Unidad Didáctica III.- Programación visual. Modelo vista-controlador

*CASO. En **SI Andalucía** tienen cada vez más proyectos que desarrollar. Además, últimamente, muchos de esos proyectos están relacionados con la web. Y los clientes son muy exigentes con el diseño de los interfaces web. El caso es que todo esto, está haciendo reflexionar a todos los miembros de la empresa. Si siguen a este ritmo de trabajo, quizás tengan que contratar a alguien más para dedicarse específicamente a realizar diseño de interfaces para las aplicaciones. Pero claro, un buen diseñador de interfaces, que además sepa programar, nos saldría por un ojo de la cara, piensa **María**, y lo comenta con el resto. En caso de ser necesario, lo mejor sería contratar a alguien que sólo se dedicara a diseñar interfaces, ya que sería más barato. Aunque en ese caso, a la hora redesarrollar un proyecto, se debe separar claramente la parte de presentación, de la parte de modelo, o sea, de los datos. Así, el que sabe programar, programaría esa parte, el modelo, y otra persona sólo se dedicaría a diseñar, pudiendo trabajar de manera independiente.*

Seguro que más de una vez, al utilizar cualquier aplicación o cualquier página web del mercado, te habrás parado a pensar sobre la complejidad que habrá tenido el desarrollo de esa aplicación o de esa página. Quizás, incluso te habrás planteado de qué manera los desarrolladores habrán tenido en cuenta sus trabajos anteriores y habrán reutilizado algo que ya tenían hecho.

El desarrollo de software es una tarea bastante complicada, que depende en gran medida de la experiencia de las personas involucradas, en particular de los desarrolladores.

Las Tecnologías Orientadas a Objetos son las más utilizadas en los últimos años para el desarrollo de aplicaciones software. Se ha comprobado que este paradigma de programación presenta muchas ventajas, entre las cuales destacan:

- eficiencia,
- disminución del tiempo de desarrollo y
- reutilización.

Respecto a la reutilización del software, podemos decir que dispone de diversos mecanismos, entre ellos:

- **componentes**: elementos de software suficientemente pequeños para crearse y mantenerse pero suficientemente grandes para poder utilizarse,
- **bibliotecas de clases:** preparadas para la reutilización, que pueden utilizar a su vez componentes, y
- **patrones de diseño**. ¿Qué es un patrón?

*CASO. **María**, **José** y el resto de integrantes de **SI Andalucía** siguen pensando sobre esa separación de papeles a la hora de desarrollar una aplicación, y empiezan a recordar algo que hace poco aprendieron. Hace un tiempo asistieron a unos cursos sobre Análisis y Diseño del software, donde les estuvieron contando la utilización de patrones en el desarrollo de aplicaciones. **Carmen** ve claramente que ahí puede estar la solución a las recientes inquietudes del equipo de trabajo, y comienza a refrescarles la memoria a sus compañeros, sobre qué era eso de los patrones.*

Conceptos básicos y origen de los patrones.

En el apartado anterior dejábamos sin responder la pregunta sobre qué es un patrón.

¿Has pensado en la respuesta? ¿Te has dado cuenta alguna vez, de que en el mundo de la informática se utilizan muchos conceptos que provienen de otros ámbitos de la vida?

Los patrones constituyen un ejemplo de ello, ya que este concepto es aplicable a multitud de cosas, y en concreto se aplica a la construcción del software, pero por similitud con la construcción de edificios.

Una definición de patrón puede ser:

> ***"Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma".***

Los patrones como elemento de la reutilización, comenzaron a utilizarse en la arquitectura con el objetivo de reutilizar diseños que se habían aplicado en otras construcciones y que se catalogaron

como completos.

Chistopher Alexander, autor de la definición anterior de patrón, fue el primero en intentar crear un formato específico para patrones en la arquitectura. Un patrón trata de extraer la esencia de un diseño para que pueda ser utilizada por otros arquitectos cuando se enfrentan a problemas parecidos a los que resolvió dicho diseño. Esa misma idea es la que se pretende aplicar en el mundo de la construcción del software.

El tal Alexander intenta resolver problemas arquitectónicos utilizando estos patrones. Para ello trata de extraer la parte común de los buenos diseños, con el objetivo de volver a utilizarse en otros diseños.

Cuando has estado de viaje en algún lugar, o simplemente paseando por tu pueblo o ciudad, ¿te has fijado en los pisos, en las iglesias o en las casas?

Si nos fijamos en las construcciones de una determinada zona, normalmente observaremos que todas ellas poseen apariencias parejas (por ejemplo en zonas del norte de España, los tejados de pizarra con gran pendiente, o bien en zonas del sur de España, que gozan de mucho sol y poca lluvia, predominan las casas blancas con tejados con muy poca pendiente, etc.). De algún modo, la esencia del diseño se ha copiado de una construcción a otra, y a esta esencia se pliegan de forma natural los diversos requisitos. Se puede decir aquí que existe un patrón que soluciona de forma simple y efectiva los problemas de construcción en tal zona.

También se puede utilizar una metáfora textil para explicar lo que es un patrón: es como la pieza que utiliza el sastre a la hora de confeccionar vestidos y trajes. De esta forma este patrón además de contener las especificaciones de corte y confección del producto final, representa a la vez, una parte de ese producto.

> ***En definitiva se puede definir un patrón como "una solución a un problema en un determinado contexto".***

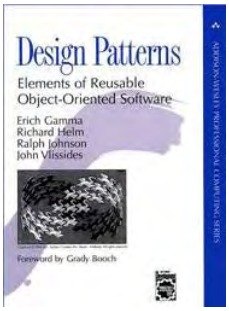
> ***Para saber más:***
> ***En este enlace puedes saber algo más sobre Christopher Alexander. Enlace a Wikipedia acerca de C. Alexander:***
> ***http://es.wikipedia.org/wiki/Christopher_Alexander*** [Versión en cache]

Patrones en el diseño de aplicaciones informáticas.

¿Pero, cuando llega el concepto de patrón al mundo del software?

Como hemos visto en el apartado anterior, el término patrón se utilizó inicialmente en el campo de la arquitectura, por Christopher Alexander, a finales de los años 70.

Ese concepto se transporta al ámbito del desarrollo de software orientado a objetos y se aplica al diseño. Probablemente ese hecho se produce en 1987, cuando Ward Cunningham y Kent Beck trabajaron con el lenguaje de programación Smaltalk y diseñaron interfaces de usuario. Decidieron, para ello, utilizar alguna de las ideas de Alexander para desarrollar un lenguaje pequeño de patrones para servir de guía a los programadores de Smalltalk. Sus ideas las plasmaron en el libro "*Using Pattern Languajes for Object-Oriented Programs*".



Desde 1990 a 1994, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (conocidos como "la banda de los cuatro") realizaron un primer catálogo de patrones de diseño. Y ya en 1994 publicaron el libro "*Design Patterns: Elements of Reusable Object-Oriented Software*" (Gang of Four, [GoF]) que se puede considerar como el libro de referencia de los patrones.

Los patrones para el desarrollo de software son uno de los últimos avances de la Tecnología Orientada a Objetos. Son una forma de resolver problemas de ingeniería del software, que tienen sus raíces en los patrones de la arquitectura.

Los diseñadores y analistas de software más experimentados aplican de forma intuitiva algunos criterios que solucionan los problemas de manera elegante y efectiva. La ingeniería del software se enfrenta a problemas diversos que hay que identificar para poder utilizar la misma solución (o casi la misma) con problemas similares.

Las metodologías Orientadas a Objetos tienen como uno de sus principios "no reinventar la rueda" para la resolución de diferentes problemas. Por lo tanto los patrones se convierten en una parte muy importante en las Tecnologías Orientadas a Objetos para poder conseguir la reutilización.

¿Por qué utilizar patrones? ¿A qué ayudan los patrones?

- Una arquitectura orientada a objetos bien estructurada está llena de patrones. Los patrones conducen a arquitecturas más pequeñas, más simples y comprensibles, y más reutilizables.
- Cada patrón permite que algunos aspectos de la estructura del sistema puedan cambiar independientemente de otros aspectos.
- Facilitan la reusabilidad, la extensibilidad de aplicaciones y el mantenimiento de las mismas.

