mejor forma de implementarlas e incluirlas a todas ellas.

Contribuciones del trabajo

El análisis realizado contribuye a comprender la importancia de la arquitectura MVC en el diseño de aplicaciones, mostrando cómo ha sido transportada desde su aplicación en sistemas sencillos a sistemas actuales de mayor complejidad y en la web.

A su vez, permite visualizar cómo las responsabilidades de los componentes de MVC han variado desde el MVC original al MVC del presente, mediante la localización de dichos componentes en los frameworks más utilizados para la construcción de interfaces de usuario.

Un resultado importante de este trabajo es que la metodología utilizada para el análisis de propiedades de MVC en distintos frameworks pone de manifiesto, a nivel implementación, que efectivamente MVC es aplicado jerárquicamente y que pueden hallarse patrones para incluir ciertas propiedades analizadas.

Por último, esta investigación sobre MVC puede utilizarse como una referencia que unifica los conceptos contenidos en una aplicación web que sigue los lineamientos de MVC.

Guía sobre los capítulos del trabajo

El trabajo se estructura en cinco capítulos. A continuación se da una breve explicación del contenido de cada uno de ellos.

MVC es presentando como patrón de diseño en el primer capítulo. Se introducen sus componentes, su funcionamiento y se enumeran a grandes rasgos, ventajas y desventajas de su aplicación en el diseño de software.

En el segundo capítulo se describe al framework para construir interfaces gráficas de usuario (GUI) contenido en VisualWorks, localizando los roles de modelo, vista y controlador y explicando cómo interactúan estos componentes. Además, se expone un conjunto de propiedades deseables en MVC, de las cuales algunas son inherentes a MVC y otras son mejoras para un mayor desacoplamiento de sus componentes. En el contexto de estas propiedades se realiza un análisis del framework descrito anteriormente. Java Swing es otro framework ampliamente utilizado para los mismos propósitos que también implementa MVC. De la misma forma, es descrito y analizado en base a las características enumeradas.

El diseño original que propone MVC presenta limitaciones para ser utilizado en las aplicaciones de negocios actuales. El capítulo 3 incluye un análisis de estas restricciones e introduce un cambio en MVC, que se adapta a aplicaciones de mayor complejidad. Estas limitaciones y consecuentes adaptaciones del diseño de MVC surgen de la creciente complejidad y tamaño de las aplicaciones modernas.

El capítulo 4 representa el pasaje de MVC al contexto de la Web, explicando la necesidad de MVC en este ambiente, los factores inherentes a la Web que hacen imposible implementar MVC en la Web en su forma original y las consecuencias de estos factores sobre distintos aspectos de MVC. Luego, se presentan dos soluciones que intentan llevar a MVC a la Web a pesar de los factores Web que complican dicho traspaso.

El capítulo 5 retoma el análisis de frameworks iniciado en el capítulo 2. Se enumeran nuevamente las propiedades de MVC pero esta vez, indicando cuáles de ellas no son aplicables en el contexto de la Web debido a las restricciones de MVC al ser transportado a la web (capítulo 4). Algunas de las propiedades se adaptan al nuevo contexto, a la vez que se agregan otras características que surgen de la Web. Luego se introduce un conjunto de frameworks que implementan MVC en la Web. Siguiendo los lineamientos de análisis utilizados en el capítulo 2 y luego de la descripción de los componentes y funcionamiento de cada framework, se lo analiza en el contexto de las propiedades de MVC listadas en este capítulo. Para una visión general de cómo las características han sido aplicadas, se realiza un resumen de la implementación de las mismas.

Si la intención del lector es simplemente conocer la definición de MVC y algunas de sus implementaciones, el primer capítulo provee una explicación abstracta de la arquitectura y a su vez, los capítulos 2 y 5 tratan sobre implementaciones puras. Por otro lado, si se desea ahondar en el análisis realizado en el presente trabajo y en la evolución de MVC, será necesario seguir estrictamente el orden de capítulos en que el trabajo ha sido estructurado.

Capítulo 1
Fundamentos de MVC

En este primer capítulo presentamos a MVC introduciendo sus componentes y su funcionamiento y enumerando a grandes rasgos, ventajas y desventajas de su aplicación en el diseño de software.

1.1 Contexto de MVC

Las interfaces de usuario son propensas a cambiar. Al agregar funcionalidad a una aplicación, es posible que tengamos que modificar menús para integrar una nueva funcionalidad al sistema. Puede que se requieran interfaces específicas para cada cliente, o portar un sistema de una plataforma a otra con un "look and feel" diferente. Aún una mejora en el sistema de ventanas puede implicar cambios en el código.

A su vez, diferentes usuarios generan distintos requerimientos en la interfaz. En el ejemplo del CD, un vendedor desea modificar la información del CD mediante un formulario (ver figura 1.1). Un gerente de ventas usa el mismo sistema para visualizar información sobre cuántas unidades se vendieron de dicho CD. El comprador solo quiere ver el precio y los temas que el CD contiene, pero no le interesa saber el stock inicial o cantidad disponible. En consecuencia, necesitamos incorporar varios tipos de interfaz de usuario. Además, es posible que se quiera incluir formas de ver el mismo objeto a través de no sólo una interfaz común, sino mediante un browser web, un PDA, etc.

De todo lo anterior notamos que, la construcción de un sistema con la flexibilidad requerida sería muy cara y propensa a errores si ligáramos la interfaz fuertemente con la funcionalidad básica. Necesitamos separar la información del CD de la forma en que es mostrado en la interfaz. Además, si no desarticulamos estos componentes desembocaríamos en el desarrollo y mantenimiento de varios sistemas de software, uno para cada interfaz de usuario, lo cual trae como consecuencia cambios que se propagarían sobre muchos módulos.

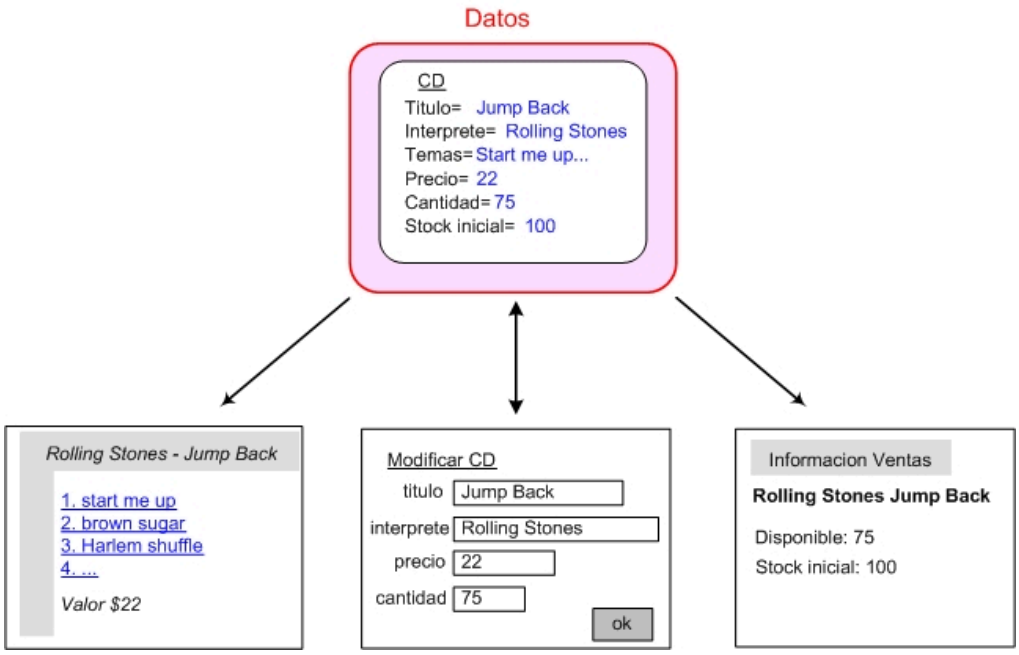


Figura 1.1 – Distintas interfaces de los mismos datos

Pretendemos que

- la misma información pueda ser presentada en forma distinta en diferentes vistas, por ejemplo, una descripción del CD y sus temas y otra con información de disponibilidad.

- la presentación y el comportamiento de la aplicación reflejen los cambios producidos por la manipulación de datos de forma inmediata: si se modifica el stock del CD, las vistas que muestren el stock deben reflejar el cambio.

- los cambios a la interfaz de usuario sean sencillos, y posiblemente realizados en tiempo de ejecución,

- el soporte de diferentes “look and feel” o portar la interfaz de usuario no afecte el código del corazón de la aplicación.

1.2 Qué es Model – View – Controller

Model View Controller es el concepto de encapsular datos y procesamiento (model) aislándolo de su manipulación (controller) y presentación (view) que forman parte de la interfaz de usuario.

1.3 Estructura de MVC

El **modelo** contiene los datos y la funcionalidad básica de la aplicación. Es una representación del mundo real. Encapsula información interna y exporta procedimientos para realizar procesamiento específico de la aplicación. Además, provee funciones para poder acceder a sus datos. El modelo es independiente de la representación específica de la salida, y del comportamiento de la entrada. Es un componente de procesamiento.

La **vista** es la presentación del modelo y se comunica con él para obtener la información necesaria que debe mostrar. Cuando decimos vista, nos referimos a una o más vistas que muestran información del mismo modelo. La vista es el componente de salida de la aplicación.

El **controlador** es un componente de entrada. Recibe pedidos en forma de eventos y los transforma en llamadas a los procedimientos que el modelo exporta para realizar algún procesamiento. Por lo tanto el controlador es el punto de acceso a la aplicación, y es el componente que activa el funcionamiento del sistema.

En nuestro ejemplo, el CD es el modelo mientras que los otros tres componentes (en la figura 1.1) son vistas de este modelo. Una posible acción sobre el modelo es la modificación de las propiedades del CD. Esta acción es realizada por un controlador que es invocado desde la interfaz correspondiente.

1.4 Relación entre componentes de MVC

Si el controlador recibe un pedido de acción y modifica al modelo, los cambios deberán reflejarse en todas las vistas del modelo para que puedan mantenerse consistentes con él. Esto es básicamente, el ciclo de acciones que sucede en una aplicación construida con estilo MVC.

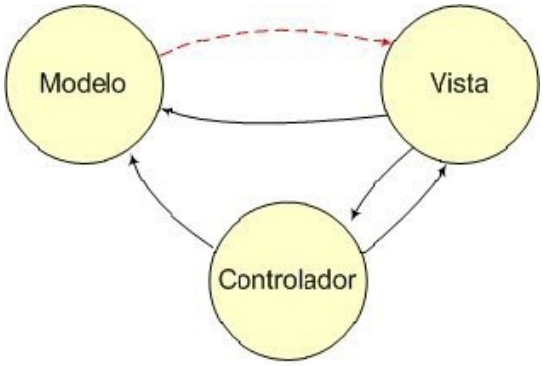


Figura 1.2- MVC Componentes y relaciones.

Para que este ciclo funcione, los componentes de MVC deben estar relacionados de alguna manera. Así, las vistas y los controladores conocen exactamente a su modelo, y para esto contienen una referencia explícita a él como puede verse en la figura 1.2. El modelo, por el contrario, no debe conocer detalles sobre qué vistas lo están mostrando ni qué controladores activarán su funcionalidad. Sin embargo, existe una relación débil, pero relación al fin, entre el modelo y los demás componentes que quieran mantenerse sincronizados con él. En el dibujo, esta relación esta marcada en rojo punteado, y si bien va desde el modelo a la vista, también podría suceder que un controlador esté interesado en sincronizarse con el modelo.

Por último, cuando una vista es creada, también se crea un controlador para ella. Este controlador conoce a la vista que lo ha creado.

1.4.1 Pattern Observer

También conocido con el nombre Publish-Subscribe, Observer [17] es un patrón de diseño que describe comportamiento de objetos. Su intención es definir una dependencia uno-a-muchos entre objetos de modo que, cuando un objeto cambie su estado, sus dependientes sean notificados y se actualicen automáticamente.

En el contexto de MVC, los datos de la aplicación son separados de los aspectos de presentación, por lo que existe la necesidad de mantener la consistencia entre estas dos partes.

Los objetos clave son **Subject** y **Observer**. El Subject puede tener cualquier número de observers que dependan de él. Todos los observers serán notificados cuando suceda un cambio en el estado del Subject y como respuesta a la notificación, cada observer interrogará al Subject para sincronizar su estado con el del Subject.

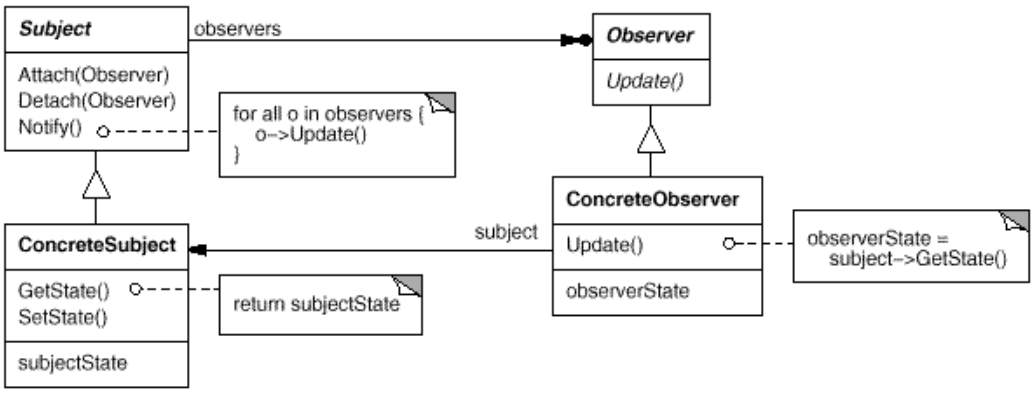


Figura 1.3 – Estructura de Observer

El **Subject** conoce a sus observers y cualquier objeto Observer puede observar un Subject. Además, el Subject provee una interfaz que permite a los observers expresar su interés en recibir notificaciones (attach() en el dibujo) y también declarar que ya no quieren recibirlas (detach()).

El **Observer** define una interfaz de actualización (update) para los objetos que deben ser notificados sobre cambios en el Subject.

ConcreteSubject almacena el estado en el cual los objetos **ConcreteObserver** están interesados y les enviará una notificación cada vez que suceda algún cambio en su estado.

ConcreteObserver mantiene una referencia a su ConcreteSubject, almacena el estado que debe ser consistente con el del Subject e implementa la interfaz definida en Observer para mantener esa consistencia.

Los objetos descriptos colaborarán de la siguiente manera: ConcreteSubject notificará a sus observers cuando ocurra un cambio que pueda provocar inconsistencia entre su estado y el de sus observers. Luego de haber sido notificado sobre un cambio, un ConcreteObserver podrá interrogar al Subject requiriendo información que será empleada para equilibrar su estado con el del Subject.

1.5 Funcionamiento de MVC

Una de las aplicaciones de Observer es el caso de un objeto que debe notificar a otros objetos sin conocer nada acerca de quiénes son ni de cómo son realmente. En otras palabras, no se desea que estos objetos estén ligados fuertemente al primero. En MVC, este patrón está implementado entre el modelo y sus dependientes. El modelo juega el rol de Subject mientras que las vistas son sus observers. Este mecanismo es la base del funcionamiento de MVC.

Todo componente de la interfaz de usuario, que esté interesado en los cambios en el estado del modelo, deberá registrarse a él. Estos componentes pueden ser vistas y también controladores.

Cuando un controlador recibe un pedido de acción sobre el modelo, lo transforma en llamadas a procedimientos exportados por el modelo. Si la acción modifica su estado actual, el modelo notifica a sus “observadores” que su estado ha sido modificado sin necesidad de indicar cual fue el cambio exactamente y produce así el disparo del mecanismo de propagación

nombrado. Con esta notificación, las vistas asociadas al modelo se actualizan recuperando información del modelo. Los controladores que estén también asociados a él, evalúan el nuevo estado y realizan cambios tales como restringir alguna acción de entrada que antes estaba permitida en la vista asociada. Un ejemplo de esto último sería deshabilitar un menú.

En resumen, cuando un controlador modifica al modelo respondiendo a una acción de usuario, las vistas y controladores se actualizan, manteniéndose consistentes con el modelo.

En cualquier momento, una vista o controlador podría dejar de estar interesado en recibir notificaciones de cambios. Por lo tanto, es posible desregistrarse del modelo con este fin.

No toda interacción comienza con una comunicación entre controlador y modelo. Puede suceder que el controlador interactué directamente con la vista sin pasar por el modelo. Si la vista tiene una lista de los temas del CD y el controlador requiere que la lista se muestre en orden alfabético, esto no necesitará de una actualización de los datos desde el modelo.

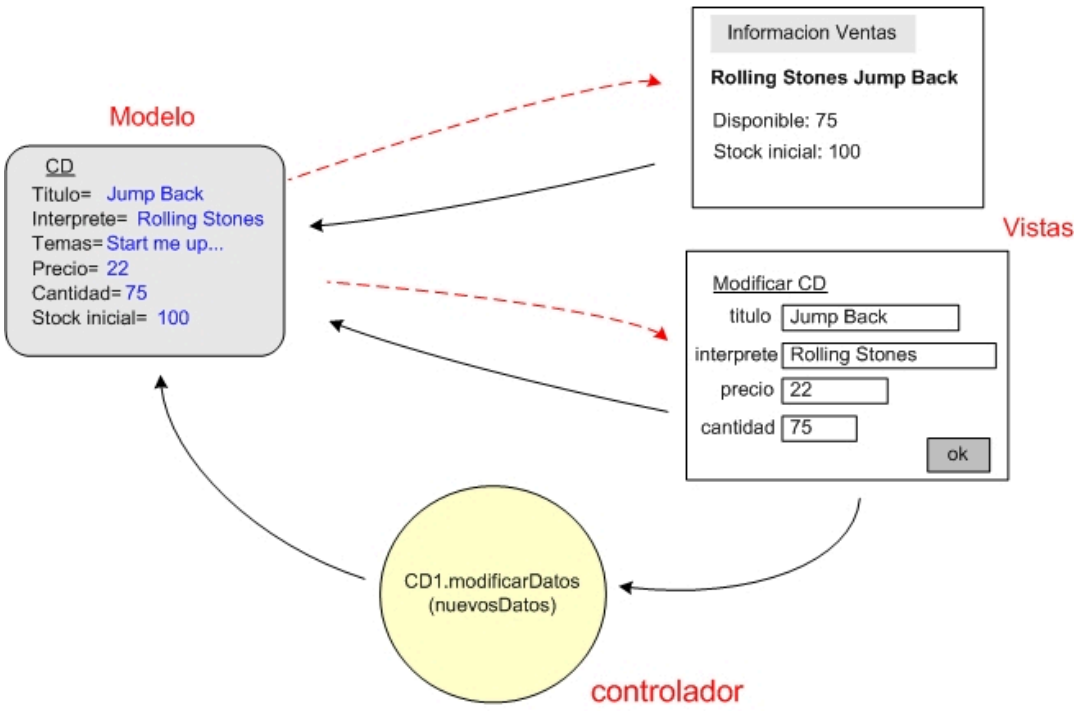


Figura 1.4 – Un ejemplo de funcionamiento MVC

En el esquema de la figura 1.4, la vista "información ventas" muestra la cantidad de unidades disponibles del CD, en este caso 75. A su vez, la segunda vista despliega información del CD y permite editar sus datos. El usuario que modifica la cantidad disponible activa al controlador que cambia los datos del CD. Esta modificación resulta en la notificación de las vistas por parte del modelo, las cuales obtendrán nuevamente la información del modelo para mantenerse consistentes con él.

1.6 Consecuencias del uso de MVC

A continuación explicamos cómo MVC permite lograr los objetivos que se mencionaron

- la misma información pueda ser presentada en forma distinta en diferentes vistas

1.6.1 Múltiples vistas del mismo modelo. MVC separa estrictamente el modelo de los componentes de interfaz de usuario. Por lo tanto, pueden implementarse varias vistas y usarlas con el mismo modelo. En tiempo de ejecución, pueden abrirse múltiples vistas al mismo tiempo, y las mismas pueden ser abiertas y cerradas dinámicamente.

- la presentación y el comportamiento de la aplicación reflejen los cambios producidos por la manipulación de datos de forma inmediata.

1.6.2 Vistas sincronizadas. El mecanismo de propagación-de-cambios del modelo asegura que todos los observers registrados sean notificados de los cambios a los datos de la aplicación en el momento justo. Esto sincroniza todas las vistas y controladores dependientes con el modelo.

- los cambios a la interfaz de usuario sean sencillos, y posiblemente realizados en tiempo de ejecución.

1.6.3 Vistas y controladores conectables. La separación conceptual de MVC permite cambiar los objetos vista y controlador de un modelo. Los objetos de interfaz de usuario pueden ser sustituidos en tiempo de ejecución.

- el soporte de diferentes "look and feel" o portar la interfaz de usuario no afecte el código del corazón de la aplicación.

1.6.4 Intercambio de 'look and feel'. Como el modelo es independiente del código de interfaz de usuario, portar una aplicación MVC a una nueva plataforma no afecta la funcionalidad básica de la aplicación. Solo se necesitan implementaciones apropiadas de la vista y controlador para cada plataforma.

Si quisiéramos que la información del CD fuese accesible vía la web, solamente sería necesario crear vistas y controladores adecuados. En otras palabras: el modelo permanecería intacto.

1.7 Algunas desventajas del uso de MVC

1.7.1 Mayor Complejidad. Seguir la estructura MVC de manera estricta no siempre es el mejor modo de construir una aplicación interactiva. La separación en modelo, vista y controlador para el caso de elementos de texto simples incrementa la complejidad sin ganar mucha flexibilidad.

1.7.2 Numero excesivo de actualizaciones. Si una sola acción de usuario resulta en muchas actualizaciones, el modelo debería saltear las notificaciones de cambios que sean innecesarias. Puede suceder que no todas las vistas estén interesadas en todos los cambios propagados por el modelo en un determinado momento. Por ejemplo, una vista con una ventana iconizada puede no necesitar actualizarse hasta que la ventana sea restaurada a su tamaño normal.

Si el usuario modifica únicamente el precio del CD, entonces solamente las vistas que muestran este aspecto del CD deberían ser actualizadas y a su vez, no deberían actualizar todos sus componentes sino aquel que muestre el valor precio.

1.7.3 Conexión fuerte entre vista y controlador. Los controladores y vistas están separados pero son componentes muy relacionados, lo que dificulta su reuso individual. Es muy

difícil que una vista pueda ser usada sin su controlador, o viceversa, con la excepción de vistas de sólo lectura que comparten un controlador que ignora todo ingreso de datos.

1.7.4 Acoplamiento estrecho de vistas y controladores a un modelo. Ambos componentes vista y controlador hacen referencias directas al modelo. Entonces los cambios en la interfaz del modelo probablemente hagan que el código de la vista y el del controlador deban ser modificados. Este problema es mayor aún si el sistema usa muchas vistas y controladores.

1.7.5 Ineficiencia de acceso de datos en la vista. Dependiendo de la interfaz del modelo, la vista requiere de múltiples llamadas al modelo para obtener la información necesaria para mostrarse. Los pedidos innecesarios al modelo sobre datos que no han cambiado, debilitan la performance si las actualizaciones son frecuentes. Una solución sería almacenar (cacheo) la información dentro de la vista misma para mejorar el grado de respuesta.

1.7.6 Cambio inevitable de vista y controlador al portar. Todas las dependencias de la plataforma de interfaz de usuario están dentro de la vista y el controlador. Sin embargo, ambos componentes además contienen código que es independiente de la plataforma específica. Si las dependencias de la plataforma están directamente embebidas en la vista y el controlador, no será posible cambiar la plataforma en tiempo de ejecución y todo el código dependiente de la plataforma deberá ser reescrito para que los componentes funcionen en la plataforma nueva al portar el sistema.

1.7.7 Dificultad al usar MVC con herramientas modernas para construcción de interfaz de usuario. Si la portabilidad no es algo indispensable, usar kits de herramientas de alto nivel o constructores de interfaces de usuario puede quitar las posibilidades de usar MVC. Usualmente, resulta caro mejorar los componentes de estas herramientas o las interfaces de usuario construidas con ellas para que estén orientadas a MVC. Para realizarlo sería necesario al menos definir un nuevo componente que encapsule el ya existente en la herramienta. Además, muchas herramientas de alto nivel (toolkits) definen su propio flujo de control y manejan eventos internamente, tales como desplegar menús pop-up o scrolling en una ventana. Finalmente, una plataforma de interfaz de usuario de alto nivel puede ya interpretar eventos y ofrecer llamadas a procedimientos (callbacks) para cada tipo de actividad de usuario. Por lo tanto, la mayor parte de la funcionalidad de los controladores ya esta provista por el toolkit, y no es necesario implementarlo explícitamente.

Capítulo 2
Uso de MVC en GUI's tradicionales

Llamaremos GUI[1] tradicional a una interfaz gráfica de usuario para una aplicación de escritorio. En este capítulo presentamos dos ejemplos de implementaciones de MVC para GUI's tradicionales. El primero es el framework para interfaces gráficas de usuario existente en VisualWorks. El segundo, es el framework Java Swing para la construcción de interfaces en Java.

Mostrar la utilización de MVC en frameworks para construir interfaces de escritorio es clave para comprender por qué MVC es la receta predominante en el diseño de aplicaciones GUI tradicionales.

Para cada framework, se identificarán los componentes que cumplen el rol de modelo, vista, y controlador respectivamente y se mostrará cómo estos componentes interactúan entre sí. Luego se listan una serie de características que se consideran conceptualmente importantes para explotar al máximo las propiedades de MVC, explicándolas fuera del contexto de un framework en particular. Seguidamente de la presentación de un framework, se toma cada una de estas propiedades indicando si el framework la soporta y cómo. En caso de ser necesario, se delineará una posible solución para implementarla.

2.1 La implementación de MVC en VisualWorks

VisualWorks es un sistema para desarrollar aplicaciones con el lenguaje orientado a objetos Smalltalk. Dentro de las clases provistas por este sistema, existe un conjunto que conforma un framework para interfaces gráficas de usuario para aplicaciones interactivas. Este framework implementa MVC. Smalltalk-80 es el primer sistema que incluyó a MVC y por esta razón, se le atribuye su origen.

[1] Del término en inglés Graphical User Interface

Las clases del framework permiten la separación de los objetos que son parte de la UI[1], de los que son parte del dominio. El programador primero construye clases que representan el dominio específico de su aplicación, esto es, la funcionalidad básica de la aplicación que constituye el Modelo. El paso siguiente es diseñar la interfaz de usuario armando una vista compuesta o ventana "pegando" componentes que son instancias de clases también provistas por el framework.

La implementación de MVC en VisualWorks consiste principalmente de tres clases abstractas cuyos nombres son *Model, View* y *Controller*. Estas clases contienen el código que representa el estado y comportamiento de cada uno de los componentes de MVC. Las subclases de estas clases almacenan comportamientos mas específicos, y están disponibles para ser instanciadas y usadas en una aplicación. Un ejemplo de clase concreta es una que representa a un menú, la cual puede instanciarse y agregarse a una interfaz.

2.1.1 Clase *Model:* soporte de dependencias para MVC

En Smalltalk, la clase *Object* contiene una colección para almacenar dependencias entre objetos. Cada dependencia está formada por un objeto y una lista de dependientes de ese objeto. *Object* se encarga de propagar los cambios de un objeto a sus dependientes, mecanismo necesario para el funcionamiento de MVC. Entonces, las dependencias pueden existir entre cualquier tipo de objeto.

La clase *Model* introduce una mejora para dependencias específicas entre un objeto **modelo** y sus dependientes. La clase que actúa como modelo, debe definirse como subclase de *Model*. *Model* retiene sus dependientes, es decir que, estas dependencias no están incluidas en la colección global de *Object*. Las vistas confían en este mecanismo para ser notificadas sobre cambios de su modelo. Cuando un modelo es asignado a una vista, la misma se agrega como un dependiente del modelo y cuando la vista se cierra, se remueve de esta colección de dependencias. El protocolo necesario para el agregado y remoción de vistas está en la clase *Object.*

[1] Interfaz de usuario

Asimismo, el protocolo de mensajes para la propagación de cambios y subsiguiente actualización de las vistas en respuesta a ellos, se encuentra en la clase *Object*. Cuando un modelo cambia se manda el mensaje *changed* a sí mismo, lo que produce el envío del mensaje *update* a sus dependientes. La implementación de *update* contiene las acciones que el objeto dependiente realiza en respuesta a la notificación. En el caso de los dependientes que son vistas, se espera que éstas recuperen la información necesaria del modelo para actualizarse y mantenerse consistentes con él.

Existen variantes de *changed* tales como *changed:unSimbolo* y *changed*:*unSimbolo with*:*unParametro* que causan el mismo efecto sobre los dependientes recibiendo el mensaje *update:unSimbolo* y *update:unSimbolo with:unParametro* respectivamente. El mensaje *changed:unSimbolo* se utiliza para indicar a los dependientes sobre qué aspecto ha cambiado. El símbolo representa un determinado cambio en el modelo. El agregado de *with:unParametro* permite enviar un parámetro adicional en el mensaje. En general este parámetro es alguna parte del modelo. Los dependientes evalúan los parámetros y realizan, de acuerdo a los mismos, una acción o no en respuesta al cambio del modelo.

Para agregar un nivel de indirección a esta comunicación entre modelo y dependientes, se puede agregar un objeto de la clase *DependencyTransformer* el cual recibe el mensaje *update* cuando el modelo cambia. Cuando el *dependencyTransformer* recibe un *update* lo "transforma" enviando mensajes específicos a determinados objetos, que son los dependientes del modelo. Por ejemplo, el mensaje

unModel onChangeSend: #unMensaje to: unDependiente

especifica que cuando el objeto *unModel* cambie, se envíe el mensaje *unMensaje* al objeto *unDependiente*. El receptor debe implementar el método correspondiente. Al enviar un mensaje de este tipo al modelo, se genera automáticamente un *dependencyTransformer* que guarda al objeto dependiente y al mensaje que debe enviarle en caso de cambiar su modelo. En realidad el que realmente es dependiente del modelo es el *dependencyTransformer* pero a los ojos del programador esta indirección es invisible.

2.1.2 La Vista de MVC en VisualWorks

La jerarquía de clases *VisualPart* y la clase *ScheduledWindow* representan al componente Vista de MVC. La interfaz de una aplicación consiste de una ventana, que es instancia de *ScheduledWindow* y de componentes de interfaz o widgets[1] que son instancias de alguna subclase de *VisualPart*. Ejemplos de widgets son *input fields, buttons*, *text displays.*

2.1.2.1 Clase VisualPart

La clase *VisualPart* es subclase de *VisualComponent* que es la superclase de otras tres jerarquías de clases con raíz en *DependentPart, Wrapper* y *CompositePart.* La primera, *DependentPart,* provee la habilidad para que las instancias de sus subclases puedan re desplegarse al recibir el mensaje *update:*. Las instancias de *DependentPart* entonces pueden redibujarse, y así ser usadas para representar aspectos del modelo. **Además esta clase provee una variable *model* para referenciar a su modelo.**

La clase abstracta *View,* subclase directa de *DependentPart,* es la superclase para la creación de vistas específicas. El framework provee numerosas subclases de *View,* como text views, list views, buttons y switches. **La clase *View* contiene la funcionalidad básica de una vista, incluyendo comportamiento para la interacción con su modelo y su controlador.** Para este fin contiene una variable de instancia llamada *controller* y el protocolo necesario para asignar un modelo, asignar y acceder el controlador. Además esta clase define el método *defaultControllerClass* que explicaremos más adelante.

Las vistas son compuestas. En otras palabras, pueden contener otras vistas en su interior, las cuales a su vez, también pueden incluir más vistas. Se dice entonces que una vista puede mantener uno o más *componentes* en su interior y estar contenida en un objeto denominado *container*.

[1] Widget es el nombre con que se denomina a un componente de interfaz de usuario en el ambiente VisualWorks

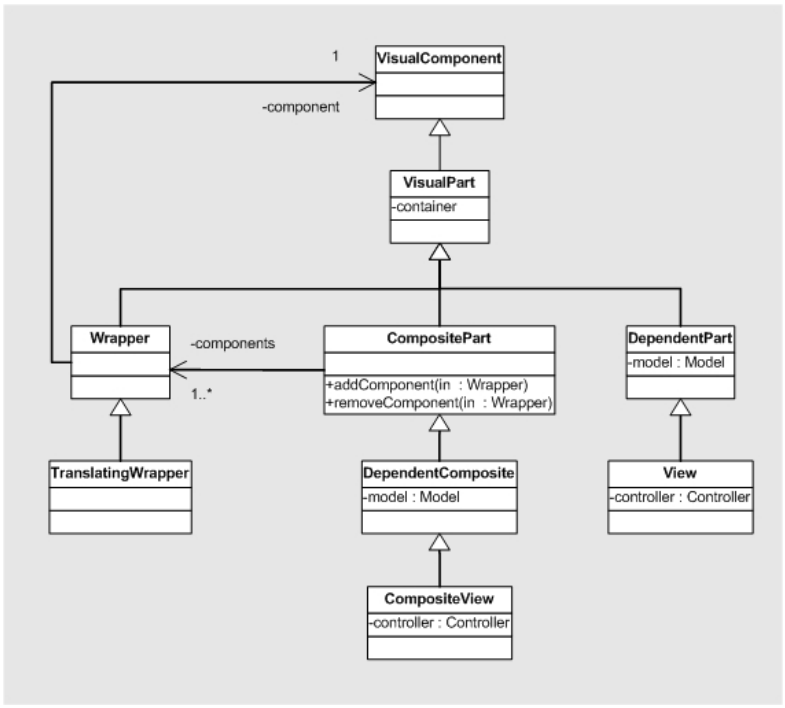


Figura 2.1 – Jerarquía de clases de VisualComponent

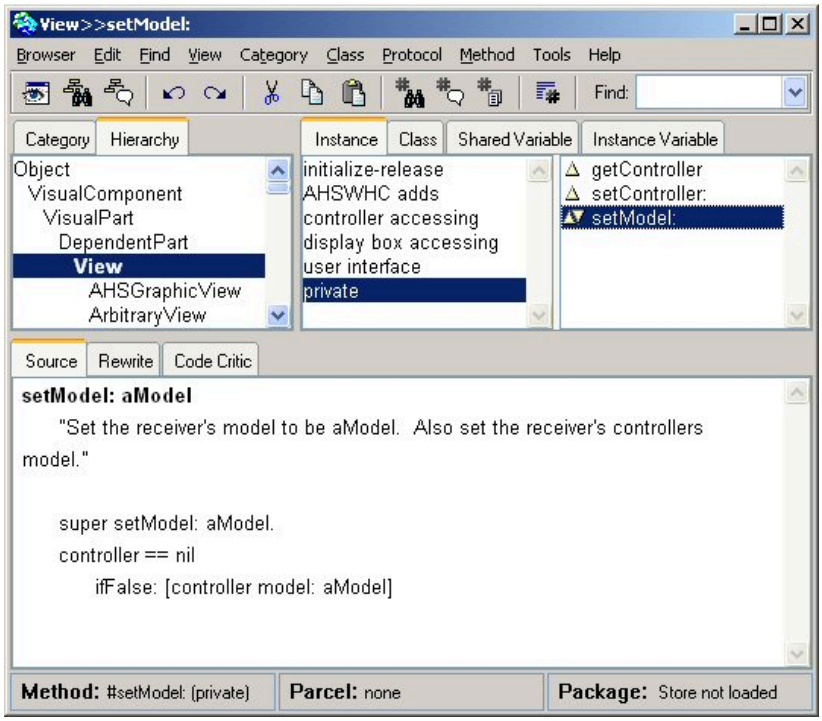


Figura 2.2 - Un SystemBrowser

Como ejemplo observamos un *SystemBrowser* de VisualWorks en la figura 2.2, donde cada uno de los paneles es un componente dentro de un solo container.

