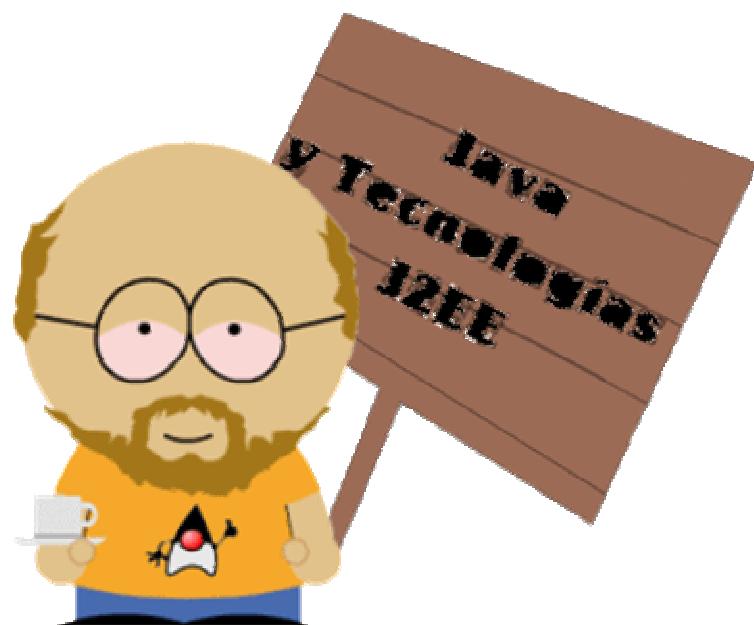


Curso Java y Tecnologías Java EE



Juan José Meroño Sánchez
Plan Formación PAS 2009
Universidad de Murcia

ÍNDICE

| | |
|---|-----------|
| ÍNDICE | 2 |
| PREFACIO | 4 |
| TEMA 1: HERRAMIENTAS DE DESARROLLO: ECLIPSE | 5 |
| 1.1.- INTRODUCCIÓN..... | 5 |
| 1.1.1.- <i>Componentes Eclipse</i> | 5 |
| 1.1.2.- <i>Espacio de Trabajo</i> | 7 |
| 1.1.3.- <i>Configuración de Eclipse</i> | 7 |
| 1.1.4.- <i>Actualización de Eclipse</i> | 7 |
| 1.2.- ARTEFACTOS DE DESARROLLO Y RUNTIME..... | 8 |
| 1.2.1.- <i>Artefactos de Desarrollo</i> | 8 |
| 1.2.2.- <i>Artefactos de Runtime</i> | 8 |
| 1.3.- AYUDANDO A ECLIPSE | 8 |
| 1.3.1.- <i>Subversion</i> | 8 |
| 1.3.2.- <i>Maven</i> | 8 |
| 1.3.3.- <i>Servidor de Integración Contínua</i> | 10 |
| TEMA 2: LENGUAJE JAVA Y POO..... | 11 |
| 2.1.- PROGRAMACIÓN ORIENTADA A OBJETOS..... | 11 |
| 2.1.1.- <i>Introducción</i> | 11 |
| 2.1.2.- <i>Conceptos Fundamentales</i> | 12 |
| 2.2.- PROGRAMACIÓN ORIENTADA A ASPECTOS..... | 14 |
| 2.2.1.- <i>Conceptos Básicos</i> | 14 |
| 2.3.- LENGUAJE JAVA | 16 |
| 2.3.1.- <i>Estructura General</i> | 16 |
| 2.3.2.- <i>Programación Java</i> | 18 |
| 2.3.3.- <i>Clases en Java</i> | 21 |
| 2.3.4.- <i>Excepciones</i> | 24 |
| 2.3.6.- <i>Novedades Lenguaje Java 5</i> | 27 |
| TEMA 3: PATRONES DE DISEÑO WEB | 30 |
| 3.1.- INTRODUCCIÓN..... | 30 |
| 3.1.1.- <i>Patrones Creacionales</i> | 30 |
| 3.1.2.- <i>Patrones Estructurales</i> | 31 |
| 3.1.3.- <i>Patrones Comportamiento</i> | 32 |
| 3.2.- PATRONES WEB | 34 |
| 3.2.1.- <i>Capa de Presentación</i> | 34 |
| 3.2.2.- <i>Capa de Negocio</i> | 35 |
| 3.2.3.- <i>Capa de Integración</i> | 36 |
| 3.3.- PATRÓN MVC | 36 |
| 3.3.1.- <i>Descripción del Patrón</i> | 36 |
| 3.3.2.- <i>Extensiones del Patrón</i> | 38 |
| 3.3.3.- <i>Inversión de Control/Inyección de Dependencias</i> | 39 |
| TEMA 4: ARQUITECTURA JAVA EE | 40 |
| 4.1.- MODELO DE CAPAS | 40 |
| 4.1.1.- <i>Tipos de Contenedores</i> | 41 |
| 4.1.2.- <i>Servicios Java EE</i> | 41 |
| 4.1.3.- <i>Ensamblado y Empaquetado</i> | 42 |
| 4.2.- ECLIPSE Y JAVA EE | 44 |
| 4.2.1.- <i>Proyectos Java EE</i> | 44 |
| 4.2.2.- <i>Desarrollo y Ejecución</i> | 47 |
| TEMA 5: TECNOLOGÍAS JAVA EE | 48 |
| 5.1.- TECNOLOGÍAS VISTA: JSF..... | 48 |

| | |
|--|------------|
| <i>5.1.1.- Introducción</i> | 48 |
| <i>5.1.2.- Ciclo de Vida JSF.....</i> | 50 |
| <i>5.1.3.- Componentes JSF</i> | 53 |
| <i>5.1.4.- El fichero faces-config.xml</i> | 58 |
| <i>5.1.5.- Facelets.....</i> | 59 |
| <i>5.1.6.- RichFaces</i> | 59 |
| 5.2.- TECNOLOGÍAS CONTROL: EJB | 61 |
| <i>5.2.1.- Introducción</i> | 61 |
| <i>5.2.2.- Tipos de EJB.....</i> | 64 |
| <i>5.2.3.- Ciclo de Vida</i> | 65 |
| <i>5.2.4.- Interceptores.....</i> | 66 |
| <i>5.2.5.- Anotaciones</i> | 70 |
| 5.3.- TECNOLOGÍAS MODELO: JPA..... | 71 |
| <i>5.3.1.- Introducción</i> | 71 |
| <i>5.3.2.- Beans de Entidad</i> | 72 |
| <i>5.3.3.- Anotaciones Básicas</i> | 74 |
| <i>5.3.4.- Anotaciones del Ciclo de Vida.....</i> | 80 |
| TEMA 6: TECNOLOGÍAS AVANZADAS JAVA EE | 82 |
| 6.1.- SERVICIOS WEB | 82 |
| 6.2.- AUTENTICACIÓN JAVA EE..... | 84 |
| <i>6.2.1.- Roles de Seguridad</i> | 84 |
| <i>6.2.2.- Restricciones de Seguridad.....</i> | 84 |
| <i>6.2.3.- Usuarios, Grupos y Realms</i> | 85 |
| 6.3.- PORTALES Y PORTLETS | 87 |
| <i>6.3.1.- Definición de Portlet</i> | 87 |
| APÉNDICE A: DIARIO DIGITAL..... | 89 |
| APÉNDICE B: CONFIGURAR PROYECTO ECLIPSE | 90 |
| B.1.- CREAR PROYECTO JAVA EE | 90 |
| B.2.- CONFIGURAR FACELETS Y RICHFACES | 90 |
| B.3.- CONFIGURAR HIBERNATE JPA Y HSQLDB..... | 95 |
| B.4.- CONFIGURAR SERVICIOS WEB | 97 |
| APÉNDICE C: TRANSPARENCIAS DEL CURSO | 102 |

Prefacio

El objetivo principal de este curso es divulgar el conocimiento de Java y de las tecnologías empleadas hoy día para el desarrollo de aplicaciones en entorno web. Puede considerarse como una introducción a los aspectos que se abordan en los cursos de desarrollo de FundeWeb.

El curso tendrá una parte teórica fuerte, que considero bastante importante para manejar algunos conceptos fundamentales a la hora de entender y abordar el desarrollo de aplicaciones web con tecnologías Java.

Muchas partes del contenido del curso están extraídas directamente del curso de FundeWeb por lo que siempre me remitiré al material publicado en la web de metodologías para dicho curso cuando se quiera ampliar conocimientos, aunque en principio el objetivo es que este manual sirva de referencia básica y suficiente.

Si tu intención es ponerte a trabajar con estas tecnologías, es necesario que te leas los manuales del curso de FundeWeb y las guías técnicas desarrolladas por el equipo de metodologías, si quieres tener una idea aproximada sin entrar en demasiado detalle este puede ser un buen punto de partida.

Tema 1: Herramientas de Desarrollo: Eclipse

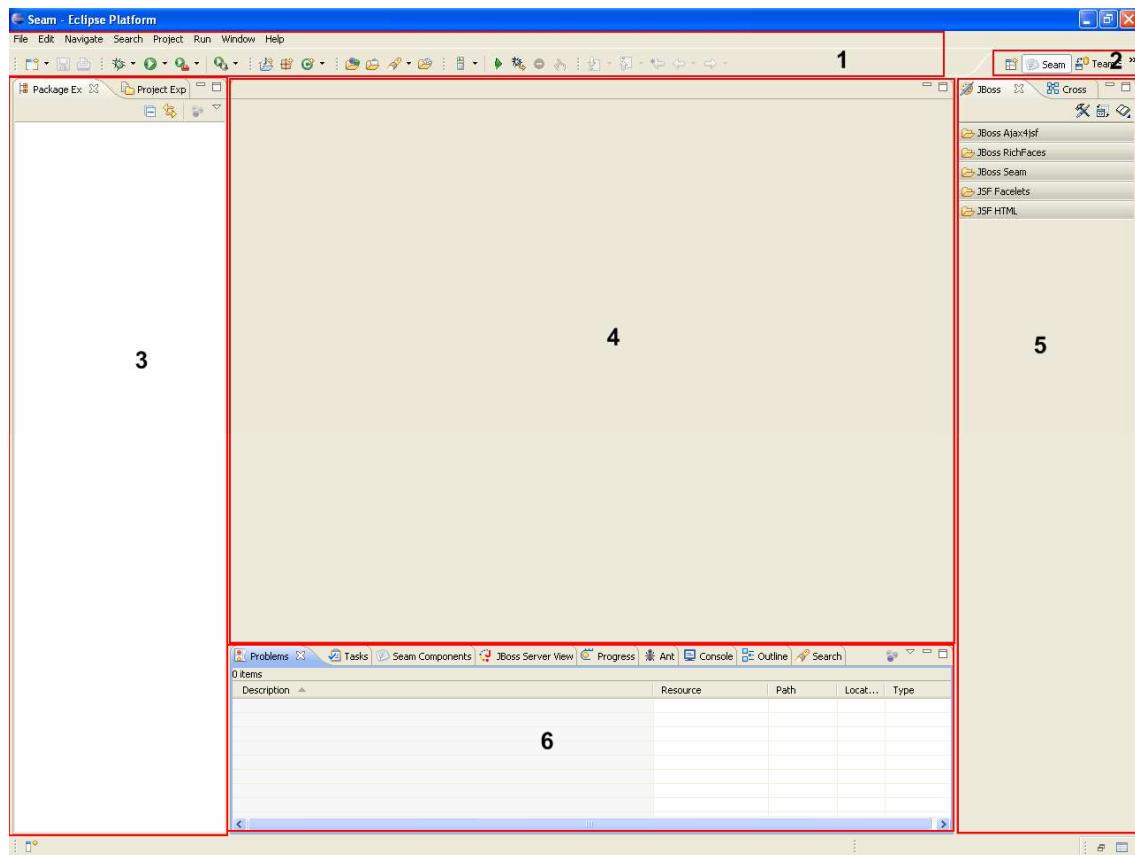
1.1.- Introducción

Existen multitud de entornos de desarrollo para Java, cada compañía con cierto peso en la tecnología Java dispone del suyo propio, JDeveloper (Oracle), NetBeans (Sun), Eclipse (IBM), etc. Todos son buenos incluso mejores que el propio Eclipse, pero este es sin duda el más ligero y adaptable. Como iremos viendo posteriormente el IDE no es un factor determinante, sólo cuando el desarrollador no es capaz de adaptarse a él le entorpece su tarea.

En un IDE vamos a buscar que facilite la edición de los archivos, la localización de errores y las tareas generales durante el desarrollo. Debe ser fácil de usar para que los desarrolladores no encuentren dificultades de adaptación, pero lo suficientemente potente como para no necesitar otras herramientas al margen.

1.1.1.- Componentes Eclipse

Vamos a describir brevemente el entorno de desarrollo de Eclipse a través de la siguiente figura:



- Zona 1: Menú -

En esta zona se incluye el menú de opciones completo, y algunos botones de acceso directo a determinadas opciones de uso común. Como el resto de las zonas es totalmente configurable por parte del usuario para adaptarla a sus necesidades.

A través de esta Zona conseguimos crear nuevos proyectos, importar o exportar información de nuestros proyectos y cualquier otra opción relativa a la gestión del proyecto y de sus archivos. En otras ocasiones no acudiremos al menú de esta zona puesto que eclipse permite acceder a diferentes opciones de manera más directa mediante los menús contextuales.

- Zona 2: Perspectiva -

En esta zona aparecen de una u otra forma las distintas perspectivas abiertas por el usuario. Una perspectiva es una configuración específica de todas las zonas destinada a ver un proyecto desde un punto de vista determinado.

Por ejemplo, si creamos un proyecto Java, disponemos de la perspectiva “Java” que nos muestra las clases java agrupadas por paquetes, y nos oculta los archivos compilados. Si ese mismo proyecto lo abrimos con la perspectiva de “Recursos” veremos todos los archivos del proyecto incluidos los compilados .class, pero no veremos las clases agrupadas por paquetes.

En definitiva esta zona me va a permitir cambiar de perspectiva dentro de mi proyecto, para visualizar diferentes puntos de vista del mismo proyecto. En algunos casos algunas perspectivas como las de Base de Datos o SVN se alejan algo del proyecto en sí, ya que muestran conexiones a BBDD o Repositorios SVN y no archivos de nuestro proyecto.

Qué se ve y cómo en cada perspectiva es configurable por el usuario, por lo que es interesante conocer que podemos resetear una perspectiva para dejarla como estaba inicialmente.

- Zona 3: Control Proyecto -

En realidad, esta es una genérica, pero que por convención se suele utilizar para navegar por los archivos del proyecto. En esta zona se pueden incluir Vistas de Eclipse. Una vista es una pestaña que muestra cierta información, tenemos multitud de vistas que muestran desde los ficheros contenidos en el proyecto, hasta los errores de compilación, mensajes de consola, monitores de TCP, etc...

Existen tres vistas principalmente empleadas para visualizar los componentes del proyecto: Project Explorer, Package Explorer y Navigator. Las tres muestran los archivos contenidos en el proyecto pero de diferente forma. Por lo que suelen aparecer en diferentes vistas del proyecto.

Como todo en eclipse uno puede incluir más vistas o cambiar las vistas que tiene disponibles en cada zona sin más que utilizar el botón de la parte inferior izquierda del editor o a través del menú de Eclipse. Arrastrando las pestañas podemos colocarlas en la posición de la zona que más nos interese.

- Zona 4: Editor -

Esta es la zona normalmente dedicada a editar el contenido de los archivos del proyecto. Eclipse dispone de diversos editores dependiendo de los plugins que tengamos instalados en el IDE. Generalmente encontraremos editores de texto, pero hay editores gráficos para archivos XML, HTML, JSP, etc.

Normalmente trabajaremos con editores textuales, pero en algunos casos puede resultar muy cómodo tener una representación visual del fichero que estamos editando.

- Zona 5: Propiedades -

Esta es otra zona dedicada a mostrar vistas y que convencionalmente se emplea para situar las vistas dedicadas a visualizar/editar las propiedades de los archivos editados en la zona central.

- Zona 6: Control Ejecución -

Esta es una zona equivalente a la zona 5 pero que se suele emplear para ubicar aquellas vistas relacionadas con el control de la ejecución del proyecto como son las vistas de consola, para ver los log's de ejecución, vistas de errores, problemas, tareas, etc.

1.1.2.- Espacio de Trabajo

Eclipse utiliza el espacio de trabajo denominado “Workspace”, para agrupar los diferentes proyectos que maneja. Podemos disponer de múltiples Workspaces en un mismo Eclipse, y dentro de estos múltiples proyectos.

Es una forma muy cómoda de mantener separados diversos proyectos en los que se trabaja para que el trabajar con unos no entorpezca a los otros. El desarrollador puede abrir tantas instancias de Eclipse con Workspaces diferentes como desee. Además puede cambiarse de espacio de trabajo en cualquier momento “File ▶ Switch Worspace”.

1.1.3.- Configuración de Eclipse

Para configurar Eclipse utilizamos la opción “Window ▶ Preferences”, desde la cual podemos actuar sobre las configuraciones de Eclipse:

- **General:** Opciones generales del entorno, apariencia, worspace, etc.
- **Java:** Opciones de compilación, entorno jre instalado, etc.
- **Run/Debug:** Opciones relativas a la ejecución, vistas de consola, etc.

Aparecen muchas otras opciones que van a depender de los plugins instalados en el editor, por ejemplo para manejar opciones de servidores, editores XML, Web, opciones para JPA, y un larguísimo etcétera.

1.1.4.- Actualización de Eclipse

Una de las opciones más interesantes de Eclipse es la posibilidad de actualizar y ampliar las funcionalidades del mismo. Podemos ampliar el IDE para incluir perspectivas, vistas, editores, etc, de forma que nos ayude a manejar nuestros proyectos de la forma más adecuada.

Para actualizar eclipse basta con acudir al menú “Help ▶ Software Updates”. Desde aquí podemos actualizar funcionalidades que tengamos instaladas, o ampliar buscando nuevas. Los desarrolladores de plugins ofrecen URL's que permiten instalar sus funcionalidades desde esta opción.

Esta es una de las características más interesantes de Eclipse, ya que nos permite tener el IDE minimizado, es decir, manteniendo activas sólo con aquellas funcionalidades que realmente nos interesan, descartando otra serie de funcionalidades que no queramos usar.

En otros entornos se precargan editores visuales Swing para desarrollo de aplicaciones Java en modo escritorio que suelen ser pesadas y que no necesitaremos si desarrollamos web.

1.2.- Artefactos de Desarrollo y Runtime

En este apartado veremos los diferentes elementos que nos ayudan a gestionar un proyecto. En primer lugar para desarrollarlo y después para ejecutarlo y probarlo. La mejor forma de descubrir estos artefactos es hacerlo de forma práctica, y es lo que haremos a lo largo del curso.

1.2.1.- Artefactos de Desarrollo

Son los artefactos empleados en la edición de nuestro proyecto, principalmente los editores de archivos y los de propiedades. Se suelen distribuir en las zonas 3, 4 y 5. El artefacto más importante obviamente es el propio editor de archivos y el explorador que nos permite navegar por los archivos del proyecto.

1.2.2.- Artefactos de Runtime

Son los artefactos empleados para el control de la ejecución. Nos permiten crear un entorno de ejecución, visualizar los log's, hacer debug de la ejecución, etc. Se suelen ubicar en la zona 6. En el caso del Debug existe una perspectiva entera dedicada a esta tarea, en la que se entremezcla la visión del editor de código con el valor de variables o expresiones durante la ejecución.

1.3.- Ayudando a Eclipse

No siempre un desarrollador está sólo en la gestión de un proyecto y aunque lo esté no tiene porqué gestionarlo de por vida. Surge la necesidad de emplear otras herramientas al margen del propio IDE pero que se integran perfectamente con esta.

1.3.1.- Subversion

Mediante Subversion vamos disponer de un repositorio de control de versiones que podemos gestionar desde el propio IDE para realizar todas las tareas relativas al control de versiones.

El control de versiones nos va a servir para mantener salvaguardado el código y para gestionar el desarrollo entre varios desarrolladores en un mismo proyecto. También nos va a ayudar a liberar versiones de nuestras aplicaciones de forma controlada.

Eclipse dispone de un plugin denominado Subclipse que integra todas las opciones realizables desde el menú contextual “Team”.

1.3.2.- Maven

Esta es una herramienta muy útil para gestionar las dependencias en proyectos grandes. En determinados entornos puede ser suficiente con el uso de subversion, si mantengo una rama con el proyecto de un determinado IDE, cualquier desarrollador que

se incorpore al proyecto es capaz de ponerse a funcionar muy rápido, haciendo un checkout de dicha rama y utilizando el mismo IDE.

En otros casos puede ser interesante que se puedan ejecutar todas las tareas del ciclo de vida de un proyecto de forma autónoma. Maven me aporta esto. Mediante ficheros de configuración denominados POM, puedo describir las tareas a ejecutar en cada una de las fases del ciclo de vida de un proyecto (compilación, test, despliegue,...), y qué librerías son necesarias para cada fase.

Maven de forma automática se descarga las librerías exactas que son necesarias y con las que se desarrolló el proyecto de repositorios públicos disponibles en internet.

- Aspecto General POM -

El aspecto que tiene un fichero de configuración de Maven (POM) es como el que muestro a continuación:

```
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.um.atica</groupId>
    <artifactId>Prueba_Maven</artifactId>
    <name>Prueba_Maven</name>
    <version>0.0.1-SNAPSHOT</version>
    <description>Ejemplo de archivo POM de Maven.</description>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.4</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Los principales apartados que vemos describen nuestro proyecto y sus dependencias:

- **Project:** Elemento raíz de cada archivo pom.xml.
- **ModelVersion:** Elemento obligatorio, indica la versión del modelo de objeto que el POM está usando.
- **GroupId:** Indica el identificador único de la organización o del grupo de elementos creados para el proyecto.
- **ArtifactId:** Indica el nombre base único del componente o artefacto primario generado para este proyecto.
- **Name:** Indica el nombre de despliegue usado para el proyecto.
- **Version:** Indica la versión del componente o artefacto generado por el proyecto.
- **Description:** Proporciona una descripción del proyecto.
- **Packaging:** Indica el tipo de empaquetamiento que genera por defecto jar.

Las dependencias van a indicar las librerías que son necesarias en cada uno de los pasos del ciclo de vida del proyecto. Por ejemplo, en este caso durante la fase de test, es necesaria la librería (el artefacto) de JUnit en su versión 4.4. Existen otros ámbitos aplicables a las dependencias, compile, provided, runtime, etc. Cada una afecta a si se incluyen en el classpath de cada fase y si deben añadirse al empaquetado final.

Maven buscará esta librería en el repositorio local del desarrollador y si no lo localiza se la descargará de un repositorio remoto según las configuraciones del propio Maven (settings.xml). Si el desarrollador usa la misma librería en varios proyectos sólo tendrá una copia en su repositorio.

Existe un software adicional (Archiva) que permite controlar los accesos a repositorios de librerías, haciendo de proxy para todos los desarrolladores y permitiendo conceder permisos y ejercer cierto control sobre estas descargas de librerías.

- Ciclo de Vida -

Existen 3 ciclos de vida diferentes en Maven, build, clean y site. Buil para construir el proyecto, clean para eliminarlo y site para generar la documentación. El ciclo de vida por defecto es build y aunque existen bastantes fases dentro de este ciclo de vida para un proyecto Maven las principales son:

- **Compile:** Ejecutada para compilar los fuentes.
- **Test:** Ejecutada para pasar los tests.
- **Package:** Ejecutada para empaquetar el proyecto.
- **Deploy:** Ejecutada para desplegar el proyecto.

- Herencia -

Los archivos POM permiten herencia, es decir, en un proyecto podemos disponer de un POM general donde se declaren todas las dependencias de cada módulo y POM que hereden de este dónde sólo se indica los id de artefactos. Cambiar la versión de una librería para todo un proyecto es más sencillo de este modo.

- Arquetipos -

Los arquetipos nos van a permitir definir una especie de plantillas de generación que nos crearán un esqueleto de proyecto, para un tipo específico de proyecto. De este modo se simplifica la generación inicial de un proyecto y se unifica su formato.

1.3.3.- Servidor de Integración Contínua

Con un proyecto configurado con Maven, existe la posibilidad de que un ordenador de forma autónoma se descargue un proyecto de un repositorio SVN, lo compile y le pase los test, devolviendo un informe al equipo de desarrollo. Cada vez que se cambie algo en el repositorio, el servidor de integración continua puede ejecutar el ciclo otra vez. Esto da otra dimensión al desarrollo de test por parte del programador, ya no es un trabajo que implica más trabajo (hacer los test y probarlos), además aporta una forma ágil y rápida de detectar problemas.

Este tipo de herramientas además suelen incorporar otro tipo de herramientas para analizar el código, dando medidas de calidad, código repetido, no usado, generación de documentación y otra serie de valores añadidos a disponer de un repositorio centralizado de código.

Tema 2: Lenguaje Java y POO

En este tema veremos algunos conceptos del lenguaje Java y de programación orientada a objetos y aspectos.

2.1.- Programación Orientada a Objetos

La Programación Orientada a Objetos (POO u OOP según sus siglas en inglés) es un paradigma de programación que usa objetos y sus interacciones para diseñar aplicaciones y programas de computadora. Está basado en varias técnicas, incluyendo herencia, modularidad, polimorfismo y encapsulamiento. Su uso se popularizó a principios de la década de 1990. Actualmente son muchos los lenguajes de programación que soportan la orientación a objetos.

2.1.1.- Introducción

Los objetos son entidades que combinan estado, comportamiento e identidad:

- El estado está compuesto de datos, será uno o varios atributos a los que se habrán asignado unos valores concretos (datos).
- El comportamiento está definido por los procedimientos o métodos con que puede operar dicho objeto, es decir, qué operaciones se pueden realizar con él.
- La identidad es una propiedad de un objeto que lo diferencia del resto, dicho con otras palabras, es su identificador (concepto análogo al de identificador de una variable o una constante).

La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que colaboran entre ellos para realizar tareas. Esto permite hacer los programas y módulos más fáciles de escribir, mantener, reutilizar y volver a utilizar.

De aquella forma, un objeto contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos. A su vez, los objetos disponen de mecanismos de interacción llamados métodos que favorecen la comunicación entre ellos. Esta comunicación favorece a su vez el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separan ni deben separarse el estado y el comportamiento.

Los métodos (comportamiento) y atributos (estado) están estrechamente relacionados por la propiedad de conjunto. Esta propiedad destaca que una clase requiere de métodos para poder tratar los atributos con los que cuenta. El programador debe pensar indistintamente en ambos conceptos, sin separar ni darle mayor importancia a ninguno de ellos. Hacerlo podría producir el hábito erróneo de crear clases contenedoras de información por un lado y clases con métodos que manejen a las primeras por el otro. De esta manera se estaría realizando una programación estructurada camuflada en un lenguaje de programación orientado a objetos.

Esto difiere de la programación estructurada tradicional, en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. En la programación estructurada sólo se escriben funciones que procesan datos. Los programadores que emplean éste nuevo paradigma, en cambio, primero definen objetos para luego enviarles mensajes solicitándoles que realicen sus métodos por sí mismos.

2.1.2.- Conceptos Fundamentales

La programación orientada a objetos es una nueva forma de programar que trata de encontrar una solución a estos problemas. Introduce nuevos conceptos, que superan y amplían conceptos antiguos ya conocidos. Entre ellos destacan los siguientes:

- Clase -

Definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas.

- Objeto -

Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos) los mismos que consecuentemente reaccionan a eventos. Se corresponde con los objetos reales del mundo que nos rodea, o a objetos internos del sistema (del programa). Es una instancia a una clase.

- Método -

Algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un "mensaje". Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un "evento" con un nuevo mensaje para otro objeto del sistema. Existen métodos de clase, que no se ejecutan sobre ninguna instancia concreta sino sobre la clase en si misma.

- Atributo -

Contenedor de un tipo de datos asociados a un objeto (o a una clase de objetos), que hace los datos visibles desde fuera del objeto y esto se define como sus características predeterminadas, y cuyo valor puede ser alterado por la ejecución de algún método. Existen atributos ligados a una clase, que comparten su valor para todas las instancias de la misma.

- Mensaje -

Una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.

Hay un cierto desacuerdo sobre exactamente qué características de un método de programación o lenguaje le definen como "orientado a objetos", pero hay un consenso general en que las características siguientes son las más importantes (para más información, seguir los enlaces respectivos):

- Abstracción -

Denota las características esenciales de un objeto, donde se capturan sus comportamientos. Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar cómo se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos y cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción.

- Encapsulamiento -

Significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente.

- Ocultación -

Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos.

- Polimorfismo -

Comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre, al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. O dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama asignación tardía o asignación/ligadura dinámica. Algunos lenguajes proporcionan medios más estáticos (en "tiempo de compilación") de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++.

- Herencia -

Las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y

el encapsulamiento permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo. Esto suele hacerse habitualmente agrupando los objetos en clases y estas en árboles o enrejados que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay herencia múltiple.

- Recolección de Basura -

La Recolección de basura o Garbage Collection es la técnica por la cual el ambiente de Objetos se encarga de destruir automáticamente, y por tanto desasignar de la memoria, los Objetos que hayan quedado sin ninguna referencia a ellos. Esto significa que el programador no debe preocuparse por la asignación o liberación de memoria, ya que el entorno la asignará al crear un nuevo Objeto y la liberará cuando nadie lo esté usando. En la mayoría de los lenguajes híbridos que se extendieron para soportar el Paradigma de Programación Orientada a Objetos como C++ u Object Pascal, esta característica no existe y la memoria debe desasignarse manualmente, en Java es clave para su funcionamiento.

2.2.- Programación Orientada a Aspectos

La Programación Orientada a Aspectos (POA) es un paradigma de programación relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos. Gracias a la POA se pueden encapsular los diferentes conceptos que componen una aplicación en entidades bien definidas, eliminando las dependencias entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables. Varias tecnologías con nombres diferentes se encaminan a la consecución de los mismos objetivos y así, el término POA es usado para referirse a varias tecnologías relacionadas como los métodos adaptivos, los filtros de composición, la programación orientada a sujetos o la separación multidimensional de competencias.

2.2.1.- Conceptos Básicos

Debido a la escasa literatura en español sobre el tema, se presenta la terminología original en inglés.

- Aspect -

Es una funcionalidad transversal (cross-cutting) que se va a implementar de forma modular y separada del resto del sistema. El ejemplo más común y simple de un aspecto es el logging (registro de sucesos) dentro del sistema, ya que necesariamente afecta a todas las partes del sistema que generan un suceso.

- Join Point -

Es un punto de ejecución dentro del sistema donde un aspecto puede ser conectado, como una llamada a un método, el lanzamiento de una excepción o la

modificación de un campo. El código del aspecto será insertado en el flujo de ejecución de la aplicación para añadir su funcionalidad.

- Advice -

Es la implementación del aspecto, es decir, contiene el código que implementa la nueva funcionalidad. Se insertan en la aplicación en los Puntos de Cruce.

- Pointcut -

Define los Consejos que se aplicarán a cada Punto de Cruce. Se especifica mediante Expresiones Regulares o mediante patrones de nombres (de clases, métodos o campos), e incluso dinámicamente en tiempo de ejecución según el valor de ciertos parámetros.

- Introduction -

Permite añadir métodos o atributos a clases ya existentes. Un ejemplo en el que resultaría útil es la creación de un Consejo de Auditoría que mantenga la fecha de la última modificación de un objeto, mediante una variable y un método setUltimaModificacion(fecha), que podrían ser introducidos en todas las clases (o sólo en algunas) para proporcionarlas esta nueva funcionalidad.

- Target -

Es la clase aconsejada, la clase que es objeto de un consejo. Sin AOP, esta clase debería contener su lógica, además de la lógica del aspecto.

- Proxy -

Es el objeto creado después de aplicar el Consejo al Objeto Destinatario. El resto de la aplicación únicamente tendrá que soportar al Objeto Destinatario (pre-AOP) y no al Objeto Resultante (post-AOP).

- Weaving -

Es el proceso de aplicar Aspectos a los Objetos Destinatarios para crear los nuevos Objetos Resultantes en los especificados Puntos de Cruce. Este proceso puede ocurrir a lo largo del ciclo de vida del Objeto Destinatario:

- *Aspectos en Tiempo de Compilación*, que necesita un compilador especial.
- *Aspectos en Tiempo de Carga*, los Aspectos se implementan cuando el Objeto Destinatario es cargado. Requiere un ClassLoader especial.
- *Aspectos en Tiempo de Ejecución*.

Muchas veces nos encontramos, a la hora de programar, con problemas que no podemos resolver de una manera adecuada con las técnicas habituales usadas en la programación imperativa o en la programación orientada a objetos. Con éstas, nos vemos forzados a tomar decisiones de diseño que repercuten de manera importante en el desarrollo de la aplicación y que nos alejan con frecuencia de otras posibilidades.

A menudo, hace falta escribir líneas de código que están distribuidas por toda o gran parte de la aplicación, para definir la lógica de cierta propiedad o comportamiento del sistema, con las consecuentes dificultades de mantenimiento y desarrollo. En inglés este problema se conoce como *scattered code*, que podríamos traducir como código disperso. Otro problema que puede aparecer, es que un mismo módulo implemente múltiples comportamientos o aspectos del sistema de forma simultánea. En inglés este problema se conoce como *tangled code*, que podríamos traducir como código enmarañado. El hecho es que hay ciertas decisiones de diseño que son difíciles de capturar, debido a que determinados problemas no se pueden encapsular claramente de igual forma que los que habitualmente se resuelven con funciones u objetos.

Existen extensiones de Java como AspectJ o frameworks como Spring AOP que permiten con este paradigma, pero nosotros nos acercaremos a este paradigma empleando los interceptores que es el mecanismo más sencillo para emplear AOP. Las intercepciones se emplean para invocar a código del desarrollador en determinados momentos del ciclo de vida de un objeto.

2.3.- Lenguaje Java

La plataforma Java está compuesta por tres componentes básicos:

- **El Lenguaje:** Un lenguaje de propósito general que emplea el paradigma de programación orientado a objetos.
- **La Máquina Virtual:** Los programas Java se compilán a un código intermedio interpretado por una máquina virtual JVM (Java Virtual Machine), lo que permite su portabilidad.
- **Las Bibliotecas:** El conjunto de API's (Application Programming Interface) que proporcionan herramientas para el desarrollo.

La plataforma Java dispone de 3 ediciones diferentes para diferentes propósitos:

- **Java ME:** MicroEdition ideada para ejecutar aplicaciones en dispositivos móviles.
- **Java SE:** Para aplicaciones en general.
- **Java EE:** Para aplicaciones corporativas o empresariales.

En general las ediciones Java se dividen en dos partes un SDK y un JRE. El SDK está pensado para el desarrollador e incorpora herramientas de compilación, debug, etc. Por otro lado el JRE incorpora sólo el entorno necesario para ejecutar aplicaciones Java previamente compiladas. Es el entorno para distribuir a los clientes.

La máquina virtual de Java necesitará tener acceso a las clases que requiera para ejecutar el programa compilado. Se denomina **ClassPath** a la ruta en la cuál Java busca las clases referenciadas por los programas que interpreta.

2.3.1.- Estructura General

Lo primero de todo es ver cuál es la estructura habitual de un programa realizado en cualquier lenguaje orientado a objetos u OOP (Object Oriented Programming), y en

particular en el lenguaje Java. Este es un sencillo ejemplo de una clase que imprime un clásico mensaje en consola.