Pero diseñar software orientado a objetos es difícil, y diseñar software orientado a objetos reutilizable es más difícil todavía. Cuando nos ponemos a diseñar y programar aplicaciones informáticas, encontrar diseños generales y flexibles es una tarea muy difícil al principio.

Entonces, ¿qué conoce un programador experto que desconoce uno inexperto?

Pues claramente podemos contestar: **reutilizar soluciones que funcionaron en el pasado, es decir, la experiencia**.

La ingeniería del software intenta aplicar patrones software para la resolución de problemas en dicho campo. Para conseguir esto debe existir una comunicación entre los distintos ingenieros para compartir los resultados obtenidos. Por tanto debe existir también un esquema de documentación con el objetivo de que la comunicación pueda

entenderse de forma correcta. El objetivo de los patrones es crear un lenguaje común a una comunidad de desarrolladores para comunicar experiencia sobre los problemas y sus soluciones.

Por tanto, al enfrentarse al diseño de una aplicación informática nueva, o incluso cuando haya que retocar alguna aplicación existente, **deberíamos pensar en la posibilidad de aplicar patrones para obtener una solución óptima**, es decir, una solución reutilizable y eficiente. Por ello, se recomienda no empezar directamente a codificar.

Existen un montón de patrones: Singleton, Bridge, Builder, Factoría Abstracta, Modelo-Vista-Controlador, etc, cada uno para un propósito distinto.

Si echáis un vistazo a algún libro sobre patrones de diseño de software, como por ejemplo alguno de los mencionados más arriba, veréis que cada patrón describe un problema recurrente y una solución. Para cada patrón se suele describir una serie de características, entre ellas:

- el nombre del patrón,
- el propósito o problema que afronta, y
- la solución que se aporta con el patrón.

Autoevaluación

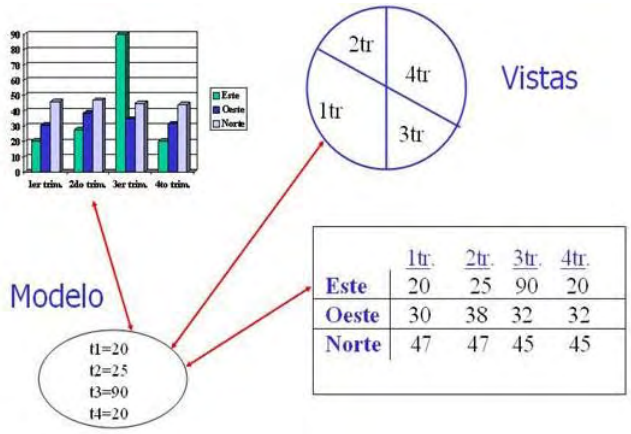
1 En el mundo de la ingeniería del software, y hablando de patrones, ¿qué afirmación es correcta?

a) Los patrones son un invento de los ingenieros de software que rara vez sirven para algo.
b) Todos los patrones sirven para resolver el mismo tipo de problema, por lo que da igual el que se utilice.
c) El objetivo de los patrones es crear un lenguaje diferente dentro de una comunidad de desarrolladores para comunicar experiencia sobre los problemas y sus soluciones.
d) Los patrones permiten reutilizar soluciones que en el pasado funcionaron.

Comprobar

*CASO. **Carmen** les ha hecho ver al resto de compañeros de **SI Andalucía**, que uno de los patrones de diseño que aprendieron en su día puede ser la solución a sus problemas a la hora de desarrollar aplicaciones. Saben que no es tarea fácil llevar a la práctica esas ideas de separar la presentación de los datos, o dicho de otro modo, de separar la vista del modelo. Aunque tienen claro que el patrón que más le interesa para sus propósitos es el patrón **Modelo-Vista-Controlador**, son conscientes de que tendrán que practicar mucho su uso en sus desarrollos.*

¿Has utilizado alguna vez una hoja de cálculo como Calc de OpenOffice o Excel de Microsoft Office? (Haz Clic en la imagen para ampliar)

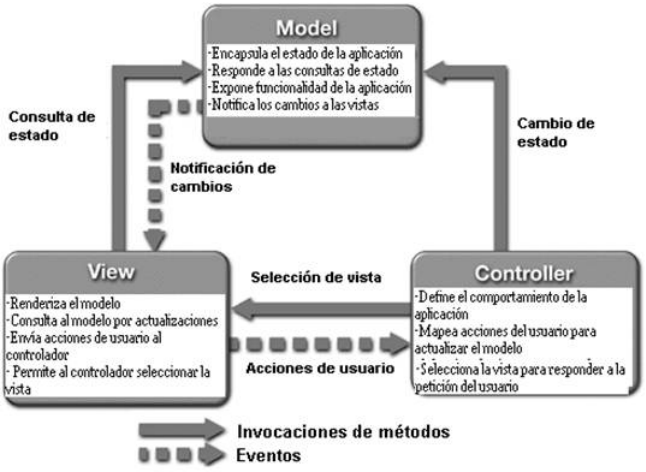


Si has utilizado alguno de esos programas, habrás comprobado que cuando introduces unos datos en las celdas de una hoja, existe la posibilidad de ver esos datos de varias formas. Es el caso que se puede apreciar en la imagen que ves junto a este texto. Los mismos datos, se pueden visualizar de varias maneras: como diagrama de barras, como diagrama de quesos, como tabla, etc.

Esos programas implementan el patrón Modelo-Vista-Controlador (MVC), el modelo lo constituyen los datos que hemos introducido en las celdas. Las vistas se encargan de mostrar los datos de la forma que se seleccione.

El patrón Modelo Vista Controlador (MVC) divide una aplicación en tres componentes:

- **El modelo** contiene la funcionalidad básica y los datos, los objetos del dominio.
- **Las vistas** muestran información al usuario (el **interfaz** de usuario).
- **El controlador** define la forma en que el interfaz reacciona a la entrada del usuario, gestiona las entradas del usuario.



El propósito de este patrón es que **los objetos del modelo no deben conocer directamente a los objetos de la vista o presentación**. Las clases del dominio encapsulan la información y el comportamiento relacionado con la lógica de la aplicación. Las clases de la interfaz (ventanas) son responsables de la entrada y salida, capturando los eventos, pero no encapsulan funcionalidad de la aplicación.

Por tanto, la vista es la capa de presentación, que es responsable de interactuar con el usuario. El modelo consiste en la lógica de negocio y datos, y el controlador consiste en el código de la aplicación que responde a los eventos de los usuarios e integra el modelo y la vista. Esta arquitectura asegura que la aplicación está desacoplada, lo que reduce las dependencias entre diferentes capas.

Si recuerdas, en el módulo *Programación en Lenguajes Estructurados*, se hablaba brevemente sobre la arquitectura Modelo-Vista-Controlador.

Puedes ver el contenido de lo que se comentaba, en el siguiente recurso:

Modelo-Vista-Controlador en Java Swing

Y así, se comentaba por ejemplo que Java Swing implementa este patrón, permitiendo por ejemplo, poder cambiar el aspecto de un componente según los datos que almacene en su modelo. Esto hace posible en Swing, entre otras cosas, que tengamos distintas representaciones gráficas de un mismo botón.

Como justificación de este patrón, como los objetivos que pretende podemos destacar:

- Se busca utilizar clases cohesivas.
- Permitir separar el desarrollo de las clases de la vista y del dominio. Así puede haber programadores realizando las tareas concernientes a los datos, a la funcionalidad de la aplicación, permitiendo trabajar de manera independiente a los diseñadores de la parte de interfaz, de la capa vista de la aplicación.
- Minimizar el impacto de los cambios en la interfaz sobre las clases del modelo.
- Facilitar la conexión de unas vistas a una capa de dominio existente.
- Permitir varias vistas simultáneas sobre un mismo modelo.
- Permitir que la capa del modelo se ejecute de manera independiente a la capa de presentación.

> ***Los objetos del modelo (dominio) no deben conocer directamente a los objetos de la vista (presentación).***
> ***Las clases del dominio encapsulan la información y el comportamiento relacionado con la lógica de la aplicación.***
> ***Las clases de la interfaz (ventanas) son responsables de la entrada y salida, capturando los eventos, pero no encapsulan funcionalidad de la aplicación.***

MVC es una arquitectura típica de diseño de GUIs.