Un componente se dibuja dentro de los límites de su container. El container que no está contenido en ningún otro, se denomina *top component.*

2.1.2.2 Clase ScheduledWindow: la ventana de la interfaz

Las ventanas de una aplicación son instancias de *ScheduledWindow* (subclase de *Window)* o bien de alguna de sus subclases. Esta clase contiene el comportamiento necesario para participar en una estructura de componentes UI. Por lo tanto, una *ScheduledWindow* es el *top component* de una estructura de componentes *VisualPart* con conexiones a la pantalla donde se despliega la ventana y al objeto *WindowManager.*

La clase *ScheduledWindow* no es parte de la jerarquía de vistas *VisualPart* pero participa en un MVC ya que tiene un controlador (de la clase *StandardSystemController)* y puede tener un modelo. Solamente puede contener un componente, instancia de *DependentPart, CompositePart*, o *Wrapper* o alguna de sus subclases. Tiene además una instancia de LookPreferences que determina su apariencia interna. Se dice que una ventana "puede" tener un modelo porque en general, una ventana no depende de un modelo. Existen muy pocas ocasiones en las que la ventana en sí necesita variar de acuerdo a cambios de un modelo. Son las partes que la componen, widgets, las que reaccionan a éstos cambios.

2.1.2.3 Clases para composición de vistas: Wrapper y CompositePart

La clase *Wrapper* es subclase de *VisualPart*. Una instancia de esta clase contiene un solo componente en una variable denominada *component* y controla la disposición de los componentes si estos últimos se encuentran dentro de una instancia *CompositePart.* Las subclases de *Wrapper* provistas por VisualWorks son las que habitualmente se utilizan.

CompositePart es otra subclase de *VisualPart* y es su versión compuesta. Las instancias de *CompositePart* se usan como contenedores dentro de la estructura de una ventana. Un *CompositePart* contiene instancias de *Wrapper* o alguna de sus subclases. A su vez, cada uno de estos componentes contiene una instancia de alguna de las subclases de *VisualComponent. CompositePart* tiene una subclase denominada *DependentComposite,* la cual permite construir una estructura de componentes donde todos ellos dependen de un solo modelo. ***CompositeView* subclasea a *DependentComposite* para referenciar un controlador**, y es la versión compuesta de *View.*

CompositePart provee mensajes para el agregado de componentes.

En resumen, dentro de la estructura de una ventana, las hojas de la estructura son instancias de *DependentPart* como lo es *TextCollectorView.* Las instancias *Wrapper* o alguna de sus subclases son nodos interiores en la estructura y contiene a un solo componente simple o compuesto. Por último, una instancia de *CompositePart* o de alguna de sus subclases es un nodo interior que contiene un número arbitrario de *wrappers.*

2.1.2.4 El método que dibuja a un componente vista

Todas las vistas responden al mensaje *displayOn:,* el cual hace posible el despliegue de un aspecto del modelo. La implementación de este método es una secuencia de mensajes enviados al objeto *GraphicsContext* que viene como parámetro en *displayOn:.*

Cuando una vista es notificada de que la información que está mostrando no es consistente con su modelo, se envía a sí misma el mensaje *invalidate* o *invalidateRectangle*: con un rectángulo como parámetro para indicar cual es el área invalidada. Este mensaje se reenvía a todos los objetos que se encuentran en el camino hacia arriba en la estructura de vistas, hasta llegar a la instancia ventana, que usualmente es instancia de *ScheduledWindow.* Esta última acumula todos los rectángulos y luego envía un mensaje *displayOn:* a su componente para que este se re dibuje con un rectángulo apropiado. Si este componente está compuesto por otros componentes, entonces la acción de dibujarse consiste en que sus componentes se dibujen. El proceso de acumulación de áreas inválidas se usa para re dibujar todas las áreas dañadas de manera eficiente, evitando que un área sea dibujada más de una vez.

2.1.3 El Controlador en VisualWorks

En VisualWorks, las vistas y los controladores trabajan de a pares. Cada controlador debe tener una vista para permitir que el usuario dispare eventos en la misma a través del mouse o el teclado. Una vista necesita estar asociada a un controlador para que cuando se genere un evento, el controlador lo tome y realice una acción en respuesta al pedido del usuario.

Cada instancia de *ScheduledWindow* y cada componente *view* tienen su controlador. Por lo tanto, un usuario interactúa con varios controladores activos a la vez. Cuando un usuario clickea sobre un componente vista que representa una lista, existen dos controladores involucrados en interpretar el evento, el de la lista y el de la ventana que la contiene. Ambos controladores responden al evento clic de distinta manera pero en este caso es el controlador de la lista el que debería responder. Existe entonces un flujo de control entre controladores gobernado por ellos mismos, en el cual cada controlador debe decidir si tomar el control o pasarlo a otro controlador.

2.1.3.1 Clase Controller

La tarea del controlador en MVC Smalltalk es manipular la funcionalidad de un modelo y una vista en particular. La clase abstracta *Controller* contiene tres variables de instancia: *model, view* y *sensor*. La última es generalmente una instancia de la clase *InputSensor* que provee soporte de bajo nivel para dispositivos de teclado y mouse.

La interpretación de cómo se comporta un dispositivo de entrada depende mucho de la aplicación. Por esta razón, la clase *Controller* presenta un comportamiento casi nulo para esta tarea, pero permite saber si la vista del controlador contiene al cursor o no en un determinado momento.

Controller provee una secuencia de control por defecto para responder a la entrada de usuario. Los métodos que proveen este comportamiento son redefinidos por las subclases de *Controller* para definir comportamiento específico.

2.1.3.2 El flujo de control de los controladores

Existen dos mecanismos de flujo de control. Uno es el que se da entre ventanas de la aplicación y otro es el que se produce dentro de una ventana. Una sola ventana puede tener el control en determinado momento, así que se necesita un objeto que determine qué ventana posee el control. Este rol es llevado a cabo por un objeto *ControlManager,* único en el sistema, que contiene a todos los controladores de cada ventana, es decir, a los controladores de cada *ScheduledWindow.*

Para determinar qué ventana tiene que obtener el control, el *ControlManager* envía a cada controlador el mensaje *isControlWanted*. El controlador que responda verdadero recibe el mensaje *startUp.* Es responsabilidad de este controlador determinar cual de los controladores de los componentes de la ventana debe tomar el control. Con este fin, cada ventana pregunta a su componente qué subcomponente requiere el control enviándole el mensaje *objectWantingControl*. Cuando un componente de la vista recibe este mensaje, este redirecciona la pregunta a su controlador mediante el mensaje *isControlWanted*. Si el componente visual es compuesto, reenviará el mensaje *objectWantingControl* a cada subcomponente. Esta cadena de mensajes finaliza cuando un componente responda verdadero al mensaje *objectWantingControl*. El controlador de la ventana enviará a este componente el mensaje *startUp.* Finalmente el componente redirecciona el mensaje a su controlador el cual obtiene así el control y comienza lo que se conoce como "secuencia de control".

El controlador que obtuvo el control se envía a sí mismo el mensaje *controlInitialize,* luego *controlLoop* y finalmente *controlTerminate.* Dentro de *controlLoop*, el controlador se envía el mensaje *isControlActive* para chequear que todavía mantiene el control. Si la respuesta es afirmativa, toma el evento que se generó y lo procesa y repite *isControlActive.* Si en algún momento el mensaje retorna falso, el *controlLoop* se da por terminado y el control vuelve al objeto que envió el mensaje *startUp* a este controlador.

Las acciones que se tomen luego de verificar que el controlador aún tiene el control, son las tareas relevantes para manejar el evento generado que dio origen a esta búsqueda del controlador adecuado.

En la clase *Controller* el método *controlLoop* no realiza ninguna acción. El método *eventLoop,* que es invocado desde *controlLoop* en subclases de *Controller* debería ser redefinido cuando un controlador quiera realizar algún tipo de acción. También *controlInitialize* y *controlTerminate* son métodos redefinidos en algunas clases provistas por el framework.

2.1.3.3 Algunas clases de la jerarquía Controller

Las vistas incluidas en el framework tienen un tipo de controlador asociado que es creado junto con ellas. Este controlador es obtenido al enviar el mensaje *defaultControllerClass* a la vista y pertenece a alguna de las clases de la jerarquía *Controller.*

La clase *ApplicationStandardSystemController* subclase de *StandardSystemController* (la cual es subclase de *Controller)* está diseñada para ser el controlador de una instancia de *ScheduledWindow* o de alguna de sus subclases. Los controladores que almacena el *ControlManager* son generalmente instancias de esta subclase. *ApplicationStandardSystemController* redefine el método *controlLoop* para que al hacer un clic con el botón <window>[1] se despliegue el menú típico de operaciones sobre ventanas que permiten moverla, redimensionarla, cerrarla, etc.

Un controlador de la clase *NoController* rechaza siempre el control por tanto la vista como el modelo asociados a él no pueden ser editados. Esta subclase de *Controller* redefine los métodos *isControlWanted* y *isControlActive* para que retornen falso.

La clase *ControllerWithMenu*, subclase directa de *Controller*, provee el comportamiento para la aparición de un menú cuando se hace clic con el botón <operate>[2], que es el menú de operaciones tales como inspeccionar un objeto o ejecutar un programa. *ControllerWithMenu* actúa como una clase abstracta para las clases de controladores específicos de aplicaciones que requieran menúes de opciones personalizados.

Controller define los métodos *blueButtonPressedEvent:, redButtonPressedEvent:* y *yellowButtonPressedEvent:* que representan el comportamiento realizado al clickear sobre los

[1] El botón <window> no se refiere a un botón específico de un mouse. Aunque en general se asocia la funcionalidad <window> con el botón del medio de un mouse convencional.
[2] Comúnmente <operate> se asocia al botón derecho de un mouse convencional.

distintos botones del mouse denominados <window>, <select>[1] y <operate> respectivamente. La implementación de estos métodos en *Controller* es vacía.

ApplicationStandardSystemController redefine el método *blueButtonPressedEvent:* para desplegar el menú de acciones posibles sobre una ventana. *ControllerWIthMenu* redefine el método *yellowButtonPressedEvent*: implementando el menú de operaciones (copiar, pegar, inspecciones de objetos,etc).

Un objeto *ControllerWithMenu* mantiene una referencia al menú en una variable denominada *menuHolder* que contiene una instancia de *Menu.* El método *initializeMenu* en *ControllerWithMenu* inicializa a la variable *menuHolder.* Es este método el que las subclases de *ControllerWithMenu* redefinen cuando se requiere un menú con operaciones específicas. Un ejemplo de clase que hereda de *ControllerWithMenu* es *SequenceController* que es el controlador para las widgets campos de texto. Un controlador de este tipo permite reaccionar ante la entrada de datos desde teclado en la widget. Redefine los métodos *yellowButtonPressedEvent:* y *redButtonPressedEvent:*. La implementación de *redButtonPressedEvent:* permite que el cursor aparezca dentro de la widget, si ésta es editable.

2.1.3.4 Clases InputSensor y Cursor

Un controlador contiene una instancia de *InputSensor*. Todos los tipos de controladores provistos por el framework poseen un sensor de la clase *TranslatingSensor* (subclase de *InputSensor).TranslatingSensor* provee funcionalidad para traducir coordinadas de acuerdo a las coordenadas de la vista del controlador. La clase *InputSensor* provee métodos que determinan qué botón ha sido cliqueado, la posición actual del cursor y qué caracter ha sido ingresado vía el teclado.

Cursor representa la forma del cursor. Existen muchos métodos de clase que devuelven una instancia *Cursor* diferente. Por ejemplo, el mensaje *normal* devuelve un cursor con la forma natural de flecha apuntando a la izquierda.

[1] <select> es el botón de selección, generalmente el botón izquierdo.

2.1.4 Conexión entre los tres componentes de MVC

Los tres componentes vista, modelo y controlador se relacionan de una determinada manera en VisualWorks. Cada instancia de *View* tiene exactamente un modelo y un controlador. El modelo para una determinada vista suele fijarse explícitamente. En cuanto al controlador de la vista, éste puede ser una instancia de cualquier clase *Controller* o una instancia de la clase que el framework tiene predestinada a ese tipo de vista. Tanto para clases vista ya definidas por el framework como para cualquier otra construida por el programador, el controlador asociado a la vista se obtiene del mensaje *defaultControllerClass*, el cual está definido en la mayoría de las subclases de *View* o que debe definirse por el usuario. Por ejemplo, la clase vista *InputFieldView,* redefine el método *defaultControllerClass* para devolver una instancia de *InputBoxController.*

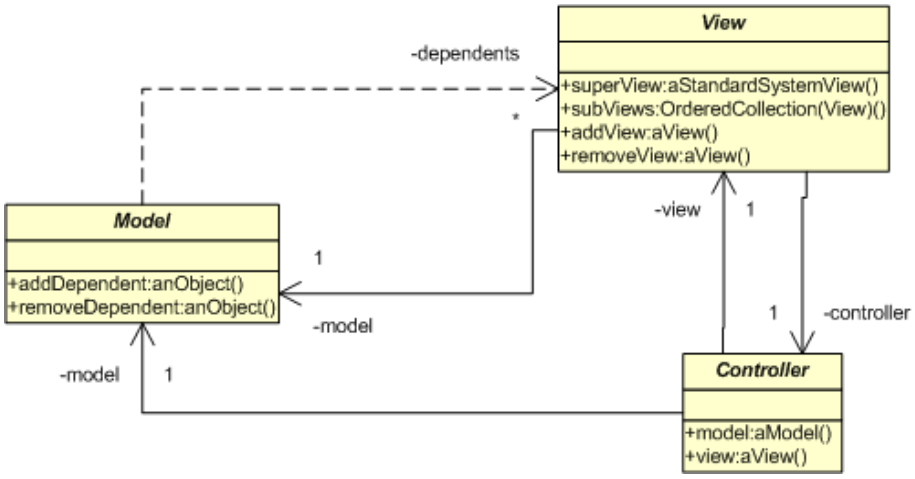


Figura 2.3- Componentes de MVC en VisualWorks.

Al enviar el mensaje *model:unObjeto* a una vista, su controlador es inicializado con una instancia de la clase controlador por defecto para este tipo de vista con *defaultControllerClass*. Además, este mensaje *model:unObjeto* establece a *unObjeto* como modelo para esa vista y ese controlador. Por último, la vista es agregada como dependiente del modelo.

Cuando se cierra una vista, se le envía automáticamente el mensaje *release*, lo que produce la ruptura del enlace entre la vista y el controlador y remueve a la vista de la colección

de dependientes de su modelo. Cuando una ventana de una aplicación se cierra, el mensaje *release* la desconecta de su modelo y controlador y además propaga el mensaje a todos sus componentes, para que ellos también sean desligados de sus modelos y controladores correspondientes.

2.1.5 Vistas Customizadas

Cuando las clases widgets definidas no son apropiadas o suficientes para el tipo de presentación que queremos darle a una parte del modelo, es conveniente crear una *Vista Customizada,* esto es, una clase vista nueva que hereda de la clase *View.* Además, es necesario un controlador para la vista, cuya clase será subclase de *Controller* o de alguna de sus subclases. Por último, se extiende *Model* con una clase destinada a cumplir el rol de modelo. Es necesario redefinir métodos como *displayOn:* y *defaultControllerClass* para que el MVC funcione adecuadamente.

2.1.6 Mejora a la implementación de MVC en Smalltalk: el framework ValueModel de VisualWorks

2.1.6.1 El ApplicationModel: mediador entre el modelo y la interfaz

Esta clase fue introducida para mediar[1] entre el modelo, al cual llamaremos "Modelo del Dominio" y la interfaz de usuario. Esta mejora permite que estos componentes no estén cargados con código que no tiene que ver con su funcionalidad específica haciéndolos más independientes entre sí.

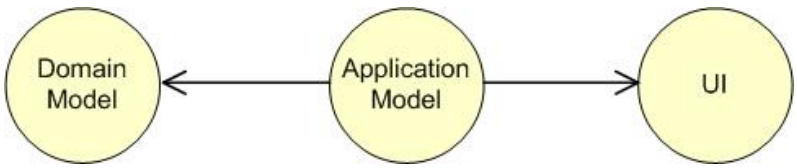


Figura 2.4 – Relación entre el modelo de dominio y la interfaz de usuario.

[1] Ver patrón *Mediator* en el Anexo

El *ApplicationModel* maneja la lógica de cómo las Vistas y sus subvistas colaboran con el modelo, para que no tengan que conocer en forma directa al modelo de dominio, como ocurre en anteriores versiones de Smalltalk.

Las aplicaciones se construyen definiendo una subclase de *ApplicationModel,* la cual provee un lugar donde almacenar todo el estado que no es parte de la UI y tampoco es parte del modelo del dominio.

2.1.6.2 El ApplicationModel: controlador de la aplicación

Una ventana de una aplicación muestra partes de un modelo a través de los componentes visuales que la componen. Cada widget usa una vista para desplegar su modelo, que es parte de un modelo más grande y su controlador, cuya clase es provista por el framework, se encarga de la interacción entre sus componentes vista y modelo. De poco serviría una interfaz donde su única funcionalidad fuese la generada dentro de cada trío MVC a nivel componente, esto es, entre cada widget, su modelo y su controlador. Por esta razón es necesario, en la mayoría de los casos agregar más dependencias y acciones sobre el modelo.

Estas acciones las realiza el A*pplicationModel,* quien suele agregarse como dependiente de un modelo de una widget. Una vez notificado de un cambio, genera él mismo cambios en el modelo de toda la aplicación, activando la actualización de otras vistas en la interfaz. Asimismo, un evento en un componente UI de la interfaz genera llamados a procedimientos que el *ApplicationModel* implementa. **Es por eso que el *ApplicationModel* es considerado a veces el "controlador de la aplicación"** ya que, en general, contiene la funcionalidad que no está incluida en los controladores de los componentes simples de la interfaz. Este es un concepto de diseño que será analizado más adelante independientemente de un framework en particular[1].

[1] El capitulo 3 trata el agregado de un cuarto objeto que resulta en un rediseño de MVC.

2.1.6.3 ValueModels: conexión entre una vista y un aspecto del modelo

El *ApplicationModel* coordina la comunicación entre los objetos UI y los objetos del dominio enlazando cada componente de la interfaz con un atributo u operación de un objeto del dominio.

Cuando el usuario ejecuta una acción sobre una widget pueden suceder dos cosas, o bien se modifica un atributo de un objeto del dominio o se llama a una operación de un objeto para su ejecución. Si por ejemplo ingresamos datos en un campo de texto, el *ApplicationModel* toma el texto y envía un mensaje a un objeto seteando alguna variable con el texto. A su vez, si un atributo de un objeto cambia, el *ApplicationModel* toma su nuevo valor y lo envía a la UI para que se mantenga consistente con su modelo.

El *ApplicationModel* se vale de un mecanismo llamado *adapter*. Un *adapter*[1] "adapta" la interfaz de un objeto de dominio y la de un objeto de la UI de manera que ambos objetos puedan comunicarse sin modificar sus protocolos. En Visualworks, este adaptador se denomina ***ValueModel*** porque define la relación entre el valor (Value) de un atributo y las widgets que dependen de este valor.

Un *ValueModel* contiene un valor llamado ***value***, un mensaje para recuperar este valor ***value*** y otro para asignarlo ***value:unValor***. Este objeto es capaz de notificar a sus dependientes cuando su valor cambia, para lo cual dispone de un mecanismo estándar para registrar interés en este valor. El *ValueModel* es así, el modelo de la widget, pero en realidad, el *ValueModel* contiene o se comunica con el Modelo real. Esta comunicación pasa desapercibida para la widget.

Existen diferentes tipos de ValueModel para cada tipo de atributo o valor. Para un tipo de atributo simple como un String, suele utilizarse una instancia de la clase *ValueHolder,* que simplemente *contiene* al string y notifica a sus dependientes cuando su valor ha cambiado. Para tipos más complejos, donde el valor está dentro de un objeto compuesto o separado del *ApplicationModel*, suele utilizarse un objeto de la clase *AspectAdaptor.*

[1] Ver patrón *Adapter* en el Anexo

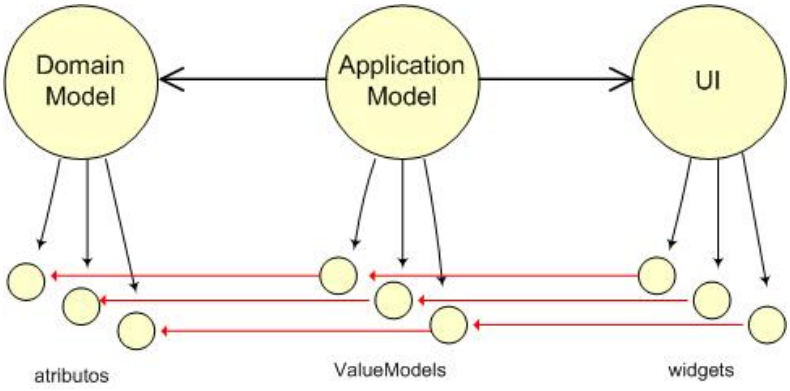


Figura 2.5- ValueModels que relacionan atributos y componentes UI.

La figura 2.5 muestra un esquema sobre cómo el *ApplicationModel* relaciona a grandes rasgos el Modelo del Dominio con la UI y cómo los componentes del Modelo del Dominio, denominados atributos, y los componentes de la UI, las widgets, se relacionan a través de ValueModels que son parte del *ApplicationModel*.

2.1.6.4 ValueModels: Dependencias entre objetos

El *ApplicationModel* colabora con los *ValueModels* para notificar a los objetos dependientes sobre cambios. En el ejemplo de la figura 2.6, el efecto de seleccionar un ítem en la lista de la izquierda, produce la actualización del gráfico de la derecha con el dibujo correspondiente al ítem seleccionado.

El último widget que muestra el dibujo seleccionado no depende de la widget que muestra la lista, sino del ValueModel de esta última, que es un objeto ítem seleccionado. Ambas widgets dependen de este ValueModel. Por lo tanto ambas reciben las notificaciones de cambios.

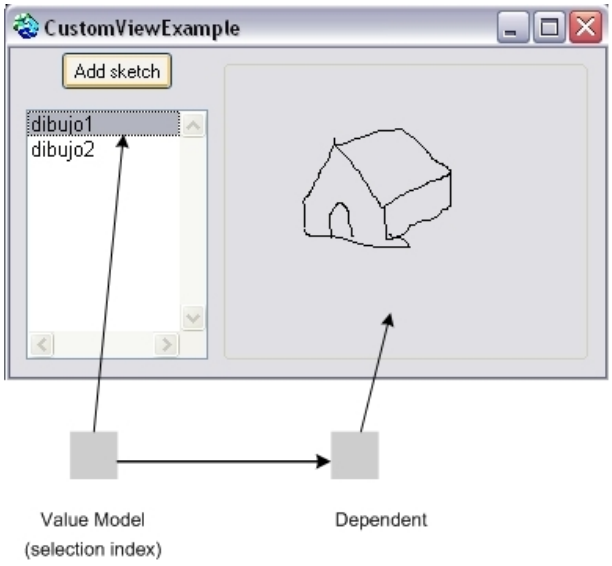


Figura 2.6

2.1.6.5 ValueModels: los mensajes value y value:

Considerando el ejemplo anterior y volviendo al contexto de MVC, observamos que el controlador de la lista, reacciona ante la selección de un dibujo y modifica al ValueModel que contiene al ítem seleccionado (selectionIndex) asignándole un nuevo ítem. En consecuencia, el modelo ha cambiado y todos sus dependientes son notificados para actualizarse. Las widget no dependen directamente del modelo, como ocurría anteriormente en Smalltalk sino que se registran como dependientes de un ValueModel. Con este nivel de indirección en el mecanismo de propagación de cambios, la interfaz no conoce directamente al modelo independizándose de él. Cuando una widget acepta información del usuario, la envía a su ValueModel, y cuando necesita recuperar la información a mostrar, envía un mensaje a su ValueModel para obtener el dato a presentar.

Los ValueModels tienen un **protocolo uniforme** para que las widgets se comuniquen con ellos permitiendo que, cualquiera sea la subclase de ValueModel que estemos usando, invoquemos a *value* para recuperar el dato a mostrar, y a *value:* para enviar al ValueModel el dato que debe almacenar. En respuesta a un cambio en el ValueModel, la widget reacciona

enviándole el mensaje *value*. Cualquier objeto que quiera ser modelo de una widget debe implementar ambos mensajes.

2.1.6.6 Ejemplo de ValueModel con más comportamiento: ProtocolAdaptor

Cuando el modelo del dominio de una widget no es de tipo simple como un string, o un número entero, un *ValueHolder* no es adecuado para contenerlo. Supongamos que el objeto del dominio es una cuenta bancaria, y que la widget muestra el saldo de la cuenta. **La widget espera un modelo con el protocolo *value* y *value:*.** Por esta razón, necesitamos una forma de comunicarla con la cuenta de manera que los mensajes que la widget envíe sean traducidos en mensajes interpretados por la cuenta y viceversa. La clase *ProtocolAdaptor* representa un tipo de *Adaptor* que sirve para este fin.

La clase *ProtocolAdaptor* es una clase abstracta subclase de *ValueModel*. Cuando un *ProtocolAdaptor* recibe un mensaje de la widget (value o value:) lo traduce a otro que el objeto del dominio, su *Subject*, pueda entender.

Este proceso se denomina "adaptación" y las distintas subclases de ProtocolAdaptor proveen otras formas de adaptación tales como aspects, indices y slots. En la siguiente figura podemos ver la relación entre el objeto del dominio, el *ProtocolAdaptor* y la widget.

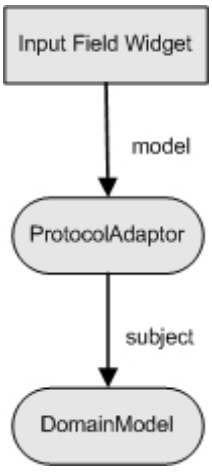


Figura 2.7 – Relación entre objeto del dominio y Widget.

La variable booleana *subjectSendsUpdate* de *ProtocolAdaptor* permite decidir cuando los dependientes reciben notificaciones de cambios en el *Subject.* Si la variable está en falso, el *ProtocolAdaptor* notifica a sus dependientes únicamente cuando recibe un *value:* desde la widget. La figura 2.8 muestra este tipo de relación. Si la variable *SubjectSendsUpdate* tiene el valor verdadero, el *ProtocolAdaptor* se agrega como dependiente del *Subject*. De esta forma cuando recibe un *update* desde *Subject* propaga la notificación a sus dependientes, las widgets, como se observa en la figura 8 en el gráfico de la derecha. De esta manera, los dependientes serán notificados siempre que el modelo cambie, independientemente de quien realice la modificación.

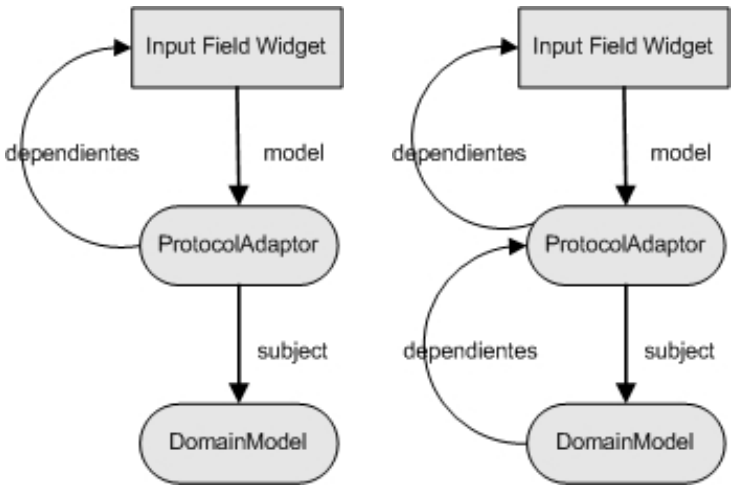


Figura 2.8- Implementación del Observer con Adaptors

Existen otras clases que cumplen función de adaptadores en el framework pero no es nuestra intención aquí profundizar sobre las distintas implementaciones que el framework ofrece.

2.2 Análisis de frameworks MVC

Con el objetivo de analizar las implementaciones que se exponen a lo largo de este trabajo, se define un método para que el análisis sea lo más uniforme posible entre ellas.

El análisis se realiza partiendo de un conjunto de características de MVC.
La utilización de este método permite:

- entender claramente sobre qué propiedades de MVC se analizan las implementaciones,
- evaluar frameworks y propuestas de implementaciones de MVC de acuerdo al grado en que soportan las propiedades del punto anterior, y
- ser usado como base para futuras implementaciones que quieran incluir estas propiedades de MVC.

Una vez listadas las características de MVC, y analizados los frameworks en el contexto de las mismas, se muestran los resultados en una tabla que contiene por un lado los frameworks y por otro las características. Una celda de la tabla representa la intersección de un framework y una característica y puede contener uno de los siguientes símbolos:

☑ El framework incluye la característica en forma explícita.

? El framework no soporta explícitamente la característica, siendo posible soportar completamente la característica.

☒ No es posible incluir la característica en el framework.

2.3 Características de MVC para el análisis de frameworks

A continuación se listan un conjunto de características que servirán de referencia para el análisis de frameworks a lo largo de este trabajo. Este grupo contiene "propiedades" que son inherentes al diseño MVC tradicional y otras que intentan mejorar este diseño desacoplando más sus componentes para lograr un alto grado de flexibilidad en el diseño y reuso de los componentes. Retomaremos el ejemplo del CD (capítulo 1) para una explicación más clara de los conceptos considerados.

2.3.1 Manejo de dependencias del modelo y propagación de cambios

Esta característica se refiere a la posibilidad de crear y mantener dependencias y a la notificación de cambios del modelo a sus dependientes.

2.3.2 Actualización inmediata de dependientes

Se refiere a si la actualización de las vistas se realiza en forma inmediata luego de la notificación de cambios y a si cumplen un rol activo en el ciclo de vida MVC que comienza con un cambio en el estado del modelo.

2.3.3 Independencia de la vista con respecto al modelo

Nos referimos a la posibilidad de que la vista sea independiente del modelo que muestra de modo que, si los nombres de los procedimientos que el modelo exporta sufrieran modificaciones o si se conectase otro modelo a la vista, estos cambios no se propaguen a ella.

Si la vista que muestra los datos de un CD de música obtiene el título del CD enviando al modelo el mensaje *CD.getTitulo()*, entonces si más tarde el método que obtiene el título del CD es otro, la vista deberá ser modificada acorde al cambio así como también todas las demás vistas

que permitan visualizar esta información. Esto sucede porque la vista accede a los datos del modelo en forma directa.

2.3.4 Separación entre Vista y Controlador: Independencia de la vista con respecto a su controlador

Esta característica permite sustituir el controlador asociado a una vista para cambiar la interacción que la misma ofrece al usuario. Un controlador "escucha" determinados eventos generados por una vista y define qué acciones se realizan en respuesta a los eventos. Para una determinada vista, el cambio de un controlador por otro podría acotar o ampliar este conjunto de eventos "escuchados", permitiendo conservar la vista independiente de su controlador asociado y al mismo tiempo, poder conectar un controlador adecuado al tipo de interacción que se requiere. La vista no tendrá que ser modificada para adaptarse a un comportamiento o procesamiento de entrada determinado. De esta manera se logra la separación y consecuente reusabilidad de los componentes.

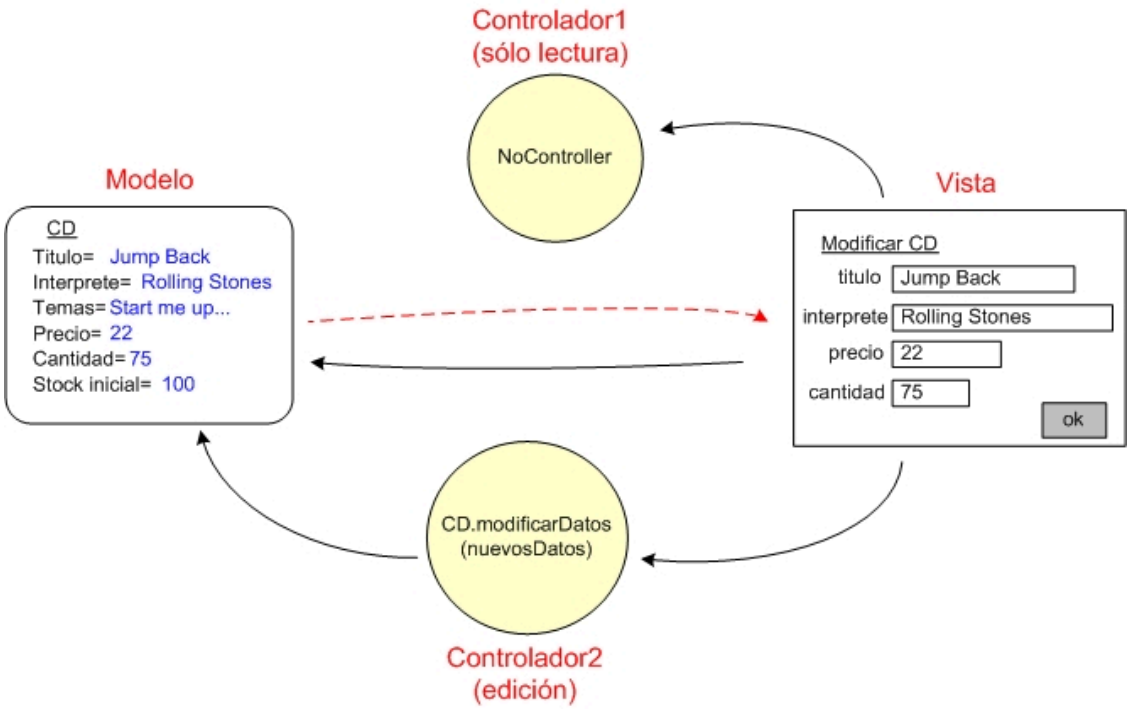


Figura 2.9 – Dos posibles controladores para la misma vista

Distintos controladores para una misma vista pueden ser vistos como una clasificación de las interacciones sobre la vista de distintos grupos de usuarios. Un ejemplo sencillo es el de un usuario que sólo ve la información de un CD pero no la modifica y otro que sí puede hacerlo (figura 2.9). La interacción con la vista varía de acuerdo al controlador asociado.