```
public class MiClase {  
    public static void main(String[] args) {  
        System.out.println("Hola Mundo!!");  
    }  
}
```

Aparece una clase que contiene el programa principal (aquel que contiene la función main()) y algunas clases de usuario (las específicas de la aplicación que se está desarrollando) que son utilizadas por el programa principal. Los ficheros fuente tienen la extensión *.java, mientras que los ficheros compilados tienen la extensión *.class.

Un fichero fuente (*.java) puede contener más de una clase, pero sólo una puede ser public. El nombre del fichero fuente debe coincidir con el de la clase public (con la extensión *.java). Si por ejemplo en un fichero aparece la declaración (public class MiClase {...}) entonces el nombre del fichero deberá ser MiClase.java. Es importante que coincidan mayúsculas y minúsculas ya que MiClase.java y miclase.java serían clases diferentes para Java. Si la clase no es public, no es necesario que su nombre coincida con el del fichero. Una clase puede ser public o package (default), pero no private o protected. Estos conceptos se explican posteriormente.

De ordinario una aplicación está constituida por varios ficheros *.class. Cada clase realiza unas funciones particulares, permitiendo construir las aplicaciones con gran modularidad e independencia entre clases. La aplicación se ejecuta por medio del nombre de la clase que contiene la función main() (sin la extensión *.class). Las clases de Java se agrupan en packages, que son librerías de clases. Si las clases no se definen como pertenecientes a un package, se utiliza un package por defecto (default) que es el directorio activo. Los packages se estudian con más detenimiento más adelante.

- Concepto de Clase -

Una clase es una agrupación de datos (variables o campos) y de funciones (métodos) que operan sobre esos datos. A estos datos y funciones pertenecientes a una clase se les denomina variables y métodos o funciones miembro. La programación orientada a objetos se basa en la programación de clases. Un programa se construye a partir de un conjunto de clases.

Una vez definida e implementada una clase, es posible declarar elementos de esta clase de modo similar a como se declaran las variables del lenguaje (de los tipos primitivos int, double, String, ...). Los elementos declarados de una clase se denominan objetos de la clase. De una única clase se pueden declarar o crear numerosos objetos. La clase es lo genérico: es el patrón o modelo para crear objetos. Cada objeto tiene sus propias copias de las variables miembro, con sus propios valores, en general distintos de los demás objetos de la clase. Las clases pueden tener variables static, que son propias de la clase y no de cada objeto.

- Herencia -

La herencia permite que se puedan definir nuevas clases basadas en clases existentes, lo cual facilita re-utilizar código previamente desarrollado. Si una clase deriva de otra (extends) hereda todas sus variables y métodos. La clase derivada puede añadir nuevas variables y métodos y/o redefinir las variables y métodos heredados.

En Java, a diferencia de otros lenguajes orientados a objetos, una clase sólo puede derivar de una única clase, con lo cual no es posible realizar herencia múltiple en base a clases. Sin embargo es posible “simular” la herencia múltiple en base a las interfaces.

- Interface -

Una interface es un conjunto de declaraciones de funciones. Si una clase implementa (implements) una interface, debe definir todas las funciones especificadas por la interface. Una clase puede implementar más de una interface, representando una forma alternativa de la herencia múltiple.

A su vez, una interface puede derivar de otra o incluso de varias interfaces, en cuyo caso incorpora todos los métodos de las interfaces de las que deriva.

- Package -

Un package es una agrupación de clases. Existen una serie de packages incluidos en el lenguaje (ver jerarquía de clases que aparece en el API de Java).

Además el usuario puede crear sus propios packages. Lo habitual es juntar en packages las clases que estén relacionadas. Todas las clases que formen parte de un package deben estar en el mismo directorio.

- Jerarquía de Clases -

Durante la generación de código en Java, es recomendable y casi necesario tener siempre a la vista la documentación on-line del API de Java de la versión en que trabajemos. En dicha documentación es posible ver tanto la jerarquía de clases, es decir la relación de herencia entre clases, como la información de los distintos packages que componen las librerías base de Java (el core).

Es importante distinguir entre lo que significa herencia y package. Un package es una agrupación arbitraria de clases, una forma de organizar las clases. La herencia sin embargo consiste en crear nuevas clases en base a otras ya existentes. Las clases incluidas en un package no derivan por lo general de una única clase.

En la documentación on-line se presentan ambas visiones: “Package Index” y “Class Hierarchy”. La primera presenta la estructura del API de Java agrupada por packages, mientras que en la segunda aparece la jerarquía de clases. Hay que resaltar una vez más el hecho de que todas las clases en Java son derivadas de la clase `java.lang.Object`, por lo que heredan todos los métodos y variables de ésta.

Si se selecciona una clase en particular, la documentación muestra una descripción detallada de todos los métodos y variables de la clase. A su vez muestra su herencia completa (partiendo de la clase `java.lang.Object`).

2.3.2.- Programación Java

En este capítulo se presentan las características generales de Java como lenguaje de programación algorítmico. En este apartado Java es muy similar a C/C++, lenguajes en los que está inspirado. Se va a intentar ser breve, considerando que el lector ya conoce algunos otros lenguajes de programación y está familiarizado con lo que son variables, bifurcaciones, bucles, etc.

- Variables -

En Java existen dos tipos de variables, las de tipos primitivos y las de referencia. Las primeras son aquellas que pertenecen a un tipo primitivo del lenguaje (*char, byte, short, int, long, float, double, boolean*). Las segundas son aquellas que referencian a un objeto o a un array.

Dependiendo del lugar donde se declaren serán variables miembro de una clase (atributos), o variables locales las que se definen dentro de los métodos o en general dentro de un bloque Java (definido por un par de {}).

A la hora de declarar una variable basta con darle un nombre para lo cual es conveniente seguir una serie de convenciones o reglas de escritura de código, e inicializarla. La forma de hacerlo es muy variada:

```
int x; // Declaración de la variable primitiva x. Se inicializa a 0
int y = 5; // Declaración de la variable primitiva y. Se inicializa a 5
MyClass unaRef; // Declaración de una referencia a un objeto MyClass.
                 // Se inicializa a null
unaRef = new MyClass(); // La referencia "apunta" al nuevo objeto creado
                        // Se ha utilizado el constructor por defecto
MyClass segundaRef = unaRef; // Declaración de una referencia a otro objeto.
                            // Se inicializa al mismo valor que unaRef
int [] vector; // Declaración de un array. Se inicializa a null
vector = new int[10]; // Vector de 10 enteros, inicializados a 0
double [] v = {1.0, 2.65, 3.1}; // Declaración e inicialización de un vector de 3
                                 // elementos con los valores entre llaves
MyClass [] lista=new MyClass[5]; // Se crea un vector de 5 referencias a objetos
                                 // Las 5 referencias son inicializadas a null
lista[0] = unaRef; // Se asigna a lista[0] el mismo valor que unaRef
lista[1] = new MyClass(); // Se asigna a lista[1] la referencia al nuevo objeto
                         // El resto (lista[2]...lista[4]) siguen con valor null
```

Se entiende por **visibilidad, ámbito o scope** de una variable, la parte de la aplicación donde dicha variable es accesible y por lo tanto puede ser utilizada en una expresión. En Java todas las variables deben estar incluidas en una clase. En general las variables declaradas dentro de unas llaves {}, es decir dentro de un bloque, son visibles y existen dentro de estas llaves. Por ejemplo las variables declaradas al principio de una función existen mientras se ejecute la función; las variables declaradas dentro de un bloque if no serán válidas al finalizar las sentencias correspondientes a dicho if y las variables miembro de una clase (es decir declaradas entre las llaves {} de la clase pero fuera de cualquier método) son válidas mientras existe el objeto de la clase.

Las variables miembro de una clase declaradas como **public** son accesibles a través de una referencia a un objeto de dicha clase utilizando el operador punto (.). Las variables miembro declaradas como **private** no son accesibles directamente desde otras clases. Las funciones miembro de una clase tienen acceso directo a todas las variables miembro de la clase sin necesidad de anteponer el nombre de un objeto de la clase. Sin embargo las funciones miembro de una clase B derivada de otra A, tienen acceso a todas las variables miembro de A declaradas como **public** o **protected**, pero no a las declaradas como **private**. Una clase derivada sólo puede acceder directamente a las variables y funciones miembro de su clase base declaradas como **public** o **protected**. Otra característica del lenguaje es que es posible declarar una variable dentro de un bloque con el mismo nombre que una variable miembro, pero no con el nombre de otra variable local que ya existiera. La variable declarada dentro del bloque oculta a la variable miembro en ese bloque. Para acceder a la variable miembro oculta será preciso utilizar el operador **this**, en la forma *this.varname*.

Uno de los aspectos más importantes en la programación orientada a objetos (OOP) es la forma en la cual son creados y eliminados los objetos. En Java la forma de crear nuevos objetos es utilizando el operador ***new***. Cuando se utiliza el operador ***new***, la variable de tipo referencia guarda la posición de memoria donde está almacenado este nuevo objeto. Para cada objeto se lleva cuenta de por cuántas variables de tipo referencia es apuntado. La eliminación de los objetos la realiza el programa denominado *garbage collector*, quien automáticamente libera o borra la memoria ocupada por un objeto cuando no existe ninguna referencia apuntando a ese objeto. Lo anterior significa que aunque una variable de tipo referencia deje de existir, el objeto al cual apunta no es eliminado si hay otras referencias apuntando a ese mismo objeto.

- Operadores -

Los operadores básicos del lenguaje Java son:

- ***Los aritméticos***: suma +, resta -, multiplicación *, división / y resto de la división %.
- ***Los de asignación***: Igual =, con suma +=, con resta -=, con multiplicación *=, con división /= y con resto de división %=.
- ***Los incrementales***: Incremento ++ y decremento --. Según se pongan antes o después de la variable incrementan su valor antes o después de su uso.
- ***Los relacionales***: Mayor >, mayor o igual >=, menor <, menor o igual <=, igual == y distinto !=.
- ***Los lógicos***: and &&, or || y negación.
- ***Para cadenas***: Concatenar +.
- ***Para bits***: Desplazamiento a derecha >>, a izquierda <<, and &, or |, xor ^, complemento ~.

Es importante a la hora de escribir expresiones en Java, tener en cuenta la precedencia de los diferentes operadores y emplear paréntesis en caso de duda.

- Estructuras de Programación -

Las estructuras de programación van a permitirnos tomar decisiones o realizar tareas de forma reiterada, vamos a ver algunas de las más importantes:

- ***Comentarios***: Nos permitirán introducir documentación en el propio código. Especial dedicatoria a los comentarios en formato Javadoc, que permiten generar un sitio web a partir de los comentarios y contenido del código fuente de una aplicación.
 - *Comentario de línea*: // Esto esta comentado hasta el final de la línea.
 - *Comentario de Bloque*: /* Puede contener varias líneas */
- ***Bifurcaciones***: Permiten tomar decisiones.
 - *Bifurcación If*: if (condición) { .. }
 - *Bifurcación If-Else*: if (condición) { .. } else { .. }
 - *Bifurcación If-Elseif-Else*: if (condición) { .. } else if { .. } else { .. }
 - *Estudio de Casos*: Empleada para verificar múltiples casos.
- ***Bucles***: Creados para iterar sobre conjuntos o repetir la ejecución de parte del código según alguna condición.

- *Bucle While*: while (condición) { .. }
- *Bucle doWhile*: do { .. } while (condición)
- *Bucle For*: for (inicialización; condición; incremento) { .. }
- **Sentencia Break**: Abandona el interior de un bloque, bien sea condicional o bucle.
- **Sentencia Continue**: Empleada en bucles para obviar el resto del bloque y pasar a la siguiente iteración.

2.3.3.- Clases en Java

Una clase es una agrupación de datos (variables o campos) y de funciones (métodos) que operan sobre esos datos. La definición de una clase se realiza en la siguiente forma:

```
public class MiClase {  
  
    /* Atributos */  
    private String nombre;  
    private String valor;  
  
    /* Métodos */  
    public String getNombre() { return nombre; }  
    public void setNombre(String nombre) { this.nombre = nombre; }  
  
    public String getValor() { return valor; }  
    public void setValor(String valor) { this.valor = valor; }  
}
```

A continuación se enumeran algunas características importantes de las clases:

1. Todas las variables y funciones de Java deben pertenecer a una clase. No hay variables y funciones globales.
2. Si una clase deriva de otra (extends), hereda todas sus variables y métodos.
3. Java tiene una jerarquía de clases estándar de la que pueden derivar las clases que crean los usuarios.
4. Una clase sólo puede heredar de una única clase (en Java no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de Object. La clase Object es la base de toda la jerarquía de clases de Java.
5. En un fichero se pueden definir varias clases, pero en un fichero no puede haber más que una clase public. Este fichero se debe llamar como la clase public que contiene con extensión *.java. Con algunas excepciones, lo habitual es escribir una sola clase por fichero.
6. Si una clase contenida en un fichero no es public, no es necesario que el fichero se llame como la clase.
7. Los métodos de una clase pueden referirse de modo global al objeto de esa clase al que se aplican por medio de la referencia this.
8. Las clases se pueden agrupar en packages, introduciendo una línea al comienzo del fichero (package packageName;). Esta agrupación en packages está relacionada con la jerarquía de directorios y ficheros en la que se guardan las clases.

Una *interface* es un conjunto de declaraciones de funciones. Si una clase implementa (implements) una interface, debe definir todas las funciones especificadas

por la interface. Las interfaces pueden definir también variables finales (constantes). Una clase puede implementar más de una interface, representando una alternativa a la herencia múltiple.

En algunos aspectos los nombres de las interfaces pueden utilizarse en lugar de las clases. Por ejemplo, las interfaces sirven para definir referencias a cualquier objeto de cualquiera de las clases que implementan esa interface. Con ese nombre o referencia, sin embargo, sólo se pueden utilizar los métodos de la interface. Éste es un aspecto importante del polimorfismo.

Una interface puede derivar de otra o incluso de varias interfaces, en cuyo caso incorpora las declaraciones de todos los métodos de las interfaces de las que deriva (a diferencia de las clases, las interfaces de Java sí tienen herencia múltiple).

Para entender mejor estos conceptos vamos a estudiar brevemente el siguiente código de ejemplo:

```
// Cualquier objeto que se pueda dibujar
public interface Dibujable {
    public void dibujar();
}

// Representa Figuras Geométricas
public abstract class Figura implements Dibujable {
    // Permite calcular el área de una figura
    public abstract float area();
}

// Una figura concreta: El cuadrado
public class Cuadrado extends Figura {
    // Lado del cuadrado
    private float lado;

    // Constructor con lado
    public Cuadrado(float l) { lado = l; }

    @Override
    public float area() {
        // El área del cuadrado
        return lado*lado;
    }

    public void dibujar() {
        // Método para dibujar un cuadrado
    }
}

// Representa un plano con figuras geométricas
public class Plano implements Dibujable {
    // Colección de figuras que forman el plano
    java.util.List<Figura> figuras;
    // Dibujar el plano
    public void dibujar() {
        for (Figura f:figuras)
            f.dibujar();
    }

    // Área que ocupa el plano
    public float areaPlano() {
        float total = 0;
        for (Figura f:figuras)
            total += f.area();
        return total;
    }
}
```

Este conjunto de clases representa un plano compuesto de figuras geométricas, que pueden ser dibujadas y que permiten calcular su área. Vamos a describir

brevemente algunas características del lenguaje que nos servirán para entender el código expuesto y para crear nuestro propio código.

- Métodos -

Son funciones definidas dentro de una clase, reciben como argumento implícito un objeto de la clase en la que se definen. Su declaración consta de una cabecera donde aparece: El cualificador de acceso (*public, private, protected*), el tipo de retorno (void si no retorna nada), el nombre del método y los argumentos explícitos.

Existe la posibilidad de sobrecargar métodos, definir métodos con igual nombre pero diferentes argumentos (en número o tipo). El compilador aplica la llamada al método sobrecargado adecuado, en función del tipo o número de argumentos que hay en la llamada.

- Paso de Argumentos -

En Java todos los argumentos se pasan por valor, tanto los tipos primitivos como las referencias de objetos y arrays. La diferencia es que en el caso de las referencias a través de estas se modifica el objeto original, cosa que no sucede con los tipos primitivos.

- Miembros de Clase -

Anteponiendo a la declaración de un método o de una variable la palabra *static*, logramos crear un miembro o método de clase. Estas variables son comunes a todas las instancias de la clase y se accede a ellas a través del operador punto precedido del nombre de la clase. Los métodos actúan igual y sólo pueden acceder a miembros estáticos de la clase.

Otra palabra en este caso *final*, sirve para definir miembros que no varían, en el caso de variables constantes y en el caso de métodos que no pueden redefinirse en subclases.

- Constructores -

Los constructores son métodos que se llaman al crear los objetos de una clase, con el fin de inicializarlos correctamente. Estos métodos carecen de valor de retorno, y suelen sobrecargarse con diferentes números de argumentos. Al constructor sin argumentos se le denomina *constructor por defecto*.

Podemos tener constructores privados, que permiten controlar la creación de objetos desde el exterior a través de un método *static (factory method)*. Dentro de un constructor se puede utilizar *this* para invocar otro constructor de la clase, o *super* para invocar un constructor de la superclase.

También existen los inicializadores (estáticos y de objeto), que son bloques *static* (o sin nombre) que se invocan al utilizar una clase por primera vez.

El proceso de creación de objetos de una clase es el siguiente:

- Al crear el primer objeto de la clase o al utilizar el primer método o variable static se localiza la clase y se carga en memoria.
- Se ejecutan los inicializadores static (sólo una vez).

- Cada vez que se quiere crear un nuevo objeto:
 - Se comienza reservando la memoria necesaria.
 - Se da valor por defecto a las variables miembro de los tipos primitivos.
 - Se ejecutan los inicializadores de objeto.
 - Se ejecutan los constructores.

Al igual que se crean los objetos con un constructor existe el método *finalize*, que permite liberar recursos y que es ejecutado al destruir un objeto.

- Herencia -

Java emplea herencia simple, es decir, cada clase extiende como máximo de una clase. Una subclase puede redefinir los métodos que herede de una superclase, salvo los final o private. Al redefinirlos puede ampliar sus derechos de acceso (convertir un método protected en public). No se pueden redefinir los métodos static.

Se pueden crear clases abstractas. Son clases de las que no se pueden crear objetos y que definen métodos abstractos que deben implementarse en las subclases. No se pueden crear métodos abstractos static, ya que no se pueden redefinir.

Para solventar las deficiencias de no disponer de herencia múltiple Java emplea el concepto de interface y una jerarquía de interfaces (al margen de la jerarquía de objetos) que si permite herencia múltiple.

A la hora de declarar variables podemos emplear las interfaces como tipos, y asignar unas variables a otras siguiendo el principio de herencia “es un”. A una variable Dibujable podemos asignarle un objeto Plano ó Cuadrado. También podemos hacer la asignación contraria (si estamos seguros) empleando el *casting* explícito.

Nos dejamos muchas características del lenguaje en el tintero como pueden ser los métodos native, las clases internas, etc... Recomiendo la lectura de cualquier libro expresamente dedicado al lenguaje Java para ampliar los conocimientos del lenguaje, en este curso con estas breves nociones será suficiente.

También estamos obviando ciertas características como el manejo de flujos, hilos de ejecución y otras API's de gran relevancia dentro del core de Java.

2.3.4.- Excepciones

A diferencia de otros lenguajes de programación orientados a objetos como C/C++, Java incorpora en el propio lenguaje la gestión de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo prácticamente sólo los errores de sintaxis son detectados durante este periodo. El resto de problemas surgen durante la ejecución de los programas.

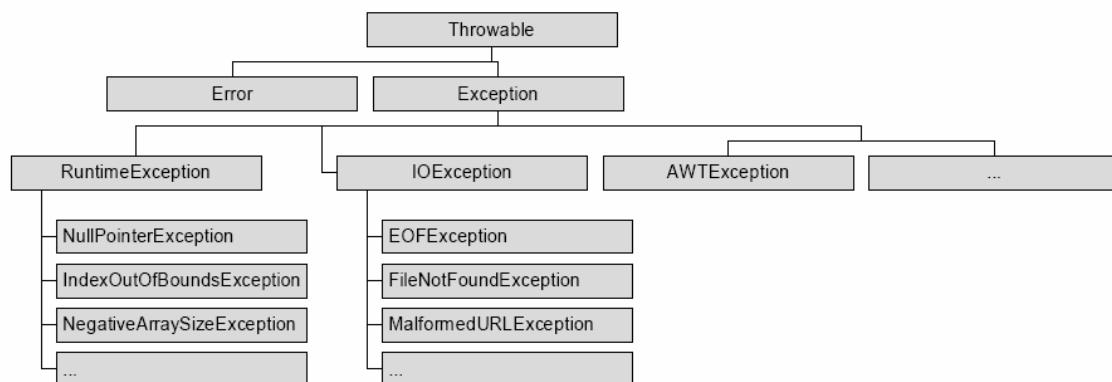
En el lenguaje Java, una ***Exception*** es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas excepciones son fatales y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras, como por ejemplo no encontrar un fichero en el que hay que leer o escribir algo, pueden ser recuperables. En este caso el programa debe dar al usuario la oportunidad de corregir el error (indicando una nueva localización del fichero no encontrado).

Un buen programa debe gestionar correctamente todas o la mayor parte de los errores que se pueden producir. Hay dos “estilos” de hacer esto:

- **A la “antigua usanza”:** Los métodos devuelven un código de error. Este código se chequea en el entorno que ha llamado al método con una serie de if elseif ..., gestionando de forma diferente el resultado correcto o cada uno de los posibles errores. Este sistema resulta muy complicado cuando hay varios niveles de llamadas a los métodos.
- **Con soporte en el propio lenguaje:** En este caso el propio lenguaje proporciona construcciones especiales para gestionar los errores o Exceptions. Suele ser lo habitual en lenguajes modernos, como C++, Visual Basic y Java.

- Excepciones Estándar -

Los errores se representan mediante dos tipos de clases derivadas de la clase **Throwable**: **Error** y **Exception**. La siguiente figura muestra parcialmente la jerarquía de clases relacionada con **Throwable**:



La clase **Error** está relacionada con errores de compilación, del sistema o de la JVM. De ordinario estos errores son irrecuperables y no dependen del programador ni debe preocuparse de capturarlos y tratarlos.

La clase **Exception** tiene más interés. Dentro de ella se puede distinguir:

- **RuntimeException:** Son excepciones muy frecuentes, de ordinario relacionadas con errores de programación. Se pueden llamar excepciones implícitas.
- **El Resto:** Derivadas de **Exception** son excepciones explícitas. Java obliga a tenerlas en cuenta y chequear si se producen.

El caso de **RuntimeException** es un poco especial. El propio Java durante la ejecución de un programa chequea y lanza automáticamente las excepciones que derivan de **RuntimeException**. El programador no necesita establecer los bloques try/catch para controlar este tipo de excepciones. Representan dos casos de errores de programación:

- Un error que normalmente no suele ser chequeado por el programador, como por ejemplo recibir una referencia null en un método.

- Un error que el programador debería haber chequeado al escribir el código, como sobrepasar el tamaño asignado de un array (genera un `ArrayIndexOutOfBoundsException` automáticamente).

En realidad sería posible comprobar estos tipos de errores, pero el código se complicaría excesivamente si se necesitara chequear continuamente todo tipo de errores (que las referencias son distintas de null, que todos los argumentos de los métodos son correctos, y un largo etcétera).

Las clases derivadas de `Exception` pueden pertenecer a distintos packages de Java. Algunas pertenecen a `java.lang` (`Throwable`, `Exception`, `RuntimeException`, ...); otras a `java.io` (`EOFException`, `FileNotFoundException`, ...) o a otros packages. Por heredar de `Throwable` todos los tipos de excepciones pueden usar los métodos siguientes:

- `String getMessage()`: Extrae el mensaje asociado con la excepción.
- `String toString()`: Devuelve un `String` que describe la excepción.
- `void printStackTrace()`: Indica el método donde se lanzó la excepción.

Para entender un poco mejor como manejar las excepciones en Java vamos a ver un pequeño ejemplo de código, y las alternativas que tenemos para controlar los errores del mismo.