- El modelo representa la estructura lógica de la GUI independientemente de su representación visual.
- Las vistas constituyen la representación visual de la GUI.
- El controlador constituye la lógica de interacción con el usuario. La mayor parte de las herramientas de desarrollo incorporan en las clases de la vista gran parte o todo el procesamiento de eventos. Con lo que el controlador queda semioculto dentro de la vista.

Entre las ventajas del patrón MVC podemos destacar:

- **Es posible tener diferentes vistas para un mismo modelo** (por ejemplo representación de un conjunto de datos como una tabla o como un diagrama de barras).
- **Es posible construir nuevas vistas sin necesidad de modificar el modelo subyacente**.
- **Proporciona un mecanismo de configuración para componentes complejos mucho más tratable que el puramente basado en eventos** (el modelo puede verse como una representación estructurada del estado de la interacción).

Autoevaluación

1 Referente al patrón Modelo-Vista-Controlador, ¿qué afirmación es correcta?

a) El patrón MVC pretende que las clases tengan baja cohesión.
b) Cuando se utilice este patrón, cada vez que se haga una nueva vista en la aplicación, será

necesario modificar el modelo subyacente.

c) El patrón disminuye el impacto de los cambios en la interfaz sobre las clases del modelo

d) El objetivo de este patrón es que el diseñador de interfaces de una aplicación sea también el programador de dicha aplicación.

Comprobar

Ejemplo de aplicación usando el patrón MVC: datos sobre personas.

Al realizar una aplicación informática, podemos utilizar innumerables vistas para representar la misma información. Supongamos una aplicación de lo más sencilla, para explicar este patrón. Digamos, que la aplicación, simplemente va a manejar información: nombre y edad de una persona.

Crearemos la aplicación que llamaremos **NombreEdad**. Más abajo dispones de los enlaces para descargarte tanto la aplicación como una presentación donde se muestra cómo construirla.

La información, los datos de la aplicación, se podrán representar de muchas formas. Pero esos datos son los mismos para todas las vistas. Es decir, para una persona tiene tal nombre y tal edad, esos datos se pueden representar de muchas formas, con la letra más grande o más pequeña, pero los datos en sí son los mismos. Claramente, el modelo va a venir representado por esa clase donde tengamos esos datos. Llamemos a esa clase **Persona**, y en principio, podría ser:

```
package mvc.model;
public class Persona {
    // Datos que almacena: el nombre y la edad
    private String nombre ;
    private String edad ;
    public Persona() {
    }
    public String getNombrePersona() {
        return nombre ;
    }
    public void setNombrePersona(String nuevoNombre) {
        this.nombre = nuevoNombre ;
    }
    public String getEdadPersona() {
        return edad ;
    }
    public void setEdadPersona(String nuevaEdad) {
        this.edad = nuevaEdad ;
    }
}
```

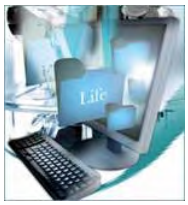
Como hemos dicho que podemos tener varias vistas sobre esos mismos datos, podemos ahora crear un interface, al que podemos llamar **Vista**, de modo que todas aquellas vistas que se realicen, implementen ese interface.

¿Y qué método o métodos deben contener esas vistas?

La idea fundamental es que cuando se produzca un cambio en el modelo, se notifique a las vistas para que se actualicen. Por tanto, podemos crear el interface **Vista** con un método al que llamemos **actualizar().**

```
package mvc.view;
public interface Vista {
    public void actualizar() ;
}
```

Entonces, según lo que acabamos de decir, será necesario que al cambiar los datos en el modelo, se notifique a todas las vistas que haya que actualizar.



- ¿Cómo puede saber el modelo, qué vistas tiene que actualizar? Tendrá que actualizar todas aquellas **vistas que se registren en él**.
- Y, ¿cómo se pueden registrar? Pues lo más fácil quizás será **crear un vector en la clase modelo**, de manera que cuando una clase vista quiera registrarse para ser notificada, se introduzca en ese vector. De este modo, cuando haya que actualizar las vistas, bastará con recorrer ese vector, y para cada elemento, notificarle que se actualice.

Por tanto, volvemos a la clase **Persona** y la modificamos añadiendo ese vector, el método que permitirá registrar las vistas y el método que las requerirá para actualizarse, como podemos ver en la siguiente porción del código de la clase, en el recurso siguiente:

Extracto de la clase Persona

Como desde el proyecto modelo estamos referenciando al proyecto **Vista**, para poder compilar y ejecutar la aplicación, tenemos que situarnos en el proyecto **Model**, y pulsando el botón derecho del ratón seleccionamos **Properties**. En el apartado Dependencias marcamos el proyecto **View**. En principio no haría falta hacer lo mismo con el proyecto **Vista**. Pero como desde el fichero fuente **Principal.java** estamos haciendo referencia al modelo, para hacer las pruebas, tenemos que proceder de manera similar.

Ahora podemos implementar cada una de las vistas. Podemos empezar por una vista con controles Swing, que nos presente la información por la consola. Como se puede ver en el código siguiente, la vista implementa el interface **Vista**. En el constructor de la clase, pasamos el modelo a representar. Y en el método actualizar se presenta la información. Recordemos que ese método actualizar es al que se llama cuando se produce un cambio en el modelo.

Código fuente de la clase VistaSwing1

En el código que se adjunta más abajo, podéis ver el resto de vistas. Para probar el código, podemos crear una clase, que podemos situar en el proyecto de las vistas, que cree una persona con unos datos. Tras eso, crearemos las tres vistas que nos mostrarán la información de esa persona. Haremos una pausa y cambiaremos los datos del modelo para ver cómo se actualizan las vistas. En este recurso que tienes a continuación, puedes ver la clase Principal:

Código fuente de la clase Principal

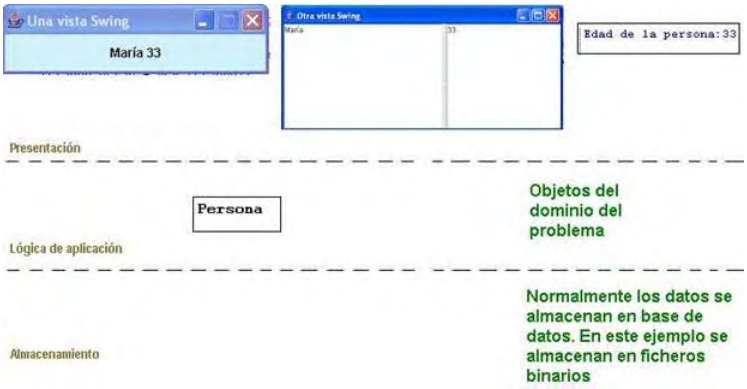
El mismo Modelo presenta en este ejemplo tres vistas: dos Swing y una vista por consola.

Como se puede ver en el listado del recurso anterior del fichero fuente **Principal.java**, y según hemos comentado, en este caso no hay interacción del usuario, sino una estructura secuencial, por lo que el controlador no existe como tal, está diluido en ese archivo fuente.

Creación de la aplicación para almacenar nombre y edad

Un sistema de información o aplicación típica suele presentar un diseño arquitectónico de varios niveles o capas como las siguientes:

- **Presentación**: interfaz gráfica, ventanas.
- **Lógica de aplicación**: objetos que representan conceptos del dominio del problema. También puede haber objetos de servicio, es decir, objetos no relacionados con el problema, pero que prestan servicio de soporte.
- **Almacenamiento**: un mecanismo persistente de almacenamiento, por ejemplo una base de datos.



JDeveloper guarda una carpeta por cada aplicación en: C:\jdevstudiobase10132\jdev\mywork (suponiendo que tu disco duro tenga asignada la letra C:). Para probar la aplicación que tienes en el siguiente enlace, tendrías que descomprimir el recurso y copiar la carpeta a dicho directorio. Y entonces, desde JDeveloper buscar esa carpeta y abrir el archivo NombreEdad.jws. Una vez incorporado en JDeveloper sólo habrá que ejecutar desde el proyecto vista, desde Principal.java.

Código fuente de la aplicación

Autoevaluación

1 De las siguientes afirmaciones referidas al patrón Modelo-Vista-Controlador, señala la correcta

a) Permite minimizar el impacto de los cambios en la interfaz sobre las clases del modelo.
b) No permite varias vistas simultáneamente sobre el mismo modelo.
c) El modelo constituye la representación de la GUI (Interfaz Gráfica de Usuario).
d) Al construir una nueva vista es necesario modificar el modelo subyacente.