Notemos que esta propiedad tiene sentido si realmente necesitamos mostrar la misma información a ambos usuarios. Donde entonces la separación del procesamiento de la entrada y de la vista, permite no tener que programar una vista para cada uno de los usuarios cuando el contenido que se visualiza es el mismo. Esta propiedad tiene mayor aplicabilidad si consideramos una vista como un componente, como lo es el *TextBox* que muestra el título del CD. En este caso tiene sentido disponer de varios controladores conectables al componente. Si se considera vista a la ventana (ver figura 2.9), el controlador de esta vista ya no es fácilmente reemplazable por otro debido a que su funcionalidad es altamente dependiente de los controladores de cada componente en la ventana.

2.3.5 Independencia del Controlador con respecto al Modelo

El controlador se comunica con el modelo de manera directa llamando a los procedimientos que éste exporta. En consecuencia, se encuentra estrechamente vinculado al modelo. Existen dos posibles razones por las cuales es necesario rediseñar esta relación entre ambos componentes:

- La posibilidad de efectuar modificaciones en la funcionalidad básica de la aplicación.
- La capacidad de proveer controladores que puedan reusarse para lo cual es necesario que los mismos sean independientes de una interfaz de modelo específica.

Una aproximación para desacoplar modelo y controlador es la utilización del patrón *Command*[1]. *Command* representa una operación como un objeto desacoplando el objeto que invoca a las operaciones de aquel que las lleva a cabo.

Implementado en la relación controlador-modelo de MVC, *command* desacopla ambos objetos de manera que el controlador no llame directamente a las operaciones que el modelo

[1] Ver Anexo para una aplicación detallada de *Command*

exporta, sino que éste instancie un objeto que representa la acción solicitada y le indique "ejecutarse"[1]. Si se produjeran cambios en la interfaz del modelo, solo sería necesario modificar el código de las clases *command* que resulten afectadas por el cambio. De esta forma, el controlador sólo decide qué acción realizar instanciando el objeto de la clase adecuada y se independiza de la interacción con el modelo que se requiera para ejecutar la acción.

En lugar de que el controlador modifique los datos del CD en forma directa (ver figura 2.9), éste instanciará un objeto de la clase *AccionModificarCD* y le indicará ejecutarse enviándole los nuevos datos del CD como parámetro. Este objeto finalmente lleva a cabo la modificación del CD modelo, posiblemente guardando el estado anterior para posibilitar un posterior deshacer de los cambios (ver figura 2.10).

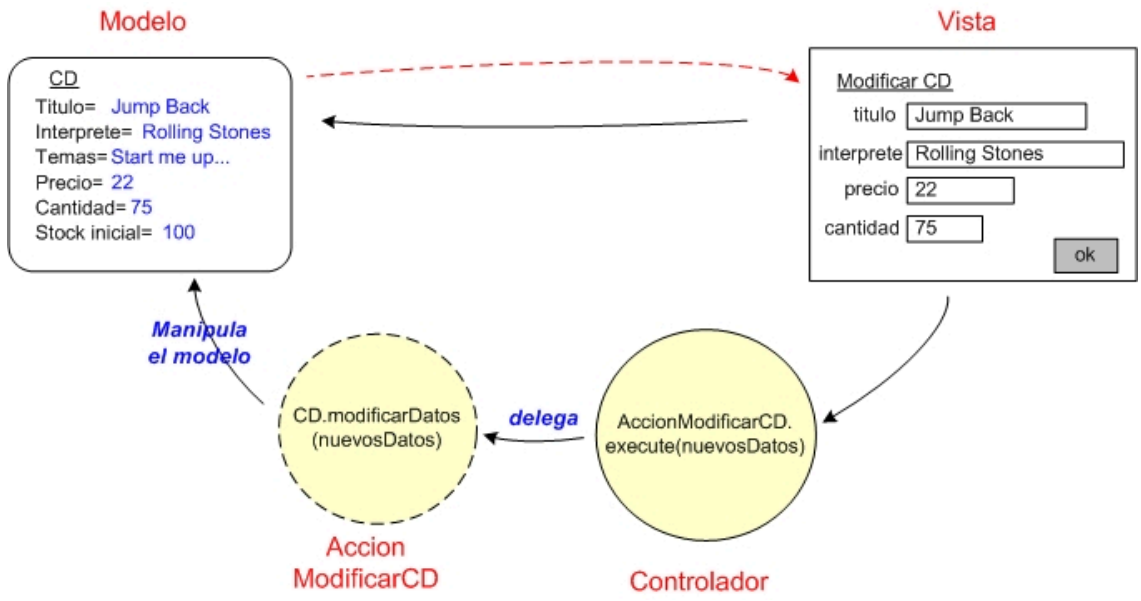


Figura 2.10- Independencia del controlador con respecto a su modelo

2.3.6 Vistas anidadas, Controladores anidados

Se refiere a que las vistas se construyan anidando componentes, por ejemplo, un panel con botones puede ser implementado como una vista compleja que contiene vistas botones anidadas. Así, como lo hemos visto en VisualWorks, las vistas compuestas actúan de la misma forma que

[1] Ver Anexo para una explicación detallada de *Command*.

las vistas simples pero con la diferencia de que contienen otras vistas en su interior. Decimos que actúan de la misma forma porque responden a los mismos mensajes, al mismo protocolo. El anidamiento de vistas en MVC es una implementación de *Composite*[1]. Asimismo los controladores de las vistas resultan anidados, y se organizan como un "árbol de controladores" con raíz en el controlador de la ventana de la interfaz, de modo de poder identificar qué componente ha generado un determinado evento y otorgar el control al controlador asociado a este componente.

2.3.7 Separación total del modelo y sus vistas

Sería ideal separar aún más al modelo de las vistas que dependen de él, es decir, que el modelo no referencie a sus dependientes ni siquiera en forma indirecta. La separación "total" se hace posible mediante un objeto intermedio que simule el rol de modelo para la vista, y que conozca al modelo real. A su vez, el desconocimiento total de las vistas es posible si los cambios en el modelo son generados solo a través de esas vistas, es decir si el modelo es "pasivo". Si el modelo es capaz de alterarse a sí mismo ("activo") o ser modificado por otro objeto que no es parte de nuestra aplicación, ya no es factible el tipo de separación aquí propuesta. La figura 2.11 muestra un esquema general de esta idea, donde el modelo CD no tiene ni siquiera una referencia débil a sus vistas y en consecuencia, puede liberarse de realizar la notificación de cambios.

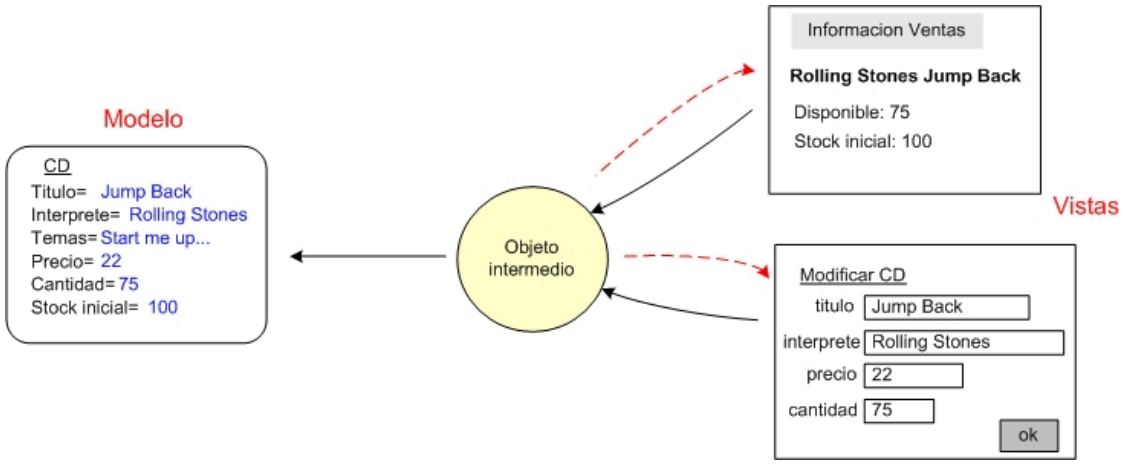


Figura 2.11 – Separación total del modelo y sus vistas

[1] Ver anexo para una explicación de *Composite*

2.3.8 Múltiples vistas para un mismo modelo y sincronización entre ellas

Nos referimos a la posibilidad de asociar varias vistas a un mismo modelo y que se mantengan sincronizadas cuando se produzca algún cambio en el modelo que observan. En nuestro ejemplo, si un usuario edita el titulo del CD y existe otro componente visual que depende de esta valor, entonces ambos deberán sincronizarse automáticamente ocurrido un cambio. Un framework que soporta esta característica provee una forma de conectar ambos componentes vista con un mismo modelo de manera que ante un cambio en el mismo, ambos componentes se actualicen y sincronicen en forma automática, es decir, sin tener que programarlo explícitamente. Notar que a medida que la aplicación crece, esta sincronización entre componentes no alcanza para completar el funcionamiento de toda la aplicación, y se hace necesario codificarla explícitamente para poder mantener la consistencia entre las distintas vistas de un mismo modelo.

Esta característica puede incluirse en distintos niveles de granularidad. Es posible que un modelo pequeño, como el caso expuesto en el párrafo anterior, tenga asociadas varias componentes y que ellas se sincronicen automáticamente. Sin embargo, si volvemos a considerar vista a una ventana como un todo que muestra a un modelo entonces la sincronización de distintas ventanas no es trivial, y es una tarea del programador de la aplicación.

2.3.9 MVC a nivel componente

El hecho de que un atributo parte de un modelo más grande, sea observado por un componente parte de una vista más grande, introduce el concepto que llamaremos MVC a "nivel componente". Tomemos el ejemplo de la vista que permite editar los datos de un CD en la figura 2.12. A "nivel componente", la ventana de edición esta formada por campos de texto, uno por cada dato del CD: título, intérprete, precio, etc. Cada uno de estos campos de texto es una vista de un dato dentro del CD. Cada campo de texto tiene su propio controlador, que sabe manejar la entrada de usuario para ese campo y tiene un modelo, que es una variable de instancia dentro de un objeto CD. Entonces, cuando un usuario modifique el precio del CD, el controlador del campo de texto actualiza el modelo, el modelo envía una notificación a sus dependientes y la vista se actualiza con el nuevo precio. Este último es un ejemplo de lo que llamamos "MVC a nivel componente".

Figura 2.12- El campo de texto marcado depende del atributo precio formando un MVC a nivel componente

Estos MVC's componen otro más grande compuesto por el objeto CD (modelo), la ventana de edición de los datos del CD (vista) y el controlador correspondiente. Existe un MVC aún mayor, o "MVC a nivel aplicación" que está constituido por el modelo de la aplicación y sus respectivas vistas.

Como propiedad nos interesa observar si un framework incluye los elementos necesarios como para el soporte de "MVC's a nivel componente", es decir, si existe la posibilidad de ligar componentes de la interfaz con aspectos del modelo.

2.4 Las propiedades de MVC a nivel aplicación y a nivel componente

Del conjunto de propiedades expuestas en la sección anterior destacamos que el concepto de "MVC a nivel componente" no solo es una característica que puede estar implementada o no en un framework, sino que incide en el resto de las características de MVC consideradas. En consecuencia, no es lo mismo que la vista sea independiente del modelo[1] a nivel componente que a nivel aplicación. Lo mismo sucede con la capacidad de conectar distintos controladores a una misma vista[2], lo cual tiene mas sentido a nivel componente.

[1] Propiedad 2.3.3
[2] Propiedad 2.3.4

2.5 Análisis del framework VisualWorks sobre las características de MVC

Para cada característica enumerada indicaremos si el framework la soporta, y de ser así, cómo lo hace.

2.5.1 Manejo de dependencias del modelo y propagación de cambios

La clase *Model* almacena la colección de objetos dependientes. Cuando un modelo específico (una subclase de *Model*) es asignado a una vista o controlador mediante el mensaje *setModel:unModelo*, son automáticamente agregados a la colección de dependientes de este modelo. Cuando ocurre un cambio en el modelo, cada dependiente en la colección recibe el mensaje *update*, pudiendo recibir al modelo o un aspecto del mismo como parámetro. Esta última característica permite al objeto dependiente obtener información sobre qué aspecto del modelo ha cambiado. Todos los dependientes contienen el mensaje *update* en su protocolo que se hereda de *Object*.

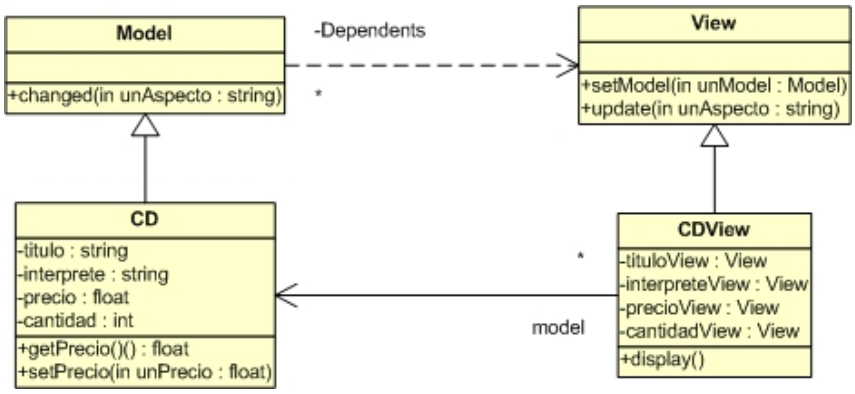


Figura 2.13- Diagrama de clases en Smalltalk para el ejemplo de CD's

Este diagrama de clases muestra a grandes rasgos una posible implementación en Smalltalk del ejemplo del modelo CD y la vista que muestra sus datos. Cuando el objeto de la clase *CDView* es creado se le asigna el objeto CD1 como modelo, llamemos CD1 a una instancia de la clase *CD*. Ante un cambio del precio a través del método *setPrecio(unPrecio),* CD1 se manda el mensaje *self changed:#precio* a sí mismo. Esta acción deriva en el envío del mensaje *update:#precio* a todos los dependientes del CD incluido el objeto *CDView*.

2.5.2 Actualización inmediata de dependientes

La actualización de las vistas se realiza en forma inmediata. Las vistas una vez creadas se mantienen a la expectativa de una notificación de cambio en el modelo y cuando esto sucede, se actualizan manteniéndose consistentes con el modelo. El mecanismo les permite recuperar a ellas mismas, los datos que necesitan para re desplegarse. Luego de actualizadas, vuelven al mismo estado en el que se encontraban, es decir, a la espera del siguiente cambio del modelo que activará un nuevo ciclo.

Siguiendo con el mismo ejemplo, si el precio del CD cambia, y el objeto *CDView* es notificado con un *update: precio*, entonces automáticamente enviará un mensaje *getPrecio()* al CD1 para actualizar el valor en la vista.

2.5.3 Independencia de la vista con respecto al modelo

Esta independencia no es una característica inherente de las primeras versiones de Smalltalk, donde era natural que una vista conociera directamente a su modelo referenciándola con una variable de instancia y comunicándose con ella mediante la interfaz que el modelo exportaba. En consecuencia, si el modelo era reemplazado por otro o los métodos cambiaban de nombre, esta vista debía ser al menos modificada para adaptarse al nuevo modelo, sino reemplazada directamente por otra, como es el caso de la figura 2.13, donde la vista obtiene el precio del CD directamente enviándole a este el mensaje *getPrecio()*.

Los *ValueModels* incluidos como intermediarios entre un componente vista y modelo, permiten lograr esta "independencia de la vista con respecto a su modelo". Ya hemos explicado como la vista siempre utiliza los mensajes *value*, para recuperar el valor a mostrar, y *value:unValor*, para asignar un valor a su modelo. El *ValueModel* adapta ambos protocolos, el de la vista y el modelo para que ambos puedan comunicarse de forma transparente, esto es, sin depender de sus protocolos específicos.

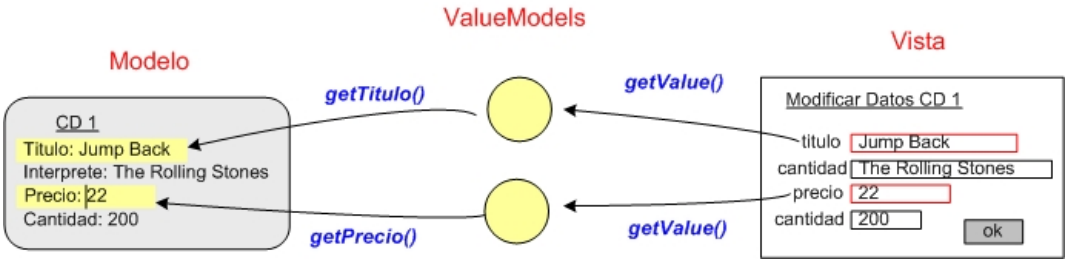


Figura 2.14 – Vista independiente del modelo

En el caso del ejemplo, dos ValueModels son objetos intermedios entre, las widgets que muestran el título y precio del CD y sus correspondientes modelos.

2.5.4 Separación entre Vista y Controlador: Independencia de la vista con respecto a su controlador

VisualWorks provee una clase controlador por cada clase vista disponible. Cuando un programador define sus propias vistas, debe también generar una clase controlador correspondiente. En ambos casos, podríamos definir varias clases controlador para una misma vista y que cada uno reaccione a determinados eventos que reflejen las operaciones permitidas sobre esa vista en un determinado momento. Además, diferentes controladores para una misma vista podrían actuar de distinta forma al recibir un mismo tipo de evento. Luego, conectaríamos el controlador apropiado de acuerdo al tipo de interacción deseada para la interfaz.

Supongamos que creamos una clase *CDTemasView* para una vista que despliega una lista de temas del CD. Esta vista está contenida en otra vista que muestra toda la información referente al CD. Definimos también una clase controlador para *CDTemasView* que llamamos *CDTemasController* y que hereda de *ControllerWithMenu* para que el controlador despliegue un menú de acciones posibles cuando un usuario hace click con el botón <yellow> sobre alguna canción de la lista.

El menú que el controlador de *CDTemasView* muestra, ofrece opciones para ver más información de dicho tema o reproducir el tema actual.

Podría ser necesario ampliar este conjunto de posibles acciones, por ejemplo, reemplazando el controlador actual por otro que permita además, borrar el tema o cambiar su nombre. Con esta finalidad, debemos definir otra clase controlador para esta misma vista, que también herede de *ControllerWithMenu* pero que amplíe el menú de acciones. Otro caso a considerar, es el de un controlador que directamente ignore la interacción del usuario sobre la lista de temas. VisualWorks provee una clase para los controladores de este tipo denominada *NoController*.

2.5.5 Independencia del Controlador con respecto al Modelo

En un diseño de MVC en VisualWorks es necesario implementar el patrón *Command* descripto, definiendo una jerarquía de clases *command* para representar cada acción posible sobre el modelo. Cuando el controlador atiende a un evento generado en el componente de la interfaz, instancia un objeto correspondiente para ejecutar la acción requerida en respuesta al evento. Luego de instanciarlo, le envía el mensaje *ejecutar(modelo)* donde el parámetro es el modelo. En la ejecución de este método, el objeto *command* interactúa con el modelo llamando a el o los procedimientos necesarios para realizar la acción solicitada.

2.5.6 Vistas anidadas, Controladores anidados

VisualWorks soporta vistas anidadas. Hemos visto cómo el framework provee clases tales como *CompositePart* (sección 2.1.2.3) que actúan como contenedoras de objetos *View*. Como resultado de esta capacidad, los controladores de las *View* son recorridos de acuerdo a la estructura en la que se organizan las widgets y ellos mismos se encargan de encontrar al controlador que ha de responder (flujo de control de los controladores, sección 2.1.3.2).

2.5.7 Separación total del modelo y sus vistas

A nivel componente en VisualWorks, existe la posibilidad de que un modelo de dominio de una widget no conozca en absoluto sobre la existencia de dependientes, si los cambios sólo se producen a través del *ValueModel* que lo referencia. La variable *subjectSendsUpdate* en un *ProtocolAdaptor* permite al *ValueModel* ser notificado o no de los cambios del modelo de dominio (su *subject)* para propagarlos a los dependientes que él se encarga de mantener (sección 2.1.6.6).

En el caso de una vista customizada, debemos definir un objeto intermedio para cumplir el rol de *ValueModel* para nuestra vista.

En la figura 2.14 los dos ValueModels se interponen entre las widgets título y precio, y sus respectivos modelos (las propiedades del CD). Si la variable *subjectSendsUpdate* del ValueModel título se encuentra en *false*, entonces el título del CD no tiene dependientes directos sino que el *ValueModel* se encarga de notificar a las widgets dependientes sobre los cambios en el modelo. Este aislamiento del modelo solo puede darse si los cambios al titulo del CD pueden solamente darse a través de cualquiera de las widgets dependientes del ValueModel título. De no ser así, la vista podría dejar de ser consistente con el modelo.

2.5.8 Múltiples vistas para un mismo modelo y sincronización entre ellas

Dos widgets pueden compartir el mismo modelo y por consiguiente mantenerse sincronizadas en forma automática. También es común agregar dependencias entre *ValueModels* de distintas widgets para que cuando uno de ellos cambie se propague el cambio a los demás *ValueModels*, resultando en la actualización de las vistas dependientes. La sincronizacion puede darse a nivel componentes View en una misma ventana o a nivel mayor, cuando hablamos de distintas ventanas que muestran un mismo modelo.

En el primer caso, los *ValueModels* son los que permiten conectar un mismo modelo con distintos tipos de componentes *View* aun cuando estos últimos no sean muy similares. Por ejemplo, si el modelo es el CD de música y la ventana contiene una lista de temas del CD y un

TextEdit para cambiar el nombre de uno de los temas, entonces al editar este nombre, el cambio debe verse también en la *View* de la lista de temas. Esta sincronización puede darse en forma automática asociando los *ValueModels* correspondientes.

2.5.9 MVC a nivel componente

El framework soporta este concepto. Un MVC a nivel componente funciona entre la vista de una widget, su controlador y su *ValueModel*. Por lo tanto, es posible que una widget, parte de una ventana, se actualice en forma independiente para mantenerse consistente con el aspecto del modelo que está mostrando ya que Smalltalk permite ligar componentes de la interfaz con aspectos del modelo.

2.6 La implementación de MVC en Java Swing

Swing es un framework para construir interfaces de usuario en Java. A diferencia de VisualWorks, que permite definir subclases de las clases que ofrece, Swing provee interfaces. Una aplicación que utiliza Swing contiene clases que implementan estas interfaces.

La arquitectura de cada componente Swing está basada en MVC pero de una manera diferente a VisualWorks. Esta arquitectura es una adaptación del MVC original y suele denominarse *separable model architecture* que quiere decir arquitectura de modelo separado. Si observamos la figura 2.15 vemos que, en este diseño, el modelo de un componente es tratado como un elemento separado siguiendo los lineamientos de MVC.

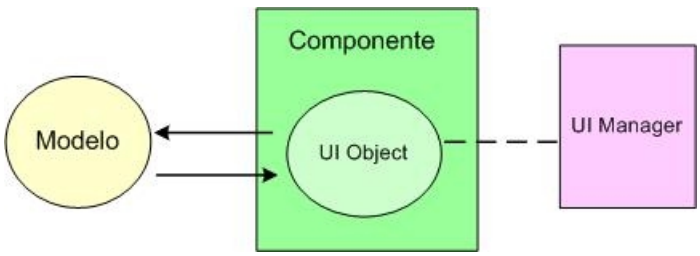


Figura 2.15 – Esquema general de un componente Swing

Como MVC requiere una comunicación fuerte entre vista y controlador, Swing unifica las responsabilidades de los roles de vista y controlador en un solo objeto UI[1]. Por lo que, tanto la apariencia como el comportamiento de un componente (el look&feel) se manejan juntos.

Además de la separación del modelo y la interfaz, el segundo objetivo de Swing es delegar las responsabilidades del View/Controller de un componente en un objeto *look&feel*[2] separado. Esto permite intercambiar el *look&feel* de un componente en cualquier momento sin tener que modificar la componente.

El objeto *Component* es el que cumple la función de vista y controlador en un componente Swing y cuando es necesario, delega esta responsabilidad en el objeto UI, también llamado *UIDelegate,* que se encarga de aspectos relativos al *look&feel* actual de la interfaz. Más adelante nos encontraremos con que existen otros objetos que hacen al rol de controlador.

La clase *JComponent* es la superclase de la mayoría de los componentes y contiene el código genérico para que un componente se dibuje o se muestre. Por ejemplo, la clase *JButton*, subclase de *JComponent* representa un componente botón pero el código que permite dibujar la etiqueta del botón se encuentra en la clase *UI Object* del *JButton* y no en la clase *JButton* misma.

2.6.1 El Modelo de un Componente Swing y el mecanismo de propagación de cambios

Swing provee las jerarquías de clases necesarias para el soporte del modelo de cada componente. Para cada componente Swing, existe una interfaz que representa al modelo del componente. **Esta implementación permite conectar cualquier objeto para actuar como modelo siempre y cuando implemente esta interfaz.** A modo de ejemplo, la clase componente *JSlider* cuenta con la interfaz *BoundedRangeModel* para que el objeto que quiera cumplir el rol de modelo de la barra deslizante implemente la interfaz.

[1] Abreviatura de User Interface (Interfaz de Usuario)
[2] Look se refiere a la presentación, Feel al manejo de eventos

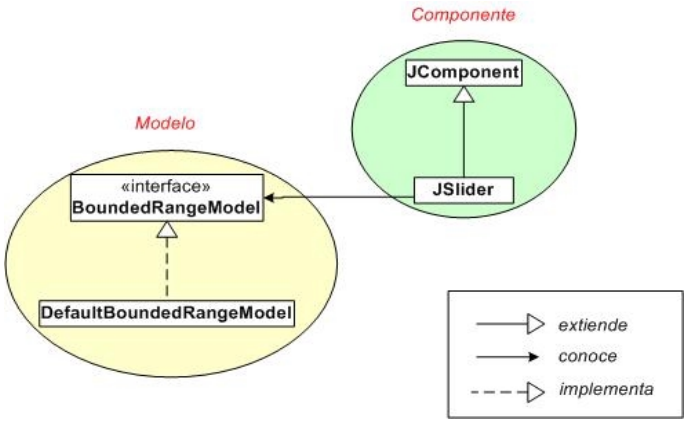


Figura 2.16- Arquitectura de la relación de un JSlider y su modelo.

Al crear un componente, el framework genera automáticamente un objeto modelo que implementa la interfaz correspondiente al componente. Estos objetos que se generan por defecto son instancias de clases de una jerarquía también provista por Swing. En la figura 2.16 se observa que la clase *DefaultBoundedRangeModel* implementa la interfaz que representa el modelo del componente y una instancia de esta clase es la generada al crear un nuevo componente *JSlider*. Este modelo puede reemplazarse por otro si así se lo desea. La clase *JComponent* contiene mensajes para poder asignar un modelo determinado a un componente.

A simple vista parece que un modelo pertenece sólo a un determinado componente. Sin embargo, cuando los datos que dos componentes esperan como modelo son similares como para soportar una única interfaz, una única instancia de una clase que implemente esta interfaz puede cumplir el rol de modelo para ambos componentes.

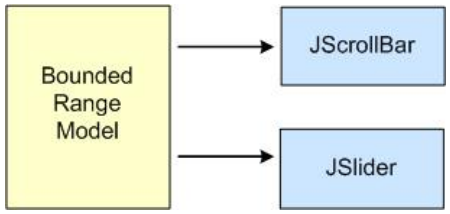


Figura 2.17 – Dos componentes comparten un mismo modelo

Un ejemplo son los componentes *JSlider* y *JScrollBar* que usan la interfaz *BoundedRangeModel* como modelo (figura 2.17). Al ser ambos componentes "vistas" del mismo modelo, se mantendrán sincronizados entre sí en forma automática.

2.6.1.1 La propagación de cambios del modelo: orientada a eventos

En la notificación de cambios del modelo a sus dependientes interviene un modelo, objetos *listeners* que son dependientes del modelo, y los objetos *evento*. Cuando el modelo cambia su estado, se genera un evento y los *listeners* o dependientes de este modelo son notificados recibiendo el evento como parámetro en la notificación.

Para que un objeto sea notificado debe, primeramente, implementar la interfaz *ChangeListener* y luego, registrarse como *listener* del modelo. El método *stateChanged* de la interfaz *ChangeListener* es el que será invocado por el framework al producirse un cambio en el modelo. Este mensaje recibe como parámetro el evento que se produjo, instancia de la clase *ChangeEvent*.

El modelo del componente contiene el protocolo para registrar y desregistrar dependientes, el cual está compuesto por los métodos *addChangeListener* y *removeChangeListener*. Ambos mensajes reciben un objeto *ChangeListener* como parámetro.

Cuando un dependiente recibe un llamado a *stateChanged*, puede recuperar desde el evento, al modelo generador del evento, y así obtener la información necesaria para realizar la tarea en respuesta al cambio en el modelo. En este tipo de notificación, el dependiente interroga directamente al modelo mismo para enterarse de qué aspecto ha cambiado.

Existe **otro tipo de notificación** que describe precisamente cuál fue el cambio. Los modelos que soportan este tipo de notificación proveen interfaces específicas para que los *listeners* las implementen y clases de objetos evento especiales para cada componente. Un ejemplo es el caso de un componente *JList,* que representa una lista de objetos dentro de una interfaz. El modelo del componente implementa la interfaz *ListSelectionModel*. Un dependiente de este modelo implementa la interfaz *ListSelectionListener* y el evento que se genera ante un cambio en la lista modelo es una instancia de *ListSelectionEvent.* Este evento es más "inteligente"

que un *ChangeEvent* ya que el dependiente que lo recibe como parámetro obtiene datos más precisos acerca de qué cambió en el modelo. Además, no necesita enviar mensajes al modelo para recuperar la información sino que puede recuperarlos desde el evento mismo. En el ejemplo nombrado, el evento *ListSelectionEvent* puede recibir mensajes para recuperar un elemento de la lista modelo que ha cambiado.

2.6.1.2 Actualización de las vistas

El modelo de un componente contiene la lista de sus dependientes, listeners, pero no tiene mayor conocimiento acerca de ellos. Un componente Swing se encarga de registrar automáticamente un *listener* a su modelo de manera de poder redibujarse apropiadamente si este éste cambia.

2.6.1.3 El modelo encapsulado

Además de permitir la separación bien definida entre modelo e interfaz, Swing agrega a sus componentes UI la capacidad de aislar totalmente el modelo. Esta característica es posible porque las clases componente contienen métodos propios a través de los cuales se accede al modelo. En lugar de enviar mensajes al objeto modelo de la componente para asignarle un valor o recuperarlo, los mensajes se envían al componente y éste los redirecciona a su modelo.

Esta característica también se hace presente en la propagación de cambios. Las clases componente proveen la habilidad de registrar *ChangedListeners* directamente al componente en lugar de a su modelo. De esta forma, el componente escucha los cambios de su modelo internamente y propaga el evento a los *listeners* registrados en el componente.

2.6.2 Vista/Controlador: Look & Feel conectables

Swing unifica las responsabilidades de los roles de vista y controlador de un componente en el objeto denominado *UI object* también llamado *UIDelegate*. Este objeto representa un look&feel específico para el componente y puede ser reemplazado por otro *UI Object* que

represente otro look&feel en tiempo de ejecución. Por esta razón, una misma componente puede tener diferentes formas de comportarse y de mostrarse. El framework incluye varios look&feel y también permite crear nuevos.

2.6.2.1 El UIDelegate: look&feel específico de un componente

Para relacionarse con su *UI Object* cada clase componente define una clase abstracta para representarlo. Por ejemplo, la clase *SliderUI* es la clase abstracta que representa al *UIDelegate* de un componente de la clase *JSlider*. *SliderUI* hereda de la clase *ComponentUI* que es la superclase de todos los *UIDelegate.*

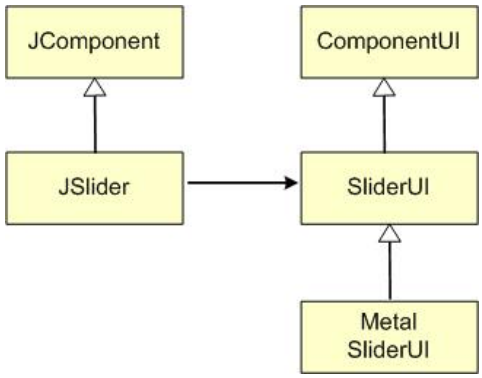


Figura 2.18 – Relación entre clases de un componente y su UIObject

Al instanciar un componente, se genera automáticamente un objeto *UIDelegate* para el mismo y éste es accesible a través del componente.

Cada look&feel disponible subclasea cada clase abstracta *UIDelegate*. Uno de los look&feel provistos es el denominado *Metal* que extiende a la clase abstracta *ButtonUI* mediante la subclase *MetalButtonUI*, a la clase *SliderUI* con la subclase *MetalSliderUI* y así siguiendo para cada clase *UIDelegate* en el framework. Entonces el objeto *UIDelegate* que se genera al crearse un componente es instancia de la subclase correspondiente al look&feel actual. Volviendo al ejemplo de *JSlider*, si el look&feel actual es *Metal*, entonces el objeto UI del componente es una instancia de *MetalSliderUI.*

2.6.2.2 UI Manager: control del Look&Feel actual

La clase abstracta *LookAndFeel* es la raíz de una jerarquía de clases abstractas donde cada una contiene la información general de un look&feel específico, tales como nombre y atributos de colores y fuentes.

Existe un look&feel actual en un determinado momento y todos los componentes de una interfaz Java Swing se muestran de acuerdo a él. *UI Manager* es un objeto a través del cual, los componentes Swing acceden a la información del look&feel actual. El objeto *UI Manager*, además del look&feel actual, contiene al look&feel por defecto y conoce qué otros están disponibles. Al mismo tiempo, permite reemplazar el look&feel actual por otro.

Es posible cambiar el look&feel luego de creados los componentes de la interfaz de usuario. Sin embargo, esto no es aconsejable, ya que los componentes no actualizan automáticamente sus objetos *UIDelegate* sino que habría que programar la actualización a mano.

2.6.2.3 Conexión de un look&feel

La clase *ComponentUI* es la superclase de todas las clases *UIDelegate* y contiene la funcionalidad necesaria para poder conectar diferentes look&feel a un determinado componente.

El método *installUI* definido en la clase *ComponentUI* instala un look&feel determinado en un componente. La instalación abarca un conjunto de tareas como seteo de fuentes, colores e instalación de un Layout Manager. Para llevar a cabo estas tareas, el objeto *UIDelegate* interactúa con el *UI Manager* para obtener la información necesaria acerca del look&feel actual, por ejemplo el color de la fuente. Además, en este método *installUI* se registran *listeners* al componente y al modelo. Estos objetos son almacenados por el *UI Delegate.*

Como ejemplo consideramos una componente lista, de la clase *JList* y un *UI Delegate* específico para la componente de la clase *BasicListUI*. Cuando este objeto es instalado en la componente, crea instancias de las clases siguientes que son internas a *BasicListUI*:

- BasicListUI.ListSelectionHandler, que implementa la interfaz *ListSelectionListener*. Un objeto de esta clase es instanciado y registrado como "listener" del "selectionModel" de la lista. Cuando la selección cambiar se redibujan las filas que cambiaron en la lista.
- BasicListUI.ListDataHandler, implementa *ListDataListener* y se agrega al modelo de la lista al instalar el *UIDelegate* y cada vez que el modelo cambie.