```
public void ejemplo(String numero, String nombreClase) {  
    // Método para dibujar un cuadrado  
    int n = Integer.parseInt(numero);  
    Class.forName(nombreClase);  
}
```

Este método convierte una cadena en un entero, e instancia de forma dinámica una clase a partir de su nombre. En este código deberíamos controlar dos posibles errores, por un lado que la cadena “numero” contuviese un número, y que la cadena “nombreClase” contuviese un nombre de clase existente. Los métodos “`parseInt`” y “`forName`” lanzan las correspondientes excepciones: `NumberFormatException` y `ClassNotFoundException`. La primera hereda de `RuntimeException`, por lo que no es necesario tratarla, pero la segunda si, en cualquier caso trataremos las dos por igual.

Una primera alternativa es lanzar las excepciones sin tratamiento alguno:

```
// Lanzo las excepciones correspondientes  
public void ejemplo(String numero, String nombreClase)  
throws NumberFormatException, ClassNotFoundException {  
    // Método de ejemplo  
    int n = Integer.parseInt(numero);  
    Class.forName(nombreClase);  
  
    // Lanzo una excepción más general  
    public void ejemplo(String numero, String nombreClase) throws Exception {  
        // Método de ejemplo  
        int n = Integer.parseInt(numero);  
        Class.forName(nombreClase);  
    }
```

También podemos tratar las excepciones en el propio método sin que este lance ninguna otra excepción. Esta es la forma normal cuando tenemos un código alternativo en caso de error.

```
// Trato las excepciones
public void ejemplo(String numero, String nombreClase) {
    // Método para dibujar un cuadrado
    try {
        int n = Integer.parseInt(numero);
        Class.forName(nombreClase);
    } catch (NumberFormatException ne) {
        // Código para tratar este fallo
    } catch (ClassNotFoundException ce) {
        // Código para tratar este fallo
    } finally {
        // Bloque opcional se ejecuta siempre haya o no excepción
    }
}
```

O podemos tratarlas y además lanzar la propia excepción u otra más general o incluso particular creada por nosotros mismos.

```
// Trato las excepciones de forma conjunta y las lanza con otro tipo
public void ejemplo(String numero, String nombreClase) throws ExpcionEjemplo {
    // Método para dibujar un cuadrado
    try {
        int n = Integer.parseInt(numero);
        Class.forName(nombreClase);
    } catch (Exception nce) {
        // Código para tratar este fallo
        throw new ExpcionEjemplo("Error Con Número o Clase.");
    } finally {
        // Bloque opcional se ejecuta siempre haya o no excepción
    }
}

// Trato las excepciones y lanza una excepción más general
public void ejemplo(String numero, String nombreClase) throws Expcion {
    // Método para dibujar un cuadrado
    try {
        int n = Integer.parseInt(numero);
        Class.forName(nombreClase);
    } catch (NumberFormatException ne) {
        // Código para tratar este fallo
        throw ne;
    } catch (ClassNotFoundException ce) {
        // Código para tratar este fallo
        throw ce;
    } finally {
        // Bloque opcional se ejecuta siempre haya o no excepción
    }
}
```

El manejo de excepciones en el lenguaje Java es clave, un buen manejo de excepciones ayudará a poder localizar de forma eficiente los problemas de código una vez que se encuentre en ejecución y también evitará ejecuciones inesperadas del mismo.

2.3.6.- Novedades Lenguaje Java 5

- Tipos de datos parametrizados (generics) -

Esta mejora permite tener colecciones de tipos concretos de datos, lo que permite asegurar que los datos que se van a almacenar van a ser compatibles con un determinado tipo o tipos. Por ejemplo, podemos crear un Vector que sólo almacene Strings, o una HashMap que tome como claves Integers y como valores Vectors. Además, con esto nos ahorraremos las conversiones cast al tipo que deseemos, puesto que la colección ya se asume que será de dicho tipo.

```
// Vector de cadenas
Vector<String> v = new Vector<String>();
v.addElement("Hola");
String s = v.elementAt(0);
v.addElement(new Integer(20)); // Daría error!!

// HashMap con claves enteras y valores de vectores
HashMap<Integer, Vector> hm = new HashMap<Integer, Vector>();
hm.put(1, v);
Vector v2 = hm.get(1);
```

- Autoboxing -

Esta nueva característica evita al programador tener que establecer correspondencias manuales entre los tipos simples (int, double, etc) y sus correspondientes wrappers o tipos complejos (Integer, Double, etc). Podremos utilizar un int donde se espere un objeto complejo (Integer), y viceversa.

```
Vector<Integer> v = new Vector<Integer>();
v.addElement(30);
Integer n = v.elementAt(0);
n = n+1;
```

- Mejoras en bucles -

Se mejoran las posibilidades de recorrer colecciones y arrays, previniendo índices fuera de rango, y pudiendo recorrer colecciones sin necesidad de acceder a sus iteradores (Iterator). Simplifica muchísimo el código y lo deja incluso más legible que en anteriores versiones.

```
// Recorre e imprime todos los elementos de un array
int[] arrayInt = {1, 20, 30, 2, 3, 5};
for(int elemento: arrayInt)
    System.out.println (elemento);

// Recorre e imprime todos los elementos de un Vector
Vector<String> v = new Vector<String>();
for(String cadena: v)
    System.out.println (cadena);
```

- Tipo enum -

El tipo enum que se introduce permite definir un conjunto de posibles valores o estados, que luego podremos utilizar donde queramos. Esto hace más legible el código y permite utilizar análisis de casos en lugares donde para hacerlo hubiese sido necesario recurrir a tipos básicos que hacen el código menos comprensible:

```
// Define una lista de 3 valores y luego comprueba en un switch
// cuál es el valor que tiene un objeto de ese tipo
enum EstadoCivil {soltero, casado, divorciado};
EstadoCivil ec = EstadoCivil.casado;
ec = EstadoCivil.soltero;
switch(ec)
{
    case soltero: System.out.println("Es soltero");
    break;
    case casado: System.out.println("Es casado");
    break;
    case divorciado: System.out.println("Es divorciado");
    break;
}
```

- Imports estáticos -

Los imports estáticos permiten importar los elementos estáticos de una clase, de forma que para referenciarlos no tengamos que poner siempre como prefijo el nombre de la clase. Por ejemplo, podemos utilizar las constantes de color de la clase java.awt.Color, o bien los métodos matemáticos de la clase Math.

```
import static java.awt.Color.*;
import static java.lang.Math.*;

public class...
{
    ...
    JLabel lbl = new JLabel();
    lbl.setBackground(white);           // Antes sería Color.white
    ...
    double raiz = sqrt(1252.2);       // Antes sería Math.sqrt(...)
```

- Argumentos variables -

Ahora Java permite pasar un número variable de argumentos a una función (como sucede con funciones como printf en C). Esto se consigue mediante la expresión "..." a partir del momento en que queramos tener un número variable de argumentos.

```
// Funcion que tiene un parámetro String obligatorio
// y n parámetros int opcionales

public void miFunc(String param, int... args)
{
    ...
    // Una forma de procesar n parametros variables
    for (int argumento: args)
    {
        ...
    }
    ...
}

...
miFunc("Hola", 1, 20, 30, 2);
miFunc("Adios");
```

- Metainformación -

Se tiene la posibilidad de añadir ciertas anotaciones en campos, métodos, clases y otros elementos, que permitan a las herramientas de desarrollo o de despliegue leerlas y realizar ciertas tareas. Veremos muchos ejemplos de esto con el tema de EJB. Se utiliza la nomenclatura **@anotación**, hemos visto durante este tema alguna como **@Override**.

Tema 3: Patrones de Diseño Web

En este tema vamos a dar hablar de los patrones de diseño, qué son y cuándo son útiles en nuestros desarrollos.

3.1.- Introducción

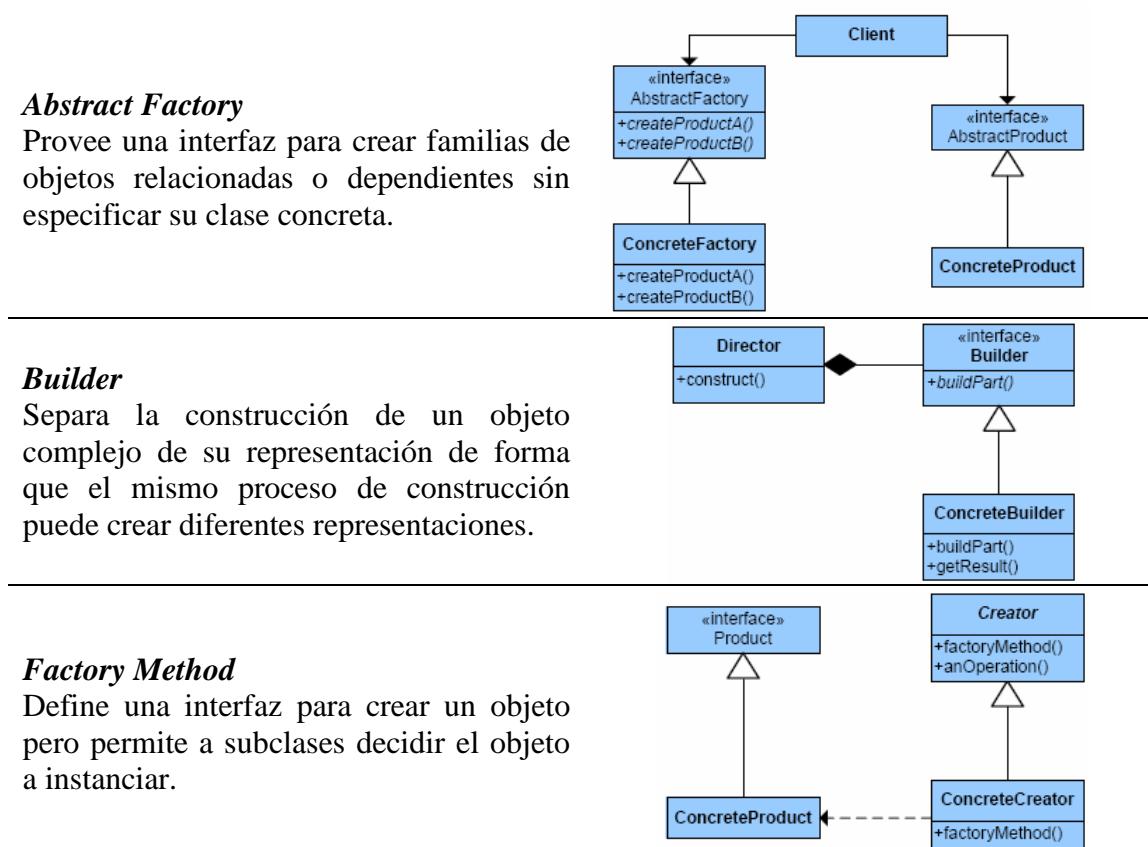
Podemos definir los patrones de diseño como esquemas predefinidos aplicables en diversas situaciones, de forma que el analista se asegura de que el diseño que está aplicando tiene ciertas cualidades que le proporcionan calidad.

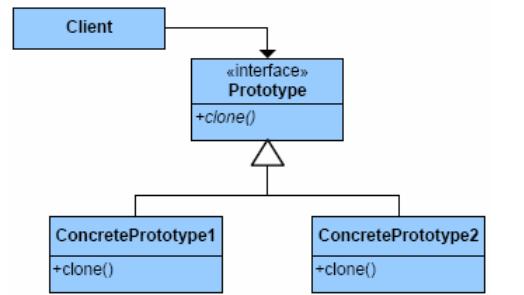
Otra forma de definirlo es: Una situación conocida en la que tenemos un problema que solucionar y una forma de solucionarlo que es apropiada para ese problema en esa situación.

Los patrones nos proporcionan soluciones existosas para esos problemas en esas situaciones, por lo que es muy útil conocerlos y aplicarlos para solucionar los problemas que encajen en su situación. El grupo de los cuatro “GoF” clasificaron los patrones según su propósito en tres categorías. En todas estas categorías hay dos estrategias, una empleando herencia “Estrategia de Clase” y otra empleando asociaciones de objetos “Estrategia de Objeto”.

3.1.1.- Patrones Creacionales

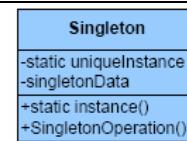
Su objetivo es abstraer el proceso de instanciación de objetos y ocultar los detalles de cómo los objetos se crean o inicializan.





Prototype

Especifica el tipo de objeto a crear usando una instancia como prototipo y copiándola para crear nuevos objetos.



Singleton

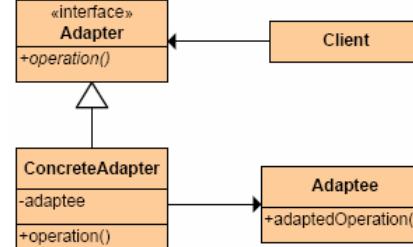
Asegura que una clase dispone de una única instancia y provee un único punto de acceso a la misma.

3.1.2.- Patrones Estructurales

Describen cómo se combinan objetos para obtener estructuras de mayor tamaño con nuevas funcionalidades.

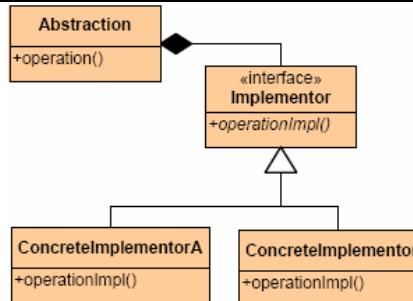
Adapter

Convierte la interfaz de una clase en otra que es la que el cliente necesita.



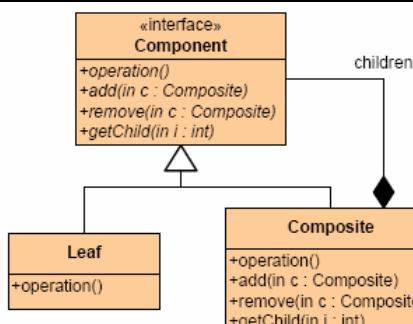
Bridge

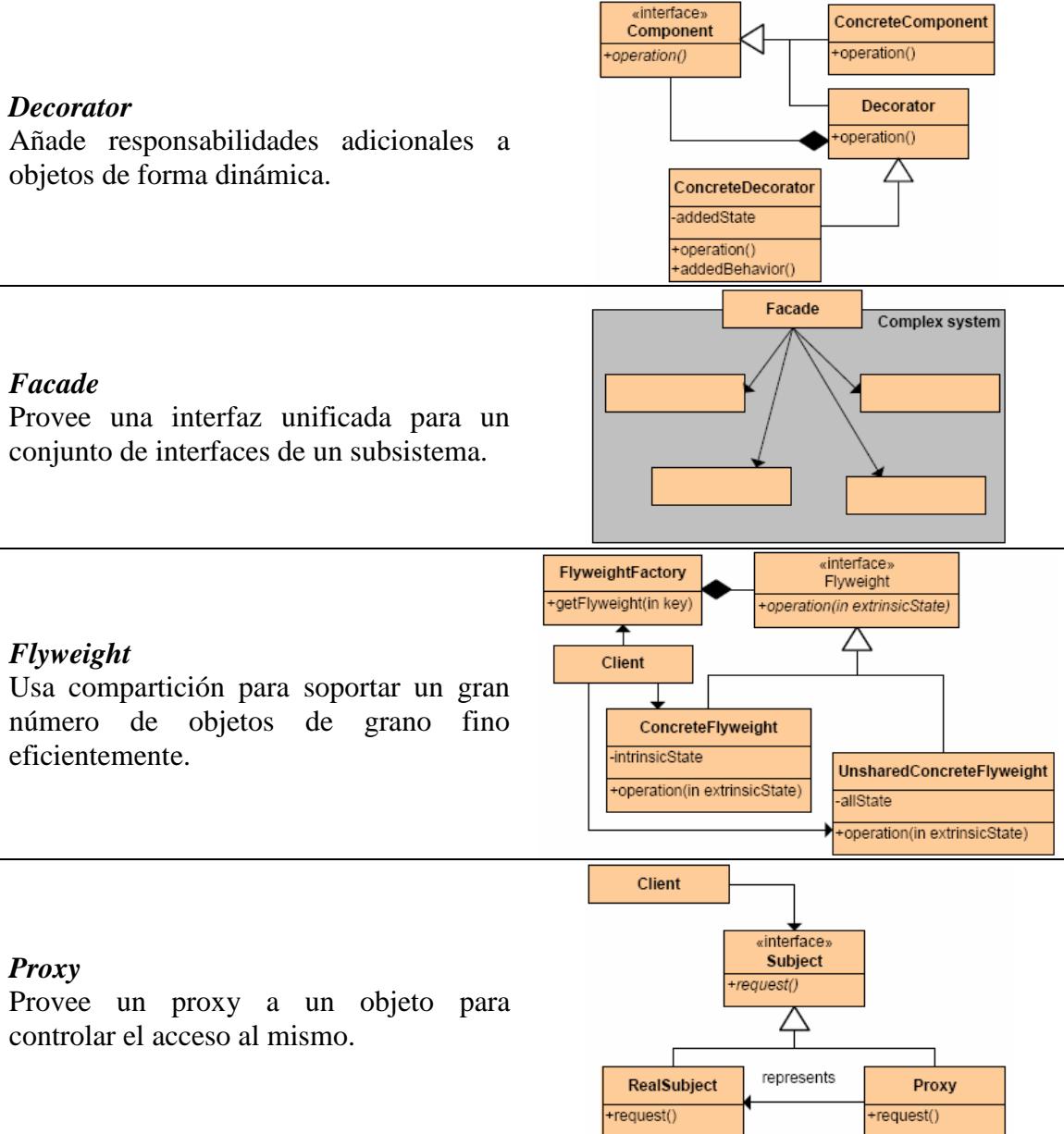
Desacopla una abstracción de su implementación de forma que ambas puedan variar de forma independiente.



Composite

Compone objetos en estructuras arbóreas que representan jerarquías parte-todo.

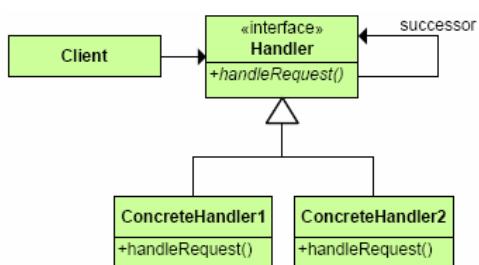




3.1.3.- Patrones Comportamiento

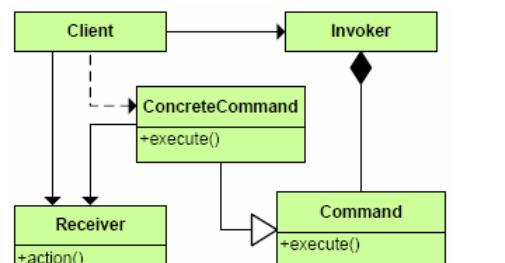
Ayudan a definir la comunicación e interacción entre los objetos de un sistema, reduciendo el acoplamiento.

Chain of Responsibility
Impide el acoplamiento del emisor y receptor de una petición dando a más de un objeto la posibilidad de manejarla.



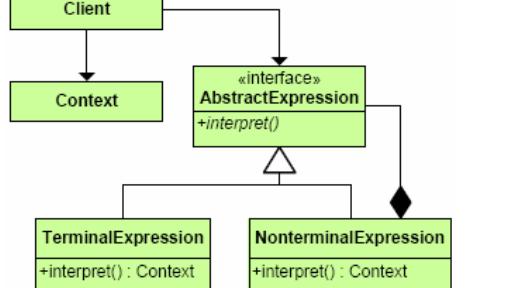
Command

Encapsula una petición como un objeto permitiendo añadir más funcionalidades a estas como parametrizarlas, encolarlas, deshacerlas o hacer log.



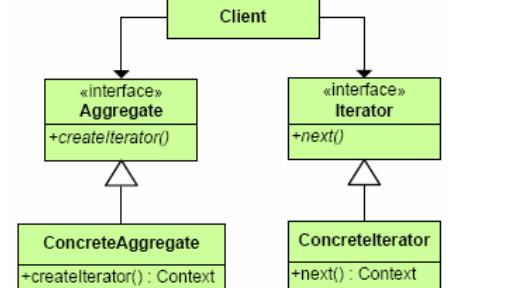
Interpreter

Dado un lenguaje define una representación para su gramática además de un intérprete que permite interpretar sentencias en dicho lenguaje.



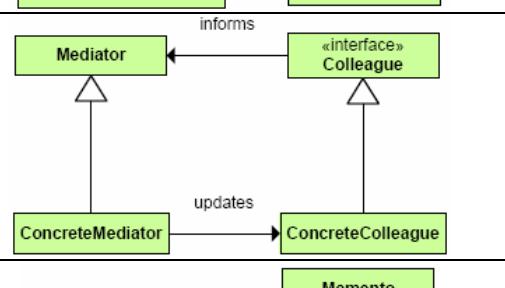
Iterator

Permite recorrer colecciones de objetos sin conocer su secuenciación interna.



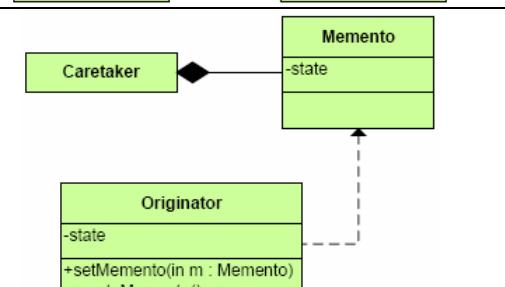
Mediator

Define un objeto que encapsula como interactúan un conjunto de objetos.



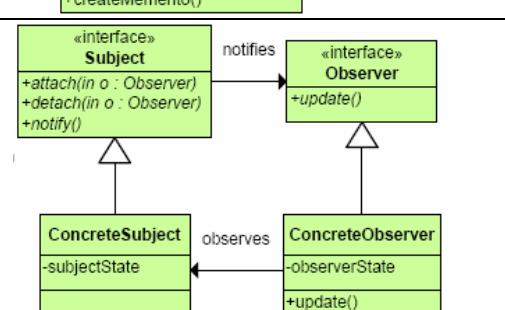
Memento

Sin violar la encapsulación captura y externaliza el estado interno de un objeto de forma que se pueda restaurar a posteriori.



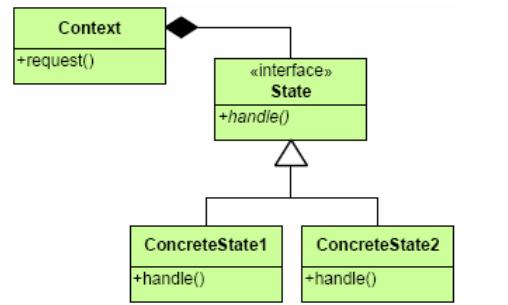
Observer

Define una dependencia uno a muchos entre objetos de forma que cuando un objeto cambia los relacionados son advertidos automáticamente.



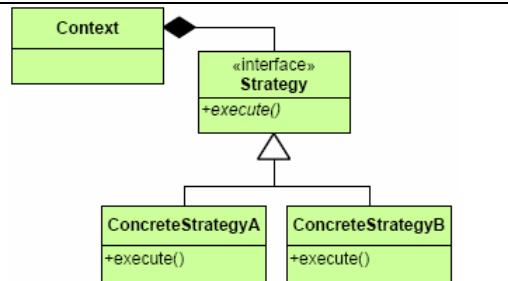
State

Permite a un objeto alterar su comportamiento cuando se modifica su estado interno.



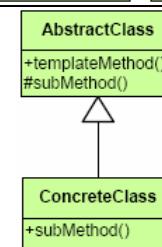
Strategy

Define una familia de algoritmos, encapsulando cada uno y permitiendo que sean intercambiados.



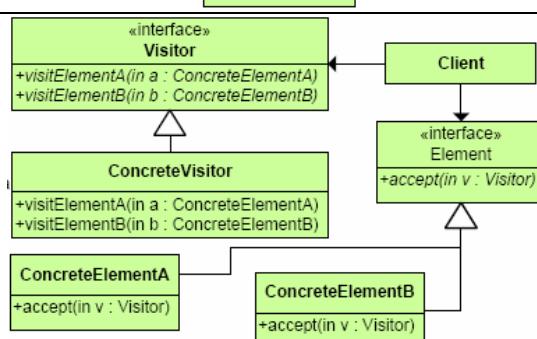
TemplateMethod

Define el esqueleto de un algoritmo delegando algunos pasos de este en alguna subclase.



Visitor

Representa una operación a realizar sobre los elementos de una estructura de objetos.



3.2.- Patrones Web

La irrupción del desarrollo web hace aparecer un catálogo específico de patrones pensados para el mundo web y más concretamente para el desarrollo de aplicaciones Java EE. En el libro “J2EE Patterns, Best Practices & Design Strategies”, aparecen 15 patrones específicos que se dividen en tres capas.

3.2.1.- Capa de Presentación

- **Decorating/Intercepting Filter:** Objeto ubicado entre el cliente y los componentes web, procesa las peticiones y las respuestas.
- **Front Controller/Component:** Un objeto que acepta todos los requerimientos de un cliente y los direcciona a los manejadores apropiados. El patrón Front Controller podría dividir la funcionalidad en 2 diferentes objetos: el Front Controller y el Dispatcher. En ese caso, El Front Controller acepta todos los

requerimientos de un cliente y realiza la autenticación, y el Dispatcher direcciona los requerimientos a manejadores apropiados.

- ***View Helper***: Un objeto helper que encapsula la lógica de acceso a datos en beneficio de los componentes de la presentación. Por ejemplo, los JavaBeans pueden ser usados como patrón View Helper para las páginas JSP.
- ***Composite View***: Un objeto vista que está compuesto de otros objetos vista. Por ejemplo, una página JSP que incluye otras páginas JSP y HTML usando la directiva include o el action include es un patrón Composite View.
- ***Service to Worker***: Es como el patrón de diseño MVC con el Controlador actuando como Front Controller pero con una cosa importante: aquí el Dispatcher (el cual es parte del Front Controller) usa View Helpers a gran escala y ayuda en el manejo de la vista.
- ***Dispatcher View***: Es como el patrón de diseño MVC con el controlador actuando como Front Controller pero con un asunto importante: aquí el Dispatcher (el cual es parte del Front Controller) no usa View Helpers y realiza muy poco trabajo en el manejo de la vista. El manejo de la vista es manejado por los mismos componentes de la Vista.

3.2.2.- Capa de Negocio

- ***Business Delegate***: Un objeto que reside en la capa de presentación y en beneficio de los otros componentes de la capa de presentación llama a métodos remotos en los objetos de la capa de negocios.
- ***Value/Data Transfer/Replicate Object***: Un objeto serializable para la transferencia de datos sobre la red.
- ***Session/Session Entity/Distributed Facade***: El uso de un bean de sesión como una fachada (facade) para encapsular la complejidad de las interacciones entre los objetos de negocio y participantes en un flujo de trabajo. El Session Façade maneja los objetos de negocio y proporciona un servicio de acceso uniforme a los clientes.
- ***Aggregate Entity***: Un bean entidad que es construido o es agregado a otros beans de entidad.
- ***Value Object Assembler***: Un objeto que reside en la capa de negocios y crea Value Objects cuando es requerido.
- ***Value List Handler / Page-by-Page Iterator / Paged List***: Es un objeto que maneja la ejecución de consultas SQL, caché y procesamiento del resultado. Usualmente implementado como beans de sesión.
- ***Service Locator***: Consiste en utilizar un objeto Service Locutor para abstraer toda la utilización JNDI y para ocultar las complejidades de la creación del

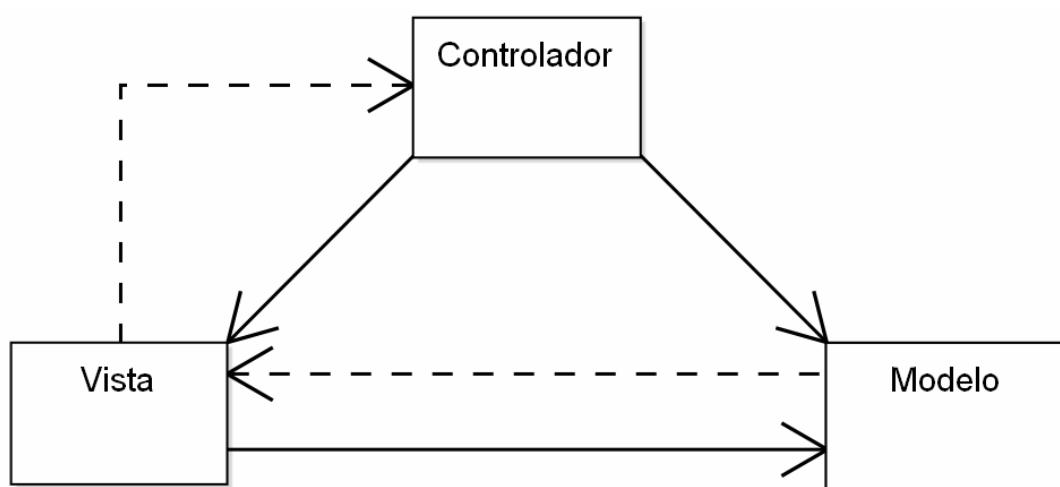
contexto inicial, de búsqueda de objetos home EJB y recreación de objetos EJB. Varios clientes pueden reutilizar el objeto Service Locutor para reducir la complejidad del código, proporcionando un punto de control.

3.2.3.- Capa de Integración

- **Data Access / Object Service / Activator:** Consiste en utilizar un objeto de acceso a datos para abstraer y encapsular todos los accesos a la fuente de datos. El DAO maneja la conexión con la fuente de datos para obtener y almacenar datos.
- **Service Activator:** Se utiliza para recibir peticiones y mensajes asíncronos de los clientes. Cuando se recibe un mensaje, el Service Activator localiza e invoca a los métodos de los componentes de negocio necesarios para cumplir la petición de forma asíncrona.

3.3.- Patrón MVC

El patrón Model-View-Controller (MVC) fue descrito por primera vez en 1979 por Trygve Reenskaug, entonces trabajando en Smalltalk en laboratorios de investigación de Xerox. La implementación original está descrita a fondo en “*Programación de Aplicaciones en Smalltalk-80(TM): Como utilizar Modelo Vista Controlador*”.



Se trata de desacoplar la vista del modelo/lógica de negocio. Para lograrlo es necesario incluir un controlador que gestiona los eventos producidos por la vista y desencadena consecuencias en el modelo, pero la vista no se comunica con este de forma directa, de forma que es factible obtener varias vistas diferentes sobre el mismo modelo.

3.3.1.- Descripción del Patrón

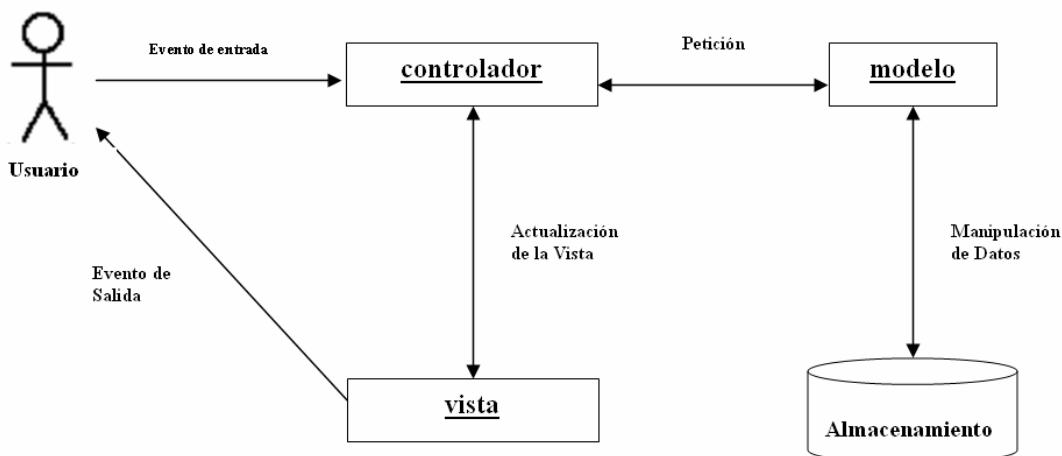
Modelo Vista Controlador (MVC) es un patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres

componentes distintos. El patrón MVC se ve frecuentemente en aplicaciones web, donde la vista es la página HTML, el modelo se aloja en el SGBD y el código del servidor se encarga por un lado de la lógica de negocio y por otro del control de los eventos llegados desde el cliente web.

Dentro de este patrón, cada uno de los componentes juega un papel que podemos describir de la siguiente forma:

- **Modelo:** Esta es la representación específica de la información con la cual el sistema opera. La lógica de datos asegura la integridad de éstos y permite derivar nuevos datos; por ejemplo, no permitiendo comprar un número de unidades negativo, calculando si hoy es el cumpleaños del usuario o los totales, impuestos o importes en un carrito de la compra.
- **Vista:** Este presenta el modelo en un formato adecuado para interactuar, usualmente la interfaz de usuario.
- **Controlador:** Este responde a eventos, usualmente acciones del usuario e invoca cambios en el modelo y probablemente en la vista.

De forma gráfica puede representarse como:



Aunque se pueden encontrar diferentes implementaciones de MVC, el flujo que sigue el control generalmente es el siguiente:

- 1) El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace)
- 2) El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos (handler) o callback.
- 3) El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, el controlador actualiza el carro de la compra del usuario). Los controladores complejos están a menudo estructurados usando un patrón de comando que encapsula las acciones y simplifica su extensión.

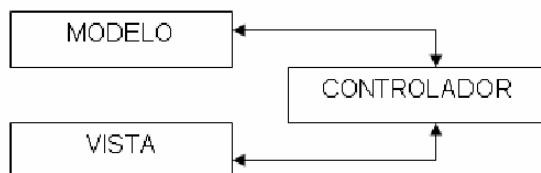
- 4) El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se reflejan los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). El modelo no debe tener conocimiento directo sobre la vista, puede emplearse el patrón observer.
- 5) La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

3.3.2.- Extensiones del Patrón

Se han desarrollado a lo largo de los años, desde la presentación de este patrón a la comunidad científica 3 variantes fundamentales, que se presentan brevemente a continuación.

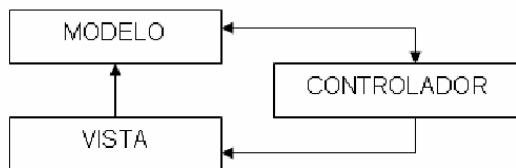
- Variante I -

Variante en la cual no existe ninguna comunicación entre el Modelo y la Vista y ésta última recibe los datos a mostrar a través del Controlador.



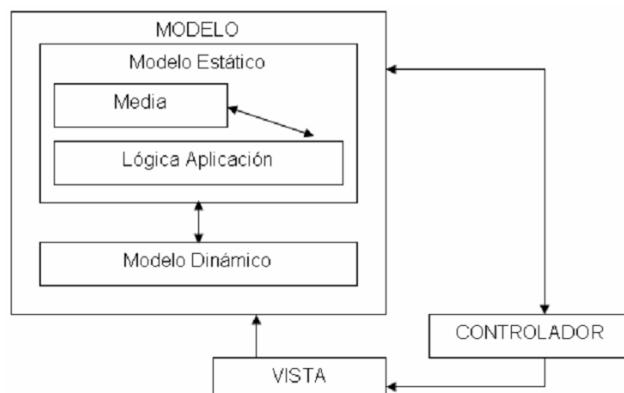
- Variante II -

Variante en la cual se desarrolla una comunicación entre el Modelo y la Vista, donde esta última al mostrar los datos, los extrae directamente del Modelo, dada una indicación del Controlador, disminuyendo así el conjunto de responsabilidades de este último.



- Variante III -

Variante en la cual se diversifican las funcionalidades del Modelo teniendo en cuenta las características de las aplicaciones multimedia, donde tienen un gran peso las medias utilizadas en éstas.



3.3.3.- Inversión de Control/Inyección de Dependencias

La Inversión de Control (IoC) y la Inyección de Dependencias (DI) son dos estrategias muy presentes en todos los marcos de desarrollo de aplicaciones Web con Java, y también empleadas conjuntamente con el patrón MVC para obtener código más desacoplado.

- Inversión de Control -

Se basa en el “Principio de Hollywood: No me llames, ya te llamo yo.” Todos los framework de uso común para desarrollo de aplicaciones web invocan en determinados momentos al código desarrollado por el programador para una determinada aplicación, en ningún caso es el programador el que invoca al código del framework. Es el esquema común para todos aquellos objetos que disponen de un ciclo de vida.

- Inyección de Dependencias -

Este es un concepto más rentable. Tradicionalmente el desarrollador debía escribir código para lograr recursos. Por ejemplo, si deseaba obtener una conexión a base de datos, debía programar el código para lograrla antes de ejecutar el código importante. Esta codificación además de ser repetitiva, era fuente de errores.

En entornos donde los objetos residen en un contenedor que los alberga y controla, es posible solicitarle al contenedor que de acceso a recursos, que los inyecte en el objeto antes de pasar el control a los métodos programados por el desarrollador.

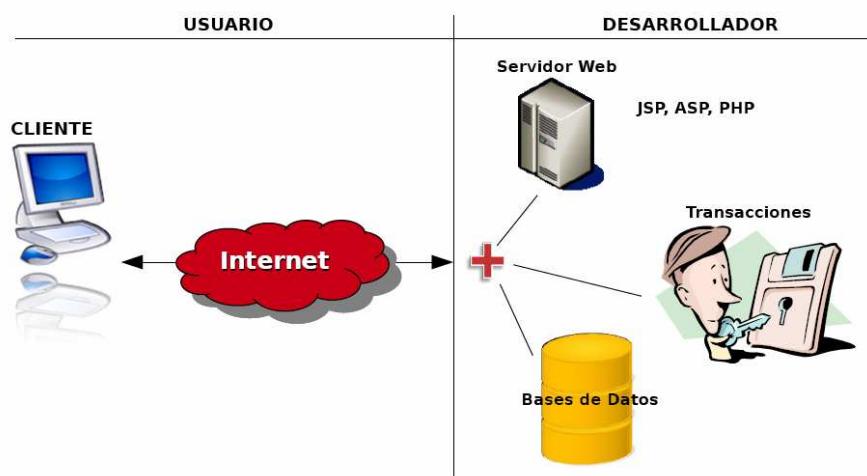
Siguiendo con el ejemplo de la conexión de base de datos, ahora usando una anotación con un nombre JNDI podemos indicar al contenedor que inyecte en la propiedad anotada el recurso con el nombre JNDI dado.

Tema 4: Arquitectura Java EE

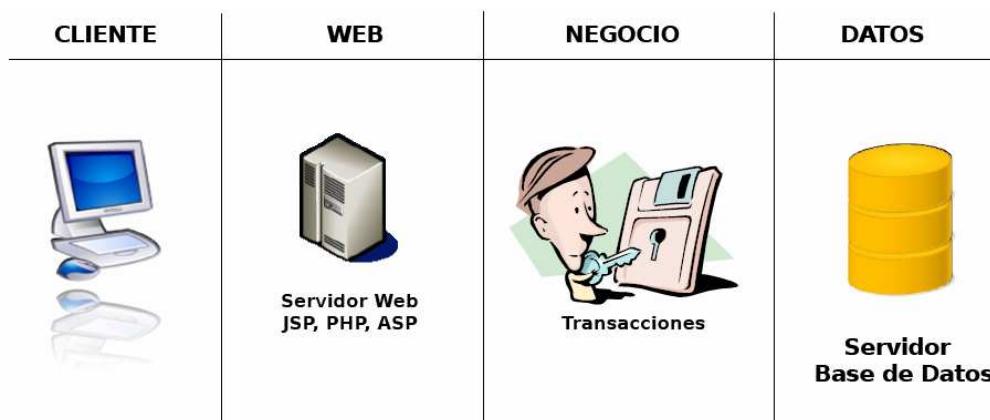
En este tema vamos a ver los componentes más importantes de la arquitectura Java EE anteriormente conocida como J2EE, y los servicios que provee y podemos emplear en las aplicaciones web.

4.1.- Modelo de Capas

Para simplificar el desarrollo y clarificar las responsabilidades de cada uno de los componentes de un aplicativo, las aplicaciones web se dividen en capas. Inicialmente en un entorno sin capas el desarrollo de aplicaciones entremezclaba todas las tareas del lado servidor.



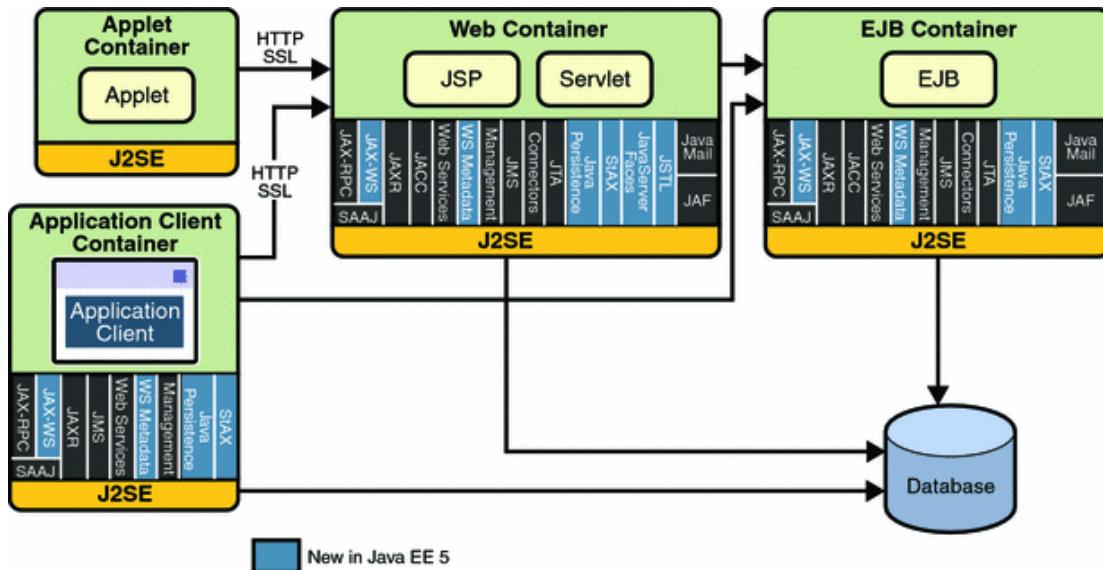
Posteriormente surge el modelo de 3 capas que separa capa cliente, servidor y de datos, y con la evolución de los servidores de aplicaciones surge la necesidad de crear una cuarta capa, para separar la lógica de negocio de la transaccional.



Este tipo de arquitectura proporciona mayor control sobre los elementos que entran en juego en el desarrollo de aplicaciones web.

4.1.1.- Tipos de Contenedores

Dentro de la plataforma Java EE disponemos de varios contenedores. Un contenedor es un entorno de ejecución específico para un conjunto de objetos de un determinado tipo y con unos fines concretos.



Por ejemplo, el contenedor Web es un contenedor específico para Servlets y páginas JSP, y que por tanto provee ciertos servicios útiles para este tipo de objetos, como son la librería JSTL de etiquetas JSP, o la infraestructura de JSF. En el contenedor de EJB no se proveen estos servicios ya que no son necesarios para los objetos que albergan.

4.1.2.- Servicios Java EE

Voy a describir muy por encima algunos de los servicios que provee Java EE y que podemos emplear en cualquier servidor de aplicaciones que siga este estándar.

- **Java Transaction API (JTA)**: Consta de dos partes y define las interfaces entre el manejador de transacciones y las partes involucradas en un sistema de transacciones distribuido, el manejador de recursos, el servidor de aplicaciones y las aplicaciones transaccionales.
- **Java Persistente API (JPA)**: Para dotar de persistencia a objetos del lenguaje.
- **Java Message Service (JMS)**: Sistema de mensajería, permite mensajes punto a punto y sistemas publicado suscrito.
- **Java Naming Direct Interface (JNDI)**: Sistema de nombrado para localizar recursos o componentes de proveedores.
- **JavaMail**: Para manejo de correo electrónico.
- **Java Beans Active Framework (JAF)**: Manejo de datos en diferentes tipos mime, ligado al uso del correo.
- **Java API for XML Processing (JAXP)**: Soporte para el manejo de XML.
- **Java EE Connector Arquitectura**: Permite el acceso a otro tipo de sistemas de información empresarial por medio del desarrollo de un plugin, similar a los conectores JDBC.

- **Java Autentificación and Authorization Service (JAAS)**: Provee servicios de seguridad de autenticación y autorización.
- **Servicios Web (JAX-WS)**: Para manejo de servicios web.

Además de todos estos servicios provee otros para manejo de protocolos HTTP/HTTPS, IIOP/RMI, JDBC, etc...

4.1.3.- Ensamblado y Empaquetado

Una aplicación estándar Java EE se divide en varios módulos, por un lado los módulos Web y por otro los módulos EJB. Cada uno de estos módulos tiene sus particularidades y su forma de empaquetar para trasladarse de un servidor a otro. Del mismo modo existe una forma de empaquetar una aplicación Java EE completa.

- Módulo EAR -

Los archivos EAR (Enterprise Application Archive) son simples archivos comprimidos en formato zip, con una estructura predefinida:

```
/*.war  
/*.jar  
/META-INF/application.xml
```

Por un lado los archivos de módulos web (.war), los archivos de módulos ejb y otras librerías (.jar) y lo más importante, el descriptor de despliegue de la aplicación, en el que se describen los diferentes módulos que contiene la aplicación y que tiene este aspecto:

```
<?xml version="1.0" encoding="ASCII"?>  
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns="http://java.sun.com/xml/ns/javaee"  
    xmlns:application="http://java.sun.com/xml/ns/javaee/application_5.xsd"  
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  
                        http://java.sun.com/xml/ns/javaee/application_5.xsd"  
    version="5">  
    <display-name>EjemploEAR</display-name>  
    <module>  
        <web>  
            <web-uri>Ejemplo.war</web-uri>  
            <context-root>Ejemplo</context-root>  
        </web>  
    </module>  
</application>
```

- Módulo WEB -

Los módulos Web se empaquetan en Web Application Archives (WAR), que son ficheros zip con un formato predefinido:

```
/*.*  
/WEB-INF  
    /web.xml  
    /classes/*.class  
    /lib/*.jar  
    /tlds/*.tld
```

El contenido de estos ficheros es muy variable dependiendo de la tecnología empleada para desarrollar la aplicación. A nivel raíz se dispondrá el contenido estático, archivos html, hojas de estilo, imágenes, y archivos jsp (si se emplean).

Dentro del directorio WEB-INF habrá un directorio de classes para las clases Java empleadas en el módulo web, un lib para las librerías necesarias y un tlds para definir las librerías de tag personalizadas.

Al igual que sucedía con el EAR existe un fichero descriptor denominado web.xml, que nos permite definir los elementos que forman parte de nuestro módulo web, principalmente serán:

- **Servlets**: Clase java con un ciclo de vida específico para atender peticiones web.
- **Filtros**: Clase java específica para filtrar peticiones web.
- **Listeners**: Clase java adecuada para manejar eventos de servlets, session o request.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="DiarioDigital" version="2.5">

    <display-name>DiarioDigital</display-name>
    <description>DiarioDigital</description>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>

    <filter>
        <display-name>RichFaces Filter</display-name>
        <filter-name>richfaces</filter-name>
        <filter-class>org.ajax4jsf.Filter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>richfaces</filter-name>
        <servlet-name>Faces Servlet</servlet-name>
    </filter-mapping>

    <listener>
        <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
    </listener>

    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.jsf</url-pattern>
    </servlet-mapping>
</web-app>
```

- Módulo EJB -

Al igual que se empaquetaban módulos Web en ficheros .war se hace lo propio con los módulos EJB en ficheros .jar. La estructura de estos es:

| |
|-----------------------|
| /*.class |
| /META-INF/ejb-jar.xml |

En este caso contienen los archivos .class del módulo y un descriptor que define los tipos de EJB contenidos en el módulo.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                               http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
    <display-name>SimpleEJB</display-name>
</ejb-jar>
```

No tiene porqué especificarse nada más en este archivo si se hace uso de anotaciones en el propio EJB.

4.2.- Eclipse y Java EE

Vamos a ver cómo manejar un proyecto Java EE a través de Eclipse. Esta no es la forma en la que se trabaja con FundeWeb, ya que en ese caso se emplea un proyecto Maven y se ejecutan objetivos Maven para obtener los compilados o empaquetados del proyecto, pero nos servirá para ver las tecnologías del tema siguiente.

4.2.1.- Proyectos Java EE

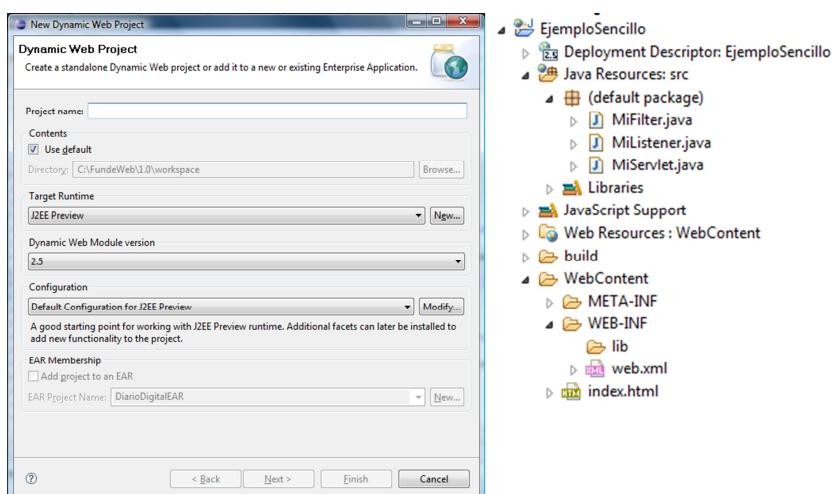
Para poder desarrollar una aplicación web vamos a necesitar un servidor de aplicaciones, pero inicialmente podemos hacerlo empleando el “J2EE Preview” de eclipse, aunque no podamos probar nada.

Vamos a crear un proyecto de ejemplo para testar las posibilidades que nos ofrece Eclipse y cómo nos facilita la vida a la hora de crear los diferentes componentes de un módulo web.

- Dynamic Web Project -

Nos permite crear un módulo web que posteriormente podemos desplegar en un servidor para testarlo desde la propia aplicación, e incluso hacer debug. La opción para crear el proyecto aparece en el menú File > New.

Generalmente se muestran algunos de los tipos de proyecto y archivo más empleados, pero si no se localiza con “Other” accedemos al listado completo, desde el que podemos seleccionar el proyecto que queremos en la carpeta Web.

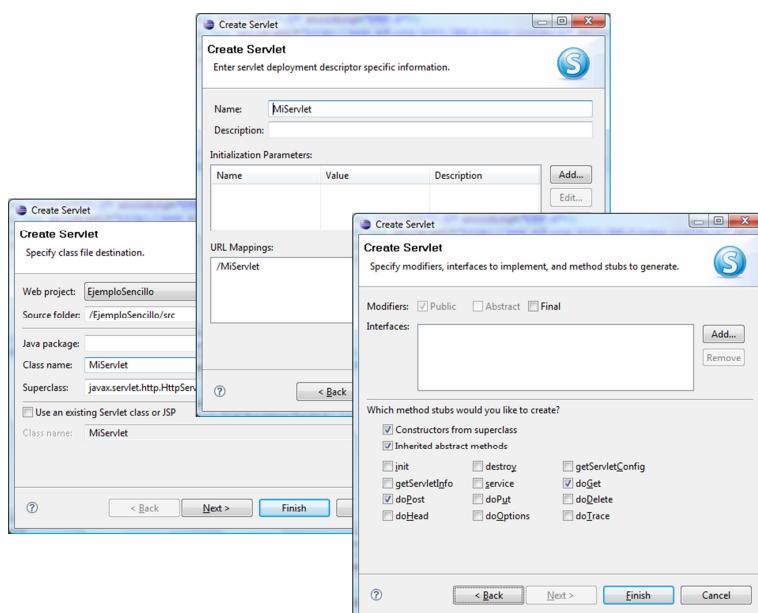


Tras crear el proyecto nos despliega una serie de directorios donde ubicaremos los diferentes archivos según la estructura de los módulos web que ya conocemos. Dentro del directorio WEB-INF tenemos el descriptor del módulo (web.xml) y fuera contenido estático como el index.html. Otros componentes como servlets, filtros y listeners aparecen en la carpeta de código fuente y se ubicarán en el directorio classes.

Ahora vamos a crear un Servlet un Filtro y un Listener con el asistente de eclipse para ver qué opciones nos ofrece y qué código genera.

- Servlet -

Desde el menú contextual de la carpeta de código fuente podemos crear un servlet. Esta opción nos lanza un asistente para crear un servlet.



En la primera nos pide el nombre del servlet, en la segunda el tipo de mapeo, y en la última los métodos a implementar. Lo normal es implementar únicamente *service*, a menos que se quiera específicamente responder a un tipo de método concreto.

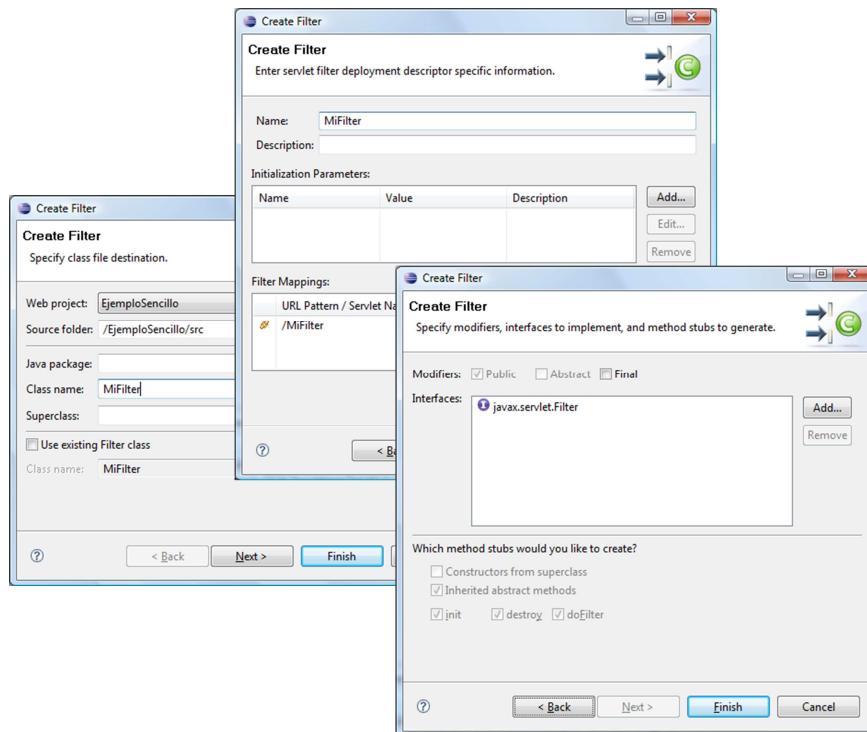
```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class MiServlet extends HttpServlet {
    public MiServlet() { super(); }
    protected void service(HttpServletRequest request,
                          HttpServletResponse response) throws ServletException, IOException {
        ..
    }
}
```

Además de crearnos la clase, este asistente nos habrá creado la entrada en el fichero web.xml declarando el servlet que acaba de crear. Es importante conocer el ciclo de vida de un servlet. Los objetos servlet son creados siguiendo el patrón singleton y cada petición requerida por parte de un cliente hacia estos es servida mediante la ejecución del método correspondiente dentro de un hilo. Si no se tiene en cuenta este aspecto podemos cometer graves errores, por ejemplo una variable miembro de una clase servlet (no estática) sería compartida por varias ejecuciones simultáneas del hilo.

de ejecución del método de atención de peticiones, dando lugar a situaciones inesperadas.

- Filtro -

Podemos obrar de igual forma para el caso de querer crear un filtro.

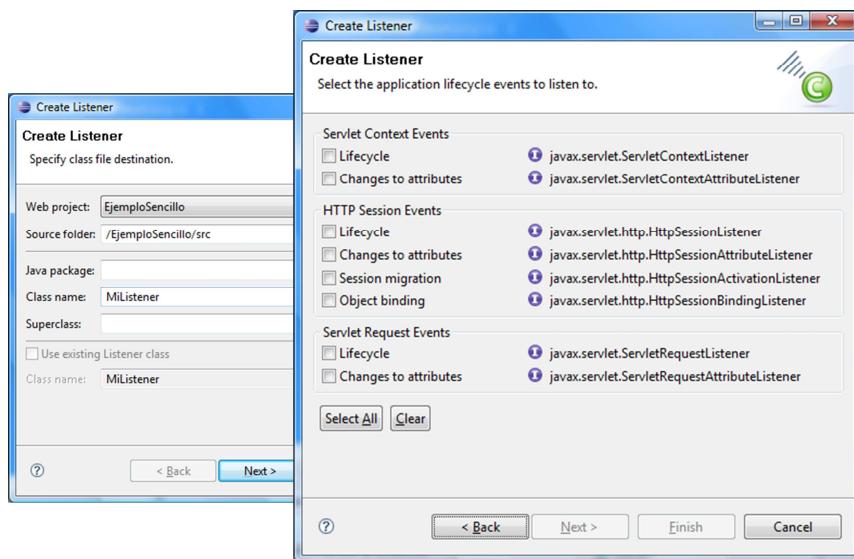


En este caso nos pide el nombre, el mapeo y posteriormente nos facilita la interface que implementa, generando la entrada en el fichero web.xml y el código:

```
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
public class MiFilter implements Filter {
    public MiFilter() { .. }
    public void destroy() { .. }
    public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException {
        // place your code here
        // pass the request along the filter chain
        chain.doFilter(request, response);
    }
    public void init(FilterConfig fConfig) throws ServletException { .. }
}
```

- Listener -

Por último crearemos un filtro del mismo modo. En este caso solicita el nombre y posteriormente pide que le digamos qué eventos son los que queremos capturar. En función de los eventos que seleccionemos la última pantalla añadirá más o menos interfaces a implementar y la clase generada tendrá más o menos métodos.



Para el caso de capturar el Lifecycle de las Sesiones se obtiene una clase como:

```
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;
public class MiListener implements HttpSessionListener {
    public MiListener() { .. }
    public void sessionDestroyed(HttpSessionEvent arg0) { .. }
    public void sessionCreated(HttpSessionEvent arg0) { .. }
}
```

Al igual que sucedía con el resto de elementos, el asistente también nos crea las entradas correspondientes en el archivo web.xml dejándolo listo para su ejecución. Podemos jugar con estos componentes, para realizar tareas como hacer un loggin de peticiones, audituar las altas de sesiones en una aplicación, etc...

4.2.2.- Desarrollo y Ejecución

Una vez que tenemos la aplicación desarrollada, o al menos iniciada, necesitamos probarla para continuar. Para hacerlo podemos emplear los servidores de la vista “Servers”. Creamos un servidor y añadimos nuestro proyecto a este servidor mediante el menú contextual, aunque se puede hacer de otras formas.

También podemos probar nuestra aplicación en servidores al margen de Eclipse, usando la opción “Export” podemos generar un WAR de nuestro proyecto y desplegarlo en cualquier servidor para probarlo.

Una vez arrancado el servidor podemos hacerlo de forma normal o en modo debug. El modo debug nos servirá para establecer puntos de ruptura en nuestro código y depurar paso a paso. Además de este tipo de debug, existe la posibilidad de monitorizar las peticiones http incluyendo la vista “TCP/IP Monitor”, que antepone un proxy entre el navegador y nuestro servidor de forma que podemos visualizar todas las peticiones y las respuestas resultado de la comunicación entre ambos.

Tema 5: Tecnologías Java EE

En este tema llegamos al punto importante, las tecnologías de desarrollo empleadas en el modelo, la vista y el control.

5.1.- *Tecnologías Vista: JSF*

De entre todas las tecnologías de vista que existen nos centrarmos en el estándar de SUN, Java Server Faces. No vamos a discutir qué tecnología o framework de desarrollo es más adecuado, la discusión no lleva a ningún sitio. Optar por el estándar nos aporta varias ventajas:

- Comunidad de desarrollo amplia y duradera.
- Apoyo de las compañías tecnológicas.
- Adaptación de las mejores ideas surgidas de otras tecnologías.

Aunque también tiene sus inconvenientes:

- No estar a la última, cierto retraso para asimilar o simplificar nuevas tecnologías.
- Depender de las implementaciones no siempre totalmente ajustadas de terceros.

De cualquier forma, apostar por otras tecnologías no estándar requiere un mayor coste de análisis y evaluación de las mismas, y las ventajas a la larga se consiguen con cualquier tecnología.

5.1.1.- Introducción

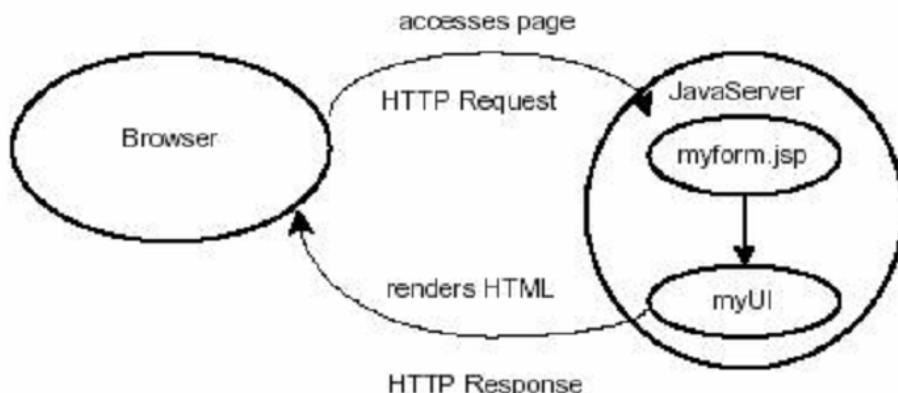
La tecnología JavaServer Faces es un marco de trabajo de interfaces de usuario del lado de servidor para aplicaciones Web basadas en tecnología Java. Los principales componentes de la tecnología JavaServer Faces son:

- Un API y una implementación de referencia para: representar componentes UI y manejar su estado; manejo de eventos, validación del lado del servidor y conversión de datos; definir la navegación entre páginas; soportar internacionalización y accesibilidad; y proporcionar extensibilidad para todas estas características.
- Una librería de etiquetas JavaServer Pages (JSP) personalizadas para dibujar componentes UI dentro de una página JSP.

Este modelo de programación bien definido y la librería de etiquetas para componentes UI, facilitan de forma significativa la tarea de la construcción y mantenimiento de aplicaciones Web con UIs del lado del servidor. Con un mínimo esfuerzo, podemos:

- Conectar eventos generados en el cliente a código de la aplicación en el servidor.
- Mapear componentes UI a una página de datos del lado del servidor.
- Construir un UI con componentes reutilizables y extensibles.
- Grabar y restaurar el estado del UI más allá de la vida de las peticiones.

Como se puede apreciar en la siguiente figura, el interface de usuario que creamos con la tecnología JavaServer Faces (representado por **myUI** en el gráfico) se ejecuta en el servidor y se renderiza en el cliente.



La página JSP, **myform.jsp**, dibuja los componentes del interface de usuario con etiquetas personalizadas definidas por la tecnología JSF. El UI de la aplicación Web (representado por **myUI** en la figura) maneja los objetos referenciados por la página JSP:

- Los objetos componentes que mapean las etiquetas sobre la página JSP.
- Los oyentes de eventos, validadores, y los conversores que está registrados en los componentes.
- Los objetos del modelo que encapsulan los datos y las funcionalidades de los componentes específicos de la aplicación.

Una de las grandes ventajas de la tecnología JavaServer Faces es que ofrece una clara separación entre el comportamiento y la presentación. Las aplicaciones Web construidas con tecnología JSP conseguían parcialmente esta separación. Sin embargo, una aplicación JSP no puede mapear peticiones HTTP al manejo de eventos específicos de los componentes o manejar elementos UI como objetos con estado en el servidor. La tecnología JavaServer Faces nos permite construir aplicaciones Web que implementan una separación entre el comportamiento y la presentación, que tradicionalmente son ofrecidas por arquitectura UI del lado del cliente.

La separación de la lógica de la presentación también le permite a cada miembro del equipo de desarrollo de una aplicación Web enfocarse en su parte del proceso de desarrollo, y proporciona un sencillo modelo de programación para enlazar todas las piezas. Por ejemplo, los Autores de páginas sin experiencia en programación pueden usar las etiquetas de componentes UI de la tecnología JavaServer Faces para enlazar código de la aplicación desde dentro de la página Web sin escribir ningún script.

Otro objetivo importante de la tecnología JavaServer Faces es mejorar los conceptos familiares de componente-UI y capa-Web sin limitarnos a una tecnología de script particular o un lenguaje de marcas. Aunque la tecnología JavaServer Faces incluye una librería de etiquetas JSP personalizadas para representar componentes en una página JSP, los APIs de la tecnología JavaServer Faces se han creado directamente sobre el API JavaServlet. Esto nos permite hacer algunas cosas: usar otra tecnología de

presentación junto a JSP, crear nuestros propios componentes personalizados directamente desde las clases de componentes, y generar salida para diferentes dispositivos cliente.

Pero lo más importante, la tecnología JavaServer Faces proporciona una rica arquitectura para manejar el estado de los componentes, procesar los datos, validar la entrada del usuario, y manejar eventos.

5.1.2.- Ciclo de Vida JSF

El ciclo de vida de una página JavaServer Faces es similar al de una página JSP: El cliente hace una petición HTTP de la página y el servidor responde con la página traducida a HTML. Sin embargo, debido a las características extras que ofrece la tecnología JavaServer Faces, el ciclo de vida proporciona algunos servicios adicionales mediante la ejecución de algunos pasos extras.

Los pasos del ciclo de vida se ejecutan dependen de si la petición se originó o no desde una aplicación JavaServer Faces y si la respuesta es o no generada con la fase de renderizado del ciclo de vida de JavaServer Faces. Esta sección explica los diferentes escenarios del ciclo de vida. Luego explica cada uno de estas fases del ciclo de vida.

Una aplicación JavaServer Faces soporta dos tipos diferentes de respuestas y dos tipos diferentes de peticiones:

- **Respuesta Faces:** Una respuesta servlet que se generó mediante la ejecución de la fase de Renderizar la Respuesta del ciclo de vida de procesamiento de la respuesta.
- **Respuesta No-Faces:** Una respuesta servlet que no se generó mediante la ejecución de la fase de Renderizar de la Respuesta. Un ejemplo es una página JSP que no incorpora componentes JavaServer Faces.
- **Petición Faces:** Una petición servlet que fue enviada desde una Respuesta Faces previamente generada. Un ejemplo es un formulario enviado desde un componente de interface de usuario JavaServer Faces, donde la URI de la petición identifica el árbol de componentes JavaServer Faces para usar el procesamiento de petición.
- **Petición No-Faces:** Una petición Servlet que fue enviada a un componente de aplicación como un servlet o una página JSP, en vez de directamente a un componente JavaServer Faces.

La combinación de estas peticiones y respuestas resulta en tres posibles escenarios del ciclo de vida que pueden existir en una aplicación JavaServer Faces:

- Escenario 1: Petición No-Faces Respuesta Faces -

Un ejemplo de este escenario es cuando se pulsa un enlace de una página HTML que abre una página que contiene componentes JavaServer Faces. Para dibujar una Respuesta Faces desde una Petición No-Faces, una aplicación debe proporcionar un mapeo FacesServlet en la URL de la página que contiene componentes JavaServer Faces. FacesServlet acepta peticiones entrantes y pasa a la implementación del ciclo de vida para su procesamiento.

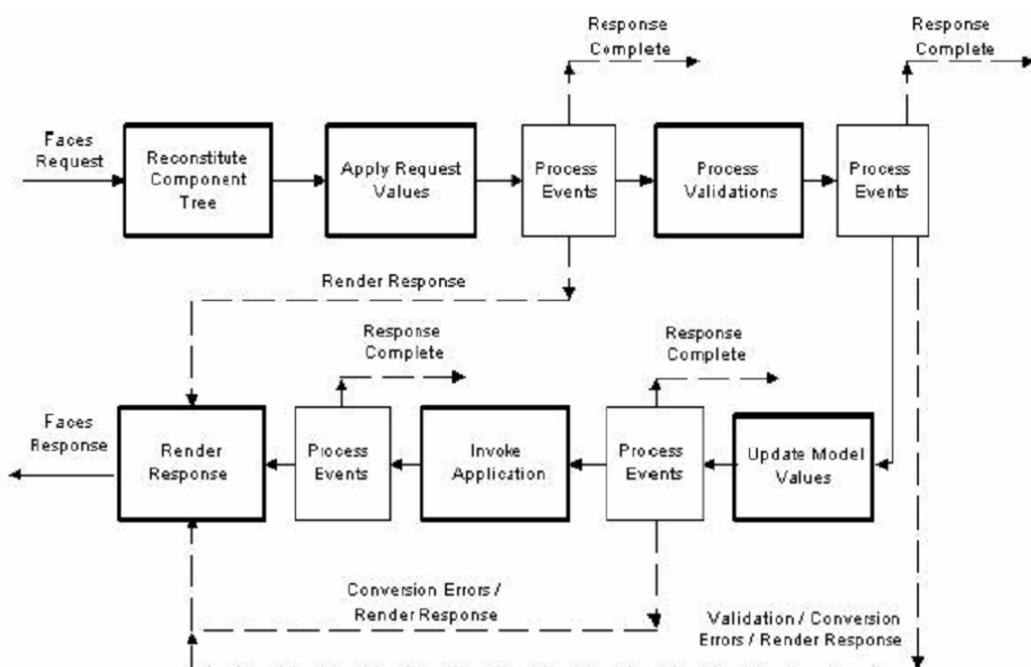
- Escenario 2: Petición Faces Respuesta No-Faces -

Algunas veces una aplicación JavaServer Faces podría necesitar redirigir la salida a un recurso diferente de la aplicación Web diferente o generar una respuesta que no contiene componentes JavaServer Faces. En estas situaciones, el desarrollador debe saltarse la fase de renderizado (Renderizar la Respuesta) llamando a `FacesContext.responseComplete`. `FacesContext`, que contiene toda la información asociada con una Petición Faces particular. Este método se puede invocar durante las fases Aplicar los Valores de Respuesta, Procesar Validaciones o Actualizar los Valores del Modelo.

- Escenario 3: Petición Faces Respuesta Faces -

Es el escenario más común en el ciclo de vida de una aplicación JavaServer Faces. Este escenario implica componentes JavaServer Faces enviando una petición a una aplicación JavaServer Faces utilizando el `FacesServlet`. Como la petición ha sido manejada por la implementación JavaServer Faces, la aplicación no necesita pasos adicionales para generar la respuesta. Todos los oyentes, validadores y conversores serán invocados automáticamente durante la fase apropiada del ciclo de vida estándar, como se describe en la siguiente sección.

La mayoría de los usuarios de la tecnología JavaServer Faces no necesitarán conocer a fondo el ciclo de vida de procesamiento de una petición. Sin embargo, conociendo lo que la tecnología JavaServer Faces realiza para procesar una página, un desarrollador de aplicaciones JavaServer Faces no necesitará preocuparse de los problemas de renderizado asociados con otras tecnologías UI. Un ejemplo sería el cambio de estado de los componentes individuales. Si la selección de un componente como un checkbox afecta a la apariencia de otro componente de la página, la tecnología JavaServer Faces manejará este evento de la forma apropiada y no permitirá que se dibuje la página sin reflejar este cambio. La siguiente figura ilustra los pasos del ciclo de vida petición-respuesta JavaServer Faces.



- Reconstituir el Árbol de Componentes -

Cuando se hace una petición para una página JavaServer Faces, como cuando se pulsa sobre un enlace o un botón, la implementación JavaServer Faces comienza el estado Reconstituir el Árbol de Componentes.

Durante esta fase, la implementación JavaServer Faces construye el árbol de componentes de la página JavaServer Faces, conecta los manejadores de eventos y los validadores y graba el estado en el FacesContext.

- Aplicar Valores de la Petición -

Una vez construido el árbol de componentes, cada componente del árbol extrae su nuevo valor desde los parámetros de la petición con su método decode. Entonces el valor es almacenado localmente en el componente. Si falla la conversión del valor, se genera un mensaje de error asociado con el componente y se pone en la cola de FacesContext. Este mensaje se mostrará durante la fase Renderizar la Respuesta, junto con cualquier error de validación resultante de la fase Procesar Validaciones.

Si durante esta fase se produce algún evento, la implementación JavaServer Faces emite los eventos a los oyentes interesados. En este punto, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta sin componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

En este momento, se han puesto los nuevos valores en los componentes y los mensajes y eventos se han puesto en sus colas.

- Procesar Validaciones -

Durante esta fase, la implementación JavaServer Faces procesa todas las validaciones registradas con los componentes del árbol. Examina los atributos del componente que especifican las reglas de validación y compara esas reglas con el valor local almacenado en el componente. Si el valor local no es válido, la implementación JavaServer Faces añade un mensaje de error al FacesContext y el ciclo de vida avanza directamente hasta la fase Renderizar las Respuesta para que la página sea dibujada de nuevo incluyendo los mensajes de error. Si había errores de conversión de la fase Aplicar los Valores a la Petición, también se mostrarán.

En este momento, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

Si se han disparado eventos durante esta fase, la implementación JavaServer Faces los emite a los oyentes interesados.

- Actualizar los Valores del Modelo -

Una vez que la implementación JavaServer Faces determina que el dato es válido, puede pasar por el árbol de componentes y configurar los valores del objeto de modelo correspondiente con los valores locales de los componentes. Sólo se actualizarán los componentes que tengan expresiones value. Si el dato local no se puede

convertir a los tipos especificados por las propiedades del objeto del modelo, el ciclo de vida avanza directamente a la fase Renderizar las Respuesta, durante la que se dibujará de nuevo la página mostrando los errores, similar a lo que sucede con los errores de validación.

En este punto, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

Si se han disparado eventos durante esta fase, la implementación JavaServer Faces los emite a los oyentes interesados.

- Invocar Aplicación -

Durante esta fase, la implementación JavaServer Faces maneja cualquier evento a nivel de aplicación, como enviar un formulario o enlazar a otra página.

En este momento, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

El oyente pasa la salida al NavigationHandler por defecto. Y éste contrasta la salida con las reglas de navegación definidas en el fichero de configuración de la aplicación para determinar qué página se debe mostrar luego.

Luego la implementación JavaServer Faces configura el árbol de componentes de la respuesta a esa nueva página. Finalmente, la implementación JavaServer Faces transfiere el control a la fase Renderizar la Respuesta.

- Renderizar la Respuesta -

Durante esta fase, la implementación JavaServer Faces invoca las propiedades de codificación de los componentes y dibuja los componentes del árbol de componentes grabado en el FacesContext.

Si se encontraron errores durante las fases Aplicar los Valores a la Petición, Procesar Validaciones o Actualizar los Valores del Modelo, se dibujará la página original. Se pueden añadir nuevos componentes en el árbol si la aplicación incluye renderizadores personalizados, que definen cómo renderizar un componente. Después de que se haya renderizado el contenido del árbol, éste se graba para que las siguientes peticiones puedan acceder a él y esté disponible para la fase Reconstituir el Árbol de Componentes de las siguientes llamadas.

5.1.3.- Componentes JSF

Los componentes UI JavaServer Faces son elementos configurables y reutilizables que componen el interface de usuario de las aplicaciones JavaServer Faces. Un componente puede ser simple, como un botón, o compuesto, como una tabla, que puede estar compuesta por varios componentes.

La tecnología JavaServer Faces proporciona una arquitectura de componentes rica y flexible que incluye:

- Un conjunto de clases UIComponent para especificar el estado y comportamiento de componentes UI.
- Un modelo de renderizado que define cómo renderizar los componentes de diferentes formas.
- Un modelo de eventos y oyentes que define cómo manejar los eventos de los componentes.
- Un modelo de conversión que define cómo conectar conversores de datos a un componente.
- Un modelo de validación que define cómo registrar validadores con un componente.

La tecnología JavaServer Faces proporciona un conjunto de clases de componentes UI, que especifican toda la funcionalidad del componente, cómo mantener su estado, mantener una referencia a objetos del modelo, y dirigir el manejo de eventos y el renderizado para un conjunto de componentes estándar.

Estas clases son completamente extensibles, lo que significa que podemos extenderlas para crear nuestros propios componentes personalizados.

Todas las clases de componentes UI de JavaServer Faces descienden de la clase UIComponentBase, que define el estado y el comportamiento por defecto de un UIComponent. El conjunto de clases de componentes UI incluido en la última versión de JavaServer Faces es:

- **UICommand**: Representa un control que dispara actions cuando se activa.
- **UIData**: Soporta ligadura de datos de colecciones de datos representadas por la instancia DataModel.
- **UIColumn**: Representa el control sobre una columna de una tabla que tiene como padre un componente UIData.
- **UIMessage**: Muestra los mensajes de un componente específico, representado por clientId. El componente obtiene el mensaje de FacesContext.
- **UIMessages**: Muestra todos los mensajes almacenados en el FacesContext.
- **UIForm**: Encapsula un grupo de controles que envían datos de la aplicación. Este componente es análogo a la etiqueta form de HTML.
- **UIGraphic**: Muestra una imagen.
- **UIInput**: Toma datos de entrada del usuario. Esta clase es una subclase de UIOutput.
- **UIOutput**: Muestra la salida de datos en un página.
- **UIPanel**: Muestra una tabla.
- **UIParameter**: Representa la sustitución de parámetros.
- **UISelectItem**: Representa un sólo ítem de un conjunto de ítems.
- **UISelectItems**: Representa un conjunto completo de ítems.
- **UISelectBoolean**: Permite a un usuario seleccionar un valor booleano en un control, seleccionándolo o deseleccionándolo. Esta clase es una subclase de UIInput.

- **UISelectMany:** Permite al usuario seleccionar varios ítems de un grupo de ítems. Esta clase es una subclase de UIInput.
- **UISelectOne:** Permite al usuario seleccionar un ítem de un grupo de ítems. Esta clase es una subclase de UIInput.

La mayoría de los autores de páginas y de los desarrolladores de aplicaciones no tendrán que utilizar estas clases directamente. En su lugar, incluirán los componentes en una página usando la etiqueta correspondiente al componente. La mayoría de estos componentes se pueden renderizar de formas diferentes. Por ejemplo, un UICommand se puede renderizar como un botón o como un hiperenlace.

La arquitectura de componentes JavaServer Faces está diseñada para que la funcionalidad de los componentes se defina mediante las clases de componentes, mientras que el renderizado de los componentes se puede definir mediante un renderizador separado. Este diseño tiene varios beneficios:

- Los escritores de componentes pueden definir sólo una vez el comportamiento de un componente, pero pueden crear varios renderizadores, cada uno de los cuales define una forma diferente de dibujar el componente para el mismo cliente o para diferentes clientes.
- Los autores de páginas y los desarrolladores de aplicaciones pueden modificar la apariencia de un componente de la página seleccionando la etiqueta que representa la combinación componente/renderizador apropiada.

Un kit renderizador define como se mapean las clases de los componentes a las etiquetas de componentes apropiadas para un cliente particular. La implementación JavaServer Faces incluye un RenderKit estándar para renderizar a un cliente HTML.

Por cada componente UI que soporte un RenderKit, éste define un conjunto de objetos Renderer. Cada objeto Renderer define una forma diferente de dibujar el componente particular en la salida definida por el RenderKit. Por ejemplo, un componente UISelectOne tiene tres renderizadores diferentes. Uno de ellos dibuja el componente como un conjunto de botones de radio. Otro dibuja el componente como un ComboBox. Y el tercero dibuja el componente como un ListBox.

Cada etiqueta JSP personalizada en el RenderKit de HTML, está compuesta por la funcionalidad del componente, definida en la clase UIComponent, y los atributos de renderizado, definidos por el Renderer. Por ejemplo, un componente UICommand, puede ser renderizado de dos formas diferentes: como un commandButton o como un commandLink. La parte command de las etiquetas corresponde con la clase UICommand, y especifica la funcionalidad, que es disparar un action. Las partes del botón y el hiperenlace de las etiquetas corresponden a un renderizador independiente, que define cómo dibujar el componente.

La implementación de referencia de JavaServer Faces proporciona una librería de etiquetas personalizadas para renderizar componentes en HTML. Es interesante conocerla para entender el código fuente de las vistas generadas por JSF. Los componentes soportados por dicha implementación son los que aparecen detallados en la siguiente tabla:

| Etiqueta | Funciones | Se renderiza como... | Apariencia |
|-----------------------|--|--|--|
| commandButton | Enviar un formulario a la aplicación | Un elemento <code><input type="tipo"></code> HTML, donde el valor del tipo puede ser <code>submit</code> , <code>reset</code> , o <code>image</code> . | Un botón |
| commandLink | Enlaza a otra página o localización en otra página. | Un elemento <code><a href></code> HTML. | Un Hiperenlace |
| form | Representa un formulario de entrada. Las etiquetas internas del formulario reciben los datos que serán enviados con el formulario. | Un elemento <code><form></code> HTML. | Sin apariencia. |
| graphicImage | Muestra una imagen. | Un elemento <code></code> HTML. | Una imagen. |
| message | Muestra un mensajes de usuario. | Texto Normal. | Muestra un mensaje almacenado en el <code>FacesContext</code> y representado por un ID. |
| messages | Muestra todos los mensajes de usuario. | Texto Normal. | Muestra todos los mensajes de usuario que hay en <code>FacesContext</code> . |
| dataTable | Representa una tabla de datos. | Un elemento <code><table></code> HTML. | Representa una tabla de datos. |
| inputHidden | Permite introducir una variable oculta en una página. | Un elemento <code><input type="hidden"></code> HTML. | Sin apariencia. |
| inputSecret | Permite al usuario introducir un string sin que aparezca el string real en el campo. | Un elemento <code><input type="password"></code> HTML. | Un campo de texto, que muestra una fila de caracteres en vez del texto real introducido. |
| inputText | Permite al usuario introducir un string . | Un elemento <code><input type="text"></code> HTML. | Un campo de texto. |
| inputTextarea | Permite al usuario introducir un texto multi-líneas. | Un elemento <code><textarea></code> HTML. | Un campo de texto multi-línea. |
| outputFormat | Muestra líneas de texto formateadas. | Texto normal formateado. | Texto normal. |
| outputLabel | Muestra un componente anidado como una etiqueta para un campo de texto especificado. | Un elemento <code><label></code> HTML. | Texto normal. |
| outputLink | Muestra un elemento de enlace con una referencia. | Un elemento <code></code> . | Representa un enlace. |
| outputText | Muestra una línea de texto. | Texto normal. | Texto normal. |
| panelGrid | Muestra una tabla. | Un elemento <code><table></code> HTML. con elementos <code><tr></code> y <code>%lt;td></code> . | Una tabla. |
| panelGroup | Agrupa un conjunto de paneles bajo un padre. | | Una fila en una tabla. |
| selectBooleanCheckbox | Permite al usuario cambiar el valor de una elección booleana. | Un elemento <code><input type=checkbox></code> HTML. | Un checkbox . |
| selectItem | Representa un ítem de una lista de ítems en un componente <code>UISelectOne</code> . | Un elemento <code><option></code> HTML. | Sin apariencia. |
| selectItems | Representa una lista de ítems en un componente <code>UISelectOne</code> . | Un elemento <code><option></code> HTML. | Sin apariencia. |
| selectManyCheckbox | Muestra un conjunto de <code>checkbox</code> , en los que el usuario puede seleccionar varios. | Un conjunto de elementos <code><input></code> HTML. | Un conjunto de CheckBox . |
| selectManyListbox | Permite a un usuario seleccionar varios ítems de un conjunto de ítems, todos mostrados a la vez. | Un conjunto de elementos <code><select></code> HTML. | Un ListBox . |
| selectManyMenu | Permite al usuario seleccionar varios ítems de un grupo de ítems. | Un conjunto de elementos <code><select></code> HTML. | Un comboBox . |
| selectOneListbox | Permite al usuario seleccionar un ítem de un grupo de ítems. | Un conjunto de elementos <code><select></code> HTML. | Un ListBox . |
| selectOneMenu | Permite al usuario seleccionar un ítem de un grupo de ítems. | Un conjunto de elementos <code><select></code> HTML. | Un comboBox . |
| selectOneRadio | Permite al usuario seleccionar un ítem de un grupo de ítems. | Un conjunto de elementos <code><input type="radio"></code> HTML. | Un conjunto de botones de radio. |

- **Modelo de Conversión -**

Una aplicación JavaServer Faces opcionalmente puede asociar un componente con datos del objeto del modelo del lado del servidor. Este objeto del modelo es un componente JavaBeans que encapsula los datos de un conjunto de componentes. Una aplicación obtiene y configura los datos del objeto modelo para un componente llamando a las propiedades apropiadas del objeto modelo para ese componente.

Cuando un componente se une a un objeto modelo, la aplicación tiene dos vistas de los datos del componente: la vista modelo y la vista presentación, que representa los datos de un forma que el usuario pueda verlos y modificarlos.

Una aplicación JavaServer Faces debe asegurarse que los datos del componente puedan ser convertidos entre la vista del modelo y la vista de presentación. Esta conversión normalmente la realiza automáticamente el renderizador del componente.

En algunas situaciones, podríamos querer convertir un dato de un componente a un tipo no soportado por el renderizador del componente. Para facilitar esto, la tecnología JavaServer Faces incluye un conjunto de implementaciones estándar de Converter que nos permite crear nuestros conversores personalizados. La implementación de Converter convierte los datos del componente entre las dos vistas.

- Modelo de Eventos -

Un objetivo de la especificación JavaServer Faces es mejorar los modelos y paradigmas existentes para que los desarrolladores se puedan familiarizar rápidamente con el uso de JavaServer Faces en sus aplicaciones. En este espíritu, el modelo de eventos y oyentes de JavaServer Faces mejora el diseño del modelo de eventos de JavaBeans, que es familiar para los desarrolladores de GUI y de aplicaciones Web.

Al igual que la arquitectura de componentes JavaBeans, la tecnología JavaServer Faces define las clases Listener y Event que una aplicación puede utilizar para manejar eventos generados por componentes UI. Un objeto Event identifica al componente que lo generó y almacena información sobre el propio evento. Para ser notificado de un evento, una aplicación debe proporcionar una implementación de la clase Listener y registrarla con el componente que genera el evento. Cuando el usuario activa un componente, como cuando pulsa un botón, se dispara un evento. Esto hace que la implementación de JavaServer Faces invoque al método oyente que procesa el evento. JavaServer Faces soporta dos tipos de eventos: eventos valueChangeListener y eventos action.

Un evento valueChangeListener ocurre cuando el usuario cambia el valor de un componente. Un ejemplo es seleccionar un checkbox, que resulta en que el valor del componente ha cambiado a true. Los tipos de componentes que generan estos eventos son los componentes UIInput, UISelectOne, UISelectMany, y UISelectBoolean.

Este tipo de eventos sólo se dispara si no se detecta un error de validación. Un evento action ocurre cuando el usuario pulsa un botón o un hiperenlace. El componente UICommand genera este evento.

- Modelo de Validación -

La tecnología JavaServer Faces soporta un mecanismo para validar el dato local de un componente durante la fase del Proceso de Validación, antes de actualizar los datos del objeto modelo.

Al igual que el modelo de conversión, el modelo de validación define un conjunto de clases estándar para realizar chequeos de validación comunes. La librería de

etiquetas jsf-core también define un conjunto de etiquetas que corresponden con las implementaciones estándar de Validator.

La mayoría de las etiquetas tienen un conjunto de atributos para configurar las propiedades del validador, como los valores máximo y mínimo permitidos para el dato del componente. El autor de la página registra el validador con un componente anidando la etiqueta del validador dentro de la etiqueta del componente.

Al igual que el modelo de conversión, el modelo de validación nos permite crear nuestras propias implementaciones de Validator y la etiqueta correspondiente para realizar validaciones personalizadas.

5.1.4.- El fichero faces-config.xml

Es el fichero principal de configuración de JSF. Describe las reglas de navegación y define los beans manejados que se emplean en la aplicación a través de la sintaxis EL.

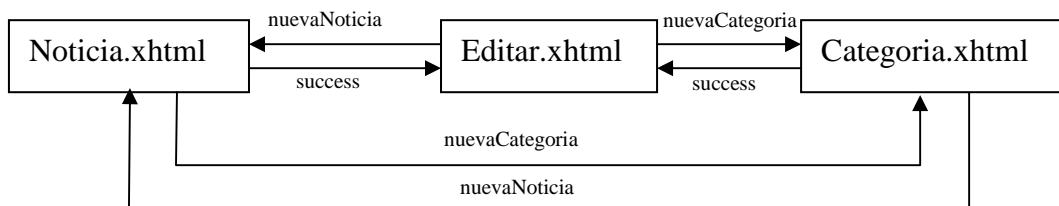
```
<managed-bean>
    <description>Noticiero</description>
    <managed-bean-name>GestorNoticias</managed-bean-name>
    <managed-bean-class>web.GestorNoticias</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<navigation-rule>
    <from-view-id>/editar/editar.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>nuevaCategoria</from-outcome>
        <to-view-id>/editar/new/categoría.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>nuevaNoticia</from-outcome>
        <to-view-id>/editar/new/noticia.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>