Comprobar

Modelo de eventos

En el pasado, un programa que quisiera saber lo que estaba haciendo el usuario, debía recoger la información él mismo. En la práctica, esto significaba que una vez inicializado, el programa entraba en un gran bucle en el que estaba continuamente haciendo pasadas para comprobar si el usuario estaba haciendo algo interesante (por ejemplo, pulsar un botón, pulsar una tecla, mover una barra o mover el ratón) y tomar las acciones oportunas. Esta técnica se conoce como polling. Además, las interfaces de usuario eran bastante pobres, careciendo normalmente de gráficos, y siendo básicamente texto.

Hoy en día la práctica totalidad de aplicaciones informáticas poseen interfaces gráficas de usuario (GUI) para presentar y pedir información al usuario, y en definitiva para interactuar con él.

La programación con GUIs es orientada a eventos. La mayoría de los eventos son sucesos asíncronos producidos por la interacción del usuario con la aplicación, y están ligados a algún elemento de la interfaz. Algunos ejemplos son:

- Pulsar un botón.
- Cambiar el tamaño de una ventana.
- Mover una barra de desplazamiento.
- Pulsar una tecla.
- Tocar alguno de los botones minimizar-maximizar-cerrar de la ventana
- Hacer un click de ratón sobre un elemento determinado

En lugar de que el programa activamente recoja todos los eventos generados por el usuario, el sistema puede avisar al programa cuando se produce un evento de interés.

Todo sistema operativo que utiliza interfaces gráficas de usuario debe estar constantemente monitorizando el entorno para capturar y tratar los eventos que se producen. El sistema operativo informa de estos eventos a los programas que se están ejecutando, y entonces cada programa decide qué hace para dar respuesta a esos eventos.

El modelo de gestión de eventos es la forma en que se generan, capturan y tratan los eventos.

En el módulo de PLE también aprendimos mucho sobre el modelo de eventos de Java. Se hablaba bastante sobre ello, sobre todo en un apartado, que te recordamos en el siguiente recurso. Sería recomendable que repasaras ese apartado para refrescar tus conocimientos al respecto.

INFORMACIÓN COMPLEMENTARIA.
Eventos en Java
Enlace a la unidad PLE-18.
Apartados 7 y 8.
O bien...
Programación guiada por eventos

De modo muy resumido, podemos recordar que cada componente Swing puede generar eventos. De ese modo:

- Se definirán las acciones del usuario sobre el componente, cambios de estado, etc.
- En cada componente se pueden registrar oyentes o escuchadores de eventos.
- Cuando el componente genere un evento, invocará a sus manejadores de eventos.

El modelo de eventos de Java propone objetos fuente en los que mediante una serie de métodos se pueden registrar otros objetos como receptores. Así cuando ocurre un evento la fuente lo notifica a todos sus receptores pasando un objeto de tipo evento que almacena cierta información sobre el evento específico.

Así pues en la captura de eventos hay que tener en cuenta que hay tres objetos implicados:

- El **objeto fuente**. Es el objeto que lanza los eventos. Dependiendo del tipo de objeto que sea, puede lanzar unos eventos u otros. Por ejemplo un objeto de tipo JLabel (etiqueta) puede lanzar eventos de ratón (MouseEvent) pero no de teclado (KeyEvent).
- **El objeto oyente (listener).** Se trata del objeto que recibe el evento producido. Es el objeto que captura el evento y ejecuta el código correspondiente. Para ello debe implementar una interfaz relacionada con el tipo de evento que captura. Esa interfaz obligará a implementar uno o más métodos cuyo código es el que se ejecuta cuando se dispare el evento.
- **El objeto de evento**. Se trata del objeto que se envía desde el objeto fuente al oyente. Según el tipo de evento que se haya producido se ejecutará uno u otro método en el oyente.

Programación visual - Patrón modelo vista controlador

Oyentes de eventos

Cada tipo de evento tiene asociado un interfaz para manejar el evento. A esos interfaces se les llama escuchadores (Listeners) ya que proporcionan métodos que están a la espera de que el evento se produzca. Cuando el evento es disparado por el objeto fuente al que se estaba escuchando, el método manejador del evento se dispara automáticamente.

Por ejemplo, el método actionPerformed es el encargado de gestionar eventos del tipo ActionEvent (eventos de acción, se producen, por ejemplo, al hacer clic en un botón). Este método está implementado en el interfaz ActionListener (implementa oyentes de eventos de acción).

Cualquier clase que desee escuchar eventos (los suyos o los de otros objetos) debe implementar el interfaz (o interfaces) pensada para capturar los eventos del tipo deseado. Esta interfaz habilita a la clase para poder implementar métodos de gestión de eventos.

Por ejemplo, un objeto que quiera escuchar eventos ActionEvent, debe implementar la interfaz ActionListener. Esa interfaz obliga a definir el método ya comentado actionPerformed. El código de ese método será invocado automáticamente cuando el objeto fuente produzca un evento de acción.

Fuentes de eventos

El objeto fuente permite que un objeto tenga capacidad de enviar eventos. Esto se consigue mediante un método que comienza por la palabra "add" seguida por el nombre del interfaz que captura este tipo de eventos. Este método recibe como parámetro el objeto oyente de los eventos.

Esto es más fácil de lo que parece. Para que un objeto fuente, sea escuchado, hay que indicar quién será el objeto que escuche (que obligadamente deberá implementar el interfaz relacionada con el evento a escuchar). Cualquier componente puede lanzar eventos, sólo hay que indicárselo, y eso es lo que hace el método "add". Ejemplo:

```
public class UnaVentana extends JFrame implements ActionListener {
JButton boton1 = new JButton("Prueba");

// Constructor
public MiVentana() {
...

// El botón lanza eventos que son capturados por la ventana
boton1.addActionListener(this);
...
}
...
public void actionPerformed(ActionEvent e){
// Manejo del evento

}
}
```

En el ejemplo anterior se habilita al boton1 para que lance eventos mediante el método addActionListener. Este método requiere un objeto oyente que, en este caso, será la ventana en la que está el botón. Esta ventana tiene que implementar el interface ActionListener para poder escuchar eventos (de hecho el método addActionListener sólo permite objetos de este interface). Cuando se haga clic con el ratón se llamará al método actionPerformed de la ventana.

Hay que señalar que una misma fuente puede tener varios objetos escuchando los eventos (si lanza varios métodos add). Si hay demasiados objetos escuchando eventos, se produce una excepción del tipo TooManyListenersException.

Hay un método "remove" que sirve para que un oyente del objeto deje de escuchar los eventos. Así, podríamos hacer:

boton1.removeActionListener(this); que en el ejemplo provoca que la ventana deje de escuchar los eventos del botón.

Autoevaluación

1 Señala la afirmación correcta de entre las siguientes:

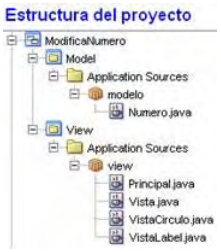
a) Cualquier clase que desee escuchar eventos (los suyos o los de otros objetos) debe implementar la interfaz (o interfaces) pensada para capturar los eventos del tipo deseado.
b) En las fuentes de eventos, el método "remove" sirve para que un oyente del objeto empiece a escuchar eventos.
c) Cada tipo de evento tiene asociado un interfaz para manejar el evento. A esos interfaces se les

llama fuentes de eventos.

d) Todas las respuestas anteriores son correctas.

Comprobar

Ejemplo de aplicación usando el patrón MVC: un número y diferentes vistas según su valor.



Veamos ahora otro ejemplo del patrón MVC en una aplicación de escritorio. Crearemos la aplicación que llamaremos ModificaNumero. A continuación tienes el enlace para descargarte el código fuente de la aplicación.

Aplicación ModificaNumero

La información, los datos de la aplicación, se podrán representar de muchas formas. Pero esos datos son los mismos para todas las vistas. Es decir, para un número, la información es la cantidad en sí y ese dato se puede representar de muchas formas: en forma de dígitos, como se muestra en una vista, o también en forma de un círculo. A mayor cantidad, el círculo será más grande.