Ambos listeners son registrados al *modelo* de la componente con el objetivo de mantener la vista sincronizada con el modelo. En este ejemplo queda claro que un componente Swing es responsable de registrar listeners apropiados al modelo para actualizarse cuando el modelo cambie.

El método *uninstallUI* deshace todas las tareas que el *installUI* realiza y deja listo al componente para instalar un nuevo look&feel.

2.6.2.4 Otras funciones de UIDelegate: layout de la interfaz y dibujo del componente

Las interfaces en Swing son anidadas, es decir que, cada componente puede contener a otros componentes. El Layout Manager es el objeto que organiza los subcomponentes de un componente para que se muestren de una determinada manera. Para este fin, necesita interrogar a los subcomponentes sobre dimensiones mínimas y máximas permitidas, dimensión ideal. El *UIDelegate* del componente es el objeto que realmente conoce esta información y por lo tanto, una vez más, el componente delega en él para devolver la información al Layout Manager.

Cuando un componente se actualiza, debe redibujarse. Esta tarea también es delegada en parte en el *UIDelegate,* porque él determina cómo debe ser el aspecto visual (look) del componente.

2.6.2.5 La ventana en Swing y la jerarquía de componentes

Todos los componentes Swing heredan de la clase *Container*. Por lo tanto, todos pueden contener a otros componentes. La clase de una ventana de una aplicación que utiliza componentes

Swing, extiende alguna de las clases siguientes: *JFrame, JDialog, JApplet, JWindow.* En general, *JFrame* es la que se extiende para crear una ventana en una aplicación de escritorio.

Estas clases contienen una instancia de la clase *JRootPane*. Esta instancia, a su vez, tiene un objeto llamado *contentPane* que es instancia de *JPanel* y que almacena todas las componentes que forman la estructura de la ventana. Un *JPanel* es un contenedor de componentes que se vale de un objeto *LayoutManager* para organizar a los componentes que contiene. Dentro del *JPanel* principal, pueden agregarse otros componentes también compuestos que pertenezcan a clases como *JTabbedPane*, *JScrollPane*, y también otros *JPanel.* Esto da forma a la estructura anidada de componentes de una ventana.

Los componentes Swing se despliegan en la ventana invocando el método *paint(Graphics g)* que está en la clase *JComponent*. Este método está implementado para primero pintar el componente receptor (llamando a *paintComponent())* y luego propagar el pintado a sus subcomponentes. Por eso, si hemos creado un componente vista y queremos redefinir el método de despliegue, debemos sobrescribir el método *paintComponent*() y no *paint*(). El método redefinido será invocado por el framework Swing cada vez que el modelo cambie o lo invocaremos indirectamente con un llamado a *repaint(). Repaint()* permite pedirle al framework que redibuje el componente receptor del mensaje.

2.6.2.6 Manejo de Eventos de Entrada: otra implementación de Observer

En el manejo de eventos de entrada (feel) Swing vuelve a implementar el patrón Observer. **Un objeto que quiera ser notificado sobre una acción ocurrida en un componente de la interfaz debe registrarse como dependiente del mismo**. Nuevamente, los objetos que intervienen son, los que originan los eventos de acción y los que escuchan estos eventos (*listeners* del evento). Estos últimos son los que realizan alguna manipulación de datos sobre el modelo de la interfaz en respuesta al evento. Cuando un evento se genere en el componente, los correspondientes métodos de los *listeners* serán invocados por el framework. **Los *listeners* de eventos sobre la componente cumplen el rol de controlador** ya que, de acuerdo con la definición de MVC, son ellos los que reciben los pedidos de usuario en forma de eventos, y los transforman en llamadas a procedimientos en el modelo.

Algunos de estos listeners se agregan al momento de instalar un objeto *UI Delegate* en la componente. Cuando se hace necesario agregar más comportamiento, pueden registrarse otros objetos listeners a un componente para realizar el comportamiento requerido.

Diferentes componentes generan distintas clases de eventos. Los componentes que generan un determinado tipo de evento contienen el protocolo de mensajes necesario para que *listeners* de este tipo de evento puedan registrarse a él. Por ejemplo, un componente *JButton* contiene los métodos *addActionListener* y *removeActionListener*. Si por ejemplo, el usuario de la interfaz hace un clic sobre el botón, éste genera un evento *ActionEvent*. Los objetos que estén interesados en atender este evento, deben implementar la interfaz *ActionListener* y registrarse al *JButton* enviándole el mensaje *addActionListener*. El framework tomará a todos los dependientes del componente e invocará un método de la interfaz que implementan, correspondiente a la acción ocurrida, pasándoles el objeto *ActionEvent* como parámetro.

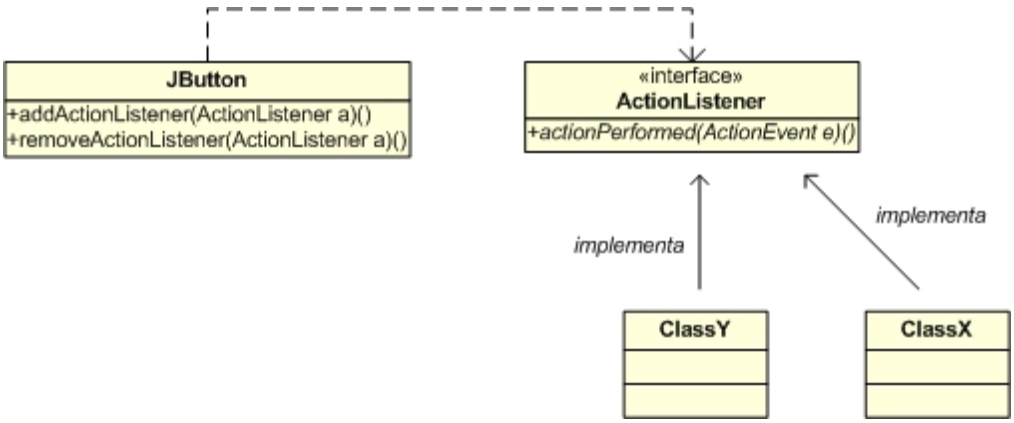


Figura 2.19 – Relación entre un componente JButton y los listeners

El framework provee clases para los tipos de evento que puedan producirse sobre un componente, como son *MouseEvent, ActionEvent;* e interfaces que los *listeners* implementan de acuerdo al tipo de evento que quieren atender. Cada interfaz *listener* contiene métodos que son invocados al ocurrir un evento específico.

El *UIDelegate* también determina qué tipo de acciones de usuario sobre el componente generarán un evento dentro del ambiente de Java. Por esta razón, solo podrán registrarse al componente, *listeners* para atender a esos eventos definidos por el *UIDelegate*.

2.6.2.7 La función de controlador del UIDelegate

El objeto *UI Delegate* es el controlador específico que viene con cada componente que se instancia y controla los eventos relacionados con actividad del mouse sobre la componente y entrada de texto. Cada clase *UI Delegate* crea y almacena un conjunto de objetos listeners sobre el modelo de la componente y sobre la componente misma. Estos últimos responden a determinados eventos sobre la componente donde se lo instaló. Consideremos nuevamente el ejemplo de la componente que representa una lista, y al objeto *UI Delegate* específico de la clase *BasicListUI*, subclase de *ListUI*. *BasicListUI* tiene además otras clases internas como:

- BasicListUI.MouseInputHandler, implementa la interfaz *MouseInputListener*, y en consecuencia responde a eventos generados por la acción del mouse.
- BasicListUI.FocusHandler, que implementa a *FocusListener*, y controla el foco de la componente desde teclado.

Al instalar un objeto *BasicListUI* en la componente lista, se instancian objetos pertenecientes a las clases nombradas en la lista de arriba donde cada uno controla determinados eventos sobre la componente.

En conclusión, Swing implementa MVC separando la definición del modelo de un componente y proveyendo la habilidad de delegar parte de las responsabilidades vista/controlador de un componente en objetos look&feel. Esta última característica permite tener objetos vista/controlador conectables.

2.7 Análisis del framework Swing sobre las características de MVC

2.7.1 Manejo de dependencias del modelo y propagación de cambios

El modelo en Swing contiene, al igual que en Smalltalk, una lista de objetos que dependen de él. La diferencia entre esta implementación del patrón Observer y la de VisualWorks, es que para que un objeto pueda registrarse como dependiente de un modelo, debe implementar una interfaz. Al ocurrir un cambio en el modelo se genera un objeto evento. Luego se toma la colección de dependientes del modelo y todos reciben un mensaje *stateChanged* con ese evento como parámetro. En el ejemplo del CD de música, supongamos que se utiliza un objeto de la clase *JList* para mostrar los temas del CD. Cuando instanciemos un objeto cuya clase implemente la interfaz *ListModel* y lo asignemos como modelo a la componente con el mensaje *setModel(unListModel),* automáticamente la componente se agrega como dependiente del modelo para recibir notificaciones de cambios en el mismo.

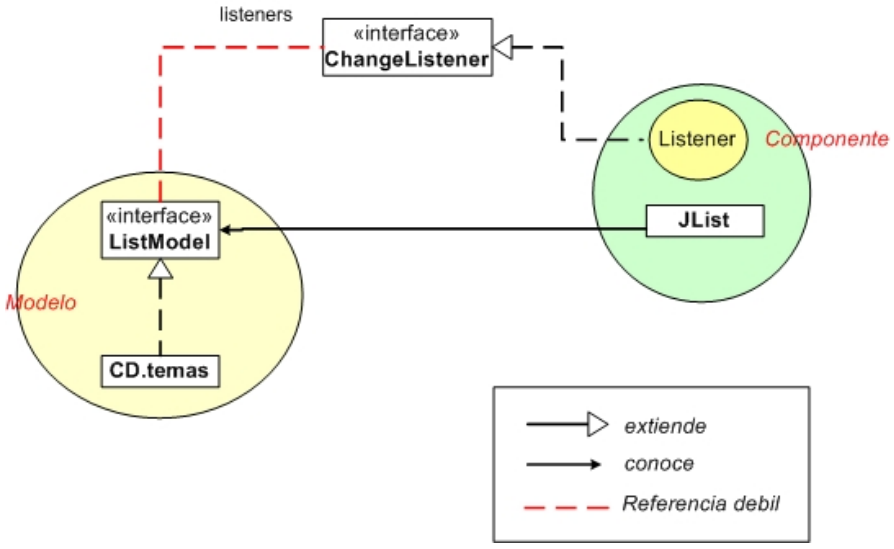


Figura 2.20 – Dependencia modelo-componente[1]

[1] El esquema solo representa algunas de las clases por simplicidad.

2.7.2 Actualización inmediata de dependientes

La actualización de las vistas se realiza automáticamente. Cuando ocurre un cambio en el modelo, los componentes vista son avisados invocando a su método *stateChanged* con un evento pasado como parámetro. Este parámetro les permite recuperar al modelo o conocer el aspecto que ha cambiado para que el componente pueda re desplegarse. Después de una actualización, las vistas se mantienen atentas al siguiente cambio.

Siguiendo con el ejemplo de los temas del CD, cuando estos datos cambian, cada listener recibe un llamado a *stateChanged* con un evento. El listener toma la lista de temas desde el evento y se actualiza de acuerdo a ella[1]. Cabe destacar que en el caso de listas, existen otras interfaces que se pueden implementar para recibir notificaciones de cambios en la misma. Por ejemplo, *ListDataListener* es una interfaz que no define un simple *stateChanged* sino métodos más especializados como *contentChanged, intervalAdded,* que sirven para que el receptor tenga mas datos acerca de qué aspecto del modelo cambió.

2.7.3 Independencia de la vista con respecto al modelo

Swing promueve esta capacidad[2] de las vistas por la manera en que una vista se asocia con el modelo. Las vistas interactúan con cualquier objeto que implemente la interfaz definida para ser modelo. Cualquiera sea el modelo que implemente esta interfaz, la vista se comunica con él únicamente mediante los métodos definidos en la interfaz. Si el objeto modelo es reemplazado por otro, la vista no sufre cambio alguno.

En nuestro ejemplo, supongamos que estamos mostrando la lista de temas del CD con un objeto de la clase *JList.* El modelo definido para un componente *JList* es un *ListModel*. El modelo, en este caso la lista de temas del CD, implementa la interfaz *ListModel* o subclasea a *AbstractListModel* (clase provista por el framework que implementa esta interfaz). Luego ligamos el modelo al *JList* y el *JList* siempre recupera los elementos de la lista con *getElements()*

[1] En el caso que el listener implemente la interfaz *ChangeListener.*
[2] En la sección 2.9.3 del capítulo 2 se describe la capacidad a la que hacemos referencia.

que es un método definido en *ListModel* e implementado por el modelo. Por lo tanto, si más tarde conectamos otro modelo, el código de la vista no cambia en absoluto.

2.7.4 Separación entre Vista y Controlador: Independencia de la vista con respecto a su controlador

Para mostrar a qué nos referimos con esta separación en Swing, consideramos el mismo ejemplo de la vista que muestra la lista de temas del CD (ver figura 2.21). Supongamos que al hacer click sobre un ítem de la lista de temas, se despliegue un menú con opciones para agregar un tema y/o borrar la canción que esté seleccionada en ese momento.

Figura 2.21 – Una vista de los temas del CD.

En Swing, es común asociar controladores (*listeners*) a los componentes UI que forman la interfaz en la inicialización de la interfaz, donde solamente deberían crearse las componentes que forman parte de la misma y especificarse el código necesario para su despliegue.

Apuntamos a la propiedad de separar la vista y el controlador de manera que conectar distintos controladores no implique modificar la vista. Por lo tanto, supongamos que ciertas opciones en el menú deben ser deshabilitadas o realizar otro tipo de acción en respuesta al clic en uno de los ítems.

El menú se compone de ítems a los cuales se asocia un *listener.* Al ítem de borrado se le asocia un *listener* que implementa el correspondiente mensaje *actionPerformed* para borrar la canción seleccionada en la lista.

Para deshabilitar el ítem de borrado necesitamos no asociar un *listener* a él ya que no es necesario realizar acción alguna o bien, asociar otro objeto *listener* que lleve a cabo una acción distinta. En cualquiera de los dos casos es conveniente que el mismo controlador sea el que se asocie a la componente Swing, fuera de la inicialización de la interfaz. Por ejemplo, podría generarse un objeto controlador para el menú, el cual asocia los *listeners* a los ítems del menú (ya sea él mismo el *listener* u otros objetos). Encapsular la definición de la interacción con el menú no tendría incidiría sobre el componente vista.

Recordar que esta separación de roles tiene sentido en los casos en que se desea mantener la misma interfaz y cambiar la interacción: el menú aparece con todos sus ítems en ambos casos. Y además, la propiedad pierde importancia si hablamos de vista a nivel ventana, como es el caso de vista que muestra la información toda del CD de música, ya que no es sencillo reemplazar un controlador que maneja la interacción con la ventana completa por otro. Igualmente, es muy poco probable que un controlador como este último pueda reusarse con otra vista.

2.7.5 Independencia del Controlador con respecto al Modelo

En la sección 2.3.5 explicamos cómo es posible desacoplar más los componentes modelo y controlador implementando el patrón de diseño *Command*[1]. En Swing, es necesario implementar este patrón mediante la definición de una jerarquía de clases de objetos *command* que represente cada acción posible sobre el modelo. Los listeners deberían entonces delegar en un objeto *command* para manipular el modelo en respuesta a un evento producido en la interfaz. El *listener* que recibe a un evento como parámetro, recupera al *source* del evento, su modelo, y luego instancia un objeto *command.* Por último invoca el método de ejecución del *command* pasándole al *source* como parámetro.

[1] Ver Anexo.

2.7.6 Vistas anidadas, Controladores anidados

Swing soporta el anidamiento de vistas. El framework provee clases contenedoras tales como *JFrame* que contienen componentes Swing. Además, la clase *JPanel* es la versión contenedora de componentes Swing, por lo que puede contener otro *JPanel* en su interior. Si bien los controladores, listeners, están escondidos dentro de cada componente, existe un mecanismo para identificar sobre qué componente se disparó el evento. Este componente luego crea un evento (instancia de una de las clases Evento del framework) para que sus listeners sean notificados y actúen en respuesta.

2.7.7 Separación total del modelo y sus vistas

El hecho de que Swing especifique una interfaz para cada componente de manera que un objeto deba implementarla para cumplir el rol de modelo beneficia a la vista notablemente pero no a la posible separación total del modelo.

Cuando mostramos la lista de temas del CD con un *JList* mencionamos que la lista de temas deberá implementar la interfaz *ListModel* para poder ser el modelo del componente. De manera que las partes del CD conocen, aunque no específicamente, el tipo de interfaz que los muestra. Esto se debe a que, dependiendo de la forma en que la lista de temas (modelo) se muestra deberá implementar distintas interfaces. Ejemplo: si la lista se muestra en un *JTable* debe implementar *TableModel*, si se despliega en un *JList* implementa un *ListModel.*

Notemos que en VisualWorks, los ValueModels permiten aislar el modelo de la forma en que está siendo mostrado a la vez que provee un protocolo uniforme para las vistas del mismo. Además, un ValueModel puede encargarse de la notificación de las vistas liberando al modelo de llevar a cabo esta tarea. Las interfaces model definidas por Swing no soportan esta propiedad.

2.7.8 Múltiples vistas para un mismo modelo y sincronización entre ellas

En la sección 2.6.1 que describe a los modelos de los componentes Swing, especificamos que cuando los datos que dos componentes esperan como modelo son similares y pueden llegar soportar una única interfaz, una única instancia de una clase que implemente esta interfaz puede cumplir el rol de modelo para ambos componentes. Asociar el mismo objeto modelo a dos componentes, produce la automática sincronización de ambos componentes cuando el modelo varíe. En Swing, esta sincronización automática no es tan flexible como en Smalltalk, ya que en Swing, los componentes deben ser muy similares como parar poder conectarlos a un mismo modelo.

Al igual que en Smalltalk, cuando ya no hablamos de modelo a "nivel componente"[1], y consideramos el modelo de una aplicación o de una vista más grande (contenedora de varios componentes), es necesario programar esta sincronización para sincronizar las vistas.

2.7.9 MVC a nivel componente

Existe un MVC a nivel componente en todo componente Swing, donde cada uno tiene modelo y look&feel propios y forman un MVC capaz de funcionar en forma independiente del resto de la interfaz.

2.8 Resumen del análisis de propiedades MVC

La característica (1) es soportada ya que ambos frameworks disponen de un mecanismo de dependencias. La actualización en respuesta a la notificación de cambios es inmediata (2). Swing utiliza un modelo de evento-listener para esta notificación y actualización.

Los ValueModels en VisualWorks y la interfaz para los modelos que Swing especifica permiten independizar la vista del modelo (3).

[1] *MVC a nivel componente*, ver sección 2.3.9

Vista y controlador están bien separados en Visualworks por lo que (4) es explícitamente soportada. En Swing, el *UIDelegate* unifica ambos componentes, pero igualmente al asociar *listeners,* que son los controladores definidos por el programador, se debe tomar la precaución de separar las responsabilidades de estos y de las vistas a las que agregan comportamiento.

	VisualWorks	Java Swing
1. Manejo de dependencias del modelo y propagación de cambios	✔	✔
2. Actualización inmediata de dependientes	✔	✔
3. Independencia de la vista con respecto al modelo	✔	✔
4. Separación entre Vista y Controlador: Independencia de la vista con respecto a su controlador	✔	?
5. Independencia del Controlador con respecto al Modelo (command)	?	?
6. Vistas anidadas, Controladores anidados	✔	✔
7. Separación total del modelo y sus vistas	✔	✘
8. Múltiples vistas para un mismo modelo y sincronización entre ellas	✔	✔
9. MVC a nivel componente	✔	✔

Tabla 1- Resumen propiedades MVC

Para incluir (5) es necesario definir clases adicionales. La propiedad (6) es inherente a las vistas en ambos frameworks, ya que éstas son orientadas a componentes. Cada componente tiene su modelo y su controlador (9). En cuanto a (7), si el modelo no es capaz de modificarse a sí mismo o por la acción de algún factor externo, VisualWorks permite que el ValueModel realice la notificación a los dependientes, aislando al modelo completamente de las vistas. Si bien Swing provee a un componente con el protocolo necesario para que una vista interactúe solo con él y no

con el modelo, el que mantiene la lista de dependientes es el modelo ya que este último debe implementar una interfaz determinada para cumplir su rol en MVC. Por último (8) es posible a nivel componente, directamente implementada por ambos frameworks, si bien en forma más flexible en VisualWorks. Para conseguir (8) a un nivel mayor tendremos que programar las acciones para la sincronización.

Capítulo 3
Evolución del diseño MVC

MVC define originalmente tres componentes:

- el modelo, que maneja la lógica negocios,
- la vista, que se encarga de la lógica de presentación
- y el controlador, que acepta e interpreta el acceso a la aplicación desde teclado y mouse.

MVC fue pensado para resolver el desarrollo de sistemas en los años 80. El grado de complejidad de los sistemas de ese tiempo se ajustaba adecuadamente a la solución que MVC presentaba. Pero las aplicaciones GUI han crecido en tamaño y complejidad.

Asimismo, no se ha encontrado una solución alternativa a MVC para el desarrollo de aplicaciones donde es necesario separar el contenido de la presentación. En consecuencia, los desarrolladores de software han intentado aplicar MVC en estos nuevos sistemas cuyas características son muy diferentes a las de sus predecesores, trasluciéndose así la incompatibilidad entre el MVC original y la aplicación siendo diseñada. Por esta razón, MVC ha sido utilizado con modificaciones a su versión original para adaptarlo al problema o tipos de problemas que se estaban resolviendo.

El objetivo principal al momento de su aparición fue básicamente, promover la separación de las aplicaciones en dos partes: por un lado el código que tenía que ver con el modelo y por otro, el relacionado con su presentación. Esta división no define un lugar u objeto encargado para todas las responsabilidades que surgen en una aplicación real. La separación de tres capas definidas por MVC es una división teórica que no se implementa directamente.

A medida que MVC se empleaba en el diseño de aplicaciones, comenzaron a traslucirse ciertas cuestiones que los desarrolladores de software tuvieron que resolver. Más aún, los frameworks para la construcción de interfaces orientadas a MVC fueron adaptándose gradualmente para resolver estas cuestiones.

3.1 Ejemplo de aplicación de MVC tradicional

El agregado de un poco de complejidad y tamaño al ejemplo de aplicación que hemos estado utilizando, nos servirá para explicar en forma clara cuáles son las cuestiones o aspectos en el diseño de MVC a los que hacemos referencia.

Supongamos que el usuario dispone de una vista donde visualiza varios CD's y decide comprar algunos de ellos. El modelo no es un único CD. Por ejemplo, la aplicación cuenta con un objeto que representa un "carrito de compras" con el fin de almacenar allí los productos que el comprador selecciona.

Consideremos un caso específico: el usuario ya ha guardado algunos ítems en su carrito de compras (*shopping cart)* entonces, en la ventana, además de una sección donde se muestran los ítems guardados hasta el momento, se despliega información acerca de otros ítems ofrecidos al comprador (ver figura 3.1). En esta ventana existe la posibilidad de agregar cualquiera de ellos al carrito de compras.

Figura 3.1- Una vista de varios CD´s y el carrito de compras

Si se opta por agregar un CD a la compra, la ventana se despliega nuevamente actualizándose el carrito con el nuevo CD (ver figura 3.2). En el idioma de MVC, cuando el usuario clickea en el botón de "agregar al carrito" (*Add to Cart*), el controlador toma el ítem seleccionado y lo agrega en el objeto carrito de ese comprador, modificando así el modelo, lo que resulta en el despliegue de la misma vista pero esta vez actualizada, reflejando el cambio.

Figura 3.2

El ejemplo anterior es el caso más sencillo de funcionamiento de MVC donde el usuario genera un evento, modifica el modelo y la vista correspondiente se actualiza en respuesta a ello.

3.2 Un ejemplo de mayor complejidad: lógica de la aplicación

Ahora planteemos una situación diferente en la cual el usuario es invitado a registrarse para poder comprar. Para realizar esta acción se requiere completar cierta información en una vista que muestra un formulario de registración. Una vez completados los datos, el usuario intenta registrarse clickeando en "continuar" (*continue),* (ver figura 3.3).

Figura 3.3

A partir de este punto pueden obtenerse dos resultados. Si ocurrió algún error, como en el caso de haber completado incorrectamente la dirección de correo, como muestra la figura 3.3, el formulario vuelve a desplegarse con algún mensaje de error indicando la falla (figura 3.4). La aplicación brinda entonces la posibilidad de enmendar el error.

Figura 3.4

Si los datos han sido ingresados en forma correcta, la vista actual es reemplazada por otra (ver figura 3.5) indicando que la registración se realizó sin problemas.

Figura 3.5

Cuando el usuario intenta registrarse haciendo clic en un botón en la interfaz, el controlador correspondiente toma el evento, lo interpreta e invoca a un procedimiento de alta de usuario en el modelo, enviándole la información ingresada como parámetro.

Del ejemplo anterior notamos cómo, de acuerdo a si la información es correcta o incorrecta, existen **dos posibles resultados**. Ahora bien, sea el controlador o no el que evalúe si los datos son completos y correctos, o sea el modelo el que realice esta verificación, se presenta el problema de decidir qué vista mostrar dependiendo del resultado de la acción de entrada.

Otro caso que sirve para ejemplificar la situación es el del "checkout": cuando el usuario decide efectivizar la compra de los CD's que eligió. Es claro que para administrar las compras, el modelo de nuestro sistema debe guardar información acerca del usuario, por ejemplo, sobre su cuenta bancaria. Al clickear en el botón *proceed to checkout* (ver figura 3.2) un controlador resulta activa y calcula el monto de la compra. Luego pregunta al modelo acerca del saldo disponible del comprador, a lo cual el modelo responde si el saldo es suficiente para realizar la compra o no lo es. Dependiendo de este resultado, o bien se realiza la compra, modificando el

saldo en el modelo y desplegando una vista que notifique al comprador de que la compra se a efectuado con éxito, o se muestra un mensaje explicando que no pudo realizarse la operación.

En este ejemplo también se presenta el problema de decidir sobre qué vista siguiente desplegar al usuario.

3.2.1 La Lógica de negocios

El modelo inicialmente contiene la **lógica de negocios** de la aplicación, que es la representación de los datos y las operaciones que se pueden realizar sobre los mismos. En nuestro ejemplo, el carrito de compras, la información de saldo del comprador y los CD´s por comprar son parte del modelo. Son ejemplos de operación sobre el modelo: la acción de agregar un CD al carrito, efectivizar la compra, consultar el saldo del comprador actual. Esta lógica debe encontrarse siempre y únicamente en el modelo, ya que de esta forma separamos los datos y las operaciones sobre los datos en un solo lugar, permitiendo modificar al modelo sin afectar otras partes de nuestra aplicación.

3.2.2. El modelo desde una visión simplista: lógica de aplicación y lógica de negocios

El problema es dónde ubicar el comportamiento relacionado con evaluar el estado resultante luego de la ejecución de una operación sobre el modelo y qué vista devolver al usuario de acuerdo a este resultado.

Es claro que este comportamiento, en teoría, no debería incluirse en la vista, dado que la misma depende en gran medida de la tecnología de la interfaz, como ser un browser web o un browser de un dispositivo móvil. *Una solución es incluir este tipo de decisiones dentro del modelo.*

Las operaciones como el alta del comprador, compra de un producto (ejemplos de pura funcionalidad básica de la aplicación) y las “decisiones” sobre qué vista desplegar quedan mezcladas en el modelo. Así, el modelo contiene información que es específica de una interfaz en

particular y por lo tanto, no alcanza el grado de independencia deseable con respecto a las vistas. Por lo tanto, no es trivial que este comportamiento deba integrarse con el modelo.

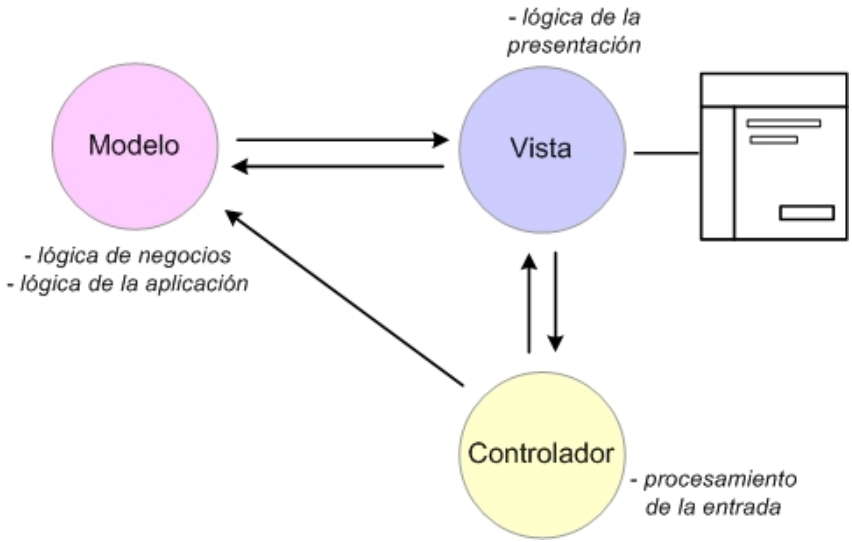


Figura 3.6 – MVC clásico

Por otro lado, el controlador de la vista actual, tal cual definido en MVC tampoco parece ser el encargado de decidir que vista siguiente debe mostrarse. De ser así, el flujo de control de la aplicación se encontraría esparcido entre los controladores de las distintas vistas.

El ejemplo de aplicación expuesto es sencillo pero útil para mostrar la presencia de ciertas tareas para las cuales no ha sido asignado un responsable, al menos entre los tres componentes inicialmente definidos por MVC. Es claro que necesitamos de un objeto que incluya este tipo de decisiones o acciones que llamamos en conjunto: "*lógica de la aplicación*".

3.3 Limitación del diseño MVC tradicional

Se deduce entonces que, en el diseño de tres capas de MVC, el comportamiento que no correspondía a la vista ni tampoco al controlador, se incluía en el modelo.

Es probable que esta solución haya funcionado en el caso de aplicaciones sencillas donde el control de flujo no es complejo, y que no haya producido consecuencias mayores en lo que respecta a la posibilidad de reuso de componentes y a las modificaciones posteriores del sistema.

Lo cierto es que, cuando el control de flujo y la coordinación de las vistas es más elaborado, se necesita separar esta "lógica de la aplicación" del modelo. De no ser así, la modificación posterior en el control de flujo de la aplicación se vuelve complicada ya que es difícil encontrar el código a cambiar para realizar la modificación. En otras palabras, no se están separando correctamente las responsabilidades.

3.4 Rediseño de MVC: controlador de la aplicación

La necesidad de añadir una cuarta capa en el diseño se hizo evidente ya en VisualWorks con la introducción del framework ValueModel, y el *ApplicationModel*. Recordemos que esta capa de objetos se encarga de coordinar el flujo entre las distintas vistas y de conectar a los distintos componentes para obtener mayor desacoplamiento.

El agregado de esta nueva capa, obligó a rediseñar el MVC clásico. El nuevo diseño al que se hace referencia hoy en día cuando una aplicación es orientada a MVC, es el que aparece en la figura 3.7.

El ***Controlador de Entrada*** cumple el mismo rol de controlador definido en el MVC clásico. El ***Controlador de la Aplicación*** es el componente agregado que se encarga de coordinar y controlar el flujo de la aplicación. El ***Modelo*** únicamente contiene los datos y la lógica de negocios de la aplicación y la ***Vista*** se mantiene tal cual en el diseño original. De esta manera el modelo resulta independiente de la aplicación y puede ser conectado a otras aplicaciones.

Asimismo, el nuevo controlador podría convertirse en mediador entre el modelo y los demás componentes. Así, el controlador de entrada puede interactuar con él para acceder al modelo. Además, el *Controlador de la aplicación* podría encargarse de la notificación de cambios del modelo y de referenciar a las vistas dependientes, separando aun más al modelo de la forma en que se lo presenta en la interfaz.

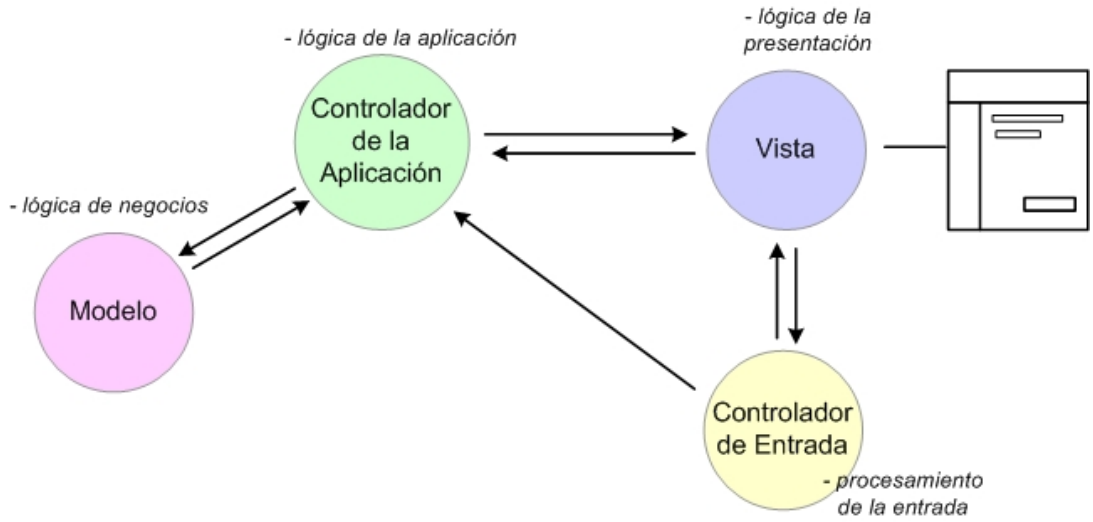


Figura 3.7 – Un MVC actual

En el ejemplo de venta de CD´s, el *controlador de la aplicación* es el que luego de conocer si el usuario logró registrarse o no, genera la vista correspondiente de acuerdo a esta situación.

3.5 Controlador de la aplicación: distintas responsabilidades

Las responsabilidades de este nuevo componente no son las mismas en todos los frameworks en que se implementa MVC. Por ejemplo, los controladores de entrada, que son quienes activan el funcionamiento de la aplicación manipulando el modelo, podrían directamente interactuar con el modelo en lugar de usar al ***controlador de la aplicación*** como intermediario, y solo delegarían en éste la generación de la siguiente vista a desplegar. Sin embargo, es útil rescatar la importancia de disponer de un responsable de la lógica de la aplicación. La aplicación desarrollada en base a esta arquitectura está formada por modelo, vistas y controladores para estas vistas, con roles bien definidos y con una separación más clara contribuyendo así a facilitar modificaciones y al posterior reuso de los distintos componentes.

Capítulo 4
Introducción a MVC en la Web

El MVC original fue ideado para su uso en ambientes GUI tradicionales. Existen ciertas características que una tecnología debe incluir para poder implementar MVC de una manera correcta o aceptable.

El mecanismo observable-observer para la comunicación entre el modelo y sus dependientes es una propiedad inherente a MVC. VisualWorks provee un mecanismo para el manejo de dependencias entre objetos sobre el cual puede construirse otro mecanismo, el de las dependencias entre modelo y vistas de una interfaz gráfica de usuario. VisualWorks también se encarga de la notificación de cambios a los dependientes (observers) cuando su modelo (observable) ha cambiado.