<navigation-rule>
    <from-view-id>/editar/new/*</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/editar/editar.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>nuevaNoticia</from-outcome>
        <to-view-id>/editar/new/noticia.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>nuevaCategoria</from-outcome>
        <to-view-id>/editar/new/categoría.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

El bean manejado permite acceder a los datos del modelo para proporcionar información y obtenerla desde los componentes mediante expresiones EL. En este caso un ejemplo de expresión podría ser, `#{GestorNoticias.listaCategorias}`, que accedería a la propiedad `listadoCategorias` del bean manejado descrito.

Las reglas de navegación describen las rutas por las que se navega en cada caso en función de la respuesta obtenida. En este caso podemos describir gráficamente las reglas de navegación de la siguiente forma:



La cadena de texto empleada en la regla de navegación puede incluirse directamente en la etiqueta que enlaza con la petición faces, u obtenerse como resultado de la ejecución de algún método.

5.1.5.- Facelets

Facelets es un framework de diseño de páginas de aplicaciones web, muy fácil de usar y configurar. Si queremos desarrollar una aplicación con JSF, entonces Facelets es el suplemento ideal para la configuración del “diseño” (aunque Facelets es independiente del lenguaje y puede ser usado en cualquier tipo de aplicación). Facelets se ajusta a JSF como un guante. Facelets nos permite definir plantillas de páginas o crear páginas completas, aunque se suele utilizar con mayor frecuencia para crear plantillas.

Otra de las causas o tal vez la principal de que usemos Facelets es debido a la limitación que tiene el contenedor OC4J, que solo soporta JSP API 2.0, con lo que solo podríamos utilizar JSF 1.1. Utilizando Facelets podemos saltar esta restricción y utilizar JSF 1.2 (JSP API 2.1) en nuestras aplicaciones.

- Propiedades de Facelets -

- Trabajo basado en plantillas.
- Fácil composición de componentes.
- Creación de etiquetas lógicas a la medida.
- Funciones para expresiones.
- Desarrollo amigable para el diseñador gráfico.
- Creación de librerías de componentes.

5.1.6.- RichFaces

RichFaces es un Framework de código abierto que añade las capacidades de Ajax a las aplicaciones JSF sin recurrir a JavaScript. RichFaces cubre todo el framework JSF incluidos el ciclo de vida, validación, conversión y, gestión de recursos estáticos y dinámicos. Los componentes RichFaces incorporan soporte para Ajax y permiten una sencilla configuración del look-and-feel que puede ser incorporado dentro de la aplicación JSF.

- Características RichFaces -

- Ampliar el conjunto de beneficios que aporta JSF mientras trabajamos con Ajax.
- Añadir las capacidades de Ajax a las aplicaciones JSF.
- Crear complejas vistas rápidamente utilizando los componentes.

- Escribir componentes propios con soporte de Ajax.
- Empaquetar recursos con las clases Java de la aplicación.
- Crear modernos look-and-feels de interfaces de usuario con tecnología basada en capas (skins).
- Crear y testear componentes, actions, listeners y páginas al mismo tiempo.
- Nos permite trabajar con Facelets y JSF 1.2.

- AJAX -

AJAX, acrónimo de Asynchronous JavaScript And XML (JavaScript asíncrono y XML), es una técnica de desarrollo web para crear aplicaciones interactivas o RIA (Rich Internet Applications). Estas aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene la comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, lo que significa aumentar la interactividad, velocidad y usabilidad en las aplicaciones.

Ajax es una tecnología asíncrona, en el sentido de que los datos adicionales se requieren al servidor y se cargan en segundo plano sin interferir con la visualización ni el comportamiento de la página. JavaScript es el lenguaje interpretado (scripting language) en el que normalmente se efectúan las funciones de llamada de Ajax mientras que el acceso a los datos se realiza mediante XMLHttpRequest, objeto disponible en los navegadores actuales. En cualquier caso, no es necesario que el contenido asíncrono esté formateado en XML.

Ajax es una técnica válida para múltiples plataformas y utilizable en muchos sistemas operativos y navegadores, ya que está basado en estándares abiertos como JavaScript y Document Object Model (DOM). AJAX es una combinación de cuatro tecnologías ya existentes:

- XHTML (o HTML) y hojas de estilos en cascada (CSS) para el diseño que acompaña a la información.
- Document Object Model (DOM) accedido con un lenguaje de scripting por parte del usuario, especialmente implementaciones ECMAScript como JavaScript y JScript, para mostrar e interactuar dinámicamente con la información presentada.
- El objeto XMLHttpRequest para intercambiar datos de forma asíncrona con el servidor web. En algunos frameworks y en algunas situaciones concretas, se usa un objeto iframe en lugar del XMLHttpRequest para realizar dichos intercambios.
- XML es el formato usado generalmente para la transferencia de datos solicitados al servidor, aunque cualquier formato puede funcionar, incluyendo HTML pre-formateado, texto plano, JSON y hasta EBML.

El principal peligro de las tecnologías de script es la diversidad de navegadores que existen en el mercado. A la hora de desarrollar hay que tratar de abarcar todos los navegadores posibles del mercado, y si escribimos código JavaScript complejo esto se complica. Para minimizar los riesgos cuando no se pueda evitar el uso de JavaScript, es fundamental el uso de librerías de manejo de DOM como JQUERY.

5.2.- Tecnologías Control: EJB

En este capítulo veremos la tecnología EJB, qué ventajas nos proporciona y cómo utilizarla dentro de las aplicaciones web que desarrollemos.

5.2.1.- Introducción

Enterprise Java Beans (EJB) es una plataforma para construir aplicaciones de negocio portables, reutilizables y escalables usando el lenguaje de programación Java. Desde el punto de vista del desarrollador, un EJB es una porción de código que se ejecuta en un contenedor EJB, que no es más que un ambiente especializado (runtime) que provee determinados componentes de servicio.

Los EJBs pueden ser vistos como componentes, desde el punto de vista que encapsulan comportamiento y permite reutilizar porciones de código, pero también pueden ser vistos como un framework, ya que, desplegados en un contenedor, proveen servicios para el desarrollo de aplicaciones empresariales, servicios que son invisibles para el programador y no ensucian la lógica de negocio con funcionalidades trasversales al modelo de dominio (a menudo requisitos no funcionales o aspectos). En la especificación 3.0, los EJB no son más que POJOs (clases planas comunes y corrientes de Java) con algunos poderes especiales implícitos, que se activan en tiempo de ejecución (runtime) cuando son ejecutados en un contenedor de EJBs.

- Desarrollo Basado en Componentes -

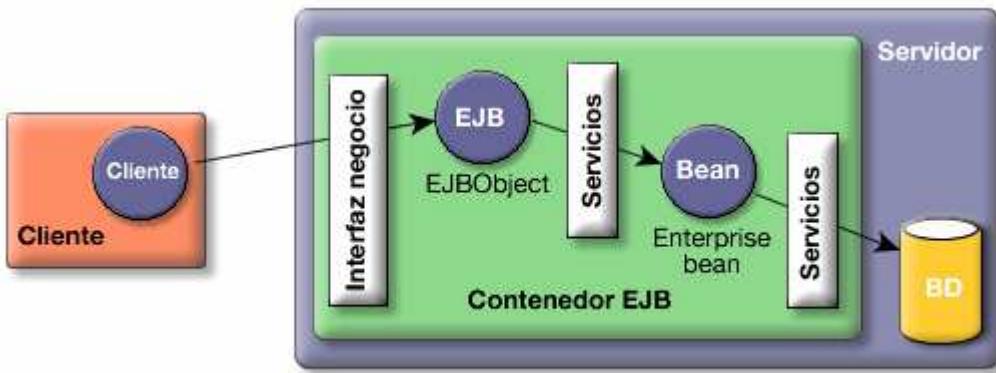
Con la tecnología Enterprise JavaBeans es posible desarrollar componentes (enterprise beans) que luego puedes reutilizar y ensamblar en distintas aplicaciones que tengas que hacer para la empresa. Por ejemplo, podrías desarrollar un bean Cliente que represente un cliente en una base de datos. Podrías usar después ese bean Cliente en un programa de contabilidad o en una aplicación de comercio electrónico o virtualmente en cualquier programa en el que se necesite representar un cliente. De hecho, incluso sería posible que el desarrollador del bean y el ensamblador de la aplicación no fueran la misma persona, o ni siquiera trabajaran en la misma empresa.

El desarrollo basado en componentes promete un paso más en el camino de la programación orientada a objetos. Con la programación orientada a objetos puedes reutilizar clases, pero con componentes es posible reutilizar un mayor nivel de funcionalidades e incluso es posible modificar estas funcionalidades y adaptarlas a cada entorno de trabajo particular sin tocar el código del componente desarrollado. Aunque veremos el tema con mucho más detalle, en este momento puedes ver un componente como un objeto tradicional con un conjunto de servicios adicionales soportados en tiempo de ejecución por el contenedor de componentes. El contenedor de componentes se denomina contenedor EJB y es algo así como el sistema operativo en el que éstos residen.

Cuando estés trabajando con componentes tendrás que dedicarle tanta atención al despliegue (deployment) del componente como a su desarrollo. Entendemos por despliegue la incorporación del componente a nuestro contenedor EJB y a nuestro entorno de trabajo (bases de datos, arquitectura de la aplicación, etc.). El despliegue se define de forma declarativa, mediante un fichero XML (descriptor del despliegue,

deployment descriptor) en el que se definen todas las características del bean o con anotaciones.

El funcionamiento de los componentes EJB se basa fundamentalmente en el trabajo del contenedor EJB. El contenedor EJB es un programa Java que corre en el servidor y que contiene todas las clases y objetos necesarios para el correcto funcionamiento de los EJBs.



En esta figura puedes ver una representación de muy alto nivel del funcionamiento básico de los EJBs. En primer lugar, puedes ver que el cliente que realiza peticiones al bean y el servidor que contiene el bean, está ejecutándose en máquinas virtuales Java distintas. Incluso pueden estar en distintos hosts. Otra cosa a resaltar es que el cliente nunca se comunica directamente con el EJB, sino que el contenedor EJB proporciona un EJBObject que hace de interfaz. Cualquier petición del cliente (una llamada a un método de negocio del EJB) se debe hacer a través del objeto EJB, el cual solicita al contenedor EJB una serie de servicios y se comunica con el EJB. Por último, el bean realiza las peticiones correspondientes a la base de datos.

Los servicios que ofrece un contenedor de EJBs son los siguientes:

- **Integración:** Proveen una forma de acoplar en tiempo de ejecución diferentes componentes, mediante simple configuración de anotaciones o XMLs. El acoplamiento se puede hacer mediante Inyección de Dependencia (DI) o usando JNDI, como se hacía en EJB 2. La integración es un servicio que proveen los beans de sesión y los MDBs.
- **Pooling:** El contenedor de EJBs crea para componentes EJB un pool de instancias que es compartido por los diferentes clientes. Aunque cada cliente ve como si recibiera siempre instancias diferentes de los EJB, el contenedor está constantemente reutilizando objetos para optimizar memoria. El pooling es un servicio que se aplica a los Stateless Session Beans y a los MDBs.
- **Thread-safely:** El programador puede escribir componentes del lado del servidor como si estuviera trabajando en una aplicación sencilla con un solo thread (hilo). El contenedor se encarga de que los EJBs tengan el soporte adecuado para una aplicación multi-usuario (como son en general las aplicaciones empresariales) de forma transparente, asegurando el acceso seguro, consistente y alto rendimiento. Aplica a los beans de sesión y a los MDBs.

- **Administración de Estados:** El contenedor de EJBs almacena y maneja el estado de un Stateful Session Bean de forma transparente, lo que significa que el programador puede mantener el estado de los miembros de una clase como si estuviera desarrollando una aplicación de escritorio ordinaria. El contenedor maneja los detalles de las sesiones.
- **Mensajería:** Mediante los MDBs es posible desacoplar por completo dos componentes para que se comuniquen de forma asincrónica, sin reparar demasiado en los mecanismos de la JMS API que los MDBs encapsulan.
- **Transacciones:** EJB soporta el manejo de transacciones declarativas que permiten agregar comportamiento transaccional a un componente simplemente usando anotaciones o XMLs de configuración. Esto significa que cuando un método de un EJB (Session Bean o MDB) se completa normalmente, el contenedor se encargará de commitear la transacción y validar los cambios que se realizaron en los datos de forma permanente. Si algo fallara durante la ejecución del método (una excepción o cualquier otro problema), la transacción haría un rollback y es como si el método jamás se hubiera invocado.
- **Seguridad:** EJB soporta integración con la Java Authentication and Authorization Service (JAAS) API, haciendo casi transparente el manejo transversal de la seguridad. Aplica a todos los Session Beans.
- **Interceptors:** EJB introduce un framework liviano y simple para AOP (programación orientada a aspectos). No es tan robusto y completo como otros, pero es lo suficientemente útil para que sea utilizado por los demás servicios del contenedor para brindarnos de forma invisible los crosscutting concerns de seguridad, transacciones, thread-safely. Además, nosotros, como programadores, podemos agregar nuevos aspectos como logging o auditoria y demás de forma configurable.
- **Acceso Remoto:** Es posible acceder de forma remota a distintos EJBs de forma sencilla, simplemente mediante la Inyección de Dependencia. El procedimiento para inyectar un componente local o uno remoto es exactamente el mismo, abstrandónos de las complicaciones específicas de RMI o similares. Este servicio aplica únicamente a los Session Beans.
- **Web Services:** Un Stateless Session Bean puede publicar sus métodos como servicios web mediante una sencilla anotación.
- **Persistencia:** EJB 3 provee la especificación JPA para el mapeo de objetos llamados Entidades a tablas.
- **Catching and Performance:** JPA provee de forma transparente, un importante número de servicios que permiten usar un caché de entidades en memoria y una lectura y escritura sobre la base de datos de alto rendimiento.

Los servicios que debe proveer el contenedor de EJBs para cada uno de nuestros Beans, deben ser especificados por el programador a través de metadata de configuración que puede escribirse como anotaciones en la propia clase o a través de un archivo XML (ejb-jar.xml).

A partir de EJB 3 se puede usar cualquiera de estas dos técnicas. Las técnicas no son exclusivas, pueden coexistir anotaciones con descriptores XML teniendo prioridad los datos del archivo descriptor sobre las anotaciones.

5.2.2.- Tipos de EJB

Los EJB3, están formados por al menos dos componentes:

- **Una interfaz (POJI):** Define los métodos públicos del EJB que estarán disponibles para ser utilizados por un cliente del EJB, la interfaz puede ser remota (Remote) o local (Local).
- **Un POJO:** Tiene la implementación de la interfaz, que es el bean en realidad.

La interfaz es necesaria ya que los elementos que usan el bean no pueden acceder a éste directamente (especialmente si se usa RMI ó Servicios Web) por lo que se programa contra la interfaz. De paso esto facilita la inyección de dependencias. La interfaz recibe el nombre de “business interface” y un bean puede tener más de una. Esto implica que diferentes accesos desde diferentes clientes, pueden proporcionar diferentes funcionalidades al usar diferentes interfaces.

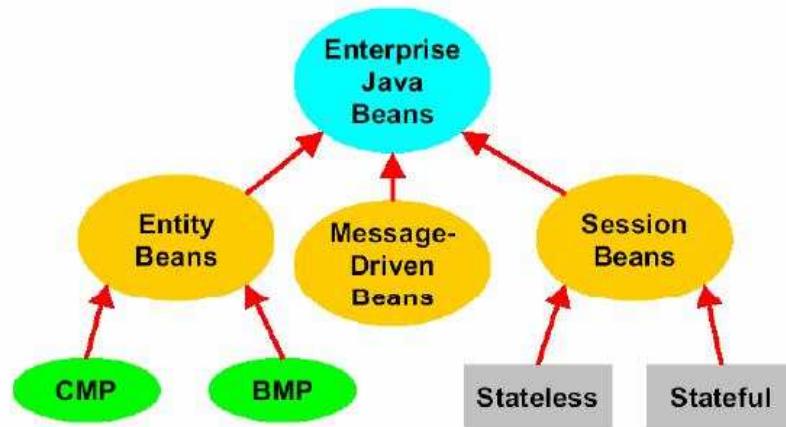
- **Interfaz Remota, Local o las Dos -**

La principal diferencia entre una interfaz local y una remota, es la manera en que el servidor de aplicaciones sirve el EJB. Con las interfaces remotas, los objetos son enviados como valores y con las locales, se envían los objetos como referencias. Existen tres tipos de EJBs:

- **Beans de Sesión (Session Beans):** En una aplicación típica, dividida en cuatro grandes capas (presentación, lógica de negocio, persistencia y base de datos), los Beans de Sesión viven en la lógica de negocio. Hay dos grandes tipos de Beans de Sesión: *Stateless* (sin estado) y *Stateful* (con estado). El primero no conserva el estado de ninguno de sus atributos de la invocación de un método a otro y el segundo conserva el estado a lo largo de toda una sesión. Los Beans de Sesión Stateless son los únicos que pueden exponerse como servicios web.
- **Message-Driven Beans (MDBs):** También viven en la lógica de negocio y los servicios que proveen son parecidos a los Beans de Sesión, con la diferencia de que los MDBs son usados para invocar métodos de forma asíncrona. Cuando se produce la invocación de un método de un MDB desde un cliente, la llamada no bloquea el código del cliente y el mismo puede seguir con su ejecución, sin tener que esperar indefinidamente por la respuesta del servidor. Los MDBs encapsulan el popular servicio de mensajería de Java, JMS.
- **Entidades (Entities):** Las entidades viven en la capa de persistencia y son los EJBs que manejan la Java Persistence API (JPA), también parte de la especificación de EJB 3.0. Las entidades son POJOs con cierta información metadata que permite a la capa de persistencia mapear los atributos de la clase a las tablas de la base de datos y sus relaciones. Existen dos tipos BMP y CMP según se gestione la persistencia por parte del bean o del contenedor.

Los beans de sesión son invocados por el cliente con el propósito de ejecutar operaciones de negocio específicas, como por ejemplo podría ser chequear la historia crediticia del cliente de un banco. El nombre sesión implica que la instancia del bean estará disponible durante una unidad de trabajo (unit of work) y no sobrevivirá a una caída del servidor. Un bean de sesión sirve para modelar cualquier funcionalidad lógica de una aplicación.

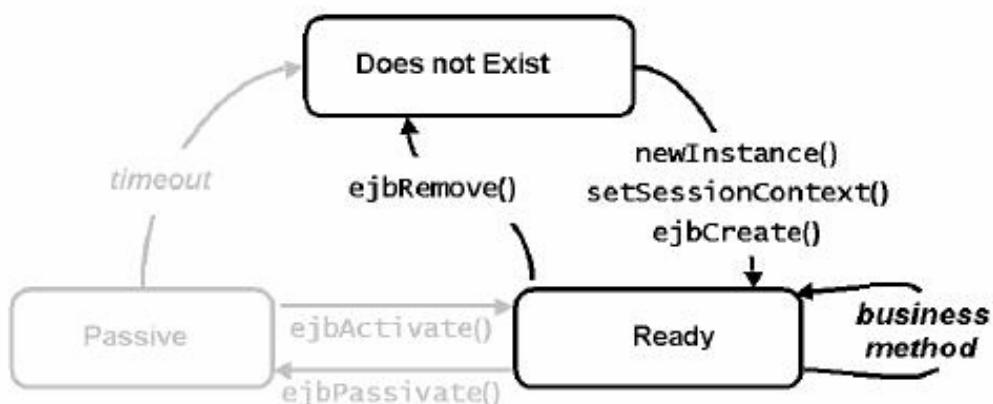
Los MDBs también procesan lógica de negocio, pero un cliente nunca invoca a un método de un MDB directamente. El sistema de mensajería asincrónica propone la utilización de una capa intermedia en la comunicación entre el productor y el consumidor del mensaje. En EJB 3, esta capa se llama MOM (Message-oriented Middleware). Básicamente la MOM es un software que permite funcionar como servidor de mensajería, reteniendo los mensajes del productor y enviándolos posteriormente al consumidor en el momento en que esté disponible para recibirlo.



5.2.3.- Ciclo de Vida

Los EJB facilitan la vida ya que dejan en manos del contenedor las operaciones de creación, el proceso de DI y otros detalles necesarios para su correcto funcionamiento. Para realizar estas operaciones el contenedor usa el concepto de “ciclo de vida” del bean, una serie de estados que indican al contenedor cómo ha de tratar el objeto. Los ciclos son:

- **Creación:** El contenedor inicializa el bean y asigna todas las DI existentes.
- **Destrucción:** El contenedor determina que no se necesita el bean y libera los recursos.
- **Pasivación:** Exclusivo de los beans con estado, serializa el contenido del bean (o lo restaura). Se llama cuando un bean no está en uso y se necesitan más recursos en el sistema, por ejemplo por haber muchas sesiones abiertas.



La parte de la figura que aparece en gris es exclusiva de los Beans con estado.

- **Callbacks** -

El ciclo de vida es útil y simple, pero llega un punto en que puede ser necesario indicar al contenedor cómo tratar algunos datos en esos ciclos, por ejemplo para cerrar una conexión a base de datos u realizar operaciones previas a la utilización del bean. Para esto utilizamos unos métodos especiales llamados *callbacks*.

Un callback es un método que se llama dado un evento. En el caso de EJB son métodos internos (no públicos) que se llaman al pasar por las fases del ciclo de vida. Estos métodos pueden ser public, private, protected o package-protected, devuelven void y se marcan con anotaciones específicas. Sólo pueden lanzar excepciones que hereden de `java.lang.RuntimeException`. Las callbacks son:

- **PostConstruct:** Llamado después de finalizar la creación del bean, se suele usar para inicializar objetos como conexión a base de datos.
- **PreDestroy:** Se llama antes de destruir el bean, se suele usar para liberar recursos asignados en PostConstruct.
- **PrePassivate:** Se llama antes de “pasivar” el objeto, se usa para limpiar recursos no serializables como conexiones a bases de datos.
- **PostActivate:** Se llama después de restaurar el objeto, para restaurar recursos no serializables.

5.2.4.- Interceptores

Los interceptores, interceptan la llamada a un objeto o método normalmente dentro de un entorno Java EE. Se trata de clases con métodos que pueden ser o no invocadas. De ser invocadas esto se realiza antes, después o antes y después de la llamada a un objeto EJB o un método dentro de un EJB.