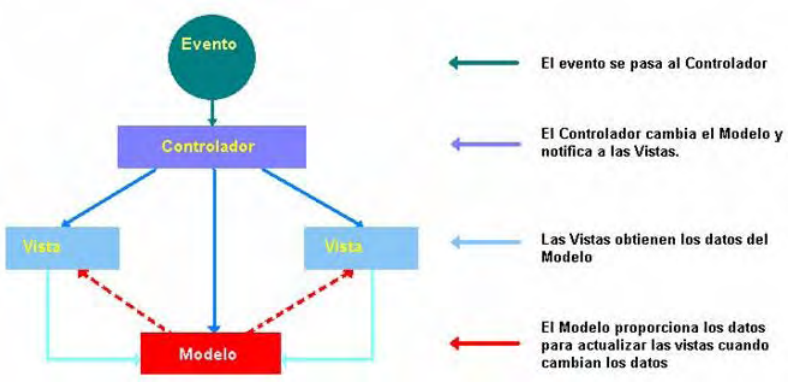
El modelo va a venir representado por esa clase que llamaremos Numero, y que es un JavaBean muy sencillo.

En el proyecto correspondiente a la Vista, encontramos el fichero fuente Principal.java, desde el que ejecutaremos la aplicación en JDeveloper. Aquí es donde se crea un formulario con un campo de entrada y un botón. Cuando se escriba cualquier número y se pulse en el botón, se disparará, el evento asociado a la pulsación del botón, que vamos a ver ahora:

```
// Evento que se dispara cuando se pincha en el botón
private void jButton1_actionPerformed(ActionEvent e) {
    // Obtener el contenido del campo de texto
    String texto = jTextField1.getText() ;
    // Establecer el contenido del campo de texto como el valor en el modelo
    Numero.setValor(texto) ;
    // Acceder al modelo para obtener los valores
   String valor = Numero.getValor() ;
    // Actualizar las vistas
    vL.actualizar(valor);
    vC.actualizar(valor);
}
```

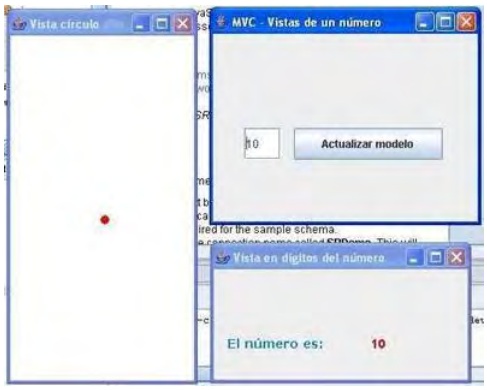
Podemos ver que la parte correspondiente al controlador de la aplicación está aquí incluida, pues coge la entrada del usuario, actualiza el modelo y luego procede a invocar los métodos actualizar() de las diferentes vistas de la aplicación.

Se podría representar esquemáticamente el uso del patrón MVC, para este ejemplo, como se puede observar en la imagen siguiente:

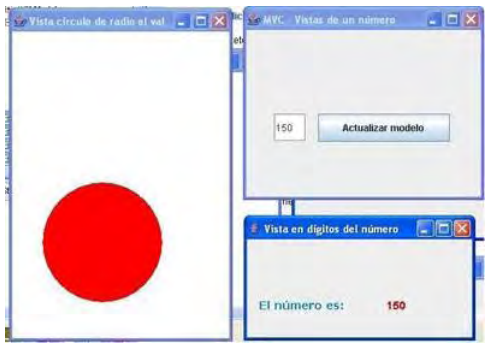


También encontramos en el proyecto que contiene la vista y el controlador, el interface Vista, que implementarán todas las vistas que queramos tener, y que consta de un método actualizar, que será el que se ejecutará como hemos visto, cuando haya un cambio en el modelo, y se tenga por tanto que actualizar las vistas.

El programa en ejecución se puede ve que presentaría el siguiente aspecto:



Posteriormente, cuando se modifique el valor del campo de entrada y se introduzca otro valor, por ejemplo 150, pinchamos en el botón para actualizar el modelo, y el aspecto será el de la siguiente imagen:



Para saber más:

En este enlace tienes un ejemplo de una aplicación de escritorio, utilizando el patrón MVC, más compleja y completa de las que hemos visto hasta ahora. Se trata de un juego de puzzle. Ejemplo de aplicación usando MVC. Juego puzzle:

http://www.chuidiang.com/java/puzzle/AppletPuzzle.php [Versión en cache]

En este otro enlace, que está en inglés, encontramos un ejemplo de uso del patrón MVC, en una aplicación de escritorio que muestra la temperatura utilizando diversas vistas. Muy instructiva, y muy recomendable que mires el código. Building Graphical User Interfaces with the MVC Pattern:

http://csis.pace.edu/~bergin/mvc/mvcgui.html [Versión en cache] ***(Versión en ingles)***

*CASO. A **Víctor** le gusta mucho el desarrollo de aplicaciones Web, y hasta ahora no había utilizado el patrón MVC en sus creaciones. Ahora ve claro, que empleando este patrón, va a poder estructurar mejor las aplicaciones y reutilizar en futuros desarrollos, gran parte de lo que haga a partir de ahora utilizando el patrón. Ahora ve claro, como va a poder ahorrar un montón de tiempo en sus desarrollos. Por ejemplo, el otro día estuvo mucho tiempo diseñando una vista web para introducir los datos de nuevos uusarios que se dan de alta eun una web, y como le quedó un diseño muy bonito, va a poder reutilizar ese diseño en muchas de las aplicaciones web que haga a partir de ahora. Por ello, Víctor está muy contento, puesto que con el tiempo que se va a ahorra, va a poder ir a la fiesta de cumpleaños de **Carmen**, cosa que dudaba hasta este momento.*

Se dijo anteriormente, que el patrón MVC surgió en aplicaciones de escritorio. Pero, ¿es en ese tipo de aplicaciones en las que más se usa?

Lo cierto es que donde más se utiliza es en el desarrollo de aplicaciones web.

Desde hace años, los desarrolladores de aplicaciones web han estado usando **HTML,** servlets y tecnología JavaServer Pages (JSP)para desarrollar interfaces de usuario basados en web.

Durante el desarrollo de una aplicación, en tanto en cuanto dicha aplicación va creciendo de tamaño y por ello va siendo más compleja de desarrollar, empiezan a surgir muchos desafíos a los que deben enfrentarse los desarrolladores. Por ejemplo, si se mezcla el código de programación de la lógica de negocio (el modelo) con el código de presentación (las vistas), se hará sumamente difícil desarrollar aplicaciones de cierto tamaño, que sean robustas y estén bien diseñadas, y por tanto sean fáciles de mantener.

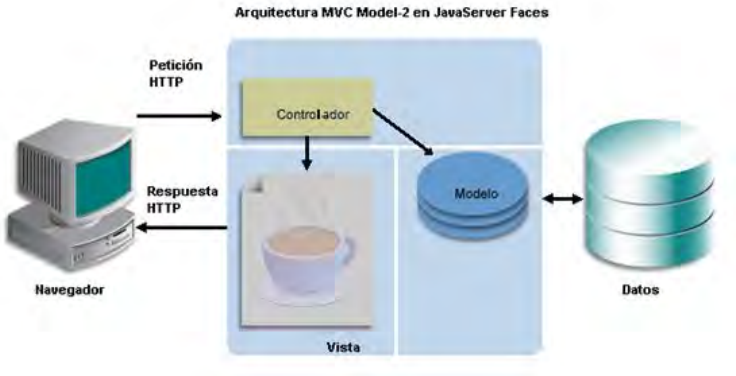
Para trabajar con la capa de presentación (las vistas) en el desarrollo de aplicaciones Web, se utilizaban servlets hasta la aparición de JSP. Las aplicaciones web sufren continuos cambios, y mantener diseñadores web con conocimientos de Java es muy costoso. Por esa razón surge JSP: debido a la necesidad de separar los roles de diseñador gráfico y desarrollador.

En el año 2004, apareció **JavaServer Faces**, una nueva tecnología en el ámbito de la capa de presentación de las aplicaciones web, que está teniendo cada vez más éxito e implantación.

JavaServer Faces (JSF) es un marco de trabajo para desarrollar interfaces de usuario para aplicaciones web estándar para Java. JSF está diseñado para desarrollar aplicaciones que se ejecuten en servidores

de aplicación Java y cuyos interfaces de usuario se representen en navegadores cliente.