La web presenta ciertos condicionamientos implementativos. Entre ellos se encuentra la conexión "sin estado" entre el cliente y servidor. Este condicionamiento trae varias consecuencias que afectan a las características de MVC, es decir, a las responsabilidades de sus componentes y a la comunicación entre ellos.

4.1 Conceptos básicos relacionados con la Web

Entre los pilares en los que se apoya la Web se encuentran el modelo cliente-servidor, el protocolo HTTP y el lenguaje HTML.

El Modelo cliente-servidor es una forma de interacción entre dos procesos que se ejecutan en forma simultánea. La comunicación está basada en una serie de preguntas y respuestas que asegura que, si dos aplicaciones intentan comunicarse, una comienza la ejecución y espera indefinidamente que la otra le responda y luego continua con el proceso. El **cliente** es la aplicación que inicia la comunicación, dirigida por el usuario. El **servidor** responde a los requerimientos de los clientes y es un proceso que se ejecuta indefinidamente.

Figura 4.1 – Comunicación básica cliente-servidor

HTML ("Hypertext Markup Language") es el lenguaje que define el formato de las páginas web, separando el contenido del formato de presentación. Una página HTML está compuesta de tags HTML (etiquetas) y contenido puro. Los tags encierran al contenido para darle un formato determinado. HTML es el lenguaje interpretado por los browsers web.

El **protocolo HTTP**[1] especifica la serie de mensajes que se envían dos aplicaciones que actúan como cliente y servidor a través de la web. Como su nombre lo indica, sirve para transferir hipertexto entre clientes y servidores. Un ejemplo de cliente HTTP es un browser Web.

Una **aplicación web** es una aplicación que se encuentra en el servidor corriendo indefinidamente para que los usuarios accedan a la misma a través de la Web. El acceso se realiza utilizando un cliente web (browser) que genera requerimientos o pedidos de páginas de esta aplicación (la vista de la aplicación). El servidor contenedor de la aplicación recibe el pedido y lo transfiere a la aplicación correspondiente, la cual devuelve la página HTML requerida por el cliente. Finalmente el servidor envía esta página al browser para que el usuario pueda interactuar con ella.

[1] Del inglés HyperText Transfer Protocol, Protocolo de transferencia de hipertexto.

4.2 ¿Por qué MVC en la Web?

Cuando comenzaron a desarrollarse las primeras aplicaciones web, y a medida que crecía el tamaño y complejidad de las mismas, se hizo necesario renovar la forma en que se diseñaban las aplicaciones. El nuevo diseño no solo separa responsabilidades entre las distintas partes que componen la aplicación, sino también entre programadores y diseñadores de interfaces.

Tal vez los desarrolladores de software de aplicaciones de escritorio hayan sido quienes decidieron "probar" transportar MVC al mundo de las aplicaciones web impulsados por los buenos resultados obtenidos al aplicar la arquitectura en las aplicaciones que ellos construían.

Otro camino posible es que, los desarrolladores de aplicaciones web hayan detectado las complicaciones posteriores al desarrollo de una aplicación en la modificación y/o extensión del sistema desarrollado, que se les presentaban como consecuencia de no separar correctamente los distintos roles en la aplicación. Cualquiera haya sido el motivo es indiscutible que, en la actualidad, MVC es considerado la arquitectura para construir aplicaciones web. Incluso también impacta sobre el diseño de aplicaciones para dispositivos portátiles, las cuales representan otro contexto importante hoy en día en el desarrollo de aplicaciones de software.

4.3 Factores inherentes a la Web que influyeron sobre el MVC original

A continuación se listan algunas características propias de la Web que obligan a cambiar el diseño MVC tradicional para poder aplicarlo en este contexto.

4.3.1 La conexión sin estado entre el cliente y el servidor

Para un usuario, una aplicación web no es más que un conjunto de páginas accesibles a través de un browser, con una determinada forma de relacionarse entre ellas, o sea, una manera de pasar desde una a otra.

El servidor contiene al conjunto de páginas que son enviadas de a una al usuario a medida que son requeridas. Luego de que una página es enviada a través de la web y llega al cliente, la conexión entre el servidor y el cliente se rompe automáticamente. Esto se debe a que la duración de una conexión HTTP es de solamente un pedido de página. Gracias a esta característica, el servidor es capaz de atender incontables requerimientos en forma simultánea. Por esta razón, cuando un usuario envía un pedido para ver una página distinta, el servidor no conoce en absoluto si este usuario ya ha requerido páginas anteriormente, o si es la primera vez que hace un pedido.

4.3.2 El lugar físico donde reside la "vista" de la aplicación web es distinto del que contiene al resto de la aplicación

La vista es la página actual que el usuario ve a través de su browser. El servidor contenedor de la aplicación a la que este usuario accede, se encuentra en otra máquina o lugar físico distinto. Por lo tanto, es claro que la presentación de la aplicación se encuentra en donde se halla el cliente y que la aplicación en sí se encuentra en el servidor.

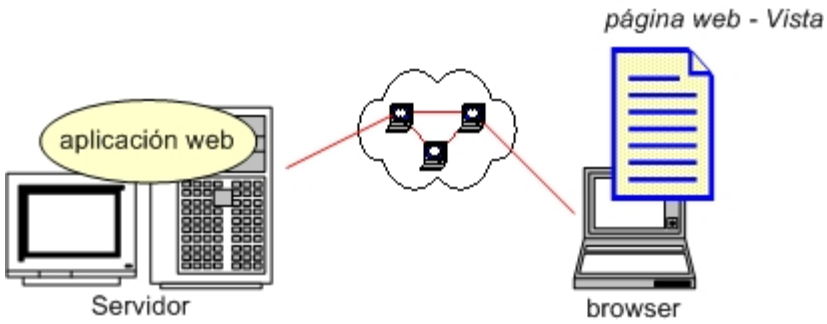


Figura 4.2

4.3.3 La diferencia entre la tecnología usada para la vista, y la usada para los otros dos componentes de MVC

Si la vista es una página en un browser web, entonces debe ser enviada en formato HTML[1] para que el browser pueda interpretarla y desplegarla, independientemente de la

[1] Del inglés Hypertext Markup Language, es el lenguaje que interpretan los navegadores web actuales.

tecnología usada para implementar nuestra aplicación. La aplicación podría estar implementada en Java o C#, pero a la hora de materializar la presentación de la aplicación, habrá que realizar las traducciones necesarias a HTML antes de enviar la vista al cliente.

4.4 Consecuencias de los factores anteriores sobre MVC

A raíz de la existencia de estos factores inherentes a la Web, es posible enumerar los efectos que originan sobre el diseño MVC.

4.4.1 Sobre la notificación de cambios del modelo

La vista es creada y enviada al cliente. Como no existe conexión real entre servidor y cliente, la vista no está conectada al modelo. Como resultado, si el modelo cambia, la vista no tiene forma de ser notificada. No es posible implementar el mecanismo de dependencias y notificación típico del MVC original.

4.4.2 Sobre las vistas y la actualización en respuesta a la notificación de cambios

Tampoco es posible mantener la consistencia entre los datos del modelo y lo que muestra la vista en forma automática. La vista no está sincronizada con él, por lo cual, ocurrido un cambio en el estado del modelo, la vista no se actualiza automáticamente. Modelo y vista serán consistentes cuando el usuario que interactúa con esta última envíe un requerimiento al servidor, resultado de alguna acción sobre la misma.

No es posible conocer cuántas vistas está viendo el usuario.

No hay posibilidad de que las transiciones desde una vista a otra sean directas, como ocurría en las aplicaciones de escritorio, donde una vista podía abrirse como resultado de un evento click en otra vista. Este pasaje se vuelve más complejo y requiere de la intervención de un

controlador. Nuevamente, como el controlador no está en el mismo lugar que la vista, tampoco el pasaje de una vista a otra es una tarea realizable sin la necesidad de comunicarse con el servidor.

4.4.3 Sobre los eventos generados en la interfaz

Los eventos no producen feedback automáticamente como suele pasar en un MVC de una aplicación de escritorio.

Como consecuencia de los distintos lugares físicos donde residen la vista y el resto de la aplicación, tanto el clic sobre un botón como la selección en una lista de la interfaz se convierten en un requerimiento HTTP. Es imposible enviar un evento click o una selección como "objetos" a través de la web, sino que alguien deberá traducir el HTTP request a un evento que la aplicación pueda interpretar.

Estos eventos WEB desembocan en el envío de un mensaje al servidor, por lo cual el feedback del evento se obtiene de forma más lenta.

4.4.4 Sobre las responsabilidades del controlador

Mientras que el trabajo del controlador era recibir un evento en forma de objeto y traducirlo en mensajes que el modelo pudiera entender, ahora su rol se reduce a la recepción de un requerimiento http y su procesamiento para la creación de ese objeto evento. Luego él u otro objeto es responsable de tomar el evento e interactuar con el modelo de la aplicación.

Además, al controlador, se le agrega la responsabilidad de seleccionar la vista a desplegar, es decir, de crearla y enviarla al cliente. Esta responsabilidad es consecuencia de la ausencia de la vista como objeto del lado del servidor. Notemos que aún en el caso de que la misma vista deba redibujarse, será necesaria la intervención del controlador.

Asimismo, recordemos que en general, la vista creaba a su controlador. El rol del controlador cambia forzadamente siendo él el encargado de crearla.

Conjuntamente con la creación de la vista, el controlador debe poner a disponibilidad de la misma los objetos del modelo que la vista necesitará para desplegarse. Como el modelo está en el servidor y la vista no, ésta no podrá interrogar por sí misma a su modelo para reunir los datos necesarios para mostrarse correctamente sino que, cuando la vista es enviada al cliente ésta ya contiene toda la información cargada. En contraste, en un MVC tradicional, cuando la vista es creada, primero recupera la información necesaria enviando mensajes al modelo y luego se despliega en la pantalla.

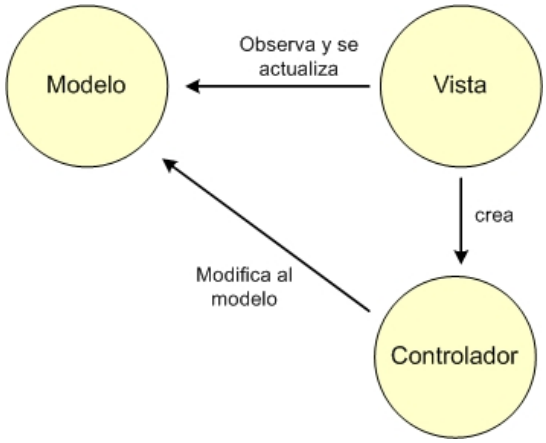


Figura 4.3 – Algunas responsabilidades de los componentes en un MVC tradicional

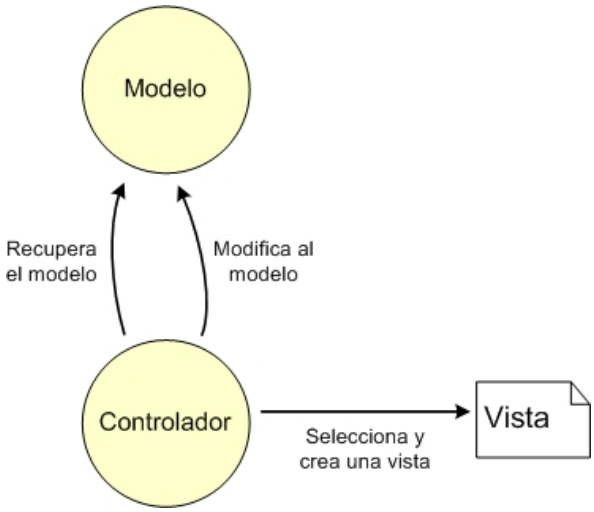


Figura 4.4 – Cambio de responsabilidades de los componentes

Otro punto a considerar es que el controlador no puede recuperar información desde la vista, por eso, todo dato que pueda necesitar de ella debe ser enviado en el requerimiento HTTP inicial.

4.4.5 Sobre los componentes que forman una vista y el MVC a nivel componente

El MVC a nivel componente tradicional[1] no es implementable en la Web. Los componentes que constituyen una interfaz, como lo es por ejemplo una lista de ítems, no observan un aspecto que forma parte de un modelo más grande. Mucho menos podrán actualizarse si este aspecto cambia. No existe el concepto de MVC's pequeños formando a un MVC más grande.

4.5 Soluciones a los "problemas" Web

Tanto la desconexión entre cliente y servidor, como la diferencia entre tecnologías utilizadas para programar la vista por un lado, y modelo y controlador por otro, son cuestiones que han sido resueltas parcialmente por los programadores web y los frameworks para construir aplicaciones que presentaremos en el capítulo siguiente.

4.5.1 Mantener la conexión entre cliente y servidor: la sesión

Existen técnicas para lograr que un servidor de aplicaciones pueda establecer una conexión con un cliente y mantenerla a través de los sucesivos requerimientos HTTP. Una forma de efectuarlo es el de mantener una sesión.

Una **sesión** es un objeto que se encuentra almacenado en el servidor y que identifica a un usuario que está accediendo a una aplicación que corre en el servidor. El usuario da inicio a esta sesión la primera vez que realiza un requerimiento de página y el servidor mantiene la sesión para recordar información sobre él, como por ejemplo, la secuencia de pasos que ha dado en una transacción bancaria. Cuando el usuario se retira, la sesión finaliza y es borrada del servidor. La generación y el mantenimiento de este objeto son habitualmente realizados por el framework que

[1] Ver capítulo 2, sección 2.3.9 "MVC a nivel componente".

el programador utilice para construir su aplicación web. Normalmente, un programador almacena en este objeto, información que sea relevante para el tipo de operación que el usuario está llevando a cabo. Volviendo al ejemplo de las ventas de CD: la sesión de usuario puede contener el estado actual en el que se encuentra la compra.

4.5.2 Diseñadores vs. Programadores: Push MVC y Pull MVC

En las primeras aplicaciones Web, los mismos programadores Java que implementaban la lógica de negocios de la aplicación eran los que diseñaban y construían la interfaz HTML. Como no eran aficionados al diseño de páginas web sino especialistas en Java, las interfaces no resultaban muy amigables y atractivas a los ojos del usuario que interactuaba con ellas.

Dado que la interfaz debe generarse con HTML, la solución a esta cuestión fue incluir un grupo de diseñadores de páginas web, expertos en HTML, que se dedicaran al diseño y construcción de la interfaz de la aplicación.

HTML es un lenguaje que solamente permite generar contenido estático. Sin embargo, habitualmente, la vista debe recuperar información del modelo y luego darle un determinado formato para que esta información sea presentada al usuario. HTML únicamente formatea el contenido y por esta razón, es necesario obtener de alguna manera este contenido "dinámico" del modelo.

4.5.2.1 Pull MVC: similitud al MVC tradicional

Una de las formas de recuperar el contenido del modelo es insertando porciones de código que no es HTML, escrito en algún lenguaje de programación (ejemplo: Java) que se ejecuta en el servidor antes de enviar la página al cliente. El código resultante es puramente HTML lo cual se envía finalmente al cliente.

Este diseño de MVC es similar al MVC tradicional ya que la vista contiene el código que recupera la información del modelo, si bien existe una diferencia fundamental con respecto al MVC tradicional, donde los lenguajes de formato y generación de contenido son uno solo.

En el ejemplo de la vista de un objeto CD, ésta sería una página que contendría código HTML más una porción de código Java[1] donde el objeto CD es recuperado para insertar las propiedades del CD en la página.

Las consecuencias de usar Pull MVC en una aplicación son:

- La **dependencia de la vista con respecto al modelo.** Si la interfaz del modelo cambia, los cambios se propagan en el código de la vista que recupera el contenido a mostrar.

- Se generan interdependencias entre **diseñadores de páginas y programadores java** y/o

- Los **diseñadores web deben poseer algunos conocimientos del lenguaje con que se programa el resto de la aplicación (java)** para independizarse de los programadores java**.**

Debemos destacar que existen actualmente mejoras a este modelo que contribuyen a facilitar el trabajo de los diseñadores de páginas quitando el código java de las páginas. Los "custom tags" son una librería de tags con la misma sintaxis que HTML pero que se traducen a código Java. De esta manera, si bien existe una etapa de aprendizaje de los tags por parte de los diseñadores, no es tan chocante como tener que agregar código Java puro en las páginas de la interfaz. Aun así, los programadores y diseñadores tienen que ponerse de acuerdo en qué tareas realiza cada grupo.

- Nuevos tags requieren de la intervención de los programadores.

- La arquitectura no obliga a usar solo tags, por lo cual sigue estando permitido escribir código java en el HTML, lo cual no promueve la separación presentación-contenido.

[1] Se utiliza a Java como ejemplo, pero puede utilizarse cualquier lenguaje que se ejecute en el servidor para la misma finalidad.

La tecnología JSP se clasifica en este tipo de MVC. Java Server Pages es una tecnología java que permite generar contenido HTML desde páginas con extensión .jsp. Estas páginas pueden contener código Java embebido que se ejecuta en el servidor. Todos los frameworks que utilicen JSP como presentación entran en este grupo: Struts es un ejemplo de ello[1].

4.5.2.2 Push MVC: la Vista independiente del modelo

Otra propuesta para resolver el problema de la diferencia entre tecnologías, es el modelo denominado "push MVC". Este esquema permite eliminar definitivamente el código java de la vista. Así, no existe código embebido de ningún lenguaje de programación en el HTML.

Este modelo se basa en un motor generador de plantillas y un compilador de páginas HTML. El compilador toma una página HTML y la parsea creando una clase Java que representa a la página. Básicamente, esta clase contiene atributos para cada elemento de la página y una forma de acceder y dar valor a cada atributo (operaciones comúnmente llamadas *get* y *set*). En otras palabras, la clase es una plantilla (template).

Cuando esta página es requerida por el cliente, se busca la clase correspondiente y se asignan valores a los atributos. Luego esta clase se convierte a código HTML que se envía finalmente al cliente.

Entonces, la vista de los datos del CD no necesita recuperar el objeto CD sino que se genera una clase *VistaCDGenerator* cuyas propiedades son las propiedades del CD y que también contiene los métodos necesarios para obtener la información así como para darle valor. Ej: la propiedad *titulo* y sus respectivos métodos *getTitulo()* y *setTitulo(unTitulo)*.

Las consecuencias de usar Push MVC en la aplicación son:

- **La vista es menos dependiente del modelo**.

[1] Struts y JSP son temas que se explican con mayor detalle en el capítulo siguiente.

- Los **diseñadores de páginas son menos dependientes de los programadores java** para cambiar el contenido de las páginas. Generalmente, las implementaciones que siguen este modelo, recompilan la página ante algún cambio en la misma, es decir, la clase Java se vuelve a generar.

- La vista solo puede contener código HTML puro, lo cual obliga a separar presentación de contenido.

El compilador XMLC que describiremos más adelante es un ejemplo de implementación de este tipo de arquitectura.

Capítulo 5
Implementaciones de MVC en la Web

Más allá de su implementación en Smalltalk, donde MVC hizo su primera aparición pública luego de ser supuestamente inventado por *Trygve Reenskaug*, los principios y conceptos que MVC proporciona han ido tomando importancia en otros lenguajes y áreas. Los desarrolladores han tomado conciencia de esta valiosa herramienta y numerosos frameworks lo han incorporado. A su vez, las aplicaciones interactivas en la Web produjeron un impacto importante en la evolución de MVC.

La Web introduce un contexto complejo de por sí, donde a las aplicaciones que corren en ella, se agregan funcionalidad y responsabilidades que las aplicaciones de escritorio (aun las de mayor tamaño) no necesitan tener en cuenta. Las aplicaciones en la web son entonces, un ejemplo de "aplicaciones complejas y de gran tamaño" a las que nos referíamos en el capítulo 3 cuando mostramos que el MVC tradicional no alcanzaba para el caso de aplicaciones de estas características.

En este capítulo presentamos y analizamos un conjunto de frameworks que implementan MVC en la Web, de la misma forma en que lo hicimos con los frameworks no Web, por dos razones: si un framework incluye estos conceptos, las aplicaciones construidas con él también los contendrán, y segundo, mediante la identificación de estas propiedades en distintos frameworks podremos concluir cuál es la mejor forma de implementarlas e incluirlas a todas ellas.

Los frameworks y tecnologías para la construcción de aplicaciones web se encargaron de resolver las complejidades que se presentan como consecuencia de los factores inherentes a la web que hemos visto en el capítulo anterior. Un ejemplo es la clase Servlet de la tecnología Java Servlets que especifica que un objeto de su clase debe ser invocado recibiendo dos objetos como parámetro pertenecientes a las clases HttpRequest y HttpResponse respectivamente. El HttpRequest contiene los datos del requerimiento HTTP, y el HttpResponse se utiliza para devolver una respuesta al cliente. En otras palabras, existe en el servidor un encargado de recibir el requerimiento, parsearlo, crear los objetos correspondientes y setear sus valores con el

resultado del parseo, para finalmente enviarlos como parámetro a un servlet definido por el programador.

Además, este framework mantiene un objeto sesión por usuario que perdura a través de distintos mensajes HTTP del mismo usuario.

Otros frameworks como Struts, abstraen aún más al programador de tener que implementar estas tareas. Como resultado, el programador de la aplicación Web puede concentrarse en el diseño puro de su aplicación sin tener que perder tiempo implementando funcionalidad relacionada con el parseo de requerimientos HTTP, creación de sesiones, etc.

Algunos frameworks ponen a disposición del programador tanto un controlador como objetos para armar la interfaz de usuario, mientras que otros solamente implementan un controlador dejando la presentación para otra tecnología a elección del programador. Un tercer tipo de framework se aboca a la construcción de la presentación de la aplicación pudiendo ser integrado con otra tecnología para el rol de controlador y modelo de la aplicación.

5.1 Redefinición de propiedades de MVC en el contexto Web

Con el objetivo de analizar las implementaciones que se listan y explican en este capítulo, partimos del conjunto de propiedades utilizado para el análisis de los frameworks VisualWorks y Java Swing.

Listaremos entonces un grupo de características las cuales, en su mayoría, ya han sido definidas en Cap. 2 sección 2.3, y determinaremos su aplicabilidad en el contexto de la Web. Si la característica descripta es aplicable, podrá ser redefinida, en caso de ser necesario, para adaptarla al contexto de la Web. A su vez se agregan a la lista, nuevas propiedades que surgen de transportar MVC al nuevo ámbito.

5.1.1 Manejo de dependencias del modelo y propagación de cambios (no aplicable)

No es posible implementar el mecanismo de dependencias y notificación propio del MVC original. En Cap. 4 sección 4.4.1 se explicó cómo esta característica resulta afectada al llevar MVC a la Web. Por lo tanto, esta propiedad no estará incluida en el análisis.

5.1.2 Actualización inmediata de dependientes (no aplicable)

Cuando la vista y el modelo residen en un mismo lugar, la actualización de las vistas se realiza en forma inmediata. Las vistas son capaces de "escuchar" los cambios en el modelo y actualizarse inmediatamente manteniendo la consistencia modelo-vista.

En el contexto de la Web no es posible mantener esta consistencia en forma automática porque la vista no es capaz de actualizarse seguidamente luego de ocurrido un cambio en el modelo. Modelo y vista serán consistentes cuando el usuario que interactúa con esta última envíe un requerimiento al servidor resultado de alguna acción sobre la misma.

5.1.3 Independencia de la vista con respecto al modelo (aplicable)

Esta propiedad definida en Cap.2 sección 2.3.3 ha sido analizada en Smalltalk y Swing (Cap.2 secciones 2.5.3 y 2.7.3, respectivamente) y es implementable en el contexto de la Web. Por lo tanto, incluiremos esta característica en el análisis de frameworks Web MVC.

5.1.4 Separación entre Vista y Controlador: Independencia de la vista con respecto a su controlador (aplicable)

Esta característica permite sustituir el controlador asociado a una vista para cambiar la interacción que la vista ofrece al usuario y es factible su implementación en una aplicación web.

5.1.5 Independencia del Controlador con respecto al Modelo (aplicable)

Hemos demostrado la necesidad de que el controlador sea independiente de la interfaz que el modelo exporta y hemos explicado que la aplicación del patrón *Command* es una solución para alcanzar esta separación. Algunos de los frameworks que se analizarán han tenido en cuenta esta abstracción entre componentes para el reuso de controladores.

5.1.6 Separación total del modelo y sus vistas (no aplicable)

Cuando analizamos esta característica en VisualWorks, mencionamos que un objeto intermedio entre el modelo y sus dependientes permite que el modelo no conozca en absoluto sobre la existencia de las vistas dependientes. En otras palabras, el objeto intermedio es el responsable de observar al modelo y notificar a las vistas dependientes. En la Web este concepto pierde sentido porque no es posible notificar a las vistas sobre cambios del modelo y que ellas se actualicen en forma inmediata como respuesta (ver 5.1.1 y 5.1.2). En consecuencia, al no existir la notificación de cambios, el modelo está desligado de sus vistas por defecto.

5.1.7 Múltiples vistas para un mismo modelo y sincronización entre ellas (no aplicable)

Puede que dos “componentes” en una página web dependan del mismo modelo y ambos estén sincronizados al recargarse la página o vista. Sin embargo, lo interesante de esta característica era el hecho de poder tener varias ventanas abiertas desplegando información del modelo y que ante un cambio en el mismo, todas las vistas reflejen el cambio. Esta propiedad depende notablemente del mecanismo de dependencias implementable solamente en el MVC no web. Supongamos tener dos browsers abiertos mostrando la misma vista. Cuando uno de ellos despliegue la vista actualizada, no hay forma de que el segundo browser reaccione actualizándose.

5.1.8 Representación del componente Vista en el servidor (agregada)

Esta propiedad se refiere a la implementación de un objeto **Vista en el servidor** formado por componentes, de manera semejante a una vista GUI. Nos referimos con "representación de vista en el servidor" a un objeto más complejo que una simple página JSP/HTML almacenada en el servidor.

5.1.9 Vistas anidadas, controladores anidados (aplicable)

Existen tecnologías o frameworks que representan a la vista como objeto en el servidor (5.1.8) y que además facilitan la programación de interfaces Web asemejándola a la de interfaces GUI, permitiendo que las vistas se construyan ensamblando componentes que contienen otros componentes todos reusables, (ver Cap. 2 secciones 2.5.6 y 2.7.6 para vistas anidadas en Smalltalk y Swing respectivamente) incluyéndose así el concepto de vistas anidadas, pudiendo asociarse controladores a cada componente visual. Dichos controladores resultan también anidados, ya que el framework debe realizar una búsqueda para hallar el componente que ha generado el evento y delegar el control en el controlador asociado. Los componentes vista son capaces de generar eventos de clases específicas (no eventos HTTP), permitiendo programar interfaces con funcionalidad compleja rememorando un MVC no web. Este concepto suele denominarse *Rich Client* y contrasta con el de *Web Client*, donde los eventos generados por la interfaz son únicamente eventos http.

5.1.10 MVC a nivel componente en la web (redefinida)

En el Cap. 4 sec. 4.4.5 se explicó como resulta afectada la implementación de MVC a nivel componente al transportarlo a la web. La comunicación modelo-vista a nivel componente no es automática y directa como en el MVC tradicional ya que el componente vista (pequeño) no puede enviarse actualizado para que el usuario note esta actualización. El usuario percibe el cambio del componente recién al enviar un requerimiento HTTP al servidor. Sin embargo si se cuenta con una representación de la vista en el servidor (5.1.8) que represente cada componente de la vista como un objeto y si es posible asociar a cada uno su propio controlador entonces

puede mantenerse la relación vista- controlador y controlador-modelo a nivel componente, obteniendo así el concepto que denominaremos **MVC a nivel componente en la web**. Cabe destacar que algunos frameworks permiten asociar una componente vista con una parte del modelo y se encargan de la interacción entre los mismos para mantener la consistencia. Esta propiedad resultará más clara luego de haber estudiado los frameworks web que se analizan en este capítulo.

5.1.11 Controlador de la aplicación (agregada)

Esta propiedad se refiere a la implementación de un controlador que contenga y maneje la lógica de la aplicación y seleccione la siguiente vista a desplegar, de manera que las modificaciones en la lógica de la aplicación se concentren en un solo lugar. Si bien esta característica se hace presente ya en frameworks no web (ver Cap. 2 sección 2.1.6.2), se vuelve obligatoria en el contexto de la Web ya que, independientemente de si la vista actual debe solo actualizarse o debe ser reemplazada por otra, el controlador deberá seleccionar la vista a enviar al cliente Web.

5.1.12 Reuso de la estructura de las vistas (agregada)

Las páginas de una aplicación web habitualmente comparten una misma estructura. Un ejemplo es cuando todas o la mayoría de las vistas poseen una barra al costado izquierdo o en el borde superior. Ante la necesidad de modificar esta estructura, es decir, la disposición de las partes de cada página, deberíamos modificar el código de todas y cada una de ellas. Sin embargo, si aislamos la estructura común en un elemento separado de manera que las páginas puedan referenciarlo o heredarlo, es posible cambiar la estructura de muchas páginas modificando sólo este elemento (ver figura 5.1). Este objeto contiene los componentes que hacen a la organización visual de todas las páginas, así como también componentes específicos comunes que las páginas específicas heredan. Luego, las páginas hijas que utilicen este objeto o extiendan a la clase (dependiendo de la implementación) adicionan las subvistas o componentes concretos de cada página.

Podemos ejemplificar esta característica volviendo al ejemplo del sistema de ventas de CD's. Supongamos que existen varias vistas que contienen un componente ubicado en la parte derecha que muestra el carrito de compras con los CD's elegidos por el usuario. Este componente puede estar ubicado en una página padre.

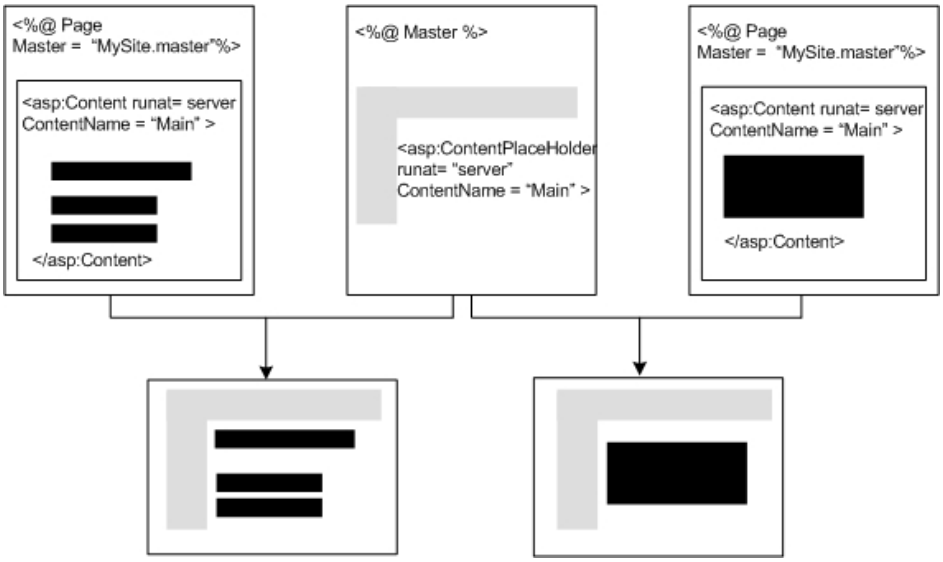


Figura 5.1 Reuso de la estructura de las vistas

5.1.13 Posibilidad de portar interfaces basadas en HTML a otras plataformas (agregada)

Cuando decimos "portar interfaces a otras plataformas", nos referimos a la posibilidad de que una aplicación tenga una interfaz HTML para acceder vía un browser HTML, y que a la vez disponga de otras interfaces para el caso de acceso vía un dispositivo móvil e incluso una interfaz GUI. Relacionamos la portabilidad de la aplicación web con la capacidad de visualizar la interfaz de la aplicación en distintos tipos de clientes ya que la aplicación en sí reside siempre en el servidor.

Contaríamos entonces con distintos tipos de interfaces para un mismo modelo. Distingamos que en este caso estamos considerando distintas interfaces del mismo modelo a nivel "soportar distintos tipos de clientes" y no distintas formas de presentar un mismo objeto.

5.1.14 Reusabilidad de los componentes visuales (agregada)

Si nos interesa disponer de distintos formatos de salida en una misma aplicación (5.1.13) es altamente deseable que un componente se separe en dos partes, una que es independiente de la salida y otra dependiente de ella. Un ejemplo de esto es un componente *Lista* que contiene comportamiento relacionado con eventos y que delegue en un objeto específico a la hora de mostrarse en el cliente. Este objeto específico puede más tarde sustituirse por otro para que el componente sea reusable y se muestre de otra manera en la interfaz. El soporte de nuevos tipos de salida se reduce de programar completamente un nuevo componente para dicha salida a agregar un objeto renderer.

5.1.15 Posibilidad de que una sola interfaz se adapte a quien accede a la aplicación (agregada)

Esta capacidad es la de programar ***una sola interfaz*** cuyos componentes puedan convertirse a la salida que el cliente que está accediendo a la aplicación espera obtener, HTML, XML, WML etc. De esta forma es posible independizar la vista de la aplicación del lenguaje de marcado en que finalmente se envía al cliente.

5.2 Análisis de implementaciones de MVC en la Web

5.2.1 Java Server Pages y Java Servlets: tecnologías para el desarrollo de aplicaciones

Java Server Pages y Java Servlets son tecnologías server-side que han sido por mucho tiempo las dominantes en la programación de aplicaciones web. Su utilización combinada permite lograr una buena separación entre lógica y presentación de una aplicación. Sin embargo, la utilización de Servlets primero y Java Server Pages más tarde fue lo que resultó en la combinación de ambas tecnologías para construir una aplicación web.

5.2.1.1 Servlet monolítico: vista y controlador acoplados

Básicamente, un Servlet es un objeto que se encuentra en un servidor y que recibe un request HTTP[1], realiza algún procesamiento y luego devuelve un response HTTP.

Antes que MVC fuese migrado a la web, la solución por defecto al problema de diseñar una aplicación con interfaz web era hacer que cada Servlet fuese responsable de todos los aspectos del procesamiento del request: lectura de parámetros, procesamiento de información, y formato de la salida. La tecnología Servlets se volvió la más aceptada para construir aplicaciones del lado del servidor. De este modo, el Servlet atiende el pedido de usuario y procesa información, accediendo al modelo de la aplicación, es decir, ejerciendo el rol de *controlador*. Luego del procesamiento de información, el mismo servlet genera la página html que se envía al browser en respuesta a la acción de usuario, para nosotros, una *vista* de la aplicación web. El diagrama que se muestra en la figura 5.2 muestra el tipo de arquitectura al que se hace referencia.

Sin embargo, a medida que el tamaño de la aplicación crece, y cuando varias acciones de usuario pueden desembocar en la misma salida o vista, la solución del servlet realizador de todas las tareas nombradas pierde utilidad. Como la arquitectura de servlet monolítico no separa el procesamiento de la entrada de la generación de la salida, es seguro que, cuando se quiera llegar a una misma página resultado desde dos acciones de entrada diferentes, nos encontremos duplicando código para generarla en los dos servlets correspondientes a cada acción de usuario. Es evidente la necesidad de la separación de responsabilidades entre los objetos encargados de materializar la interfaz.