Gracias a esto podemos ejecutar código que puede preparar un entorno de ejecución de un método, discernir si se cumplen las condiciones para su ejecución, o es preferible que se lance una excepción en lugar de ejecutar el método, o como último ejemplo, resetear variables o características del entorno para la próxima vez que se ejecute el método interceptado u otro que requiera de ellas. Sus ventajas principales son:

- Permiten al desarrollador un mayor control sobre el flujo de sus programas.
- Se llaman antes y después de la invocación a un método.
- Se aplican a todos los métodos de negocio de un EJB.
- Permiten modificar valores de entrada a métodos.
- Permiten modificar valores de retorno.
- Gestionar las excepciones.
- Interrumpir llamadas completamente.
- Realizar análisis y optimización de métodos.
- Influencias de AOP (es una de las formas de implementar un desarrollo AOP).

Podemos poner un ejemplo de cómo se llama desde el código a un interceptor de método mediante anotaciones. De todas formas no es necesario sólo esto, también hay que configurar el servidor para que acepte las anotaciones correctas y crear las clases interceptoras.

```
@Stateless
public class PlaceBidBean implements PlaceBid {

    @Interceptors(ActionBazaarLogger.class)
    public void addBid(Bid bid) {
        ...
    }

    public class ActionBazaarLogger {
        @AroundInvoke
        public Object logMethodEntry(InvocationContext invocationContext)
            throws Exception {
            System.out.println("Entering: " + invocationContext.getMethod().getName());
            return invocationContext.proceed();
        }
    }
}
```

Como vemos, declaramos mediante una anotación la clase encargada de interceptar la llamada. Esa anotación **@Interceptors** se puede usar a nivel de método o de clase según convenga, y puede declarar a varios interceptores como si fueran parámetros para que sean ejecutados en orden. A continuación en esa clase declaramos un método que se ejecutará al llamar a addBid.

Algunos detalles a comentar sobre el método de ActionBazaarLogger: la anotación **@AroundInvoke** sólo se puede usar en un método de la clase por lo que si queremos otro interceptor habrá que declarar otra clase. Por otra parte, el método siempre debe recibir un único parámetro, del tipo *InvocationContext*.

Sobre el código del interceptor, es importante llamar al método proceed o nuestra lógica de negocio no se ejecutará. De hecho lo normal es llamar a proceed o lanzar una excepción, que abortará la ejecución del método del bean. Otros métodos de *InvocationContext* nos permiten recuperar el nombre del método al que llamamos, de los valores de los parámetros que pasamos al bean o incluso modificarlos.

- Tipos de Interceptores -

Todos los interceptores pueden interceptar el ciclo de vida de un bean. También se puede definir un método interceptor en el propio bean (también llamado interceptor interno), mediante la anotación *aroundInvoke*. Los interceptores (externos) se pueden dividir en tres tipos:

- Default (interceptor por defecto)
- Class-level (interceptor de clase)
- Method-level (interceptor de método)

Existe la posibilidad de declarar interceptores globales (default interceptor) que se ejecutan para todos los métodos de todos los beans. No se pueden especificar con anotaciones, se necesita editar el descriptor de la aplicación, pero resultan muy útiles para tareas como logging.

Como antes hemos dicho, solo se pueden definir en el fichero *ejb-jar.xml*. Primero tenemos que indicar cual es el método *AroundInvoke* del interceptor con un elemento **interceptor**. Después tenemos que registrar el interceptor en el elemento **assembly-descriptor**, con un elemento **interceptor-binding**. Se utiliza * para indicar que el ejb-name que declara a DefaultInterceptor, es un interceptor por defecto.

```
<ejb-jar ...>
...
<interceptors>
    <interceptor>
        <interceptor-class>
            org.jboss.seam.ejb.DefaultInterceptor
        </interceptor-class>
        <around-invoke>
            <method-name>intercept</method-name>
        </around-invoke>
    </interceptor>
</interceptors>
...
<assembly-descriptor>
    <!--Interceptor Defecto, aplicado a todo metodo de todo bean desplegado -->
    <interceptor-binding>
        <ejb-name>*</ejb-name>
        <interceptor-class>
            org.jboss.tutorial.interceptor.bean.DefaultInterceptor
        </interceptor-class>
    </interceptor-binding>
    ...
</assembly-descriptor>
...
</ejb-jar>
```

Y la clase interceptora:

```
package org.jboss.tutorial.interceptor.bean;

import javax.ejb.AroundInvoke;
import javax.ejb.InvocationContext;

public class DefaultInterceptor {
    @AroundInvoke
    public Object intercept(InvocationContext ctx) throws Exception {
        System.out.println("**** Default intercepting " + ctx.getMethod().getName());
        try {
            return ctx.proceed();
        }finally {
            System.out.println("**** DefaultInterceptor exiting");
        }
    }
}
```

Los interceptores a nivel de clase, interceptan todos los métodos de negocio del EJB donde se registran. La anotación *Interceptors*, también puede definir un array de interceptores de clase. Puede configurarse con anotaciones o en el fichero *ejb-jar.xml*. Vamos a ver un ejemplo de interceptor de clase:

```
@Stateless
@Interceptors ({TracingInterceptor.class, SomeInterceptor.class})
public class EmailSystemBean {
    ...
}
```

Los interceptores a nivel de método, interceptan el método de negocio anotado con *Interceptors* de un bean. La anotación *Interceptors*, también puede definir un array

de interceptores de método. Puede configurarse con anotaciones o en el fichero *ejb-jar.xml*. Vamos a ver un ejemplo de interceptor de método:

```
package org.jboss.tutorial.interceptor.bean;
import javax.interceptor.Interceptors;
import org.jboss.tutorial.interceptor.bean.AccountsConfirmInterceptor;

@Stateless
public class EmailSystemBean {
    ...
    @Interceptors({AccountsConfirmInterceptor.class})
    public void sendBookingConfirmationMessage(long orderId) {
        ...
    }
    ...
}
```

Otra faceta de los interceptores es su capacidad de interceptar las llamadas de cambio de estado (*PostConstruct*, *PreDestroy*, *PrePassivate*, *PostActivate*). Para ello declaramos un interceptor donde los métodos, en lugar de usar la anotación *@AroundInvoke* usan *@PreDestroy*, *@PostConstruct*, *@PrePassivate* o *@PostActivate* (las mismas anotaciones que definen los métodos de estado en el objeto).

Se puede indicar por parte de un bean, que se excluyan los interceptores por defecto para toda la clase. También se puede indicar que para un método de un bean, se excluya de la intercepción de los interceptores por defecto y de clase. Para obtener este comportamiento podemos utilizar tanto anotaciones como el fichero descriptor de EJBs.

```
@ExcludeDefaultInterceptors
public class EmailMDB implements MessageListener {
    ...
}

public class EmailMDB implements MessageListener {
    ...
    @ExcludeClassInterceptors
    @ExcludeDefaultInterceptors
    public void noop() {
        System.out.println("<In EmailSystemBean.noop business method");
        System.out.println("Exiting EmailSystemBean.noop business method>");
    }
    ...
}
```

Los interceptores son invocados en el orden en que fueron declarados en la anotación. Los interceptores de clase se invocan antes que los interceptores de método. Los métodos interceptores definidos en los propios beans, se invocan al final. Si una clase interceptor tiene superclases, los interceptores de la superclase se ejecutan antes que los de la subclase. Si un bean con un método interceptor, tiene una superclase con otro método interceptor, se invoca en primer lugar el método interceptor de la superclase. Si se sobrescriba el método interceptor en la subclase, entonces el método interceptor de la superclase no es invocado.

En resumen el orden de invocación de los interceptores es:

- Interceptores Externos
 - Interceptores por defecto, si hay declarados

- Interceptores de clase, si hay declarados
- Interceptores de método, si hay declarados
- Métodos interceptores del propio bean

Dentro de cada grupo, el orden de invocación de los interceptores queda determinado por el orden de izquierda a derecha, de las anotaciones y después los del fichero descriptor de EJBs.

5.2.5.- Anotaciones

Como ya hemos mencionado, debemos indicar al contenedor de EJB qué servicios debe proveer a nuestro Bean, y para ello disponemos de los descriptores o de las anotaciones. Vamos a ver las anotaciones básicas para los Session Beans. El tema de las anotaciones de los Beans de Entidad lo dejaremos para el apartado de JPA donde se describirán con más detalle.

- **@Stateful:** Indica que el Bean de Sesión es con estado. Sus atributos son:
 - **name** - por defecto el nombre de la clase pero se puede especificar otra diferente.
 - **mappedName** - si se quiere que el contenedor maneje el objeto de manera específica. Si incluimos esta opción nuestra aplicación puede que no sea portable y no funcione en otro servidor de aplicaciones.
 - **description** - descripción de la anotación.
- **@Stateless:** Indica que el Bean de Sesión es sin estado. Sus atributos son:
 - **name** - por defecto el nombre de la clase pero se puede especificar otra diferente.
 - **mappedName** - si se quiere que el contenedor maneje el objeto de manera específica. Si incluimos esta opción nuestra aplicación puede que no sea portable y no funcione en otro servidor de aplicaciones.
 - **description** - descripción de la anotación.
- **@Init:** Especifica que el método se corresponde con un método create de un EJBHome o EJBLocalHome de EJB 2.1. Sólo se podrá; llamar una única vez a este método. Sus atributos son:
 - **value** - indica el nombre del correspondiente método create de la interfaz home adaptada. Sólo se debe utilizar cuando se utiliza el bean anotado con un bean con estado de la especificación 2.1 o anterior y que disponga de más de un método create.
- **@Remove:** Indica que el contenedor debe llamar al método cuando quiera destruir la instancia del Bean. Sus atributos son:
 - **retainIfException** - indica si el Bean debe mantenerse activo si se produce una excepción. Por defecto a false.
- **@Local:** Indica que la interfaz es local.
- **@Remote:** Indica que la interfaz es remota.
- **@PostActivate:** Invocado después de que el Bean sea activado por el contenedor.
- **@PrePassivate:** Invocado antes de que el Bean esté en estado passivate.

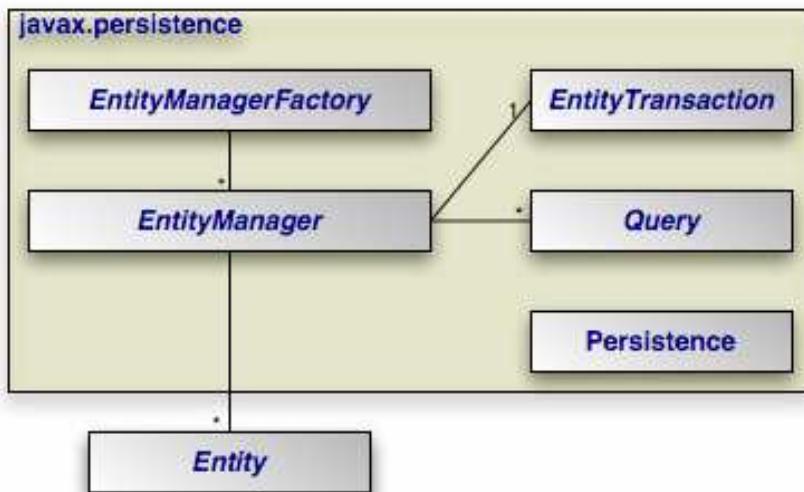
Generalmente las anotaciones empleadas serán @Stateless y @Stateful, para indicar el tipo de EJB que estemos utilizando. Más raro será el uso del resto de anotaciones, empleadas en casos más particulares.

5.3.- **Tecnologías Modelo: JPA**

JPA es el estándar para gestionar persistencia de Java, definida como parte de EJB3 y que opera con POJO's. Se trata de tratar las entidades de base de datos como objetos Java, o lo que es lo mismo, dotar de persistencia a los objetos del lenguaje Java.

5.3.1.- Introducción

En esta figura se muestra la interacción de los diversos componentes de la arquitectura de JPA:



El principal componente de la arquitectura es el “**EntityManager**”. Generalmente en un servidor de aplicaciones con contenedor de EJB esta entidad nos será proporcionada y no es necesario recurrir al factory, pensado para entornos sin contenedor de EJB (por ejemplo en un entorno de test). La clase **Persistence** dispone de métodos estáticos que permiten obtener una instancia de **EntityManagerFactory** de forma independiente al proveedor de JPA y a partir del que se puede obtener el **EntityManager** necesario.

Mediante **EntityManager** podemos obtener objetos “**Query**” para cargar objetos persistidos con determinados criterios. JPA emplea el lenguaje de consulta JPQL ó SQL. También podemos realizar operaciones CRUD sobre los objetos persistentes “**Entity**”.

Cada instancia de **EntityManager** tiene una relación uno a uno con una **EntityTransaction**, permitiendo el manejo de operaciones sobre objetos persistentes con la idea global de transacción, de forma que se ejecuten todas las operaciones o ninguna.

Por otro lado la unidad de persistencia es lo que aporta la información necesaria para enlazar con la base de datos relacional adecuada.

- **Persistence.xml** -

Es donde se definen los contextos de persistencia de la aplicación. Se debe situar dentro del directorio META-INF del fichero JAR que contiene los bean de entidad.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">

    <persistence-unit name="defaultPU" transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>jdbc/NombreDataSource</jta-data-source>
        <properties>
            <property name="hibernate.hbm2ddl.auto" value="validate"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.default_schema" value="NOMBRE"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.Oracle10gDialect"/>
            <property name="hibernate.transaction.manager_lookup_class"
                value="org.hibernate.transaction.OC4JTransactionManagerLookup"/>
            <property name="hibernate.query.factory_class"
                value="org.hibernate.hql.classic.ClassicQueryTranslatorFactory"/>
            <property name="hibernate.transaction.flush_before_completion" value="true"/>
            <property name="hibernate.cache.provider_class"
                value="org.hibernate.cache.HashtableCacheProvider"/>
        </properties>
    </persistence-unit>
</persistence>
```

Donde los elementos XML tienen el siguiente significado:

- **persistence-unit “name”**: Especifica un nombre para el contexto persistente. Si únicamente se especifica uno, no habrá que incluir su nombre cuando se recupere el EntityManager (con la anotación @PersistenceContext o @PersistenceUnit).
- **jta-data-source, non-jta-data-source**: Nombre JNDI del DataSource, también indica si el DataSource cumple con la API JTA. También debemos crear un DataSource para la conexión a la base de datos.
- **properties**: Establecen una serie de valores para el tipo de base de datos utilizada.

5.3.2.- Beans de Entidad

Los EJB de entidad están directamente relacionados con los datos de la aplicación, son objetos que mantienen en memoria los datos que maneja la aplicación, las entidades que disponen de persistencia, Noticias, Usuarios, Clientes, etc...

Los Beans de Entidad normalmente mapean (mantienen una relación en memoria) las tablas de una base de datos relacional, aunque también es posible que mantengan la persistencia de los datos en ficheros, por ejemplo un XML, o en LDAP. En cualquiera de los casos el objetivo de un Beans de Entidad es almacenar los datos en memoria desde una fuente persistente y mantener una sincronización total entre el estado de los datos entre la memoria y la fuente de datos.

Por esta razón se dice que los Beans de Entidad son los EJB que sobreviven a caídas del sistema, ya que en caso de un fallo del sistema, los datos que hay en memoria están guardados en el dispositivo persistente (que sobrevive al fin del programa por cualquier causa), con lo cual, cuando se reinicie el servidor se recuperaran sin ningún problema.

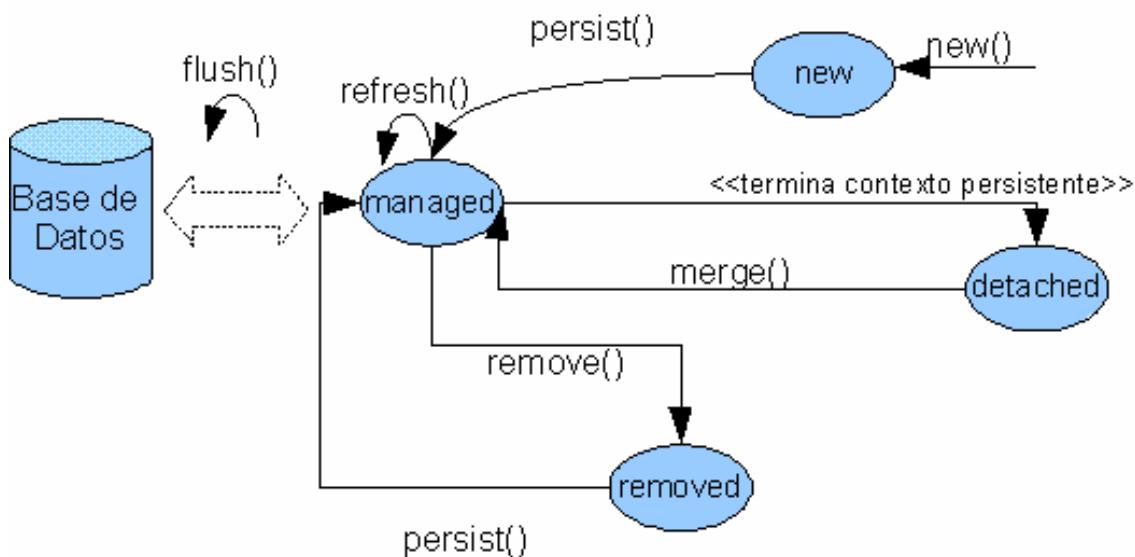
Este tipo de EJB abstrae totalmente la capa de persistencia del sistema y funciona como una herramienta de traducción de base de datos relacional a objeto, por ejemplo, se podría cambiar el nombre de una columna de una tabla y el resto de capas no se daría cuenta, ya que a esa columna se accedería a través de un método get/set del Bean de Entidad.

Dentro de los Beans de Entidad hay dos formas de manejar la persistencia, se puede dejar en manos del programador BPM BEANS o en manos del contenedor CMP BEANS. En el primer caso la responsabilidad de cómo y dónde se almacenan los datos recae en la reescritura de los métodos necesarios por parte del programador. En el segundo caso sólo hay que especificar los tipos de acceso en declaraciones de métodos y el Servidor de Aplicaciones donde se ejecuten (dentro de su correspondiente contenedor) se encargará de automatizar el proceso sin necesidad de programación extra de código por parte del programador.

- Ciclo de Vida -

Engloba dos aspectos, la relación entre el objeto Entidad y su contexto a persistir y por otro lado la sincronización de su estado con la base de datos. Para realizar estas operaciones la Entidad puede encontrarse en cualquiera de estos cuatro estados:

- **New:** Nueva instancia de la Entidad en memoria sin que aún le sea asignado su contexto persistente almacenado en la tabla de la base de datos.
- **Managed:** La Entidad dispone de contenido asociado con el de la tabla de la base de datos debido a que se utilizó el método *persist()*. Los cambios que se produzcan en la Entidad se podrán sincronizar con los de la base de datos llamando al método *flush()*.
- **Detached:** La Entidad se ha quedado sin su contenido persistente. Es necesario utilizar el método *merge()* para actualizarla.
- **Removed:** Estado después de llamarse al método *remove()* y el contenido de la Entidad será eliminado de la base de datos.



5.3.3.- Anotaciones Básicas

Una entidad JPA no es más que un POJO que dispone de una serie de anotaciones, que indican qué y cómo se persiste dicho objeto. Estas anotaciones vincula cada propiedad del POJO con las de la base de datos y establecen un vínculo único.

- **@Entity** -

Indica que el objeto es una entidad. A través de la anotación **Table** se indica la tabla de base de datos que contiene objetos de este tipo.

```
@Entity  
@Table(name= "USUARIOS")  
public class Usuario {  
  
    @Id  
    private String nick;  
    ...  
    public Usuario() { }  
  
    // Getters y Setters  
}
```

Toda entidad debe tener una clave primaria que veremos se identifica con la anotación **Id**, y disponer de un constructor vacío. Como norma debe emplearse nombre singular para las clases Java y plural para las tablas de base de datos que las contengan.

- **@Id, @IdClass, @EmbeddedId** -

Se emplean para declarar claves primarias de clases, bien de forma simple **@Id** o compuesta **@IdClass** y **@EmbeddedId**. En el caso de las claves compuestas se tiene que definir una clase para representar a la clave primaria, y redefinir el método *equals* para comprobar que dos instancias de una clase sean iguales.

```
@Entity  
@IdClass(EntidadId.class)  
public class Entidad {  
  
    @Id  
    private int id;  
    @Id  
    private String nombre;  
    ...  
    public Entidad() { }  
  
    // Getters y Setters  
}  
  
public class EntidadId {  
  
    int id;  
    String nombre;  
  
    public boolean equals(Object o) {  
        // Código que comprueba si las dos entidades son iguales  
    }  
  
    public int hashCode() {  
        // Segun las buenas practicas cuando se sobreescribe el metodo  
        // equals() hay que sobrescribir tambien el hashCode()  
    }  
}
```

```
@Entity
public class Entidad {

    @EmbeddedId
    @AttributeOverrides({
        @AttributeOverride(name = "id", column = @Column(name = "ID",
            nullable = false, precision = 5, scale = 0)),
        @AttributeOverride(name = "nombre", column = @Column(name = "NOMBRE",
            nullable = false, length = 50)),
    })
    private EntidadId id;
    ...
    public Entidad() { }

    // Getters y Setters
}

@Embedded
public class EntidadId {

    int id;
    String nombre;

    public EntidadId() { }

    public boolean equals(Object o) {
        // Codigo que comprueba si las dos entidades son iguales
    }

    public int hashCode() {
        // Segun las buenas practicas cuando se sobreescribe el metodo
        // equals() hay que sobreescribir tambien el hashCode()
    }
}
```

La recomendación es usar **@EmbeddedId** y usar el nombre de la clase con el sufijo Id para la clave primaria.

Para generar los valores de una clave primaria, es decir, que los genere JPA automáticamente, podemos utilizar la anotación **@GeneratedValue**. Esta anotación genera la clave primaria de cuatro formas diferentes:

- **AUTO**: Es el valor por defecto, y deja al proveedor de persistencia elegir cual de las siguientes tres opciones va a utilizar.
- **SEQUENCE**: Utiliza una secuencia SQL para obtener el siguiente valor de la clave primaria.
- **TABLE**: Necesita una tabla con dos columnas, el nombre de la secuencia y su valor (es la estrategia por defecto utilizada en TopLink).
- **IDENTITY**: Utiliza un generador de identidad como las columnas definidas con auto_increment en MySQL.

- **@Column** -

JPA define que los tipos primitivos, clases "envoltorio" de tipos primitivos, java.lang.String, byte[], Byte[], char[], Character[], java.math.BigDecimal, java.math.BigInteger, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Timestamp son mapeados de forma automática en columnas, es decir, no es necesario marcarlos. Además todos aquellos objetos que sean de tipo *Serializable* y estén marcados con la anotación **@Basic**.

Si lo que queremos es que una variable, de las que son mapeadas de forma automática, no sea tomada como un campo deberemos usar la anotación **@Transient**.

Además incluye formas de mapear cada campo indicando nombres de columna y otros valores como podemos ver en el ejemplo.

```
@Entity
@Table(name= "USUARIOS")
public class Usuario {
    @Id
    private String nick;

    @Column(name= "PASSWORD", nullable=false)
    private String pass;

    @Column(name= "E-MAIL", nullable=false)
    private String mail;

    @Lob
    @Basic(fetch=FetchType.LAZY)
    private byte[] imagen;
    ...
    public Usuario() { }

    // Getters y Setters
}
```

La anotación **@Column** permite definir varias propiedades. La propiedad *name* especificar el nombre de la columna donde va a ser persistido el campo, si esta propiedad no está presente el framework de persistencia usará el nombre de la variable como nombre de la columna. La propiedad *nullable* indica si la columna acepta valores null o no, si no se incluye el valor por defecto es true. Además esta anotación soporta otras muchas propiedades como pueden ser *columnDefinition*, *length*, *precision*, *scale*, *insertable*, *updatable* y *table* para definir propiedades específicas de la columna.

La anotación **@Basic** sirve para marcar cualquiera de los tipos de datos que son persistidos de forma automática además de, como se ha dicho anteriormente, de las clases que sean Serializable y permite definir las propiedades *optional* y *fetch*. La propiedad *optional* funciona igual que la propiedad *nullable* de la anotación **@Column**. La propiedad *fetch* permite definir cuándo se debe cargar el campo en cuestión, el valor **FetchType.LAZY** indica que el campo se va a cargar de forma "perezosa", es decir, el campo se cargará sólo cuando se acceda a su información, esto supone una consulta más para la obtención del campo, pero en casos de información que no es necesaria y supondría una gran carga en memoria es muy beneficioso. El valor **FetchType.EAGER** indica que el valor será cargado cuando se cargue el resto del objeto. El valor por defecto de la propiedad *fetch* es **FetchType.EAGER**.

La anotación **@Lob** indica que el contenido de un campo básico será guardado como LOB (Large Object). Si por ejemplo utilizamos esta anotación para marcar un campo que contenta un String o caracteres el framework lo mapeará a una columna CLOB (Character Large Object). Si es de otro tipo (generalmente byte[]) será mapeado a una columna de tipo BLOB (Binary Large Object).

- **@Embeddable** -

Esta anotación sirve para designar objetos persistentes que no tienen la necesidad de ser una entidad por si mismas. Esto es porque los objetos Embeddable son identificados por los objetos entidad, estos objetos nunca serán persistidos o accedidos por si mismos, sino como parte del objeto entidad al que pertenecen.

```

@Entity
@Table(name="USUARIOS")
public class Usuario {

    @Id
    private String nick;
    ...
    @Embedded
    @AttributeOverrides({@AttributeOverride(name="codigoPostal",
                                             column=@Column(name="COD_POS")),
                         @AttributeOverride(name="direccionPostal",
                                             column=@Column(name="DIR_POS"))})
    private Direccion direccion;
    ...
    public Usuario() { }
    // Getters y Setters
}

@Embeddable
public class Direccion implements Serializable {

    private String direccionPostal;
    private String ciudad;
    private int codigoPostal;
    private String pais;

    public Direccion() { }

    public boolean equals(Object o) {
        // Codigo que comprueba si las dos entidades son iguales
    }
    public int hashCode() {
        // Segun las buenas practicas cuando se sobreescribe el metodo
        // equals() hay que sobreescibir tambien el hashCode()
    }
    // Getters y Setters
}

```

Los campos contenidos en la clase marcada como `@Embeddable` son columnas de la entidad a la que pertenecen y son cargadas, de la base de datos, a la vez que la entidad.

- ***@OneToOne*** -

La anotación `@OneToOne` es usada para marcar relaciones unidireccionales y bidireccionales. Basándonos en nuestro ejemplo, pensemos que queremos tener una entidad que contenga los datos básicos del usuario, nick y password, y otra entidad con los detalles de este. Esto encajaría perfectamente en una relación 1-1. En el ejemplo se muestra una relación uno-a-uno unidireccional.

```

@Entity
@Table(name="USUARIOS")
public class Usuario {

    @Id
    private String nick;

    @Column(name="PASSWORD", nullable=false)
    private String pass;

    @OneToOne
    @JoinColumn(name="PERFIL_USUARIO_ID",
                referencedColumnName="PERFIL_ID", updatable=false)
    private Perfil perfil;
    ...
    public Usuario() { }
    // Getters y Setters
}

```

```

@Entity
@Table(name= "PERFILES")
public class Perfil {

    @Id
    @Column(name= "PERFIL_ID")
    private int id;

    @Column(name= "E-MAIL", nullable=false)
    private String mail;

    @Lob
    @Basic(fetch=FetchType.LAZY)
    private byte[] imagen;
    ...
    public Perfil() { }
    // Getters y Setters
}

```

- **@OneToMany, @ManyToOne** -

Las asociaciones 1-N y N-1 son las más comunes que nos vamos a encontrar. En este tipo de relación una entidad tiene dos o más referencias de otra. Poniendo un ejemplo un usuario puede publicar varias noticias, pero una noticia sólo puede haber sido publicada por un usuario. Esta relación sería:

```

@Entity
@Table(name= "USUARIOS")
public class Usuario {
    ...
    private Set<Noticia> noticias = new HashSet<Noticia>(0);
    // Getters y Setters
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY,
               mappedBy = "usuario")
    public Set<Noticia> getNoticias() { return this.noticias; }
    public void setNoticias(Set<Noticia> noticias) { this.noticias = noticias; }
    ...
}

```

La anotación **@OneToMany** indica que un usuario puede publicar varias noticias. La anotación **@OneToMany** tiene una propiedad llamada *cascade*, que define, con qué tipo de operaciones se realizarán operaciones en "cascada".

La propiedad *cascade*, puede tener los siguientes valores:

- ***CascadeType.PERSIST***: Cuando persistamos la entidad todas las entidades que contenga esta variable serán persistidas también.
- ***CascadeType.REMOVE***: Cuando borremos la entidad todas las entidades que contenga esta variable se borrarán del mismo modo.
- ***CascadeType.REFRESH***: Cuando actualicemos la entidad todas las entidades que contenga esta variable se actualizarán.
- ***CascadeType.MERGE***: Cuando hagamos un "merge" de la entidad todas las entidades que contenga esta variable realizarán la misma operación.
- ***CascadeType.ALL***: Todas las operaciones citadas anteriormente.

La propiedad *mappedBy = "usuario"* indica el nombre de la entidad Usuario en el objeto Noticia. Notar que se podría declarar la anotación también encima de la declaración del atributo, al igual que se hace en el ejemplo encima del método *get*.