ADF Faces (Application Development Framework), es la implementación de JSF que hace Oracle, obteniendo más componentes y más potentes, que la implementación de referencia de JSF.



ADF Faces implementa el patrón MVC. La mayoría de **frameworks web**, incluyendo ADF Faces, imponen alguna variación del patrón MVC. La variación Model-2 es una variación del patrón MVC específica para aplicaciones web. Los puntos principales son:

- El modelo puede constar de objetos Java "de toda la vida" (**POJOs**), **EJBs**, u otra cosa.
- La vista puede ser JSPs u otra tecnología de visualización.
- El controlador se implementa siempre como un servlet.

El controlador consiste en un servlet que es responsable de recibir peticiones de clientes web y desempeñar un conjunto lógico de pasos para preparar y despachar una respuesta. El modelo consiste en la lógica de negocio, los datos de la aplicación, y la vista puede ser JSPs o cualquier otra tecnología de visualización.

Por tanto, cuando en una aplicación de este tipo, un usuario hace clic en un botón que le aparece en alguna web en su navegador, ese clic genera un evento y ese servlet controlador captura el evento y lo reconduce hacia donde tenga que ir, despacha así la petición para que quien tenga que generar una respuesta al cliente, lo haga.

SERVLETS

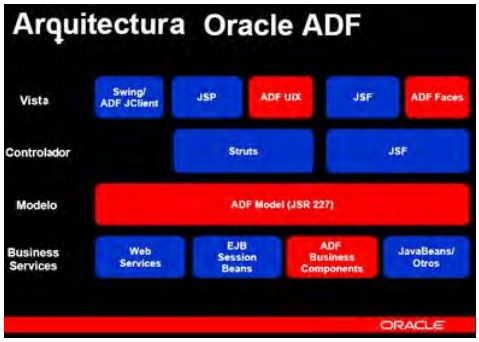
> ***Para saber más:***
> ***En el siguiente enlace podrás encontrar información sobre servlets , desde el principio y con ejemplos. Servlets (Básico):***
> ***http://www.programacion.net/java/tutorial/servlets_basico/*** [Versión en cache]
> ***Puedes encontrar más información sobre servlets y además sobre JSP en el enlace siguiente. Servlets y JSP:***
> ***http://www.programacion.net/java/tutorial/servlets_jsp/*** [Versión en cache]

Arquitectura Oracle ADF.

Acabamos de hablar anteriormente de ADF Faces, pero ¿dónde podemos enmarcar esta tecnología? Pues como vamos a ver de inmediato, ADF Faces forma parte de la arquitectura ADF. ¿Y qué es la arquitectura ADF?

La arquitectura de ADF (Application Development Framework), se basa en el patrón MVC, (Modelo - Vista - Controlador), que separa la programación de una aplicación en 3 capas. ADF se basa en estas tres capas y construye, en base a ellas, cuatro capas que conforman su arquitectura.

Se pueden observar las capas mencionadas en la imagen siguiente.



Las capas son:

- **Vista**: asociada a generar la interfaz de usuario que será expuesta a los usuarios que empleen el sistema.
- **Controlador**: responsable de gestionar el flujo de la aplicación y coordinar la interacción entre las capas de Modelo y Vista.
- **Modelo**: encargada de la gestión de los datos y el manejo de las reglas de negocio propias de la aplicación.
- **Business Services** (Servicios de Negocio). Esta capa se encarga de lograr la persistencia con la base de datos y alojar la lógica de negocio que requiera la aplicación. Una característica de usar esta capa, es que acepta diferentes tecnologías para su implementación. Es decir se puede emplear EJB, Business Components for Java (BC4J), u otra tecnología estándar para obtener la persistencia con la base de datos. En esta capa hay componentes para la administración de las transacciones con una fuente de datos objeto relacional

Como se puede apreciar, la arquitectura ADF de Oracle es bastante compleja y su estudio exhaustivo no es objeto de este curso. Pero podemos resumirla en que:

- En la capa vista se puede optar por utilizar multitud de tecnologías: Swing, JSP, ADF Faces, etc.
- Respecto al controlador a usar en una aplicación, puede utilizarse el que lleva por defecto o bien un controlador Struts.
- Respecto a la tecnología para alojar la lógica de negocio, se puede optar también por una gran variedad de tecnologías: Web Services (Servicios Web), EJB Session Beans (Beans de Sesión EJB), etc.

La tecnología ADF Faces constituye una librería de componentes JavaServer Faces de Oracle, que soporta una gran variedad de clientes que van desde navegadores web hasta unidades móviles.

> ***Para saber más:***
> ***En este enlace se hace una introducción a la arquitectura ADF de Oracle. También en ese sitio web, hay una gran cantidad de artículos y tutoriales sobre ADF, ADF Faces, etc. Oracle Application Development Framework (ADF):***
> ***http://www.jdeveloperla.com/joomla/index.php?option=com_content&task=view&id=38&Itemid=48*** [Versión en cache]

Elementos de ADF Faces.

Bueno, pues ya hemos hablado un poco de qué es ADF y también sobre ADF Faces. Pero, ¿exactamente de qué se compone, o qué elementos importantes conforman ADF Faces?

En el modelo de programación ADF Faces hay unos términos fundamentales, que habrá que tener en cuenta al desarrollar aplicaciones, como se aprecia en los siguientes puntos:

- **Componente de interface de usuario (UI), también llamado control o componente**: Un objeto con estado, mantenido en el servidor, que proporciona funcionalidad específica para interactuar con un usuario final. Los componentes UI son JavaBeans con propiedades, métodos y eventos. Están organizados en una vista, que es un árbol de componentes normalmente visualizado como una página.
- **Beans de respaldo (Backing beans):** son JavaBeans especializados en recoger valores de componentes UI e implementar métodos oyentes de eventos.
- **Eventos y oyentes**: se utiliza el modelo evento/oyente de JavaBeans y también utilizado por Swing. Los componentes UI (y otros objetos) generan eventos, y pueden registrarse oyentes para manejar esos eventos.
- **Navegación:** consiste en la habilidad de moverse de una página a la siguiente. La tecnología JavaServer Faces tiene un potente sistema de navegación que está integrado con oyentes de evento especializados.
- **Validadores:** responsables de asegurar que el valor introducido por un usuario es aceptable. Podrán asociarse a un componente UI, uno o más validadores.
- **Renderers**: responsables de mostrar un componente UI y traducir una entrada de usuario al valor del componente. Los renderers pueden ser diseñados para trabajar con uno o más componentes UI, y un componente UI se puede asociar con muchos renderers diferentes. En nuestro caso, trabajaremos con el render que viene por defecto.
- **Convertidores:** convierten el valor de un componente a un String y desde un String para visualizar.
- **Internacionalización**. JavaServer Faces, y por ello ADF Faces, cuenta entre sus características con la facilidad de internacionalizar las aplicaciones. De modo, que según la configuración del navegador del usuario y los ficheros que el desarrollador haya dispuesto con la información de la aplicación en el idioma que se quiera aportar, se mostrará una aplicación en el idioma del país del usuario que navegue por la aplicación web. En el para saber más del apartado que tienes más adelante, donde se describe la realización de una agenda telefónica, hay un tutorial sobre este asunto.

Como elementos fundamentales de la tecnología ADF Faces tenemos los JavaBeans. Se utilizan para crear los componentes de interface de usuario de ADF Faces. Como con los componentes Swing, cada componente JSF es un JavaBean en toda regla. Los componentes de ADF Faces están diseñados para trabajar con beans de respaldo.


Beans de respaldo

Un bean de respaldo define las propiedades y la lógica asociadas con los componentes de interface de usuario utilizados en una página. Cada propiedad del bean de respaldo está unida a un ejemplar de un componente o a su valor. Un bean de respaldo también define un conjunto de métodos que realizan funciones para el componente, como por ejemplo, validar los datos del componente, manejar los eventos que dispara el componente, y realizar el procesamiento asociado con la navegación entre páginas cuando el componente se activa.

A diferencia de componentes Swing, los componentes ADF Faces se ejecutan en el lado del servidor, no en el cliente.