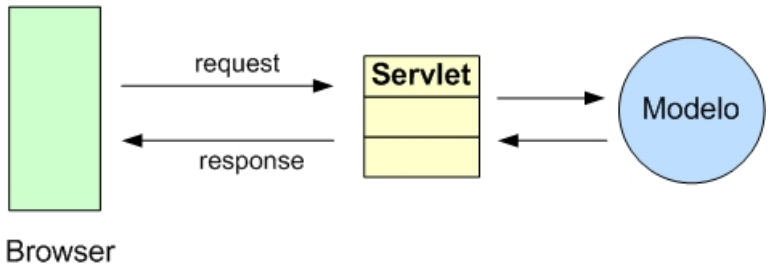


Figura 5.2 – Arquitectura de servlet monolítico

[1] Ver Cap.4 introducción a MVC en la web.

Otro problema es que un Servlet se encuentra realizando un llamado a *out.println()* por cada línea HTML de la página que se envía al browser, lo cual se convierte en un gran problema a medida que las aplicaciones crecen en tamaño y cuando la construcción de las páginas HTML está a cargo de diseñadores y no de programadores Java. Esta división de tareas es común en el desarrollo de aplicaciones de gran tamaño y complejidad.

5.2.1.2 JSP Model 1: vista y controlador acoplados

Java Server Pages intentó en principio reemplazar a Servlets. Una página JSP se traduce y compila a una clase Servlet con la diferencia de que no es necesario generar el código HTML explícitamente. JSP permite escribir las páginas HTML e insertar código Java dentro de las mismas.

Apenas surgió esta tecnología, toda la aplicación era desarrollada usando solamente páginas JSP.

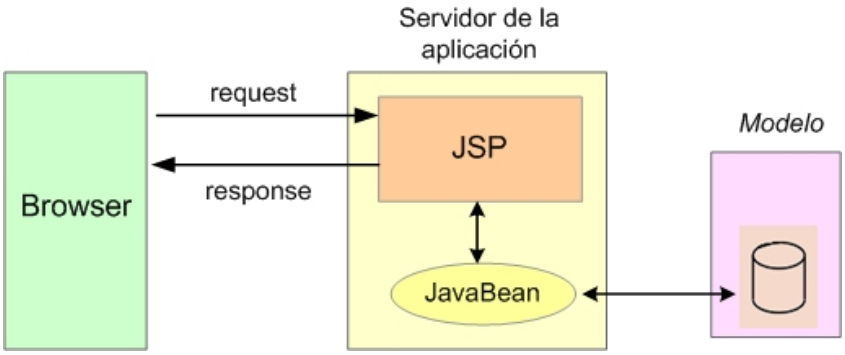


Figura 5.3 – JSP Model 1

De la misma manera que había ocurrido con Servlets, una página JSP era responsable de procesar el request, acceder al modelo y devolver una respuesta al cliente. En esta arquitectura

existe separación entre modelo y presentación debido a que todo el acceso a datos se realiza usando java beans[1].

Model 1 funciona aceptablemente para aplicaciones sencillas. Pero en aplicaciones grandes ocurre que la página JSP se carga excesivamente de código Java.

Esta acumulación en exceso de código Java en las páginas JSP de una aplicación web trae varias desventajas. Primeramente, un diseñador de páginas web no es entendido en código Java, y por lo tanto, cuanta mayor cantidad de código Java haya en la página, el diseñador necesitará consultar más al programador Java.

Además de la consecuencia sobre la separación de roles al desarrollar y mantener la aplicación, nos encontramos nuevamente con que las responsabilidades de controlador y vista en la aplicación están fusionadas en un componente. Como es de esperar, este acoplamiento no permite que la aplicación sea fácil de extender y/o modificar. En otras palabras, no se dispone de las ventajas de un diseño orientado a MVC. Un cambio en la lógica de negocios o en los datos de la aplicación podría obligarnos a retocar cada página JSP afectada por los cambios.

5.2.1.3 JSP Model 2: MVC para la web

La arquitectura que nos interesa es una combinación de JSP y Servlets que permita desarrollar una aplicación web orientada a MVC. Los servlets se encargan del procesamiento de la entrada y la interacción con el modelo, mientras que JSP se utiliza específicamente para la presentación de la aplicación.

Esta arquitectura suele denominarse *JSP Model 2* o simplemente *Model 2*. Los seguidores de este diseño utilizan los nombres *Model 2* y *MVC* indistintamente, ya que para ellos, la arquitectura sigue los lineamientos de MVC.

En la siguiente figura un esquema general es presentado para dar idea sobre los componentes que intervienen en el *Model 2* y la comunicación entre ellos.

[1] En http://java.sun.com/developer/onlineTraining/Beans/ puede hallarse información sobre JavaBeans.

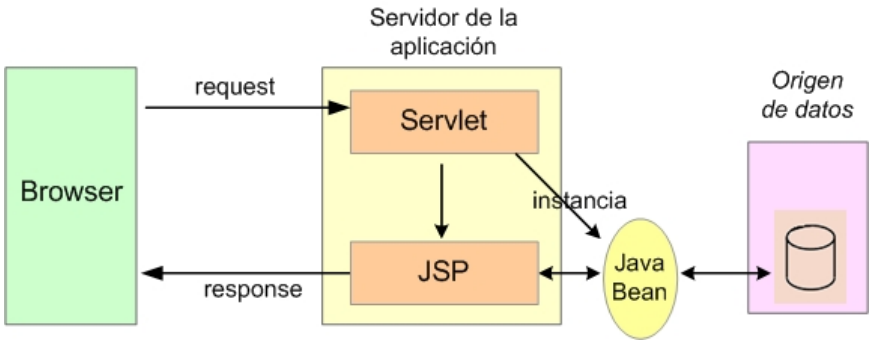


Figura 5.4 – JSP Model 2 o MVC

Básicamente, un Servlet recibe un requerimiento de usuario y lo procesa para luego manipular de alguna manera al modelo. El servlet también recupera cualquier dato necesario que la página JSP necesite y decide qué vista, o sea, qué página JSP es la que debe devolverse al cliente. La página JSP accede a los datos puestos a disposición por el servlet, recuperándolos de la sesión del usuario actual, en forma de JavaBeans.

Como resultado, la lógica de procesamiento es eliminada de las páginas JSP, conteniendo éstas solamente el código java necesario para la recuperación de los beans, es decir, para acceder al modelo a fin de desplegar la información.

En conclusión, **JSP es la presentación de la aplicación** o *Vista* de MVC; y **los Servlets actúan como *Controladores***. El framework Servlets provee una clase Servlet de la cual se derivan los Servlets de nuestra aplicación. Además, una vez programado nuestro Servlet, el framework recibe pedidos http e invoca al método principal del Servlet con el request como parámetro en un objeto de la clase HTTPServletRequest. Este framework también provee objetos Session para mantener información que debe ser persistente a través de distintos requests.

5.2.1.4 Custom tags: no más código Java en la Vista JSP

JSP es un ejemplo de tecnología para la construcción de la presentación de una aplicación web. No es utilizada solamente en combinación con Servlets.

No es conveniente para los desarrolladores de páginas web que las páginas JSP contengan código Java aun cuando éste solo se utilice para recuperar información necesaria para la presentación, ya que se sigue requiriendo de conocimiento de sintaxis Java.

JSP evolucionó haciendo disponible una librería de tags conocida como JSTL[1]. Estos tags están implementados en Java y se usan en la página JSP como cualquier otro tag HTML. Como resultado, no es necesario embeber código Java en las páginas JSP. Además la sintaxis de dichos Tags es simple (parecida a los tags HTML) y son reusables.

5.2.1.5 JSP- Servlets y las propiedades de MVC

Las páginas JSP recuperan la información del modelo a través de objetos JavaBean. La página depende de los métodos o propiedades de estos objetos de modo que, si ellos son modificados, todas las páginas que hagan referencia a estos métodos deberán ser modificadas. De aquí que la vista JSP está ligada fuertemente con el modelo de la aplicación (5.1.3).

Dado que una página JSP encapsula toda la lógica de presentación y un servlet contiene el procesamiento e interacción con el modelo, entonces vista y controlador no están acoplados (5.1.4).

Esta arquitectura no implementa de por sí un command para independizar a un servlet del modelo al que accede (5.1.5). Sin embargo, la arquitectura no impide la implementación de este diseño.

No existe una representación jerárquica de la vista en el servidor (5.1.8). Un Servlet no puede obtener más información desde la vista que la que viene como parámetro en el

[1] JavaServerPages Standard Library, http://java.sun.com/products/jsp/jstl/index.jsp

requerimiento http. Las páginas JSP generan solo eventos http, no existe el concepto de MVC a nivel componente (5.1.10), ya que no es posible asociar un controlador a un componente UI en la vista (5.1.9). Los eventos los genera la página.

Los Servlets procesan la información de entrada a la aplicación y a su vez toman decisiones sobre el flujo de la aplicación, direccionando directamente a una página HTML. Como resultado, el flujo de control de la aplicación y el procesamiento de la entrada no están separados (5.1.11) lo cual puede resultar en duplicación del procesamiento de entrada y en control de flujo diseminado en demasiados objetos.

JSP hace posible el uso de tags para incluir otra página JSP dentro de otra. Por lo tanto, un sector común a todas las páginas podría guardarse separado e incluirse en todas ellas. Sin embargo, no es a lo que apuntamos en 5.1.12. Si la disposición de las partes que componen la estructura de un grupo de páginas debe cambiar, y todas comparten la misma estructura habrá que recorrerlas una por una realizando la modificación. Un ejemplo es un simple encabezado que debe pasar a ser una barra vertical a la izquierda.

Una página JSP permite insertar código java no solo en código HTML sino que también puede insertarse en otro lenguaje de marcado tal como WML. En consecuencia, podemos tener una página llamada bienvenidaHTML.jsp para un cliente que espera un documento HTML como respuesta y otra bienvenidaWML.jsp para clientes wireless, soportando así distintos formatos de salida para la misma aplicación (5.1.13). Sin embargo, no tenemos la posibilidad de programar una sola página que se renderice al formato correspondiente (5.1.15) sino que habrá que hacer una vista nueva para cada tipo de formato o dispositivo que pueda acceder a nuestra aplicación.

Los tags que son los componentes visuales de la página JSP no son reusables. No tienen independencia con respecto al formato en el que se muestran en el cliente (5.1.14). Los tags son específicos del lenguaje de marcado que se envía al cliente.

5.2.2 Struts framework: Controlador para aplicaciones web

Struts es una de las implementaciones Java de MVC en el dominio de las aplicaciones web provisto por el Apache Yakarta Project[1]. Struts implementa un único controlador que rutea los requerimientos http a objetos manejadores que se encargan de la coordinación entre el modelo y los componentes vistas.

Struts promueve el uso de MVC en la web. El corazón del framework es una capa que cumple la función de controlador. Esta capa está implementada con una combinación de tecnologías estándar como Java Servlets, JavaBeans, JSP y XML.

Además de un controlador central, Struts define clases que deben extenderse para trabajar en cooperación con él.

5.2.2.1 Componentes del framework

Struts utiliza el framework Servlets. El componente más importante es un servlet de la clase *ActionServlet,* configurable vía objetos de la clase *ActionMapping*. Esta última clase representa a los objetos que contienen un string y un nombre de clase *Action*. Cuando un requerimiento llega al *ActionServlet*, éste busca entre los *mappings* al que contiene el string que coincida con el requerimiento para instanciar e invocar al *Action* de la clase correspondiente. En resumen, el *ActionServlet* mapea eventos, que en el idioma de la web son requerimientos HTTP, a clases *Action.* Las clases nombradas son definidas por Struts, algunas de las cuales heredan de las clases provistas por el framework Servlets.

5.2.2.2 El rol de Controlador: ActionServlet y Actions

La tarea del controlador es realizada por el *ActionServlet*. *ActionServlet* crea y usa las clases *Action, ActionForm* y *ActionForward.* El archivo struts-config.xml contiene la

[1] http://jakarta.apache.org/

información acerca de las *Actions* y configura al *ActionServlet*. A medida que se va creando la aplicación web, se extienden las clases *Action* y *ActionForm* para resolver al problema de cómo responder a un requerimiento de usuario. Entonces el *ActionServlet* sabe, gracias al archivo de configuración, qué objetos instanciar e invocar. El flujo de la aplicación esta contenido en este archivo.

ActionForm es una clase abstracta que es subclaseada para cada form de entrada definido por el programador. Representa al conjunto de datos que es seteado o actualizado con los valores de los campos de un form HTML. Cuando un *Action* requiere de un determinado *ActionForm* (esto está especificado en el archivo struts-config.xml), y antes de delegar en el *Action,* Struts chequea si el formulario ya existe, y de no ser así, lo crea. Luego setea los valores del formulario con los correspondientes valores que vienen en el requerimiento HTTP y se lo pasa a la acción como parámetro en un método execute() que todos los *Action* deben implementar.

Los objetos *Action* encapsulan llamadas a la lógica de negocios de la aplicación, es decir, a la funcionalidad de la aplicación. Además, estos objetos, luego de interactuar con el modelo, deciden cual es la próxima vista a desplegar.

5.2.2.3 Un modelo para Struts: JavaBeans

El modelo en Struts está formado por el estado del sistema y las acciones que pueden realizarse sobre él. Struts provee la capa que actúa como controlador de la aplicación y no especifica una determinada tecnología a ser usada como representación del modelo, aunque en general se usa JavaBeans. Las propiedades de los JavaBeans son el estado, mientras que sus métodos son las acciones de las que hablamos. En algunos casos las acciones no son métodos de los JavaBeans sino que están implementadas en los objetos Action. Struts promueve la idea de que un objeto *Action* debe llamar a una acción del modelo, o sea traducir la acción a una entendible por el modelo. Para nosotros entonces un objeto *Action* complementa la tarea del controlador.

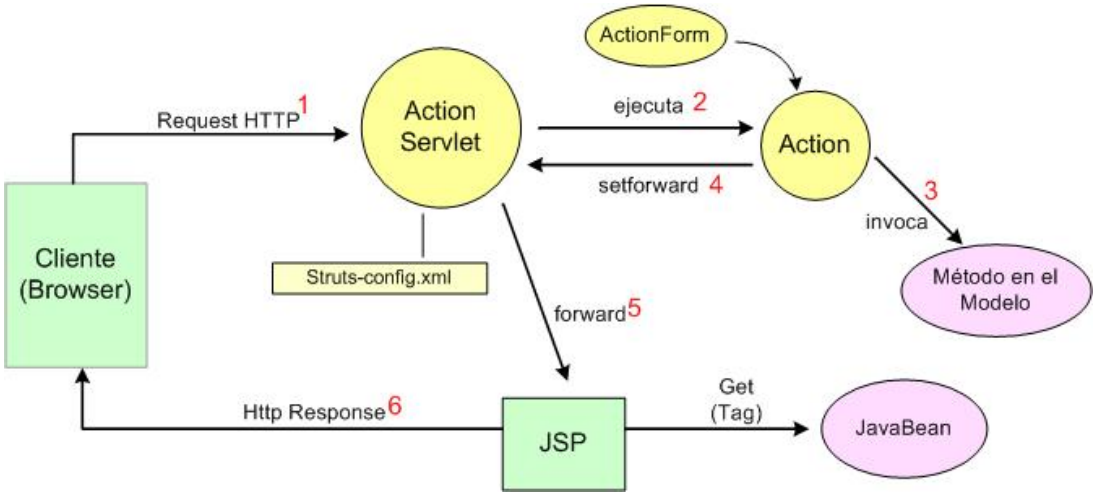


Figura 5.5 – Funcionamiento y componentes de Struts

Ejemplos de otras tecnologías estándar que proveen acceso a datos son Enterprise Java Beans[1] y JDBC[2].

5.2.2.4 Java Server Pages y Taglibs de Struts para el rol de Vista

La vista de la aplicación desarrollada con Struts se construye generalmente con JSP, si bien Struts soporta otros tipos de tecnologías para la vista[3]. Struts provee una colección de tags *Jakarta-Taglibs*[4] para JSP que se suman a los ya definidos por JSP (5.2.1.4). Estos tags permiten eliminar completamente el código Java que se mezcla con el código HTML para desplegar correctamente la información al usuario. De esta manera, el diseñador de las páginas no tiene por qué saber programar en Java.

[1] En http://java.sun.com/products/ejb/index.jsp puede encontrarse información sobre la tecnología EJB y su especificación.
[2] JDBC API, http://java.sun.com/products/jdbc/
[3] Velocity Templates, XSLT, y otros sistemas para la presentación.
[4] Para más información sobre Taglibs http://jakarta.apache.org/taglibs/

5.2.2.5 Interacción de los componentes del framework

Supongamos que se está enviando una página con un formulario al servidor. El *ActionServlet* recibe el requerimiento y lo procesa para obtener los parámetros que vienen en el requerimiento y además conocer la acción a la que debe invocar. Para instanciar el *Action* adecuado deberá inspeccionar el archivo de configuración donde se asocian las urls con las clases *Action* correspondientes. Además, si el *Action* requiere de un *ActionForm,* el *ActionServlet* crea el *Action* y el *ActionForm* correspondiente seteando sus valores con los parámetros de entrada. Por último, el *ActionServlet* invoca al método execute() del *Action,* con el *ActionForm* creado como parámetro.

El *Action* realiza las operaciones correspondientes utilizando los datos del *ActionForm* y comunicándose con el modelo de la aplicación. Finalmente selecciona un *ActionForward* especificando un string. El *ActionServlet* toma este *forward* y redirecciona a la página jsp especificada en él. La página jsp puede contener tags provistos por Struts, los cuales hacen referencia a JavaBeans para mostrar los datos en la página. Estos Beans pueden ser un *ActionForm*, o cualquier otro Bean disponible para esta página.

5.2.2.6 Struts y las propiedades de MVC

Dado que JSP es la vista de Struts, los comentarios para algunas propiedades son los mismos que para la arquitectura JSP-Servlets. La librería de tags JSP provista por Struts es un aporte importante, pero aún en estos tags se puede acceder directamente a los objetos del modelo enviándoles mensajes que ellos exportan, mensajes que al ser modificados propagarían la modificación al código de las páginas. La vista no se independiza de la interfaz del modelo (5.1.3) a no ser que utilicemos siempre ActionForms, creados por Struts y previamente seteemos sus valores antes de direccionar a la página JSP.

El ActionServlet es el único punto de entrada a la aplicación, que procesa el requerimiento http delegando en un Action. Estos dos objetos realizan en conjunto la tarea de controladores mientras que la lógica de presentación está aislada en las páginas JSP. Es posible conectar un

controlador o Action diferente a una página JSP en cualquier momento. La separación de roles entre vista y controlador está bien definida (5.1.4).

Struts implementa un command donde el controlador es el ActionServlet y los objetos Action son los objetos command que se comunican con el modelo (5.1.5).

Las propiedades 5.1.8, 5.1.9 y 5.1.10 se mantienen al igual que en la sección 5.2.1.5 ya que son consecuencia de usar JSP. El agregado de nuevos tags no influye en estas propiedades.

El ActionServlet procesa la información de entrada a la aplicación y delega en un Action. El ActionServlet controla el flujo de la aplicación. Los Actions deciden cual es la próxima vista a desplegar especificando un string. El ActionServlet usa el string para redireccionar a la página asociada al mismo enviando la página al cliente. De modo que un Action decide a que "estado" pasar donde este estado está asociado a una página concreta en el ActionServlet. El rol de control de la aplicación está implementado en el framework (5.1.11).

Struts es una capa controlador para las aplicaciones Web, por eso no introduce ninguna mejora en desligar el contenido y componentes de la interfaz de la forma en que se disponen en la página (5.1.12). Los comentarios sobre las propiedades 5.1.13, 5.1.14 y 5.1.15 no cambian para Struts, dado que dependen de la tecnología JSP.

5.2.3 WebActions framework

Este framework utiliza una combinación de scripts[1], Server pages, y objetos comunes para implementar los componentes que lo constituyen. El framework básicamente implementa las acciones que realiza un controlador de la aplicación, y define clases abstractas para el procesamiento de entrada y la generación de la salida de la aplicación. Como resultado, la aplicación podría producir distintos tipos de salida, y no solo la interpretada por un browser web.

[1] Programa pequeño que realiza procesamiento HTTP, un Servlet es un ejemplo de script implementado en Java.

5.2.3.1 Input Controller: procesamiento de la entrada

Este componente se implementa con un script o servlet y se encarga de procesar y validar la entrada que, en el contexto de la Web, es el parseo del requerimiento HTTP y la extracción de los parámetros. Este objeto *InputController* escoge a un *ApplicationController* y le envía los datos extraídos del requerimiento.

El *InputController* es único para todas las páginas de la aplicación web lo cual permite tener en un único lugar, el código que procesa los eventos de entrada, de manera que cualquier modificación en esta funcionalidad se realiza sobre este objeto.

La clase que representa al *InputController* es una clase abstracta, por lo tanto es necesario definir una clase *HTTPInputController* que procese la entrada HTTP. Además es posible extender la clase abstracta para procesar otros tipos de entrada (ver figura 5.6).

5.2.3.2 Application Controller: capa de Lógica de la aplicación

El *ApplicationController* se implementa como un objeto ordinario y **no es único en la aplicación**. Cada *ApplicationController* se dedica a un área de la aplicación. Juntos forman una capa de objetos conteniendo el código que tiene que ver con el flujo de la aplicación y separando a las otras tres capas que son: la lógica de negocios, el procesamiento de entrada y la presentación; a la vez que permite la comunicación entre las mismas.

Un *ApplicationController* controla errores, mantiene referencias a objetos del modelo, ejecuta acciones sobre el mismo y determina qué vista es la siguiente a desplegarse.

El *ApplicationController* es el punto de entrada a cualquier tipo de información persistente, siendo el único objeto a través del cual puede accederse a la funcionalidad del modelo. Además, en lugar de guardar información suelta en la sesión, la almacena en objetos business que también son accedidos mediante métodos desde el *ApplicationController*.

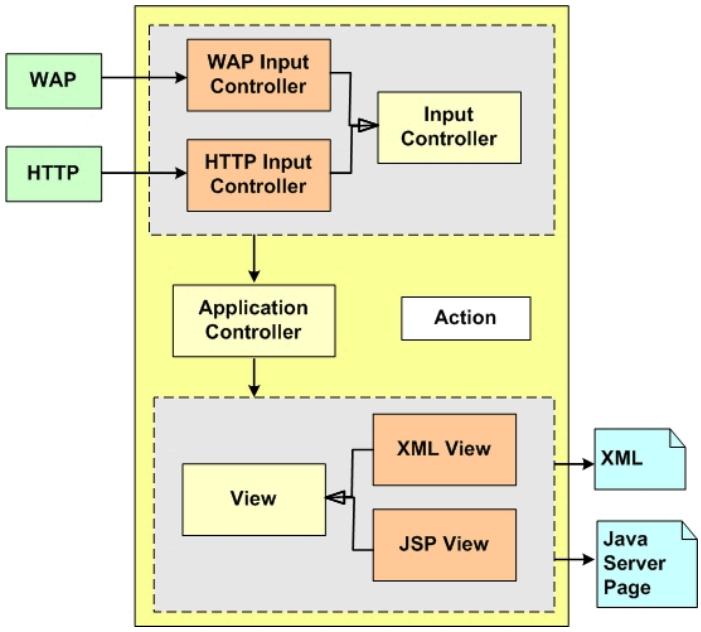


Figura 5.6 – Arquitectura de WebActions

5.2.3.3 Comunicación con el modelo: clase Action

Al recibir un requerimiento http, el *InputController* delega en un *ApplicationController* correspondiente al tipo de acción requerida. El *ApplicationController* selecciona una acción adecuada, dependiendo esta selección, de la entrada de usuario que el *InputController* le ha enviado y del estado actual de la aplicación. Las acciones se representan con objetos que heredan de la clase *Action*, que es una implementación de Command[1]. Cuando un *Action* es ejecutado, recibe al *ApplicationController* actual como parámetro. En el código del *Action* se lleva a cabo la acción requerida sobre el modelo, siempre accediendo a él vía mensajes al *ApplicationController*.

5.2.3.4 Objetos Business: modelo de la aplicación

Estos objetos representan al modelo de la aplicación y son objetos comunes que contienen solamente lógica de negocios. En consecuencia, no conocen de la existencia de los

[1] *Command* en Anexo

demás componentes del framework. El *ApplicationController* es el único que manipula estos objetos.

5.2.3.5 Vista de WebActions

Las vistas se implementan con ServerPages, que pueden acceder al *ApplicationController* y los objetos business. Solamente debería incluirse en estas páginas, código relacionado con la presentación de la información. Los creadores de este framework recomiendan el uso de tags customizados de JSP para remover definitivamente el código java de las páginas.

En la figura 5.6, además de la implementación Server Page que sirve solamente para desplegar la información en un browser, aparece una implementación XML que genera la salida para que la información pueda ser enviada no solo a un browser sino también a un applet, o una aplicación GUI.

5.2.3.6 Funcionamiento del framework: ejecución de una acción sobre la aplicación

El *InputController* recibe un requerimiento http desde el cliente y luego detecta al *ApplicationController* que corresponde para atender este requerimiento. Los *ApplicationController* están almacenados en un objeto sesión. El *InputController* extrae información del requerimiento para enviárselos al *ApplicationController.* El *ApplicationController* es el que determina la acción a invocar. La determinación depende de los datos de entrada enviados por el *InputController* más la información que el *ApplicationController* conoce, como ser, las secuencias de acciones permitidas sobre la aplicación.

El *Action* ejecuta la acción requerida por el usuario sobre la aplicación accediendo a la funcionalidad del modelo a través del *ApplicationController.* El *Action* setea el siguiente "estado" en el *ApplicationController* con un mensaje *setNextPage("un estado").*

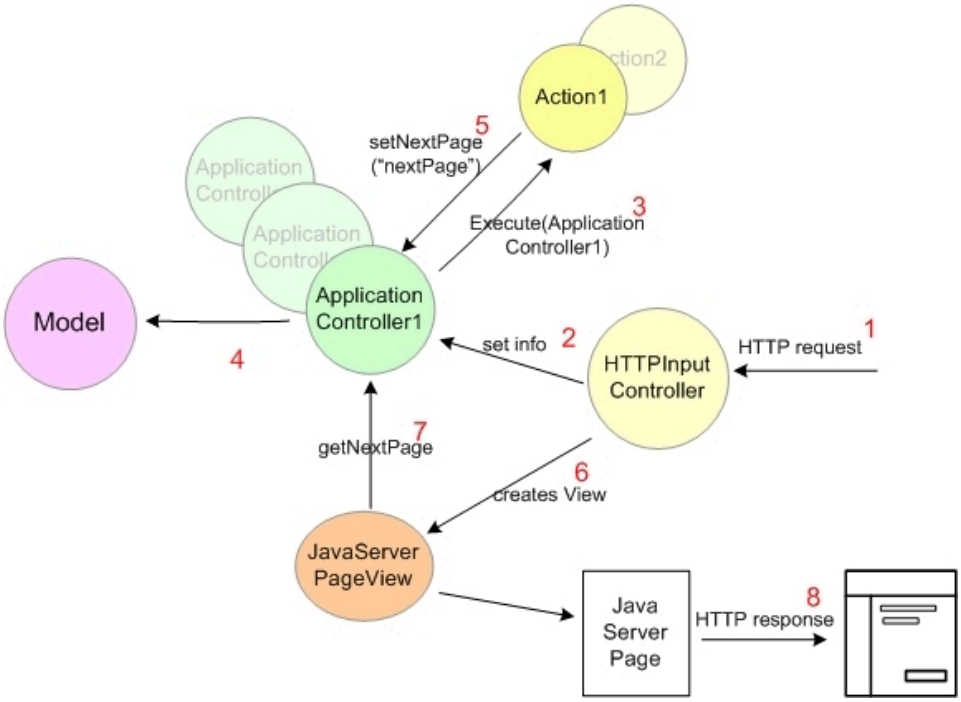


Figura 5.7- Esquema del funcionamiento de WebActions

Para la salida, el *InputController* crea un objeto *View* correspondiente al tipo de requerimiento esto es, *JSP View* en el caso de requerimiento HTTP. El *ApplicationController* contiene el siguiente "estado" al cual debe pasarse y por esta razón, el objeto *View* genera finalmente la salida HTML correspondiente consultando previamente al *ApplicationController* sobre este estado. La salida resultante es enviada al cliente.

5.2.3.7 WebActions y las propiedades de MVC

Este framework no especifica una determinada tecnología para la vista, sino que define una clase View de la cual cuelgan vistas específicas. Por lo tanto, la independencia de la vista con respecto al modelo o a los objetos que ella accede dependerá de las tecnologías con que la misma sea implementada (5.1.3).

El InputController procesa el requerimiento http e invoca a un ApplicationController. Luego el ApplicationController y el Action correspondiente se encargan del procesamiento de la información y el objeto View en el framework se dedica a la presentación (5.1.4).

WebActions implementa un command. De alguna manera, un ApplicationController delega en un objeto Action para que este realice las llamadas a la funcionalidad del modelo (5.1.5). Los Actions son commands.

No tiene sentido el análisis de las propiedades 5.1.8, 5.1.9, 5.1.10 y 5.1.12 en el contexto de este framework, ya que este no provee una implementación para la vista. La posibilidad de portar una interfaz específica a otra de otro tipo (5.1.13) depende de si el programador las implementa. El framework solo sugiere subclasear la clase View con un objeto por cada tipo de vista que la aplicación tenga y por lo tanto existen distintas interfaces para distintos tipos de acceso y no una sola que sea adaptable (5.1.14, 5.1.15). En resumen, no existen impedimentos para implementar estas características relacionadas con la vista en una aplicación diseñada con la arquitectura.

Los ApplicationController son los que deciden qué Action será invocada. Los Actions setean la siguiente View a desplegar y el ApplicationController completa el trabajo generando la vista específica correspondiente al tipo de requerimiento de entrada (html, xml, etc). Pero el rol de control de la aplicación está contenido en varios ApplicationControllers y no en un único objeto (5.2.11). El Action no está ligado al tipo de salida que se devuelve al usuario ni tampoco a la entrada, pudiendo ser invocada desde distintos tipos de interfaces.

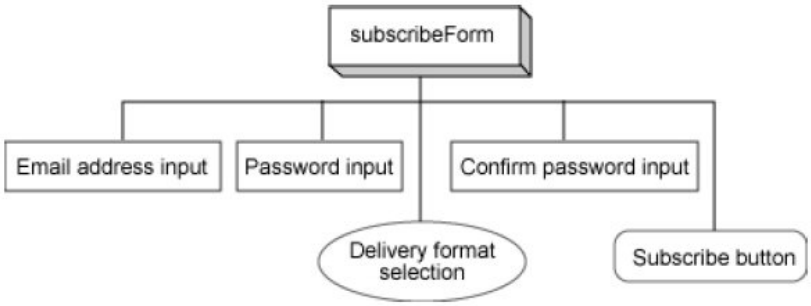
5.2.4 Java Server Faces framework

Java Server Faces (JSF) es un framework basado en MVC para construir interfaces Web en Java. JSF provee básicamente un servlet controlador de la aplicación, al igual que Struts, y ofrece un modelo de componentes para construir interfaces junto con manejo de eventos como lo hace Swing.

5.2.4.1 Componentes del framework

El framework provee un conjunto de *componentes UI* reusables para construir una vista o página JSF. Estos componentes se anidan uno dentro de otro formando un árbol de componentes. JSF implementa el *manejo de eventos* como lo hace Swing, definiendo interfaces que un objeto debe implementar para ser invocado al producirse un evento.

Los objetos *Action* que se asocien a un componente, en general a un botón, son los que contienen algún tipo de procesamiento en el servidor, invocando a la funcionalidad del modelo, y por último seleccionan y retornan un string que representa el siguiente "estado" al cual pasar. Un archivo de configuración *faces-config.xml* contiene asociaciones entre strings y páginas JSF. El objeto *NavigationHandler* toma un string y utiliza el archivo de navegación para encontrar la página específica que debe retornarse al cliente. Estas asociaciones, similares a los forward de Struts, se denominan "*reglas de navegación*" en JSF. También puede asociarse directamente un string en lugar de una acción en el caso de que no se requiera procesamiento previo al pasaje a la siguiente página.



5.8 Ejemplo de árbol de componentes.

Los componentes pueden tener asociados objetos que validen el valor que se asignó o va a ser asignado a los componentes. Estos objetos deben implementar alguna de las interfaces que el framework provee para este fin. JSF permite que un componente sea independiente del objeto que lo convierte a la salida esperada por el cliente. Por ejemplo, un objeto *HTMLRenderer*

asociado a un componente lo convierte a código HTML para ser enviado a un browser. Es posible asociar otro *Renderer* a ese mismo componente para que la salida sea diferente.