Autoevaluación

1 Sobre ADF Faces, ¿qué afirmación es correcta?

a) Oracle ADF es una parte, o dicho de otro modo, forma parte de la arquitectura de ADF Faces.
b) Es exactamente lo mismo, un sinónimo, que Java Server Faces.
c) Los componentes ADF Faces se ejecutan en el lado cliente.
d) ADF Faces es un marco de trabajo para desarrollar interfaces de usuario para aplicaciones Web estándar para Java .

Comprobar

Ejemplo de aplicación web: bienvenida.

Esta primera aplicación, simplemente consta de una primera vista: index.jsp, donde el usuario debe identificarse. Si el usuario es "Aguadulce" y la contraseña es "1234", entonces pasa a otra vista: bienvenida.jsp, que visualiza un mensaje de bienvenida.

En el desarrollo de la misma se utiliza un **bean manejado**, para almacenar el nombre del usuario y la contraseña. Ese bean, que forma parte del modelo, carga los datos en la vista index.jsp, cuando el usuario teclea los datos. Si la identificación es correcta, se navega a la vista bienvenida.jsp, en la cual se utilizan los datos almacenados en el bean, para mostrar un mensaje de bienvenida.

También se puede apreciar cómo se valida la entrada del usuario, es decir, cómo se comprueba si el usuario y la contraseña son los esperados, asociando un método del bean manejado al atributo action del componente botón. Por eso, cuando se pulsa en el botón para entrar en la aplicación se ejecuta el método asociado al action y por tanto se valida. Esa validación devuelve un String "bien" si es correcta o "mal" si es incorrecta. Entonces, por las reglas de navegación definidas en el fichero faces-config.xml se procede a acceder a la página de

bievenida si todo fue bien.

En la presentación que tienes a continuación, puedes ver los pasos en su construcción.

Construir el proyecto bienvenida

Proyecto de bienvenida

Ejemplo de aplicación web: agenda telefónica. Introducción.

Para entender los conceptos fundamentales que hemos comentado, ¿qué te parece si construimos una pequeña aplicación web, consistente en una agenda telefónica?.

Se podrá consultar el listado de contactos que tenemos en ella, añadir, editar y eliminar contactos.

En primer lugar, hay que comentar que JDeveloper dispone de varias plantillas para crear aplicaciones web. Cuando creamos nuestra aplicación elegimos como plantilla: Web Application [JSF, EJB, TopLink]. De esta manera, la herramienta crea automáticamente dos proyectos: Model y ViewController. En el proyecto del modelo estará la información relativa a los datos, mientras que en la vista se alojarán las vistas necesarias para visualizar la agenda con los componentes que se quieran.

En el proyecto Model almacenaremos la parte correspondiente al modelo de la aplicación. En este caso, en la agenda, nos tenemos que preguntar qué datos son los que nos va a interesar guardar. Decidimos que en el modelo, los datos que vamos a guardar sobre cada contacto de la agenda serán:

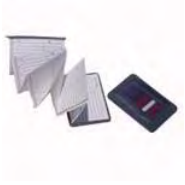
- un identificador,
- el nombre,
- los apellidos y
- el teléfono.

Por ello, creamos un JavaBean, en Nombre.java. Aquí vemos parte del código, se trata de métodos set y get para cada uno de los atributos.

Tenemos que observar también en el código que esta clase implementa un interface. Un objeto serializable es un objeto que se puede convertir en una secuencia de bytes. Para que un objeto sea serializable, debe implementar el interface java.io.Serializable. Este interface no define ningún método. Simplemente se usa para marcar aquellas clases cuyas instancias pueden ser convertidas a secuencias de bytes (y posteriormente reconstruidas).

```
package model;
public class Nombre implements Serializable {
    private String id;
    private String nombre;
    private String apellidos;
    private String telefono;
    public Nombre() {
    }
    public void setId(String newValue) {
        id = newValue;
    }
    public String getId() {
        return id;
    }
    public void setNombre(String newValue) {
        nombre = newValue;
    }
    public String getNombre() {
        return nombre;
    }
       ...
}
```

Para guardar la agenda utilizaremos un fichero. Así cada vez que se inicie la aplicación, se cargarán los contactos que haya en el fichero. En ese proceso de carga, se irán leyendo cada uno de los elementos de la agenda y se irán cargando en una lista. Podemos utilizar por ejemplo una colección, una lista LinkedList. Posteriormente, cuando creemos la vista en la que visualicemos la agenda, asociaremos el contenido de esa lista con el componente que queramos que nos muestre los contactos de la agenda.



Por esa razón, tendremos una clase, también en el modelo, a la que hemos llamado Datos, que va a contener lo necesario para añadir elementos a esa colección, editarlos, eliminarlos, etc, así como los métodos para leer de disco y grabar a disco la agenda. El código de esta clase y de toda la aplicación, lo tienes en un enlace más adelante casi al final de este mismo apartado, pero destacamos de entre todos, un método de esta clase:

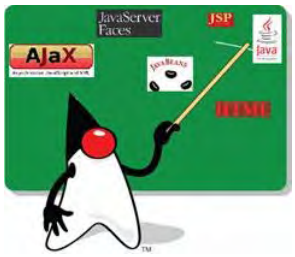
```
// Devuelve la colección con todos los elementos de la agenda
public java.util.Collection getAll() {
    return personas;
}
```

Como se ve, este método simplemente devuelve la colección de personas. Esa colección se rellena al leer la agenda del disco, que es lo que har

Creando el proyecto Agenda Telefónica

Ejemplo de aplicación Web: agenda telefónica. Creando vistas.

Ahora, podríamos pasar a realizar la vista que nos muestre los contactos, así como los botones necesarios para guardar la agenda, dar de alta un elemento, editar un elemento, etc.



En este punto, comentamos que la tecnología JavaServer Faces, y por tanto ADF Faces, permiten utilizar multitud de tecnologías para la capa de presentación. Por ello, podríamos utilizar HTML, JSP, etc. Por defecto, en la implementación de referencia JavaServer Faces de Sun, se emplea JSP, que también en los asistentes de Oracle es la opción por defecto. Haremos lo propio, y utilizaremos ficheros .jsp para cada vista que necesitemos.

Creamos la vista que llamaremos listado.jsp, obviamente en el proyecto ViewController. Como hemos dicho a lo largo del tema, el modelo es el que tenga que ser, que en este caso ya lo hemos definido también, y para ese modelo podemos tener diversas vistas. Nos vamos a decantar para ver la agenda, por elegir un componente tabla, que es el más adecuado probablemente, para visualizar cualquier lista de elementos, sea una agenda de personas, o cualquier otra lista.

En el recurso denominado **Construir la vista listado.jsp** que tienes más abajo, se muestra cómo construir esta vista, siguiendo el asistente que aporta JDeveloper, siendo muy sencilla de realizar. Pero debemos tener en cuenta antes de proceder a realizar dicha vista dos cosas:

- **Asegurarnos de que la vista pueda acceder al modelo**, tal y como ya vimos al construir aplicaciones de escritorio, en las dependencias del proyecto ViewController, indicando que tenga acceso al proyecto que contiene al modelo.
- **Tenemos que crear un bean**, en el caso de la nomenclatura JavaServer Faces hablamos en este caso de bean manejado, **que permita tener disponible la colección de contactos de la agenda**, durante todo el tiempo que el usuario esté navegando en su sesión.

En JavaServer Faces, se puede especificar qué objetos estarán disponibles a lo largo del ciclo de vida de la aplicación. En las aplicaciones Java web tradicionales, disponemos de objetos que se crean con unas etiquetas determinadas en jsp.

En las aplicaciones JSF, se puede configurar e inicializar los beans en el fichero de configuración: faces-config.xml, de modo que se creará el bean (si no existía antes), se inicializa y se almacena en el ámbito que se le indica. Es lo que se denomina un bean manejado o managed bean.