5.2.4.2 Tags JSF

Las vistas se construyen escribiendo páginas que usan los tags JSF. Los tags JSF representan componentes y además permiten asociarlos a acciones, manejadores de eventos y métodos para validar.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>


    <h:form  formName="imageMapForm" >

    <h:command_button id="US" label="English" commandName="English"
      action="success">
            <f:action_listener type="net.jackwind.jsf.LocaleEventHandler" />
    </h:command_button>

    <h:command_button id="CN" label="####"  commandName="Chinese"
      action="success">
            <f:action_listener type="net.jackwind.jsf.LocaleEventHandler" />
    </h:command_button>

    </h:form>
```

En el ejemplo anterior se importan las clases JSF necesarias y se define un formulario con el tag *form*. El tag *command_button* representa a un componente *button*. El tag *action_listener* indica la forma en que puede asociarse un objeto *listener*. Cuando el usuario hace click sobre el botón, envía datos en el formulario al servidor, el cual tomará esta información e invocará a los *listeners* correspondientes. El valor del atributo *action* del tag *command_button* indica que al hacer click en el botón se devuelve la página "success" al cliente. Este es un pase directo a una página en particular. Más adelante vemos cómo realizar algún procesamiento previo al direccionamiento a una página.

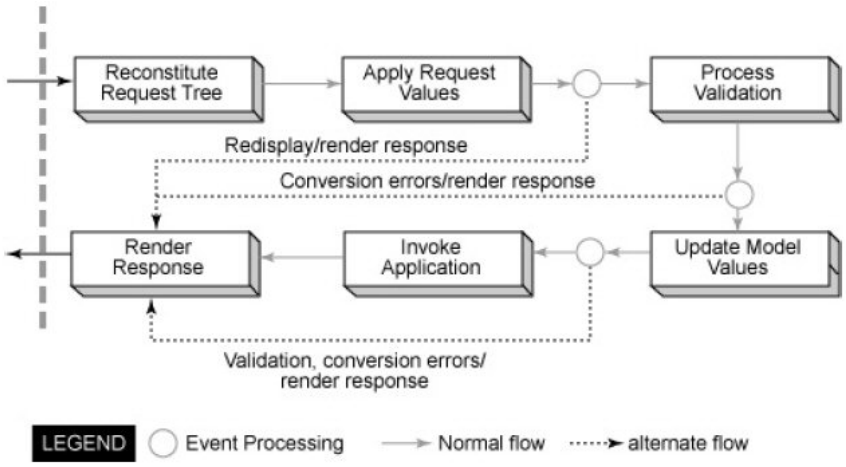
5.2.4.3 El modelo de una aplicación JSF

El modelo en las aplicaciones JSF son objetos JavaBean al igual que en cualquier aplicación Java. Los Beans pueden crearse directamente en la página JSF o se le puede pedir al framework que se encargue de generarlos y administrarlos. Para utilizar el segundo método, los beans se listan en el archivo *faces-config.xml* especificando para cada uno un descriptor (como se lo llama en la página que lo usa), su clase, su alcance, propiedades y también valores iniciales para las mismas.

5.2.4.4 Funcionamiento del framework

Supongamos que el usuario completa datos en un formulario y luego hace un click sobre un botón o un link lo cual genera un requerimiento al servidor. El framework recibe el request y reconstruye la página enviada armando el árbol de componentes de la página JSF. Luego conecta manejadores de eventos y métodos de validación a los componentes según corresponda (etapa 1).

Luego cada componente actualiza su valor con la información que viene en el request http (etapa 2). Una vez que todos los componentes han sido actualizados, lo siguiente es aplicar los métodos de validación asociados a los componentes sobre el valor actual del componente (etapa 3). Si ocurre que un valor no es válido entonces el framework devuelve esta misma página al cliente con un mensaje de error agregado.



5.9 - Ciclo de un requerimiento de página JSF

JSF ofrece la posibilidad de ligar directamente ciertos componentes de la interfaz con partes del modelo mediante una expresión *valueRef*. Por ejemplo:

<faces:input_text id=”nombre” *valueRef*=”alumno.nombre”>

La propiedad *nombre* del modelo, en este caso *alumno*, está asociada con el valor del componente que se genera a partir del tag anterior. Así, todos los componentes de la página actual que tengan asociado un *valueRef* modificarán a su modelo correspondiente. De la misma manera, si existen errores de conversión de tipos, el framework reenvía la misma página al cliente.

A continuación JSF maneja el evento click originado al comienzo del ciclo, para lo cual el framework ejecuta la acción que ha sido especificada en el tag correspondiente al botón o link mediante el atributo *actionRef*. Las acciones son las que interactúan con el modelo y su funcionalidad devolviendo finalmente un string que representa el siguiente “estado” o página a enviar al cliente. El objeto *NavigationHandler* toma este string y busca en el archivo de configuración una regla de navegación de la página actual. La página resultado especificada en este archivo que corresponda al string es la que se envía al cliente. En este último paso, el framework convierte el árbol de componentes de la página en la salida adecuada para devolverla en el response http.

En la figura 5.9 aparecen tres lugares donde se manejan eventos a lo largo del ciclo de un requerimiento. Por ejemplo, luego de que los componentes son actualizados con los datos que vienen en el request, si se producen eventos y existen *listeners* asociados al evento, éstos serán invocados por el framework. Los eventos son de dos tipos en JSF: eventos *value_changed* y eventos *action*. Los eventos *value_changed* se generan cuando cambia el valor de un componente, los eventos *action* se producen cuando se hace un click sobre un botón o link.

5.2.4.5 JavaServer Faces y las propiedades de MVC

La especificación de JavaServer Pages no designa una fase en el ciclo del requerimiento en la cual los componentes de la interfaz recuperen la información a mostrar del modelo. Los componentes poseen un valor local que debe ser actualizado si el modelo de la aplicación varía. No esta claro en la especificación quien debe realizar esta tarea. Las vistas son JSP y por ser este un Pull Model se deduce que la recuperación de la información desde el modelo es directa

ligando tags JSF con objetos del modelo (5.1.3). Notemos que en JSF la creación de objetos en el servidor por parte del framework depende puramente de los tags que se incluyen en la página jsf, como es el caso de asociar listeners a un componente.

El comportamiento del controlador se encuentra dentro de los objetos Action y en los listeners que se asocian a los componentes de la vista. La separación de roles entre vista y controlador está definida (5.1.4).

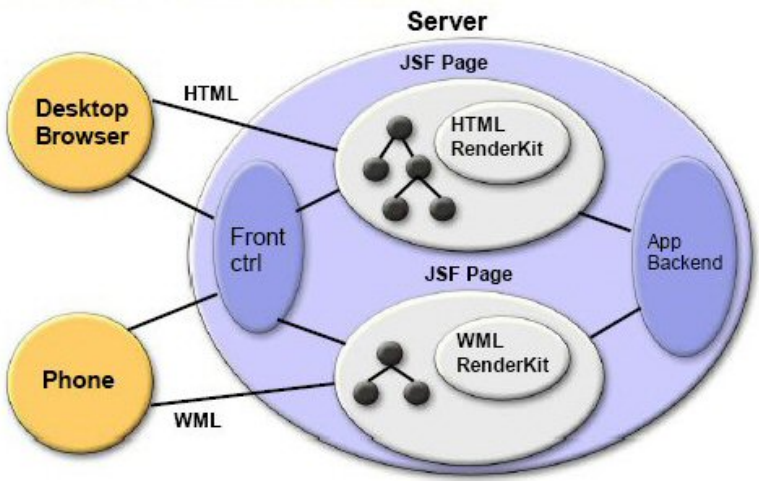
JSF implementa un command donde el controlador es el controlador central del framework y los objetos Action son los objetos command que se comunican con el modelo (5.1.5). JSF provee un command más elaborado ya que previamente a la etapa del ciclo donde los Actions son invocados, pueden llamarse a otros objetos listeners asociados al componente que generó el evento. O sea que no solo un objeto resulta invocado por el controlador central.

JSF provee la posibilidad de anidar vistas (5.1.9) y de asociar un controlador a cada una de ellas cuando sea necesario tal cual lo hace Swing. La capacidad de programar cada componente es uno de los objetivos de JSF. Las vistas se representan en el servidor como un árbol de componentes (5.1.8) y cada componente puede tener asociado un controlador o varios controladores (listeners). Además, JSF permite ligar un componente con un aspecto específico del modelo e incluso, la especificación designa una etapa donde el valor del componente es asignado al aspecto del modelo en forma automática. Por lo tanto, JSF implementa este concepto de "MVC a nivel componente en la web" (5.1.10) como en Smalltalk, en cuanto a que es posible tener un componente vista asociado a un controlador y a un modelo y que la interacción sea "independiente" de los objetos que los contienen.

El encargado de manejar el flujo de la aplicación es el objeto NavigationHandler que utiliza el archivo de reglas de navegación faces-config.xml para determinar dado un string, la siguiente vista a enviar en el response. El objeto Action especifica un string y el NavigationHandler es responsable de direccionar a una página en particular correspondiente al string indicado (5.1.11).

Los componentes UI de JSF son independientes de la forma en que se muestran en el cliente (5.1.14). El objeto Renderer asociado al componente es el que determina el lenguaje de marcado usado para mostrar el componente del lado del cliente. Se pueden conectar distintos

Renderers a un componente UI provisto por JSF, lo que hace posible que una aplicación soporte interfaces en HTML, XML, XML etc simultáneamente (5.1.13).



5.10 – Arquitectura JSF

5.1.15 no es soportada porque no es posible programar una sola interfaz cuyos componentes se rendericen automáticamente de acuerdo al tipo de entrada del requerimiento. Los componentes son reusables, pero los renderers deben asociarse explícitamente.

La situación con respecto a la propiedad de reuso de la estructura de las vistas (5.1.12) se repite como en JSP. No olvidemos que JSF pages son Java Server Pages con nuevos tags incluidos.

5.2.5 Una implementación de Push MVC: tecnología Enhydra XMLC

Enhydra XMLC es una tecnología para la construcción de la presentación de una aplicación web. Su característica más significativa es que, a diferencia de su competidor JSP, Enhydra permite separar definitivamente el código HTML de la lógica de aplicación y de todo código generador de contenido[1].

[1] Ver push MVC, Cap. 4, sección 4.5.2.2.

5.2.5.1 XMLC: generador de plantillas

XMLC es un generador de plantillas para contenido dinámico. Provee un mecanismo para crear contenido dinámico desde plantillas de documentos HTML y XML. Este compilador básicamente convierte una página HTML/XML a una clase Java. Las páginas "convertidas" o compiladas quedan representadas (usando el DOM, Document Object Model) por una clase que es modificada por la aplicación para crear la página HTML dinámicamente.

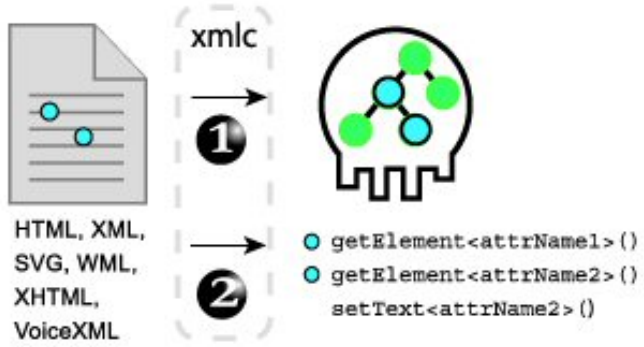


Figura 5.11 - XMLC crea un DOM y metodos setText/getElement nombrados de acuerdo a los id's encontrados durante la compilación

5.2.5.2 Composición y funcionamiento de XMLC

XMLC está compuesto por un compilador y un ambiente de ejecución para el desarrollo de la presentación de las aplicaciones.

En la "fase de pre-compilación", el diseñador y desarrollador identifican las áreas de contenido dinámico que una página contiene y les asignan nombres que se utilizan luego como los atributos ID de los tags HTML/XML; por ejemplo <td id="nombreAlumno">. Los atributos ID permiten identificar a los elementos que deben ser generados dinámicamente o cuyo contenido debe recuperarse del modelo cada vez que la página es requerida nuevamente.

Un ejemplo sencillo es el siguiente:

```
<html>
        <body>
                Hi there, the local time is <span id="time">now</span>.
        </body>
</html>
```

Una vez identificadas las áreas de contenido dinámico, y ya en la etapa de compilación, el compilador XMLC genera una clase Java para finalmente construir una representación DOM de la página original.

5.2.5.3 Document Object Model

El "Modelo de Objetos de Documento" es una interfaz que permite acceder a elementos de un documento HTML vía programación esto es mediante código Java, en el caso de Enhydra. DOM define una representación en objetos de los elementos individuales y el contenido del documento, con métodos para obtener y asignar los valores de las propiedades de dichos objetos. También proporciona métodos para agregar y eliminar elementos del documento.

De esta manera, una página HTML queda representada como un árbol de elementos, donde cada uno debe contener sus propios métodos y propiedades. Los elementos implementan una interfaz Node que contiene un conjunto de métodos y propiedades relacionadas con la estructura de árbol de la página. Ejemplos de estos métodos son los de agregar un nodo, borrar un nodo, insertar, obtener un nodo, hijos de un nodo, etc. Además es posible obtener un nodo dado su ID.

A continuación mostramos la clase generada por XMLC a partir del documento HTML:

```
public class HelloHTML extends org.enhydra.xml.xmlc.html.HTMLObject
  implements org.w3c.dom.html.HTMLDocument {

  SpanElement getElementTime() {
    return getElementByName("time");
  }
```

```
   void setTextTime(String time) {
      ... // código DOM para reemplazar el texto en el tag span con 'time'
   }

    ... // todo el código necesario para implementar DOM
}
```

XMLC implementa esta especificación para separar el código Java del código HTML. La clase Java (el template) que representa la página del ejemplo anterior contendrá un atributo con nombre *Time* que podrá ser accedido mediante el mensaje *getElementTime()* y modificado con el mensaje *setTextTime(unValor)*.

Luego, estas clases Documento generadas por el framework son utilizadas para asignar valores dinámicos y luego enviar la página al cliente, para lo cual será necesario que el template sea traducido al formato correspondiente de salida, HTML en este caso. La manipulación de los objetos generados también se realiza usando la API DOM. El ejemplo siguiente muestra una clase que crea un objeto de la clase *HelloHTML*:

```
public class Hello implements HttpPresentation {
   public void run(HttpPresentationComms comms) {
      HelloHTML page = new HelloHTML();
      String now = (new java.util.Date()).toString();
      page.setTextTime(now);
      comms.response.writeHTML(page);
   }
}
```

El método *run* asigna un valor al componente de texto *Time* y luego el documento es enviado al cliente en el formato HTML como se observa en la ultima línea del método.

Este procedimiento es compatible con todo tipo de documento XML, incluyendo WML. Por lo tanto se soportan distintos clientes con el mismo tipo de contenido dinámico simplemente reemplazando las fuentes HTML con páginas WML.

5.2.5.4 XMLC como tecnología para la vista de la aplicación

Las ventajas de utilizar esta tecnología para construir las páginas de la aplicación son:

- no se agregan tags que no sean los HTML típicos
- el diseñador de páginas no utiliza código Java. Solamente es necesario identificar los elementos con atributos ID.
- el diseñador puede cambiar el diseño de las páginas sin depender del programador de la aplicación, ya que el compilador recompila automáticamente las páginas modificadas.

Si bien esta tecnología solamente se utiliza para la construcción de vistas en una aplicación web, beneficia al diseño de MVC permitiendo desacoplar el contenido y el formato de la vista. XMLC puede integrarse con otro framework como Struts que cumpla el rol de controlador de la aplicación.

5.2.5.5 XMLC y las propiedades de MVC

La independencia de la vista con respecto al modelo (5.1.3) es una consecuencia directa de utilizar XMLC para construir la presentación de una aplicación web, ya que la vista no se comunica con el modelo para recuperar la información que debe mostrar independizándose totalmente del modelo. Los componentes identificados como de contenido dinámico no sufren variaciones si la interfaz del modelo cambia porque la asignación de valores se encuentra fuera de la página o vista. Esta asignación de valores sin embargo deberá ser realizada posiblemente en el código de algún controlador u otro objeto que decida cual es la próxima vista a enviar al usuario. Como consecuencia, la separación de responsabilidades de la vista y del controlador no es clara (5.1.4) ya que la asignación de valores suele realizarse dentro de un controlador que decide cual es la siguiente vista. La lógica de presentación se esparce en los controladores. Esto afecta a la reusabilidad de los controladores y a la conexión de distintos controladores para una vista.

Hemos mencionado que XMLC es una tecnología para la programación de la vista y que entonces debe estar integrada con otro framework que cumpla el rol de controlador. Asimismo, dado que XMLC propone páginas con HTML exclusivamente y una interfaz de acceso a los

componentes de la vista, la implementación de un *command* que independice al controlador del modelo (5.1.5) no es una característica a analizar en este diseño.

Los templates generados a partir de las páginas HTML, contienen un árbol de elementos que luego serán renderizados para enviarse al cliente. Un ejemplo es una tabla con filas donde cada fila contiene columnas en su interior. Una página HTML está representada en el servidor (5.1.8) por un objeto documento raíz y todos los elementos parte del documento se obtienen y asignan en forma de árbol. De aquí que XMLC conserva el anidamiento de componentes de las vistas (5.1.9). Sin embargo, el framework no permite asociar naturalmente un controlador a cada componente. De esta situación se deduce que no hay MVC a nivel componente (5.1.10).

La forma en que se representa a la vista en el servidor presenta diferencias con la manera en que lo hace JSF. XMLC genera una representación de las partes de la página HTML pero no asocia eventos a estos componentes ni los invoca, ya que no es una tarea propia de una tecnología pura de presentación.

Se concluye que el beneficio de quitar la lógica de presentación de las páginas influye sobre las propiedades que requieren de una relación fuerte entre componente visual y controlador. La vista se representa en forma de árbol en el servidor pero no hay eventos asociados ni controladores propios de un nodo del árbol de componentes.

La implementación de un controlador de aplicación depende de la tecnología utilizada para cumplir el rol de controlador de la aplicación (5.1.11).

Enhydra XMLC no incluye características avanzadas para la vista porque su objetivo es simplificar la programación de la vista pero con la idea de mantener solamente HTML o XML estándar. Por otro lado, solamente genera templates para manipular el contenido de una página del lado del servidor sin tener en cuenta propiedades compartidas entre varias páginas de la aplicación como puede ser la disposición de los componentes en las mismas (5.1.12).

Tampoco es posible reusar los componentes que forman la vista porque un componente solo puede renderizarse a un lenguaje de marcado en particular (5.1.14). Sin embargo, XMLC no solo parsea código HTML, sino XML, WML generando templates para cada tipo de vista (5.1.13). La propiedad 5.1.15 no es soportada por el framework.

5.2.6 ASP.NET WebForm Pages Framework

Este framework permite crear WebForms para construir las páginas Web de una aplicación Web y además permite codificar la interacción del usuario con estas páginas. El framework fue construido sobre el ASP.NET framework que corre en un servidor para generar y administrar estos webforms. A su vez, ASP.NET utiliza al .NET framework. Básicamente, un WebForm es capaz de generar automáticamente código HTML, o sea, se convierte en una página HTML que se envía a un browser. Los lenguajes con los que se puede programar la lógica de interacción de la interfaz son Visual Basic .net, C# y J#.

5.2.6.1 Las partes de un WebForm

Un WebForm se compone de dos partes: una parte visual que se guarda en un archivo, y una parte lógica que se guarda en otro archivo separado. La parte visual se construye con componentes "Server Controls" y también con tags HTML estándar. La parte lógica, también llamada "código-de-atrás" del Webform, contiene código que representa comportamiento de interacción con la interfaz.

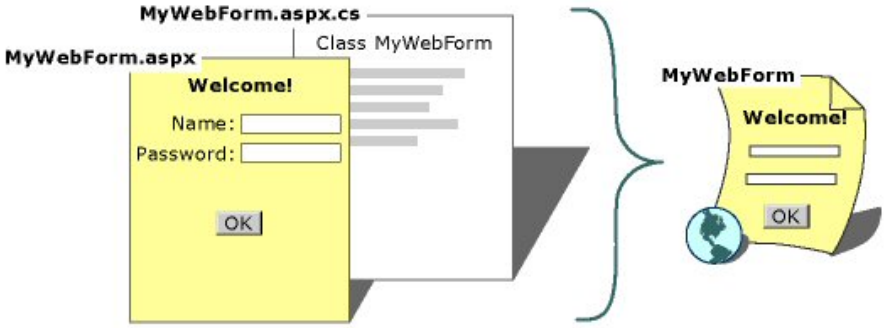


Figura 5.12- Ambos archivos forman un solo WebForm.

La clase de la página de atrás (.cs en la figura 5.12) se crea como subclase de la clase System.Web.UI.Page. Todas las clases de las páginas de atrás de una aplicación se compilan en un solo archivo. De esta clase compilada se deriva el archivo .aspx correspondiente.

Cuando un usuario pide una página al servidor a través de un browser, el WebForm dinámicamente produce la salida HTML a enviarle.

5.2.6.2 Contribución del framework: interfaces web elaboradas

El aporte fundamental de este framework es que la programación de la interfaz de la aplicación Web es muy similar a programar una interfaz GUI. Este objetivo es posible ya que, la parte visual de un Webform se construye ensamblando componentes[1] y seteando propiedades en ellos, en lugar de asignar valor a atributos en tags HTML. Esto es posible porque estos componentes se manipulan del lado del servidor y luego se convierten automáticamente a elementos HTML. Además, se programa la respuesta a eventos generados por el framework del lado del servidor. Como resultado nos abstraemos de la separación entre cliente y servidor programando la interfaz como si esta separación no existiese.

Los *ServerControls* en un WebForm son "abstracciones" del contenido de una página HTML y de la interacción entre browser y servidor. Los *manejadores de eventos* son métodos que reciben un evento y un generador del evento como parámetros y realizan diversas acciones, por ejemplo, llamadas a la funcionalidad de la aplicación.

El framework ASP.NET se encarga de capturar el evento http desde el cliente y entregárselo al servidor junto con la información del evento. El framework interpreta el mensaje, genera un evento GUI e invoca al manejador asociado a este evento. Asimismo, se encarga de mantener el estado de una página y sus controles.

5.2.6.3 Componentes y programación de la Vista: Server Controls, Eventos y Manejadores de Eventos

Los *ServerControls* **componen la Vista** y son objetos que generan, casi todos, el evento click. Los eventos como mouseOver que se producen con mucha frecuencia no son generados por los ServerControls para no generar tantas llamadas al servidor.

[1] Esta facilidad se debe en su mayor parte a las herramientas como Visual Studio .net que facilitan la construcción de los webforms con "drag and drop" de componentes en lugar de programar directamente la página aspx.

Un manejador de evento está contenido en la página de atrás del WebForm y está asociado a un evento determinado generado por algún componente de la interfaz. Cuando un manejador es invocado recibe como parámetros al objeto origen del evento y un evento con información sobre el mismo. Los manejadores se ligan a un evento a través un objeto "delegate" que contiene la lista de manejadores para un evento. Lo anterior es una implementación del patrón "observer" donde los manejadores dependen del componente al que observan y responden recibiendo al evento como parámetro. Estos manejadores cumplen la función de **controladores** ya que reciben un evento de la interfaz y manipulan al modelo, de acuerdo a la definición original de controlador de MVC.

Los eventos también pueden ser generados por la aplicación, un ejemplo es *BeginRequest* que se genera cada vez que se pide una página de la aplicación. El objeto session también genera eventos como SessionStart, SessionEnd. Registrar manejadores a estos eventos da la posibilidad de realizar alguna tarea de inicialización y finalización.

El framework provee dos tipos de *ServerControls*. Ambos tipos generan eventos para registrar manejadores y mantienen su estado cuando se hace postback[1]: por ejemplo si tenemos un *Textbox* en el cual se ingresó algún texto, el control contendrá ese valor cuando la página se reenvíe al cliente. Por último, saben convertirse a código HTML para enviarse al cliente.

HTML Server Controls: son elementos HTML programables en el servidor que se convierten a un elemento HTML. Cada HTML Server Control mapea a un elemento HTML conocido, y sus propiedades coinciden con los atributos del tag HTML, permitiendo manipular los atributos como propiedades en el servidor.

Web Server Controls: son controles que no mapean con un elemento HTML único y entonces pueden convertirse a distintos elementos HTML al enviarse la página al cliente. Por eso las propiedades de un Web Control no son uno a uno con los atributos que tiene el tag HTML al que finalmente se convierten. Contienen lógica más elaborada, pueden actuar como "contenedores" y propagar eventos además de detectar un browser en particular y crear una salida HTML acorde a éste.

[1] Es la situación en la que una página que ya ha sido requerida y enviada al cliente, es devuelta al servidor posiblemente con datos ingresados por el usuario.

5.2.6.4 Funcionamiento del framework: ciclo de vida de un WebForm

Como aquí toda la "lógica" se procesa en el servidor, cada acción de usuario sobre la interfaz genera un pedido al servidor, requiere de algún procesamiento en el mismo y termina con la devolución de una página al cliente.

Cuando el usuario accede a una página por primera vez, el framework toma el pedido, creando un objeto Page correspondiente y a todos los componentes (controls) que lo componen. El siguiente paso es invocar a un metodo *onload* del objeto Page, en el cual el programador asigna valores a los componentes visuales para que sean mostrados en el cliente. Luego de la inicialización cada componente se convierte en salida HTML adecuada resultando en la página HTML que es enviada al cliente.

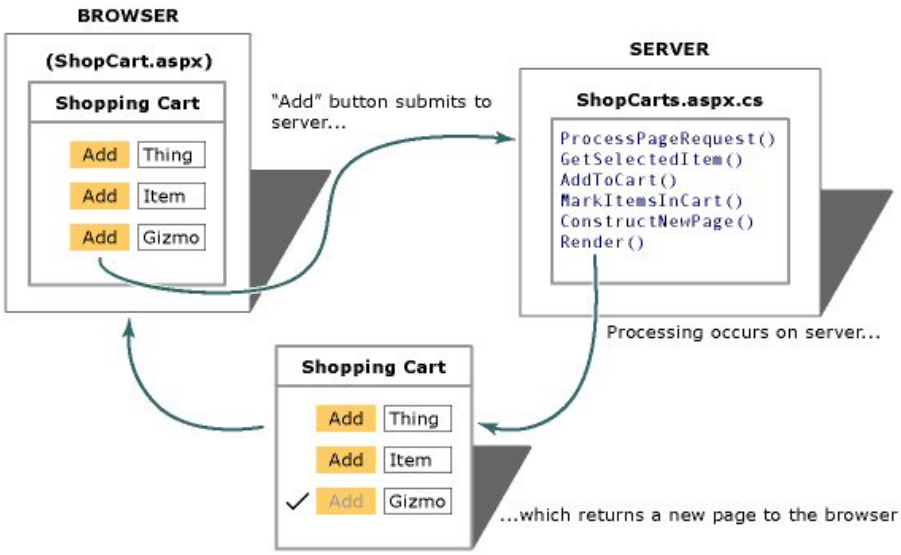


Figura 5.13 – Ejemplo de ciclo

Supongamos que el usuario completó algunos datos en un formulario y hace un click sobre un botón de la interfaz, lo cual genera un pedido al servidor. El framework toma el pedido junto con toda la información que necesita desde el cliente, vuelve a generar el objeto Page, esta vez *detectando que es un postback,* crea los objetos ServerControls y setea sus propiedades con

los valores ingresados en el formulario. Entre otras acciones, genera un evento de la clase correspondiente (clase del framework) e invoca a los manejadores de este evento. El código del manejador podría tomar los valores de los *ServerControls* y realizar alguna acción con ellos, guardándolos en el modelo. También podría asignarse un valor nuevo a algún *ServerControl*. Una vez ejecutado el código del manejador, el objeto Page se convierte en salida HTML para devolver la página al cliente.

Las ventajas con respecto a un ciclo común en la Web son:

- Se guarda el "view state" de los *ServerControls*: cuando se envía información dentro de un control, el framework mantiene estos valores para que más tarde puedan reenviarse al cliente automáticamente, esto es, sin tener que programarlo.
- Detección de si una página es pedida por primera vez o no (postback), lo cual posibilita programar acorde a la situación.

5.2.6.4 ASP.NET WebForms y las propiedades de MVC

Los componentes de un WebForm no recuperan la información del modelo de la misma manera que lo hacen los tags JSP. La asignación de valores a los componentes la realiza el programador en la página de atrás del WebForm. Por lo tanto, aun si la interfaz del modelo cambia, la vista no resulta afectada, sino la página de atrás (5.1.3).

Cuando un componente visual genera un evento, el framework invoca a los manejadores asociados al mismo. El comportamiento está entonces encapsulado en estos métodos que son los controladores. Por lo tanto puede cambiarse un controlador por otro y cambiar el tipo de interacción con el componente visual, o ServerControl (5.1.4). Sin embargo, se asignan valores a los componentes vista desde estos mismos métodos, incluyéndose así parte de la lógica de presentación en los controladores.

Los manejadores de eventos invocados por el framework son métodos de un page y no objetos (5.1.5). El framework busca el método adecuado, lo invoca y delega en él. El diseño no incluye a un controlador central que delega en objetos Action para acceder al modelo de la aplicación.

El flujo de control de la aplicación en este diseño se esparce entre los distintos manejadores de eventos. El diseño de clases del framework no separa los roles del flujo de la aplicación en un objeto especial como lo es el ActionServlet de Struts, sino que el programador mismo direcciona a una página aspx específica, generalmente como última acción del método manejador de evento. Por lo tanto, el flujo de la aplicación no está unificado (5.1.11) y tampoco separado del comportamiento de los controladores.

Cuando el framework recibe un requerimiento http crea un objeto Page en el servidor y luego lo utiliza para devolver una página al cliente. Un Page es la vista del MVC representada en el servidor (5.1.8), donde cada componente que forma la vista es capaz de generar eventos que no son simples eventos http. Un Page está compuesto por una jerarquía de componentes ServerControls. En otras palabras, una vista está formada por otras vistas anidadas (5.1.9) a las cuales se asocian controladores (manejadores de eventos). Aquí, el concepto de "MVC a nivel componente" (5.1.10) surge gracias a la implementación de vista en el servidor, a la estructura de la misma compuesta de ServerControls y a la posibilidad de asociar controladores a cada uno de ellos. Además, los controles poseen una propiedad Databinding cuyo valor es asignado con un aspecto del modelo, por ejemplo databinding= objeto.getData(), lo cual es una asociación del componente con su modelo para poder mostrar este valor al momento de la renderización. Sin embargo a diferencia de JSF, si un usuario escribe un valor para el componente y envía la página de vuelta al servidor, ASP.NET no asigna este valor de entrada al modelo, sino que esta acción debe ser programada explícitamente.

WebForms soporta la propiedad 5.1.12 de varias formas: creando una clase que hereda de *Page* y que contenga controles pre-cargados, creando un *ServerControl* que cumpla la función de plantilla especificando "áreas" para cada página y usando este *ServerControl* en todas las paginas que tengan la misma estructura, y por ultimo, valiéndose de *UserControls* para áreas comunes tales como menús, encabezados. Cabe notar que en el último caso, habrá que ubicar el *UserControl* en cada página, lo cual resulta arriesgado porque es muy fácil modificar la estructura de la página por separado.

Los HTMLControls se convierten en código HTML. Para construir una interfaz que se envíe al cliente en otro formato, por ejemplo en XML o WML, el framework provee controles que se convierten a este formato. Entonces es posible tener dos interfaces distintas en la aplicación web (5.1.13) para ambos formatos de salida. El ASP.NET MobileControls de

ASP.NET 1.1 [1] incluye controles separados para construir páginas web adecuadas para browsers pequeños. Entonces los controles que generan HTML son distintos de los que generan WML. No hay reusabilidad de los componentes porque no son independientes de la salida que generan (5.1.14). En consecuencia tampoco (5.1.15) es soportada en este framework.

5.3 Conclusiones del análisis de propiedades MVC en la Web

5.3.1 Independencia de la vista con respecto al modelo

Tenemos *independencia de la vista con respecto al modelo* cuando la vista no debe ser modificada si la interfaz del modelo cambia. El problema de la vista dependiente del modelo surge comúnmente cuando la información que la vista despliega es obtenida directamente por ella vía la interfaz que el modelo exporta.

Esta propiedad es incluida en Enhydra XMLC ya que la vista no contiene código alguno de recuperación de información del modelo. En ASP.NET la vista tampoco recupera la información de su modelo sino que los valores a cada componente son asignados en la página de atrás de un WebForm.

En JSP, la implementación de esta característica depende casi siempre del programador. Struts es el único que provee objetos *ActionForm*, los cuales pueden ser utilizados para recuperar información del modelo siendo estos objetos los que se adaptan a los cambios del modelo y no la propia vista. Los *ActionForms* deberían usarse aún para almacenar información de solo lectura y no solamente para enviar formularios al servidor. Los restantes frameworks utilizan JavaBeans que generalmente son el modelo directo.

Como los diseñadores de páginas web prefieren trabajar únicamente con HTML, se intenta quitar de las vistas el código que obtiene datos del modelo. Sin embargo, del análisis previo puede deducirse que la mejor solución, también en la web, es recuperar la información directamente en la vista como en el MVC no web para no esparcir lógica de presentación en algún otro objeto (controladores) de la aplicación. Pero para poder incluir la independencia de

[1] Es la ultima versión disponible actualmente

vistas, tenemos que disponer de un objeto como el ValueModel de Smalltalk que permita la no propagación a la vista de los cambios en el modelo. El mejor ejemplo de este diseño en la web, son los ActionForms de Struts.

5.3.1.1 Otra forma de obtener independencia: separación de contenido y formato en una vista

El concepto de vista independiente del modelo puede implementarse de modo que la vista no solo sea independiente de la interfaz del modelo sino del *contenido* que la vista despliega. Para soportar esta propiedad es necesario separar totalmente la vista en dos partes: contenido y formato del contenido. En JSP denominamos "contenido" al código java que recupera la información del modelo y "formato de contenido" es código HTML puro. En XMLC no existe más código que HTML, y el compilador genera una clase que representa una plantilla de contenido para una determinada página, la cual luego es asignada con valores tomados del modelo real. La plantilla es un objeto que reúne los datos que la vista debe mostrar. Un tercer responsable combina la plantilla y la página HTML para obtener como resultado la vista actual que debe enviarse al cliente. Es importante asegurarse de que la recuperación de datos del modelo este incluida en estos objetos de contenido y no en algún controlador o componente de la aplicación.

La implementación de esta separación contenido-formato debería ser un objeto que represente el contenido de la página y otro que sea la página en sí. El objeto de contenido posee variables para cada dato de la página. De esta manera la vista solo contendría formato de datos.

5.3.2 Separación entre Vista y Controlador: Independencia de la vista con respecto a su controlador

Con la utilización de JSP sin Servlets, el procesamiento se insertaba en las mismas páginas JSP. Al combinar JSP con Servlets las tareas de procesamiento se encuentran en estos últimos objetos, lo cual independiza a la vista de su controlador. Frameworks como ASP.NET y XMLC asignan valores a variables de la vista dentro del controlador, lo cual no beneficia el reuso de controladores. Estos frameworks no cumplen con esta característica. JSF sí implementa este

concepto incluyendo las ventajas de ASP.NET y haciendo que las vistas sigan encargándose de obtener la información del modelo.

	JSP Servlets	Struts	WebActions	JavaServer Faces	XMLC	ASP.NET WebForms
1. Independencia de la vista con respecto al modelo	✗	?	?	✗	✓	✓
2. Separación entre Vista y Controlador	✓	✓	✓	✓	?	?
3. Independencia del Controlador con respecto al Modelo	?	✓	✓	✓	?	✗
4. Representación de Vista en el servidor	✗	✗	?	✓	✓	✓
5. Vistas anidadas, Controladores anidados	✗	✗	?	✓	?	✓
6. MVC a nivel componente en la web	✗	✗	?	✓	✗	✓
7. Controlador de la aplicación	✗	✓	✗	✓	?	✗
8. Reuso de la estructura de las vistas	✗	✗	?	✗	✗	✓
9. Portar interfaces HTML a otras plataformas	✓	✓	?	✓	✓	✓
10. Reusabilidad de componentes visuales	✗	✗	?	✓	✗	✗
11. Interfaz adaptable a quien accede a la aplicación	✗	✗	✗	✗	✗	✗

Tabla 2- Resumen propiedades MVC en la web

Si la vista es capaz de recuperar información del modelo por sí misma, esto es, no necesita de la asignación por parte del controlador, entonces este último contiene solamente código de procesamiento y puede ser reemplazado por otro para posibilitar distintos tipos de interacción con la misma vista. A su vez, si se cuenta con algún objeto como el ActionForm de Struts, de manera que el controlador lo reciba para procesar la entrada, podríamos tener dos vistas diferentes que generen el mismo tipo de ActionForm e invoquen al mismo Action reusando el mismo controlador y soportando distintas vistas (diferentes looks: HTML, WML).

5.3.3 Independencia del controlador con respecto al modelo

La propiedad relacionada con la *implementación de Command* se reduce a implementar objetos *Action*. Struts y WebActions los incluyen en su diseño. JSP- Servlets no provee objetos *Action* pero no impide que el programador pueda crearlos explícitamente separando controlador y acciones. JSF hereda la implementación de *Command* de Struts.

5.3.4 Representación del componente vista en el servidor

JSP no representa a la vista como objeto del lado del servidor. Una página JSP no es capaz de almacenar información a través de sucesivas comunicaciones servidor-cliente, lo cual es uno de los objetivos principales de mantener un objeto vista "vivo" en una aplicación web.

XMLC representa a la vista con un árbol de componentes cuyos nodos pueden ser recorridos de acuerdo a su estructura.

ASP.NET y JSF también representan la interfaz en el servidor con un árbol de componentes: una vista anidada cuyas subvistas tienen asociados uno o más controladores. Además, sus componentes generan eventos GUI y son capaces de "mantener" sus valores a través de sucesivos pedidos de la página, sin necesidad de asignarlos cuando la misma vista es re enviada al cliente.

5.3.5 Vistas anidadas, controladores anidados

A excepción de XMLC que incluye el concepto de vistas anidadas pero no el de controladores anidados, los frameworks cuya vista se implementa con anidamiento de componentes permiten asociar controladores a cada uno de ellos y hasta varios controladores para un mismo componente. *ASP.NET y JSF son buenos ejemplos de cómo implementar esta propiedad. JSP está muy lejos de esta idea al igual que Struts.*

5.3.6 MVC a nivel componente en la Web

Para soportar esta propiedad debe contarse con una representación de la vista en el servidor y debe ser posible asociar los componentes de la vista con manejadores de eventos que interactúen con el modelo. El manejador asociado a un componente es el controlador del mismo. Cabe destacar que es posible asociar varios controladores a un componente porque en realidad los controladores se ligan a los eventos disparados por un componente. ASP.Net soporta esta propiedad y además permite que un componente pueda recuperar un valor directamente del modelo al renderizarse la página que lo contiene. JSF es aún más fuerte en este concepto porque la asociación componente-aspecto del modelo no solo se utiliza al momento de renderizar la vista sino que, si el componente tiene un valor asignado por el usuario en la interfaz, el framework toma este valor y lo asigna al modelo del componente. El framework realiza la modificación del modelo automáticamente tal como lo hace Smalltalk.

Una diferencia importante entre ambos frameworks es cómo se implementan estos controladores. Un controlador a nivel componente en ASP.NET es un método en la página de atrás del webform que contiene a la componente. En JSF, un controlador siempre es un objeto listener instanciado por el framework que responde a un evento generado por una componente. Los controladores de los componentes de una misma página no son métodos en una sola clase, por lo cual los "MVC a nivel componente" están formados por tres objetos modelo, vista y controlador, tal cual el MVC tradicional de Smalltalk.

5.3.7 Controlador de la aplicación

Struts y JSF incluyen un archivo de configuración donde se encapsulan las reglas de navegación de la aplicación y un objeto controlador de la aplicación que utiliza este archivo para controlar el flujo de la misma. *Esta implementación es el mejor modo de encapsular la navegación y flujo de la aplicación en un solo lugar*. WebActions reparte el procesamiento de la entrada y las reglas de navegación en varios objetos ApplicationController. ASP.NET no tiene en cuenta la idea de controlador de la aplicación.

5.3.8 Reuso de la estructura de las vistas

El reuso de la estructura de las vistas puede lograrse en ASP.NET pero no de una forma confiable, ya que esta propiedad no fue uno de los objetivos de la tecnología. Sin embargo, ASP.NET 2.0 incluirá este concepto con "Master Pages" [16]. Una *MasterPage* es una página ASP.NET que provee una plantilla para otras páginas e incluye comportamiento a nivel de página y también funcionalidad. Básicamente, estas páginas definen objetos *ContentPlaceHolders* que identifican el contenido que debe ser sobrescrito por las páginas hijas. La página resultante es la unión de las *MasterPage* y la página que la utiliza. Para sobrescribir, las páginas hijas especifican un control *Content* con el nombre de algún *ContentPlaceHolder* de la *MasterPage.*

Tiles Framework[1] es un framework que se usa con páginas JSP para separar el contenido de una página de su estructura. Por lo tanto *Tiles* puede usarse en *Struts*, *WebActions,* y cualquier otra arquitectura donde JSP sea la tecnología para diseñar la vista.

Ambos frameworks implementan la misma propiedad de formas diferentes.

[1] http://struts.apache.org/userGuide/dev_tiles.html para una descripción detallada de Tiles Framework.

5.3.9 Posibilidad de portar interfaces basadas en HTML a otras plataformas

Si bien algunos frameworks como Struts fueron pensados para incluir solo una interfaz HTML, actualmente todos los frameworks permiten disponer de varias interfaces. JSP ya permite embeber código WML en lugar de HTML. En un nivel mayor de abstracción, ASP.NET incluye esta posibilidad con librerías de componentes HTML, otra librería de componentes que se renderizan a WML por ejemplo para enviarse a dispositivos móviles. JSF también incluye este concepto pero separando los componentes de su objeto renderer.

5.3.10 Reusabilidad de los componentes visuales

De los frameworks expuestos, *JSF es el único que incluye esta idea de separación entre la representación del componente y el componente renderizado vía la asociación de un objeto render adecuado a la salida esperada.* ASP.NET v 2.0 [16] también contendrá esta propiedad proveyendo un único conjunto de controles y una arquitectura extensible para permitirles renderizarse a múltiples tipos de salida. En dicha arquitectura existe un *adapter* para cada tipo de formato de salida (o para cada dispositivo que pueda acceder a la aplicación). Los *adapters* realizan la conversión del componente para ser visualizado por el cliente. Pueden agregarse nuevos *adapters* para soportar nuevas tecnologías o tipos de clientes.

5.3.11 Posibilidad de que una sola interfaz se adapte a quien accede a la aplicación

Para contener una sola interfaz que se renderice de acuerdo a quien accede a la aplicación es necesario que el framework correspondiente se encargue de asociar objetos renderer a los componentes en forma dinámica. JSF no realiza esta tarea pero al menos separa a los componentes del objeto que renderiza lo cual es obligatorio para soportar esta propiedad. La programación de una sola interfaz independiente de la plataforma es uno de los avances de ASP.NET v 2.0 [16] donde los controles y el propio objeto *Page* reconocen el tipo de browser y, valiéndose de la renderización adaptativa (adapters para los controles), generan la salida correcta para ese tipo de browser.

Capítulo 6
Conclusiones

En este último capítulo se exponen primeramente, conclusiones que resultan del análisis realizado en los capítulos anteriores, relacionando conceptos e implementaciones explicados a lo largo del trabajo.

Posteriormente se presentan, como resultado de la experiencia obtenida en esta investigación, algunas conclusiones finales relevantes a criterio del autor y se proponen posibles trabajos futuros relacionados con el tema de esta tesis.

6.1 Sobre las propiedades de MVC

Luego de introducir los conceptos Push y Pull MVC en Cap.4 y una vez proyectadas las propiedades de MVC sobre los frameworks analizados en Cap.5 es interesante distinguir cómo determinados frameworks, dependiendo del tipo de MVC que implementan, incluyen o no ciertas propiedades de MVC.

6.1.1 Push-Pull MVC frameworks: efectos sobre las propiedades de MVC

En Cap. 4 sección 4.3.3 mencionamos que una de las cuestiones propias del ambiente Web que influye sobre el desarrollo de aplicaciones de software, es que la tecnología con la cual se implementa la vista es distinta de la que se utiliza para programar el resto de la aplicación, ya que la vista, en todos los casos, debe ser capaz de renderizarse a HTML para que el usuario pueda visualizar y/o interactuar con la interfaz.

Por otro lado, la vista debe ser capaz de obtener de alguna manera[1], la información del modelo que debe presentar y formatear. Optar por un Push MVC o Pull MVC acorta estas diferencias.

[1] En Cap. 4 sección 4.5.2 se explica esta necesidad.

Un Pull MVC, como JSP, extiende el conjunto de Tags HTML que se convierten a código Java permitiendo recuperar el contenido del modelo desde la misma vista. De esta manera, para independizar a la vista de su modelo, habrá que diseñar objetos intermedios, como los *ActionForm* de Struts, que abstraigan a la vista del modelo real.

Un Push MVC quita definitivamente la lógica de recuperación de datos en la vista y por esta razón, no existe la necesidad de diseñar la recuperación de información para que la vista sea independiente de su modelo.

Concluimos que la elección entre uno de estos modelos influye notablemente sobre el grado de *independencia de la vista con respecto al modelo*[1].

	Push MVC	Pull MVC
Independencia de la vista con respecto al modelo	✔	?
Separación de roles vista y controlador	?	✔

Tabla 3 - Propiedades de MVC proyectadas sobre los meta-frameworks

La implementación de un Push MVC o un Pull MVC también afecta a la *separación de responsabilidades del controlador y la vista.*

Un Pull MVC favorece a una separación clara de responsabilidades entre ambos componentes ya que la vista se encarga de recuperar la información que muestra y por lo tanto, los controladores solo realizan procesamiento de entrada, manipulación del modelo y direccionamiento a una vista.

En un Push MVC, los valores a los componentes que forman parte de la vista, se asignan en otro lugar de la aplicación y no en la vista misma. En consecuencia, esta asignación suele encontrarse esparcida en los controladores, especialmente en aquellos que redireccionan a la vista

[1] La definición de esta característica se encuentra en Cap.2 sección 2.3.3

en cuestión. Una buena práctica posible para esta cuestión es la de crear un objeto encargado de asignar valores a una determinada vista, el cual sea invocado por todos aquellos controladores que direccionen a ella.

6.1.2 Las propiedades de la vista Web

Originalmente se aplicaba MVC con el propósito de que el modelo fuese desconocedor de la o las interfaces de usuario de la aplicación. La interfaz era en general, el componente que se sustituía con mayor frecuencia ante la necesidad de modificar la forma en que el usuario interactuaba con la aplicación. Esta situación se da aun más en el contexto web cuando se intenta soportar distintos tipos de formato de salida.

Si observamos el agregado de propiedades de MVC en Cap.5 sección 5.1, ellas están en su mayoría relacionadas con los cambios que el contexto Web produce en el componente Vista. La vista se convierte en un objeto capaz de materializarse de distintas maneras cuando se presenta al usuario y de abstraerse del contenido que despliega. La vista, como parte de una aplicación MVC, evoluciona y alcanza altos niveles de reusabilidad.

6.2 Las desventajas del MVC original en el contexto Web

Consideremos nuevamente las *desventajas de utilizar MVC* que fueron listadas en el primer capítulo de este trabajo (ver Cap 1. sección 1.7) y transportémoslas al ambiente Web:

6.2.1 Mayor complejidad cuando la aplicación es sencilla

Es prácticamente imposible que MVC vuelva más compleja una aplicación en la web. Las aplicaciones web son complejas por defecto.

6.2.2 Número excesivo de actualizaciones

No es posible que el número de actualizaciones sea excesivo ya que la actualización de la vista no es automática sino por demanda. Al decir por demanda nos referimos a que es la vista la cual inicia una consulta al modelo para verificar si se produjeron cambios o para modificar al mismo, desplegándose nuevamente como segundo paso.

6.2.3 Conexión fuerte entre vista y controlador

La desarticulación de la vista y el controlador es un objetivo que se mantiene en el contexto web. La propiedad de separación entre vista y controlador ha sido analizada en distintos frameworks (Cap.5). A pesar de que la vista y el controlador se encuentran en lugares distantes, es habitual encontrar que las responsabilidades de ambos no estén divididas correctamente.

6.2.4 Acoplamiento estrecho de vistas y controladores a un modelo

Las propiedades analizadas de *independencia de la vista con respecto al modelo* e *independencia del controlador con respecto al modelo* demuestran que es posible "debilitar" este acoplamiento, y por lo tanto, esta desventaja de MVC resulta disminuida.

6.2.5 Ineficiencia de acceso de datos en la vista

Si la vista de la aplicación web se implementa con JSP simple, entonces no hay forma de que la vista cachee datos para no tener que consultar al modelo cada vez que deba mostrarse. Sin embargo, en el caso de un *Rich Client*, como ASP.NET, el mismo framework se encarga de mantener el "viewState" de una vista, es decir, los valores de los componentes de la interfaz. Por lo tanto en este caso, no es necesario asignar valores a los componentes cuando la página es requerida varias veces seguidas.

6.2.6 Cambio inevitable de vista y controlador al portar

"Portar" en el contexto Web se relaciona con la posibilidad de transportar una interfaz HTML a otro tipo de interfaces. A su vez, los controladores deben ser capaces de procesar no solo requerimientos HTTP. Para que la vista pueda conservarse a través de distintas plataformas habrá que considerar incluir propiedades como 5.1.13, 5.1.14[1] las cuales impulsan a no tener que implementar una interfaz para cada tipo de salida que se desee soportar o para que distintos tipos de usuarios puedan interactuar con nuestra aplicación.

6.2.7 Dificultad al utilizar MVC con herramientas modernas para construcción de interfaces de usuario

Hoy en día, los frameworks de última generación son siempre acompañados de una herramienta para asistir en la construcción de una interfaz. Cada vez son más los programadores que aprenden a utilizar un framework mediante dicha herramienta. Esto último es producto de la alta complejidad de los componentes reusables que los frameworks proveen y a que las herramientas ayudan a abstraerse de ciertos aspectos de implementación que no tienen que ver con la funcionalidad básica de la aplicación siendo construida. Por lo tanto, una herramienta para construir una interfaz de usuario afecta considerablemente el diseño de la aplicación que estamos desarrollando. Asimismo, hacer uso de un framework y no de la herramienta se vuelve irrealizable en términos de tiempo y facilidad.

En conclusión, es muy difícil incluir propiedades de MVC que la herramienta no tenga en cuenta.

[1] Explicadas en el Cap.5.

6.3 Conclusiones finales

6.3.1- Puede apreciarse que las desventajas que MVC poseía en los comienzos de su utilización fueron desapareciendo o debilitándose a medida que empezó a ser aplicado en contextos de mayor complejidad, en particular, en el de las aplicaciones web.

6.3.2- Se destaca que este trabajo intenta reafirmar la importancia de contar con un diseño MVC que abarque todos los conceptos inherentes a dicha arquitectura, reforzando la idea de que un buen diseño MVC es el resultante de incluir todas sus propiedades.

6.3.3- El autor observa que determinados frameworks en el mercado no contienen todas las propiedades inherentes a la arquitectura. Además, estas librerías de clases suelen acompañarse de herramientas de alto nivel que, si bien facilitan la construcción de aplicaciones con interfaces de usuario, vuelven prácticamente imposible cambiar o mejorar el diseño de la aplicación para que éste resulte orientado a MVC.

6.3.4- Dado el alto grado de abstracción que los frameworks de última generación ofrecen al programador, está en los creadores de dichos frameworks el incluir y promover los conceptos de MVC para que los desarrolladores que utilicen estas herramientas puedan obtener y comprobar todas las ventajas de MVC en las aplicaciones de software que construyan.

6.4 Trabajo Futuro

6.4.1- El conjunto de propiedades utilizado para el análisis de los frameworks se propone como una metodología para el estudio de otros frameworks no incluidos en esta tesis, de manera que el análisis no quede restringido únicamente a este grupo. Todo framework MVC o propuesta

de diseño MVC puede ser analizado con el mismo criterio con el objetivo de ser evaluado y de decidir si realmente implementa la arquitectura y en qué grado soporta estas propiedades, pudiendo establecerse a su vez, una clara comparación con respecto a los frameworks aquí contenidos. También podría definirse algún tipo de técnica que permita sistematizar el análisis para poder aplicarlo en la posterioridad a otros frameworks, con mayor facilidad y exactitud.

6.4.2- Dado el constante crecimiento de las aplicaciones de software en tamaño, complejidad y contextos, podrían agregarse nuevas propiedades que emerjan de la utilización de MVC en estos nuevos contextos, extendiéndose así, la metodología de análisis empleada en esta tesis. Ejemplos de nuevos contextos en pleno crecimiento son el ambiente de las aplicaciones mobile y el de las aplicaciones con interfaces personalizadas.

6.4.3- Si bien el grupo de propiedades utilizadas en el análisis no se extiende solamente al contexto de la Web, ya que hemos incluido en dichas propiedades el soporte para tipos de interfaces no Web y si bien algunos de los frameworks analizados (ej: JSF) ya los implementan, solamente se ha analizado en detalle el traspaso de MVC a la Web y los impactos que este contexto produce en la arquitectura (capítulo 4). Sin embargo, el estudio realizado sobre frameworks que soportan otros tipos de vistas, deja una puerta abierta a analizar elaboradamente cómo MVC cambia y se adapta a nuevas aplicaciones y qué obstáculos se han superado con tal fin.

6.4.4- Las propiedades que en los frameworks analizados en el Capítulo 5, no han sido implementadas o lo son en menor grado que el deseable y las soluciones propuestas en el mismo capítulo para incluir dichas propiedades, pueden ser consideradas en conjunto como base para el diseño e implementación de otros frameworks que, utilizando las mismas tecnologías, extiendan a los primeros. Estos nuevos frameworks incluirían las características que dicha tecnología aún no posee, promoviendo al desarrollo de aplicaciones Web que resulten siempre orientadas a MVC.

Glosario

El glosario contiene los términos que se utilizan con frecuencia a lo largo del trabajo. Los vocablos que pertenecen exclusivamente a una sección específica han sido omitidos aquí. Estos últimos se encuentran aclarados o explicados en la misma sección donde se los menciona.

API Application Programming Interface: La interfaz externa de una plataforma de software, tal como un sistema operativo, que es utilizada por sistemas o aplicaciones construidas sobre ella.

Aplicación o Aplicación de Sofware Un programa o conjunto de programas que satisface los requerimientos de un cliente.

Aplicación de escritorio El término se refiere a una aplicación de software que no corre en la web.

Browser web Navegador web que se comunica con una aplicación web. También puede utilizarse solamente Browser cuando se habla de Browser en un dispositivo móvil tal como un PDA.

Clase Es el bloque fundamental de construcción en los lenguajes orientados a objetos. Una clase especifica y encapsula su estructura de datos interna así como la funcionalidad de sus instancias u objetos.

Clase Abstracta Una clase que no implementa todos los métodos que son definidos en su interface. Una clase abstracta define una abstracción común para sus subclases.

Clase Concreta Una clase cuyos objetos pueden ser instanciados. En contraste con una clase abstracta, todos los métodos son implementados en una clase concreta.

Cliente Cliente denota un componente o subsistema que utiliza la funcionalidad ofrecida por otros componentes. Mayormente, se menciona el término para referenciar a un cliente web. También se utiliza, en algunos casos, como sinónimo de usuario final.

Componente Una parte encapsulada de un sistema de software. Un componente tiene una interfaz que provee acceso a él y sus servicios. Sirven como bloques para construir la estructura de un sistema. En un

lenguaje de programación, componentes pueden ser módulos, clases, objetos o un conjunto de funciones relacionadas. Un componente que no implementa todos los elementos de su interface es denominado componente abstracto.

Contenedor Un nombre común para estructuras de datos que contienen un número de elementos, Ej: listas, conjuntos.

Diseño En el contexto de este trabajo, diseño es la actividad realizada por un desarrollador de software que resulta en la arquitectura de software de un sistema. Suele denominarse con esta palabra al resultado de esta etapa.

Dispositivo móvil Término con el cual se denomina en este trabajo a un teléfono celular o PDA, especialmente para marcar la diferencia con una computadora de escritorio.

Framework Un software semiterminado para ser instanciado. Un framework define la arquitectura para una familia de sistemas o subsistemas y provee los bloques básicos para crearlos. También define las partes que deben ser adaptadas para lograr funcionalidad específica. En un ambiente orientado a objetos un framework consiste de clases abstractas y concretas. La instanciación de tal framework consiste de componer y subclasear las clases existentes.

GUI Graphical User Interface. Interfaz Gráfica de Usuario.

Instancia Un objeto generado de una clase específica. El término es utilizado como sinónimo de objeto en un ambiente orientado a objetos.

Interfaz Término que hace referencia a la vista o presentación de un modelo (en MVC) o denominación del conjunto de métodos que un objeto exporta para ser invocado por otros objetos (ver Cap.1 Pág.5). También es, en Swing, el nombre que se le da a la definición de un conjunto de métodos u operaciones. Las clases que implementen una interfaz deben implementar todos los métodos definidos en la misma.

Look and Feel Expresión general utilizada para denominar a la vista y la interacción que ofrece la misma en un sistema de software.

Mensaje En un sistema orientado a objetos, el término mensaje es utilizado para describir la selección y activación de una operación o método de un objeto.

Método Una operación realizada por un objeto que se encuentra especificado en una clase. El término también puede ser utilizado como sinónimo de metodología: un conjunto de reglas y/o técnicas usado para desarrollar un sistema de software.

Método abstracto Una interfaz para una operación de una clase que debe ser definido por las subclases.

Módulo Una entidad sintáctica o conceptual de un sistema de software. Usualmente utilizado como sinónimo de componente o subsistema.

Objeto Una entidad en un sistema orientado a objetos. Los objetos responden a los mensajes mediante la ejecución de una operación (método). Un objeto puede contener datos y referenciar a otros objetos que en conjunto definen el estado del objeto.

Observer Observador. Nombre del patrón de diseño y también término con el que se denomina a uno de los objetos que intervienen dentro del patrón o al rol que éste cumple.

Patrón de diseño Una solución a un problema recurrente que define un conjunto de objetos y la interacción entre ellos, así como sus responsabilidades para el funcionamiento de la solución. Un patrón de diseño también especifica el contexto donde puede ser utilizado y el alcance o aplicabilidad. Un patrón es luego implementado para resolver un problema en particular. Ej: Observer es implementado en MVC para resolver la comunicación entre modelo y vistas (ver Cap.1 para Observer y anexo de Patterns).

PDA Personal Digital Assistant. Un dispositivo organizador electrónico móvil.

Plataforma Hardware y/o software que un sistema utiliza para su implementación. Las plataformas de software incluyen sistemas operativos, librerías y frameworks. Una plataforma implementa una máquina virtual con aplicaciones corriendo sobre ella.

Protocolo Conjunto de métodos definidos en una clase.

Responsabilidad Funcionalidad de un objeto o componente en un contexto específico. Una responsabilidad está usualmente definida por un conjunto de operaciones.

Renderización Es el proceso por el cual un componente parte de una vista o una vista se despliega para su visualización por parte del usuario final.

Rol La responsabilidad de un componente en un contexto de componentes relacionados, posiblemente en la implementación de un patrón de diseño. En este trabajo, los roles son mayormente modelo, vista y controlador.

Servidor Un componente o subsistema que responde a requerimientos de clientes. Cuando un requerimiento llega el servidor lo atiende o delega la tarea a otros componentes.

Subclase Clase que hereda de una o más clases.

Subject Sujeto. Uno de los objetos que intervienen en el patrón de diseño *Observer* (ver Cap. 1).

Superclase Una clase de la cual otra clase hereda.

Bibliografía y documentación consultada

[1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, "Pattern-Oriented Software Arthictecture: A System of Patterns", Wiley, 1996.

[2] Glenn E. Krasner, Stephen T. Pope, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System", 1988.

[3] Glenn E. Krasner, Stephen T. Pope, "A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80", 1988.

[4] Steve Burbeck, "Applications Programming in Smalltalk-80: How to use Model-View-Controller (MVC)", 1987, 1992.
http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

[5] Mark Andrews, "Introducing Swing Architecture"
http://java.sun.com/products/jfc/tsc/articles/getting_started/getting_started2.html

[6] Todd Sundsted, "MVC meets Swing"
http://www.javaworld.com/javaworld/jw-04-1998/jw-04-howto.html

[7] Amy Fowler, Sun Microsystems 1994 – 2004 "A Swing Architecture Overview"
http://java.sun.com/products/jfc/tsc/articles/architecture/

[8] VisualWorks Application Developer's Guide, Cincom Systems, 1993-2003.

[9] VisualWorks GUI Developer's Guide, Cincom Sytems, 1993-2003.

[10] Malcom Davis, "Struts, an open-source MVC implementation"
http://www-106.ibm.com/developerworks/java/library/j-struts/

[11] Alan Knight, Naci Dai, "Objects and the Web", IEEE Software, March/April 2002

[12] The Struts User Guide
http://jakarta.apache.org/struts/userGuide/index.html

[13] "Introduction to WebForm Pages", Microsoft Visual Studio .Net 2003 Documentation.

[14] UI Development with JavaServer Pages,
https://www6.software.ibm.com/developerworks/education/j-jsf/index.html

[15] David Geary, "A First Look At JavaServer Pages", Part 1, Part 2, Java World.
http://www.javaworld.com/javaworld/jw-11-2002/jw-1129-jsf.html

[16] Alex Homer, Dave Sussman, Rob Howard,
"A First Look at ASP.NET v 2.0", Microsoft .Net Development Series, Addison-Wesley, 2004.

[17] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides,
"Design patterns, Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[18] David H. Young
"A friendly game of tug of war: XMLC vs. JSP", April 2002.
http://www.theserverside.com/articles/article.tss?l=XMLCvsJSP

[19] Bobby Woolf and Knowledge Systems Corporation
"Understanding and using ValueModels", 1994.

[20] XMLC User's Manual
http://xmlc.objectweb.org/doc/doc/xmlc/user-manual/index.html

Anexo

Patrones de Diseño

Este anexo incluye las definiciones de los patrones de diseño orientados a objetos que han sido referenciados en el trabajo.

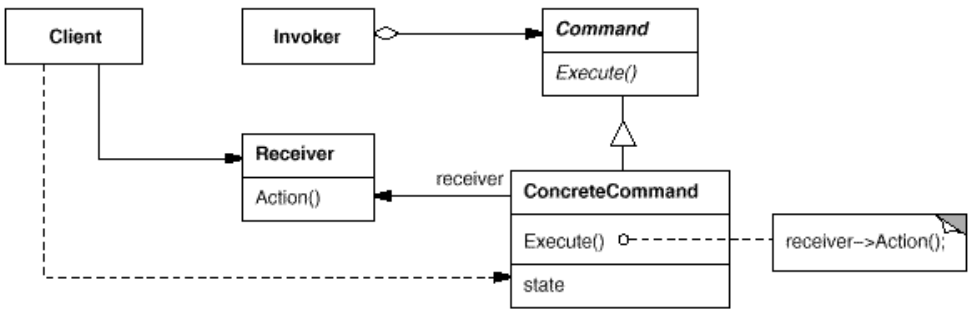
1. Command

Representa una acción como un objeto. Permite hacer y deshacer operaciones y guardar un historial de las mismas.

Se utiliza cuando se desea:

- parametrizar un objeto para ejecutar una acción.
- ejecutar operaciones en diferentes tiempos. Un objeto Command puede tener una vida independiente del pedido original.
- realizar acciones "hacer" y "deshacer". El método *execute* del Command puede guardar el estado de cualquier objeto para un posterior "deshacer".

Estructura



- Command: declara la interfaz para ejecutar una operación.
- ConcreteCommand: implementa *execute( )* invocando a los métodos de Receiver.
- Client: crea un ConcreteCommand y fija su Receiver.

- Invoker: le indica al Command que debe ejecutarse.
- Receiver: sabe cómo ejecutar las operaciones que el ConcreteCommand le solicita.

Consecuencias

- Command desacopla el objeto que llama a las operaciones de aquel que la lleva a cabo.
- Facilidad para agregar nuevos comandos, ya que no hay que cambiar ninguna clase existente.

Implementación

- Un ConcreteCommand puede necesitar guardar información adicional para proveer "rehacer" y "deshacer". La información puede consistir en: emisor y receptor del mensaje, argumentos de la operación y valores del receptor que se modifiquen al aplicar la operación.
- El Receiver debe proveer operaciones para volver a un estado anterior.

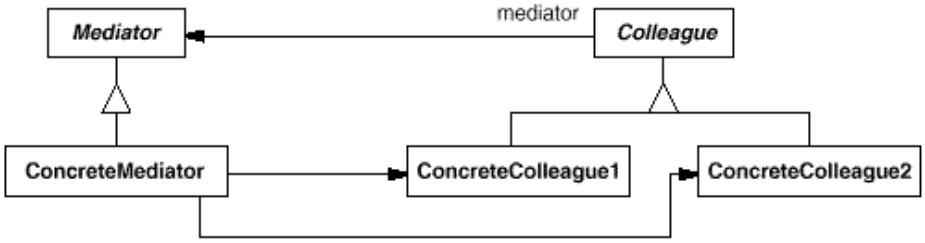
2. Mediator

Define un objeto que encapsula la interacción entre un conjunto de objetos.

Se usa cuando:

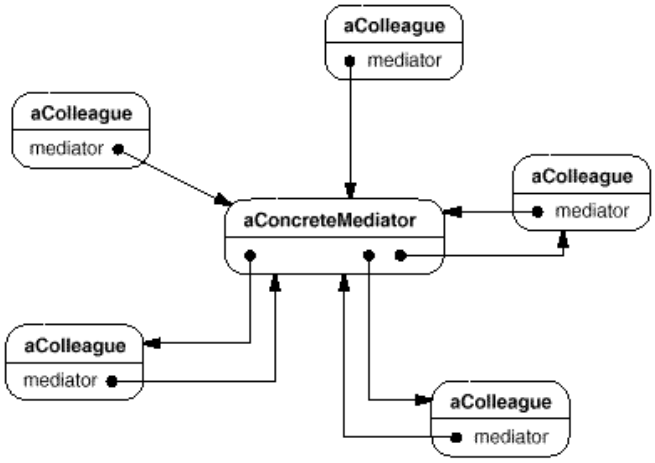
- Un conjunto de objetos se comunican de manera compleja. Con Mediator se solucionan las dependencias no estructuradas y se facilita la compresión de la comunicación.
- Reusar un objeto es difícil porque se comunica con muchos otros. Mediator soluciona ésto haciendo que todos se comuniquen con él.

Estructura



- Mediator: define la interfaz para la comunicación con los Colleague.
- ConcreteMediator: implementa la cooperación entre los ConcreteColleagues y mantiene referencia a ellos.
- ColleagueClasses: Cada clase Colleague debe conocer al Mediator. Cada Colleague se comunica con su Mediator cuando de otra manera hubiese tenido que comunicarse con uno o más Colleague.

Una estructura típica de objetos debería quedar así:



Consecuencias

- Un Mediator reemplaza una comunicación muchos a muchos con una de uno a muchos. Este tipo de relaciones es más fácil de entender, mantener y extender.
- Debido a que el Mediator se comunica con todos los demás, puede convertirse en un objeto complejo y difícil de mantener.

Implementación

Se puede omitir la declaración de la clase Mediator si todos los Colleagues trabajan con un solo Mediator.

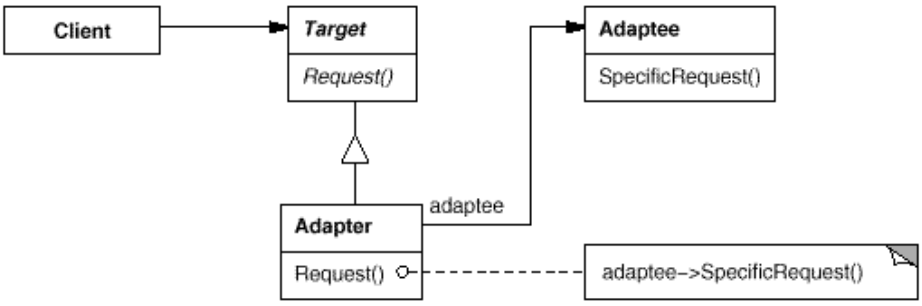
3.Adapter

Convierte la interface de una clase en otra interface que el usuario necesite. El Adapter permite que dos clases trabajen juntas aunque tengan interfaces distintas.

Se usa cuando:

- Querés usar una clase existente y la interface no es la compatible (no es la se necesita).
- Querés crear una clase reusable que coopere con clases no conocidas. Las clases no necesitan tener interfaces compatibles (pluggable adaptor o aspect adaptor).
- Necesitas adaptar muchas subclases, y no podes adaptar cada uno por separado, entonces adaptas a la clase padre.

Estructura



- Target: define la interface del dominio que el cliente usa.
- Client: Colabora con objetos conforme a la interface del Target.
- Adaptee: define una interface existente que necesita ser adaptada.
- Adapter: adapta la interface del Adaptee a la interface del Target.

Colaboraciones

El cliente llama a operaciones en una instancia del Adapter, y este llama a las operaciones del Adaptee para hacer lo pedido.

Consecuencias.

- El Adapter puede agregar funcionalidad además de adaptar una interface.

El Adapter puede solo cambiar el nombre de los métodos o soportar un conjunto de operaciones completamente diferentes.

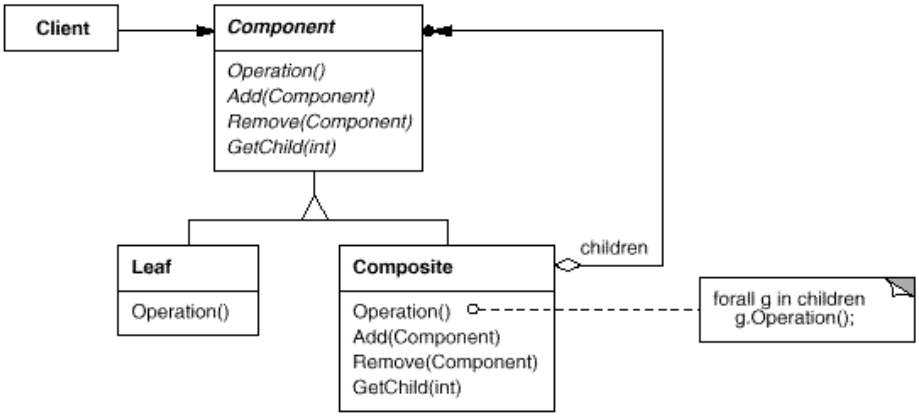
4. Composite

Compone objetos en estructura de árbol. Composite permite tratar objetos simples y composiciones de manera uniforme.

Se usa cuando:

- Se quiere representar jerarquías de objetos.
- Se quiere que el cliente ignore la diferencia entre las composiciones y los objetos individuales.

Estructura



- Component: declara la interface para objetos de la composición. Implementa el comportamiento base para la interface común a todos los objetos. Declara la interface para acceder y manejar a los hijos.
- Leaf: representa las hojas y declara las primitivas para esos objetos.
- Composite: define el comportamiento de un componente con hijos, los almacena e implementa las operaciones relacionadas con ellos.
- Client: manipula los objetos de la composición a través de la interface de Component.

Colaboraciones

Los clientes usan la interface de Component para interactuar con los objetos del componente.

Consecuencias

- El cliente no detecta si esta comunicándose con una hoja o con un Composite ya que usan el mismo protocolo.
- Hace que el cliente trabaja más fácil debido a que maneja estructuras de árbol con la simpleza de un objeto simple.
- La facilidad de agregar nuevos componentes al Composite trae como consecuencia la dificultad de restringir el tipo de los objetos, lo cual puede sobregeneralizar el diseño.

Implementación

- La clase Component debe definir tantas operaciones comunes entre Leaf y Composite como sea posible. La Clase Component provee implementación básica de esas operaciones, y tanto Leaf como Composite las sobrescribirán.
- Algunas Operaciones del Composite, tales como add o remove es importante definirlas en Component, debido a que el cliente se maneja con esa interface y no la de Composite). Otra utilidad de definirlas en Component es el chequeo de errores, ya que por error se podría llamar a operaciones del Composite (add o remove por ejemplo) desde Leaf lo cual no es correcto y debe ser corregido.
- Los métodos de Composite definidos en Component son implementados para que generen una excepción y Composite los redefine con las operaciones concretas.