Así por ejemplo, como a nosotros nos interesa tener disponible la colección con la lista de contactos, disponible en toda la sesión durante la que estaremos navegando, tenemos que definir en el fichero faces-config.xml como un bean manejado:

```
<managed-bean>
    <managed-bean-name>DatosBean</managed-bean-name>
    <managed-bean-class>model.Datos</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

En las etiquetas anidadas que acabamos de ver:

- La primera etiqueta declara un nombre para referenciar al bean.
- La siguiente línea indica la clase que implementa el JavaBean.
- La siguiente se refiere al ámbito en que se almacena el bean, que en este caso es en la petición, o sea, la sesión.

El fichero faces-config.xml es un fichero fundamental en toda aplicación JSF. Constituye el controlador de la aplicación, ya que en él se

almacenan como hemos visto los beans manejados, y otras cosas, como las reglas de navegación, que determinarán a qué página se debe navegar desde una página dada.

Como puede verse en su extensión, se trata de un fichero XML. Observamos en el contenido del fichero, que respecto a su sintaxis, todo elemento que se abre, se cierra con el mismo nombre precedido de la barra (/). Por ejemplo, para declarar un bean manejado, abrimos la declaración con la etiqueta y al final cerramos la declaración con

En este momento habrá que **decidir cómo queremos ver la lista de contactos** de la agenda. Elegiremos usar una tabla, de modo que cada contacto se muestre en una fila. Así, cuando tengamos rellena la tabla con algunos datos, el aspecto que presentará la agenda en ejecución será el de la imagen que acompaña a este texto. Puede observarse que en la tabla, además del los datos de las personas, se ha añadido un botón de radio para permitir la selección de un contacto, de cara a poder modificarlo o eliminarlo.

Construir la vista listado.jsp

Ejemplo de aplicación web: agenda telefónica. Vistas para editar y para dar de alta.

Procedemos ahora a crear la vista para dar de alta un contacto (alta.jsp). Una vez que la creemos, tendremos que asociar la pulsación del botón de añadir contacto, que pusimos en la página de listado de contactos, para que navegue desde listado.jsp hasta alta.jsp

Construir la vista alta.jsp

Tendremos una vista parea editar un contacto que seleccionemos. Si nos fijamos en la vista edicion.jsp, en modo diseño, en JDeveloper, veremos que en los campos de texto utilizamos un bean manejado denominado seleccionado. Está definido en el fichero faces-config.xml:

```
<managed-bean>
    <managed-bean-name>seleccionado</managed-bean-name>
    <managed-bean-class>model.Nombre</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
```

¿Para qué se utiliza ese bean? Para pasar información de una página a otra. Concretamente, cuando en la vista listado.jsp, se seleccione un contacto y se pinche en el botón Editar, el método que se va a ejecutar es el siguiente:

```
public String cmdEditar_action() {
// Si hay alguna fila seleccionada

if  (table1.getSelectedRowData() != null){
   // Obtener la fila  seleccionada
   Nombre nom =  (Nombre)table1.getSelectedRowData();

   // Obtenemos el  contexto
   FacesContext context =  FacesContext.getCurrentInstance();
   Application app = context.getApplication();

   // Establecer en  el bean de sesión el valor para cogerlo en la siguiente pantalla
   app.createValueBinding("#{seleccionado}").setValue(context,  nom);
   // Retornamos esto para que la regla de navegación  nos lleve a la vista de edición
   return  "editar";
   }
   else
   return null ;
   }
```

Ahí podemos ver cómo recogemos el contacto seleccionado actualmente en la tabla y establecemos el valor del bean manejado seleccionado, justamente con ese objeto contacto. De este modo, al cambiar de página, el bean sigue disponible y por tanto al acceder a la vista edicion.jsp, los campos de datos del contacto aparecerán con los valores cargados.

Se puede observar también en el código anterior cómo se obtiene el contexto de la aplicación. Cuando se está desarrollando una aplicación y se hace un arrastrar y soltar, JDeveloper crea metadatos para asociar el modelo y los componentes JSF. En el código anterior vemos cómo a través de ese contexto de la aplicación podemos acceder a los datos de la fila seleccionada actualmente.

Como habrás visto cuando navegas por Internet, la página inicial de cualquier sitio web suele ser una página llamada index, que suele denominarse index.htm o index.jsp, etc. En nuestro caso construiremos una index.jsp que simplemente lo que hará será dirigirse a la vista listado.jsp que ya hemos construido:

```
<?xml version='1.0' encoding='windows-1252'?>
<jsp:root  xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
  <jsp:scriptlet>
       response.sendRedirect("faces/listado.jsp");
   </jsp:scriptlet>
</jsp:root>
```

 Construir la vista index.jsp

El proyecto completo lo tienes en el siguiente enlace:

Aplicación Agenda Telefónica

Para saber más:

JavaServer Faces cuenta entre sus características, con la facilidad de internacionalizar las aplicaciones, como ya se ha comentado. En este enlace se describe paso a paso cómo hacer una aplicación con estas características. JSF - Internacionalización:

http://www.jdeveloperla.com/joomla/index.php?option=com_content&task=view&id=118&Itemid=103 [Versión en cache]

Lista de componentes que podemos utilizar en ADF Faces, con una imagen (en los componentes representables), y con la etiqueta que se utiliza en el fichero fuente .jsp. La página está en inglés y es del sitio web de Oracle. Tag library information:

http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/jsf/doc/tagdoc/core/imageIndex.html imageIndex.html [versión en cache]

Sabías que...

Los conceptos de la programación orientada a objetos tienen origen en Simula 67 [Versión en cache]***, un lenguaje diseñado para hacer simulaciones, creado por Ole-Johan Dahl*** [Versión en cache] ***y Kristen Nygaard*** [Versión en cache] ***del Centro de Cómputo Noruego en Oslo*** [Versión en cache]***. Trabajaban en simulaciones de naves, y pensaron en agrupar los diversos tipos de naves en diversas clases de objetos, siendo responsable cada clase de objetos de definir sus propios datos y comportamiento.***

Lecturas recomendadas:

"Introducción a la tecnología JavaServer Faces." Programación en castellano. Tutorial introductorio sobre JavaServer Faces, con un ejemplo de aplicación

"JavaServer Faces in Action". Kito D. Mann. Manning Publications Co. Estudia qué es JavaServer Faces, cómo funciona y comenta similitudes y diferencias con otros frameworks y tecnologías como Struts, Servlets, Portlets, JSP, y JSTL. Se implementa un caso de estudio completo a lo largo del libro. Está en inglés.

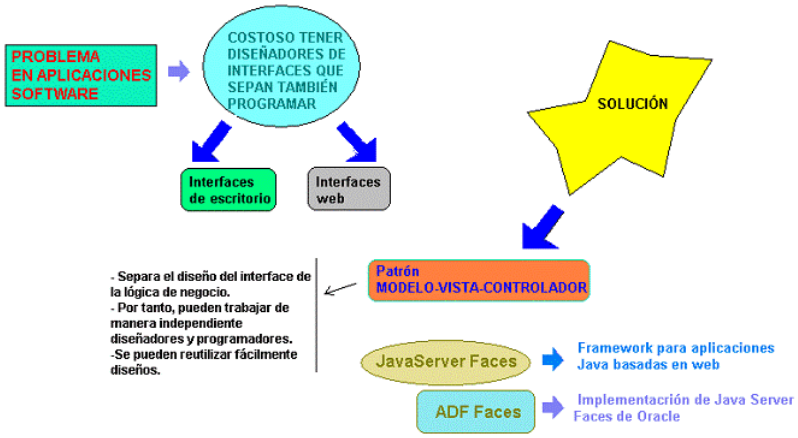
"Processing file uploads".

Oracle tecnology network [versión en cache]. Artículo de Oracle que explica cómo utilizar el componente de ADF Faces que sirve para subir archivos.

"Core JavaServer Faces". David Geary & Cay Horstmann. Editorial Addison Wesley. Estudia JavaServer Faces haciendo especial hincapié en cómo integrarlo con bases de datos. Muestra también cómo crear nuevas librerías de etiquetas, y cómo usar buenas prácticas y hábitos de programación con JSF. Está en inglés.

Mapa Conceptual.

Haz clic sobre la imagen siguiente para ver a pantalla completa los conceptos fundamentales de la unidad en forma de mapa conceptual.


PROBLEMA EN APLICACIONES SOFTWARE
COSTOSO TENER DISEÑADORES DE INTERFACES QUE SEPAN TAMBIÉN PROGRAMAR
Interfaces de escritorio
Interfaces web
SOLUCIÓN
- Separa el diseño del interface de la lógica de negocio.
- Por tanto, pueden trabajar de manera independiente diseñadores y programadores.
-Se pueden reutilizar fácilmente diseños.
Patrón MODELO-VISTA-CONTROLADOR
JavaServer Faces
Framework para aplicaciones Java basadas en web
ADF Faces
Implementacrión de Java Server Faces de Oracle