```

@Entity
@Table(name = "NOTICIA")
public class Noticia implements java.io.Serializable {
    ...
    private Usuario usuario;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "USUARIO", nullable = false)
    @NotNull
    public Usuario getUsuario() { return this.usuario; }
    public void setUsuario(Usuario usuario) { this.usuario = usuario; }
    ...
}

```

La anotación `@ManyToOne` indica que varias noticias pueden haber sido publicadas por un mismo usuario. La anotación `@JoinColumn(name="Usuario", nullable = false)` define el nombre de la columna en la tabla de la base de datos y que este no puede ser null.

- `@ManyToMany` -

La asociación N-N es muy común, aunque no tanto como la anterior. Esta relación es, además, una de las más complicadas a la hora de ser manejada, ya que implica tablas intermedias para la relación.

Vamos a ver un ejemplo con las entidades Address y Tag. La tags se utilizan para agrupar addresses que por ejemplo, una persona posee. Una tag no es más que un String que puede ser añadida a una colección de direcciones. Tag y Address tienen una relación N-N bidireccional. En la base de datos, esta información se almacena mediante relaciones 1-N con una tercera tabla que almacena las claves primarias. La anotación `@ManyToMany` se utiliza junto con las anotaciones `@JoinTable` y `@JoinColumn`.

```

@Entity
@Table(name = "t_address")
public class Address {
    @Id @GeneratedValue
    private Long id;
    private String street;

    @Column(length = 100)
    private String city;

    @Column(name = "zip_code", length = 10)
    private String zipcode;

    @Column(length = 50)
    private String country;

    @ManyToMany(cascade = CascadeType.PERSIST)
    @JoinTable(name = "t_address_tag",
        joinColumns = {
            @JoinColumn(name = "address_fk") },
        inverseJoinColumns = {
            @JoinColumn(name = "tag_fk") })
    private List<Tag> tags = new ArrayList<Tag>();
    ...
}

@Entity
@Table(name = "t_tag")
public class Tag {
    @Id private String name;

    @ManyToMany
    private List<Address> addresses;
    ...
}

```

El identificador `@id` de la clase Tag, se llama `name` y es de tipo `String`. Su valor se establece manualmente, ya que no se indica la anotación `@GeneratedValue`. Ambas clases utilizan la anotación `@ManyToMany` para indicar que la relación es bidireccional. Cualquiera de las clases puede definir el atributo de la tabla de unión (anotación `@JoinTable`), pero en el ejemplo lo define la clase Address. La tabla de unión entre addresses y tag, se llama `t_address_tag`.

5.3.4.- Anotaciones del Ciclo de Vida

Existen otra serie de anotaciones adicionales que nos pueden ser útiles para programar aspectos o afinar en la interacción con la base de datos.

- **`@EntityListeners`**: Se pueden definir clases oyentes (listeners) con métodos de ciclo de vida de una entidad. Para hacer referencia a un listener se debe incluir esta anotación seguido entre paréntesis de la clase: `@EntityListeners(MyListener.class)`
- **`@ExcludeSuperclassListeners`**: Indica que ningún listener de la superclase será; invocado por la entidad ni por ninguna de sus subclases.
- **`@ExcludeDefaultListeners`**: Indica que ningún listener por defecto será; invocado por esta clase ni por ninguna de sus subclases.
- **`@PrePersist`**: El método se llamará justo antes de la persistencia del objeto. Podría ser necesario para asignarle la clave primaria a la entidad a persistir en base de datos.
- **`@PostPersist`**: El método se llamará después de la persistencia del objeto.
- **`@PreRemove`**: El método se llamará antes de que la entidad sea eliminada.
- **`@PostRemove`**: El método se llamará después de eliminar la entidad de la base de datos.
- **`@PreUpdate`**: El método se llamará antes de que una entidad sea actualizada en base de datos.
- **`@PostUpdate`**: El método se llamará después de que la entidad sea actualizada.
- **`@PostLoad`**: El método se llamará después de que los campos de la entidad sean cargados con los valores de su entidad correspondiente de la base de datos. Se suele utilizar para inicializar valores no persistidos.
- **`@FlushMode`**: Modo en que se ejecuta la transacción: `FlushModeType.AUTO` (por defecto) y `FlushModeType.COMMIT`.
- **`@NamedQuery`**: Especifica el nombre del objeto query utilizado junto a EntityManager. Sus atributos son:
 - **`name`** - nombre del objeto query.
 - **`query`** - especifica la query a la base de datos mediante lenguaje Java Persistence Query Language (JPQL)
- **`@NamedQueries`**: Específica varias queries como la anterior.
- **`@NamedNativeQuery`**: Específica el nombre de una query SQL normal. Sus atributos son:
 - **`name`** - nombre del objeto query.
 - **`query`** - especifica la query a la base de datos.
 - **`resultClass`** - clase del objeto resultado de la ejecución de la query.
 - **`resultSetMapping`** - nombre del SQLResultSetMapping definido (se explica más abajo).
- **`@NamedNativeQueries`**: Específica varias queries SQL.

- **@SQLResultSetMapping:** Permite recoger el resultado de una query SQL. Sus atributos son:
 - **name** - nombre del objeto asignado al mapeo.
 - **EntityResult[] entities()** - entidades especificadas para el mapeo de los datos.
 - **ColumnResult[] columns()** - columnas de la tabla para el mapeo de los datos.

```
@NamedNativeQuery(name="nativeResult", query="SELECT NOMBRE,APELLIDOS FROM USUARIOS WHERE USUARIO_ID= 123", resultSetMapping = "usuarioNamedMapping")  
  
@SqlResultSetMapping(name="usuarioNamedMapping",  
    entities = { @EntityResult(entityClass = mi.clase.Usuario.class,  
        fields = { @FieldResult(name="usuarioId", column="USUARIO_ID"),  
                   @FieldResult(name="nombre",column="NOMBRE"),  
                   @FieldResult(name="apellidos", column="APELLIDOS") } )  
})
```

- **@PersistenceContext:** Objeto de la clase EntityManager que nos proporciona todo lo que necesitamos para manejar la persistencia. Sus atributos son:
 - **name** - nombre del objeto utilizado para la persistencia en caso de ser diferente al de la clase donde se incluye la anotación.
 - **unitName** - identifica la unidad de la persistencia usada en el bean en caso de que hubiera más de una.
 - **type** - tipo de persistencia, TRANSACTION (por defecto) | EXTENDED.
- **@PersistenceContexts:** Define varios contextos de persistencia.
- **@PersistenceUnit:** Indica la dependencia de una EntityManagerFactory definida en el archivo persistence.xml. Sus atributos son:
 - **name** - nombre del objeto utilizado para la persistencia en caso de ser diferente al de la clase donde se incluye la anotación.
 - **unitName** - identifica la unidad de la persistencia usada en el bean en caso de que hubiera más de una.

Tema 6: Tecnologías Avanzadas Java EE

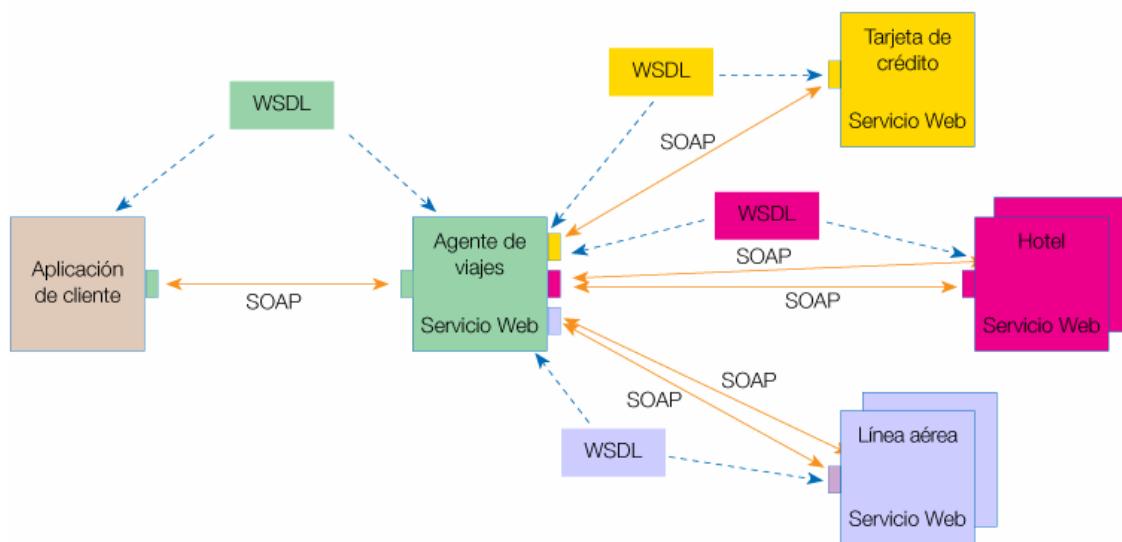
En este capítulo veremos algunas tecnologías de reciente aparición en el desarrollo de aplicaciones web, o más concretamente de reciente auge ya que todas tienen bastante solera.

6.1.- Servicios WEB

Existen múltiples definiciones sobre lo que son los Servicios Web, lo que muestra su complejidad a la hora de dar una adecuada definición que englobe todo lo que son e implican. Una posible sería hablar de ellos como un conjunto de aplicaciones o de tecnologías con capacidad para interoperar en la Web. Estas aplicaciones o tecnologías intercambian datos entre sí con el objetivo de ofrecer unos servicios. Los proveedores ofrecen sus servicios como procedimientos remotos y los usuarios solicitan un servicio llamando a estos procedimientos a través de la Web.

Estos servicios proporcionan mecanismos de comunicación estándares entre diferentes aplicaciones, que interactúan entre sí para presentar información dinámica al usuario. Para proporcionar interoperabilidad y extensibilidad entre estas aplicaciones, y que al mismo tiempo sea posible su combinación para realizar operaciones complejas, es necesaria una arquitectura de referencia estándar.

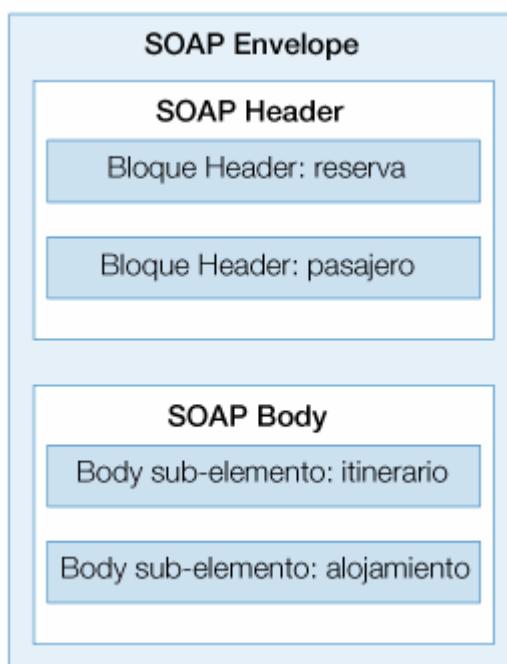
El siguiente gráfico muestra cómo interactúa un conjunto de Servicios Web:



Según el ejemplo del gráfico, un usuario (que juega el papel de cliente dentro de los Servicios Web), a través de una aplicación, solicita información sobre un viaje que desea realizar haciendo una petición a una agencia de viajes que ofrece sus servicios a través de Internet. La agencia de viajes ofrecerá a su cliente (usuario) la información requerida. Para proporcionar al cliente la información que necesita, esta agencia de viajes solicita a su vez información a otros recursos (otros Servicios Web) en relación con el hotel y la compañía aérea. La agencia de viajes obtendrá información de estos recursos, lo que la convierte a su vez en cliente de esos otros Servicios Web que le van a

proporcionar la información solicitada sobre el hotel y la línea aérea. Por último, el usuario realizará el pago del viaje a través de la agencia de viajes que servirá de intermediario entre el usuario y el servicio Web que gestionará el pago.

En todo este proceso intervienen una serie de tecnologías que hacen posible esta circulación de información. Por un lado, estaría SOAP (Protocolo Simple de Acceso a Objetos). Se trata de un protocolo basado en XML, que permite la interacción entre varios dispositivos y que tiene la capacidad de transmitir información compleja. Los datos pueden ser transmitidos a través de HTTP, SMTP, etc. SOAP especifica el formato de los mensajes. El mensaje SOAP está compuesto por un envelope (sobre), cuya estructura está formada por los siguientes elementos: header (cabecera) y body (cuerpo).



Para optimizar el rendimiento de las aplicaciones basadas en Servicios Web, se han desarrollado tecnologías complementarias a SOAP, que agilizan el envío de los mensajes (MTOM) y los recursos que se transmiten en esos mensajes (SOAP-RRSHB).

Por otro lado, WSDL (Lenguaje de Descripción de Servicios Web), permite que un servicio y un cliente establezcan un acuerdo en lo que se refiere a los detalles de transporte de mensajes y su contenido, a través de un documento procesable por dispositivos. WSDL representa una especie de contrato entre el proveedor y el que solicita. WSDL especifica la sintaxis y los mecanismos de intercambio de mensajes.

Durante la evolución de las necesidades de las aplicaciones basadas en Servicios Web de las grandes organizaciones, se han desarrollado mecanismos que permiten enriquecer las descripciones de las operaciones que realizan sus servicios mediante anotaciones semánticas y con directivas que definen el comportamiento. Esto permitiría encontrar los Servicios Web que mejor se adapten a los objetivos deseados. Además, ante la complejidad de los procesos de las grandes aplicaciones empresariales, existe una tecnología que permite una definición de estos procesos mediante la composición de varios Servicios Web individuales, lo que se conoce como coreografía.

6.2.- Autenticación Java EE

Dentro de una aplicación Java EE puede que queramos securizar el acceso a determinados recursos. Las aplicaciones Java EE proveen una forma estándar de hacer esto aunque en muchos casos para aplicaciones complejas no resulta tan sencillo.

6.2.1.- Roles de Seguridad

Podemos definir los roles de seguridad necesarios para acceder a un recurso como un servlet mediante anotaciones en el propio servlet o bien a través del fichero web.xml. También se puede hacer lo mismo con un EJB en el fichero ejb-jar.xml.

```
package srv;

import java.io.IOException;
import javax.annotation.security.DeclareRoles;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@DeclareRoles({"administrador", "usuario"})
public class Ejemplo extends HttpServlet {

    protected void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        ServletOutputStream out = response.getOutputStream();

        response.setContentType("text/html");
        out.println("<HTML><BODY bgcolor=\"#FFFFFF\">");
        out.println("request.getRemoteUser = " + request.getRemoteUser() + "<br>");
        out.println("request.isUserInRole('administrador') = " +
            request.isUserInRole("administrador") + "<br>");
        out.println("request.getUserPrincipal = " +
            request.getUserPrincipal() + "<br>");
        out.println("</BODY>");
        out.println("</HTML>");
    }
}
```

Esta anotación equivale a incluir en el fichero web.xml la entrada:

```
<security-role>
    <role-name>administrador</role-name>
</security-role>
<security-role>
    <role-name>usuario</role-name>
</security-role>
```

Emplear el método *isUserInRole* nos ayuda a conocer los roles de que dispone el usuario que ejecuta la petición. Otro método interesante es *getUserPrincipal* que nos da el usuario conectado, o mejor dicho, el usuario que solicita el recurso. Obviamente es mejor utilizar roles, ya que nos independizan algo más la aplicación y la gestión de permisos.

6.2.2.- Restricciones de Seguridad

Además de los roles, se deben indicar restricciones de seguridad para securizar los mismos, y el método de autenticación empleado para solicitar las credenciales al cliente.

```

<security-constraint>
    <web-resource-collection>
        <web-resource-name>Permiso Ejemplo</web-resource-name>
        <url-pattern>/Ejemplo</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>administrador</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
</security-constraint>
<!-- LOGIN CONFIGURATION-->
<login-config>
    <auth-method>BASIC</auth-method>
</login-config>

```

Esta declaración dentro del fichero web.xml, estaría protegiendo las llamadas al Servlet “Ejemplo”, e impidiendo el acceso a este a cualquier usuario que no fuese administrador. El patrón puede contener un asterisco final lo que permitiría proteger directorios completos. Existen varios tipos de transport-guarantee: *NONE*, utiliza http y por otro lado *CONFIDENTIAL* e *INTEGRAL* que hacen uso de https. También hay varios tipos de auth-method: *BASIC*, *DIGEST*, *FORM* y *CLIENT-CERT*. Para los dos primeros es el navegador el que solicita las credenciales y para el tercero se define un formulario personalizado, el último utiliza certificado digital.

6.2.3.- Usuarios, Grupos y Realms

El único paso que queda es dar de alta los usuarios y asociarlos a grupos y roles. Esto es particular de cada servidor, en el caso de OC4J, se incluye en JAZN que es el proveedor de JAAS específico de Oracle. Lo ideal es modificar la configuración de JAZN para la aplicación concreta, modificando el fichero orion-application.xml.

No conviene modificar el resto de archivos ya que se utilizan para el control de acceso a la administración del propio servidor. Podemos emplear un provider LDAP.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<orion-application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion-
application-10_0.xsd" deployment-version="10.1.3.3.0"
    default-data-source="jdbc/OracleDS"
    schema-major-version="10" schema-minor-version="0" >
...
<jazn provider="LDAP" location="ldap://www.example.com:636" default-realm="us" />
...
</orion-application>

```

O bien utilizar un provider XML y declarar nosotros un fichero de usuarios denominado principals.xml, que incluiremos junto con el orion-application.xml.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<orion-application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion-
application-10_0.xsd" deployment-version="10.1.3.3.0"
    default-data-source="jdbc/OracleDS"
    schema-major-version="10" schema-minor-version="0" >
...
<jazn provider="XML" location=".//jazn-data.xml" default-realm="example.com">
    <property name="jaas.username.simple" value="false"/>
</jazn>
</orion-application>

```

Y el fichero jazn-data.xml situado junto al orion-application.xml donde incluimos los usuarios y los roles a los que pertenecen para este nuevo realm. No es una estrategia muy extensible pero puede resultar útil para aplicaciones muy pequeñas con pocos usuarios.

```
<?xml version="1.0" encoding="UTF-8" standalone='yes'?>
<jazn-data
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/jazn-
data-10_0.xsd" filepath="" OC4J_INSTANCE_ID="">
<!-- JAZN Realm Data -->
<jazn-realm>
    <realm>
        <name>example.com</name>
        <users>
            <user>
                <name>admin</name>
                <credentials>!admin</credentials>
            </user>
            <user>
                <name>test</name>
                <credentials>!test</credentials>
            </user>
        </users>
        <roles>
            <role>
                <name>usuario</name>
                <description>Usuarios</description>
                <members>
                    <member>
                        <type>user</type>
                        <name>test</name>
                    </member>
                </members>
            </role>
            <role>
                <name>administrador</name>
                <description>Administradores</description>
                <members>
                    <member>
                        <type>user</type>
                        <name>admin</name>
                    </member>
                </members>
            </role>
        </roles>
    </realm>
</jazn-realm>
</jazn-data>
```

Para el servlet anterior, al invocarlo se nos pediría usuario y clave, y caso de introducir admin/admin, el resultado obtenido sería:

```
request.getRemoteUser = example.com/admin
request.isUserInRole('administrador') = true
request.getUserPrincipal = [JAZNUserAdaptor: user@example.com/admin]
```

Existe la posibilidad de utilizar providers personalizados, clases Java que provean la gestión de usuarios o como ya hemos visto usar directorios LDAP. Este tipo de alternativas son interesantes cuando se desarrolla una aplicación sin conocer su ubicación final. Por ejemplo, puede resultar interesante emplear JAAS para securizar una aplicación, y luego desarrollar un provider o emplear LDAP según el escenario en el que se despliegue la aplicación, sin que haya que modificar nada del código de esta y manteniendo por completo su seguridad.

6.3.- Portales y Portlets

Existen productos en el mercado denominados Servidores de Portales o Portal Servers cuya misión es permitir el desarrollo rápido de portales que integran el acceso a información, servicios y aplicaciones. Se generan a través de portlets o componentes de presentación. Un portal es una aplicación con las siguientes características:

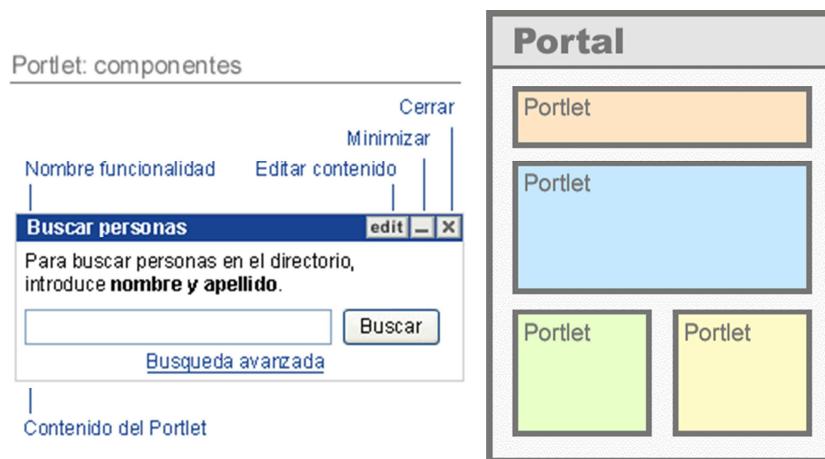
- Basado en web: servidor web y cliente navegador
- Personalizable
- Single sing-on: Existe una integración de sistemas que permite un punto de identificación único para cada usuario.
- Agregación de contenidos de diversas fuentes

6.3.1.- Definición de Portlet

Son componentes web gestionados por un contenedor que tras la petición de un usuario generan y presentan contenidos dinámicos de forma identificable en el interfaz de usuario del portal como componentes de contenido. El portlet permite la personalización, la presentación, y la gestión de la seguridad. Los portlets se diferencian de los servlets por ser estos componentes de servidor con perspectiva funcional.

Según la Java Specification Request 168 y la WSRP (Web Services for Remote Portals), que tratan de definir los estándares para el desarrollo de portlets y su interoperabilidad, son componentes web basados en Java, y gestionados por un contenedor de portlets que procesa peticiones y genera contenido dinámico. Los portales usan portlets como componentes de interfaz de usuario que proveen de una capa de presentación a los sistemas de información.

Ejemplos típicos de portlets preconfigurados pueden ser noticias provenientes de un CMS, email, cotizaciones, metereología, foros, encuestas, formularios, canales RSS, WebServices, integración de aplicaciones, herramientas de análisis, herramientas de trabajo en grupo.



El contenido generado por los portlets se denomina "fragmento". Es código XHTML, HTML, WML, etc. Los fragmentos agregados resultantes de la operación de varios portlets constituyen un documento que se traduce en el interfaz del portal. Estos elementos se disponen a través de una "retícula" o rejilla.

- Funcionamiento -

El portlet, al igual que los contenidos tiene su propio ciclo de vida. Los portales actúan como gestores de presentación de los portlets. Los usuarios realizan peticiones durante su sesión de navegación, y los portlets generarán contenidos dependiendo del perfil del usuario que realice las peticiones.

Según la especificación JSR 168 tiene tres fases:

- **Inicio (Init)**: el usuario, al interactuar con el portal arranca el portlet activando el servicio.
- **Gestión de peticiones (Handle requests)**: procesa la petición mostrando diferentes informaciones y contenido según el tipo de petición. Los datos pueden redirigir en sistemas diferentes. Dentro de esta fase se encuentra la Presentación, en la que el portlet da salida a la información en formato código para su visualización en el navegador.
- **Destrucción, (Destroy)**: elimina el portlet cuyo servicio deja de estar disponible.

- Ventajas -

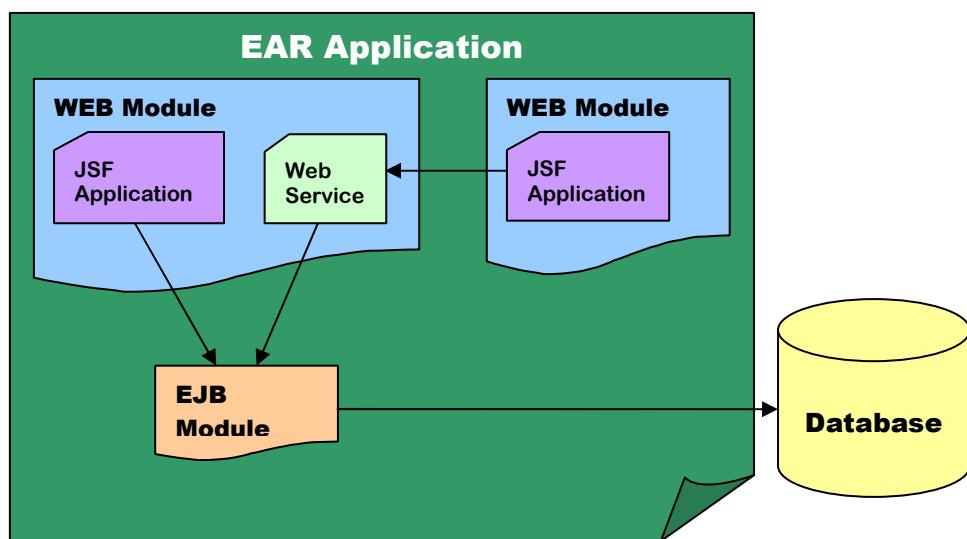
- **Rapidez de desarrollo**: Crear un portal con un servidor de portales se hace en plazos relativamente breves. Los portlets recogen funcionalidades de uso frecuente y las hacen disponibles para integrarlas en un Portal. Las ventajas derivadas es que ya están generados y debido a su uso frecuente se convierten en patrones que recogen las "mejores prácticas" del mercado.
- **Altas capacidades de Personalización**: No se trata sólo de presentar determinados portlets a diferentes usuarios, sino también de las diversas posibilidades de presentación de información de cada portlet.
- **Consolidación e integración de sitios dispersos**: Mediante un servidor de portales se centraliza la gestión de sitios dispersos ahorrando en costes de mantenimiento.
- **Administración delegada**: Mediante un sistema de permisos se delega el mantenimiento de portales o secciones agilizando los procesos de publicación.

- Inconvenientes -

- **Mecanización y despersonalización del diseño**: La traslación más evidente del Portlet a un interfaz de usuario son las "cajitas", que le otorgan ese aspecto "nukulado" impersonal y estético. El dejar que sea el portlet quien controle el aspecto final del portal puede repercutir en una despersonalización de la imagen de la compañía.
- **La rapidez en el desarrollo puede resultar una desventaja**: Cuando no se tienen unos objetivos claramente definidos. Decidirse por una herramienta de este tipo, simplemente, porque es rápida de desarrollar e instalar puede ser perjudicial y favorecer desarrollos sin visión estratégica.
- **Pérdida de perspectiva de usuario**: El sistema permite que una persona con conocimientos, a través de un checklist y una rejilla decida "cómo va a ser un portal". Algo que es una ventaja puede convertirse en un problema si los responsables de desarrollo de ese portal no adoptan el punto de vista del usuario y sus necesidades reales.

APÉNDICE A: Diario Digital

En este apéndice describo las características de la aplicación que vamos a desarrollar durante el curso. Se trata de un diario digital. En este diario se muestran las noticias recibidas de un teletipo (a través de un servicio web), así como las generadas por los redactores del propio diario. Para no complicar demasiado y para poder probar de forma autónoma la aplicación, lo que haremos será crear dos módulos web y un módulo ejb, siguiendo el siguiente esquema:



Siguiendo este esquema conectándonos a un módulo web leeremos las noticias del diario (creadas por los propios redactores) y conectándonos al otro módulo leeremos las noticias recibidas por el teletipo (a través del servicio web). En nuestro caso serán las mismas (para no complicar la cosa demasiado).

En principio para cada noticia es suficiente con que tenga: título, contenido, fecha y categoría (para ubicarlas en su sección correspondiente). También es necesario distinguir noticias que deben aparecer en portada de las que no. Los diseños de las páginas son muy sencillos:

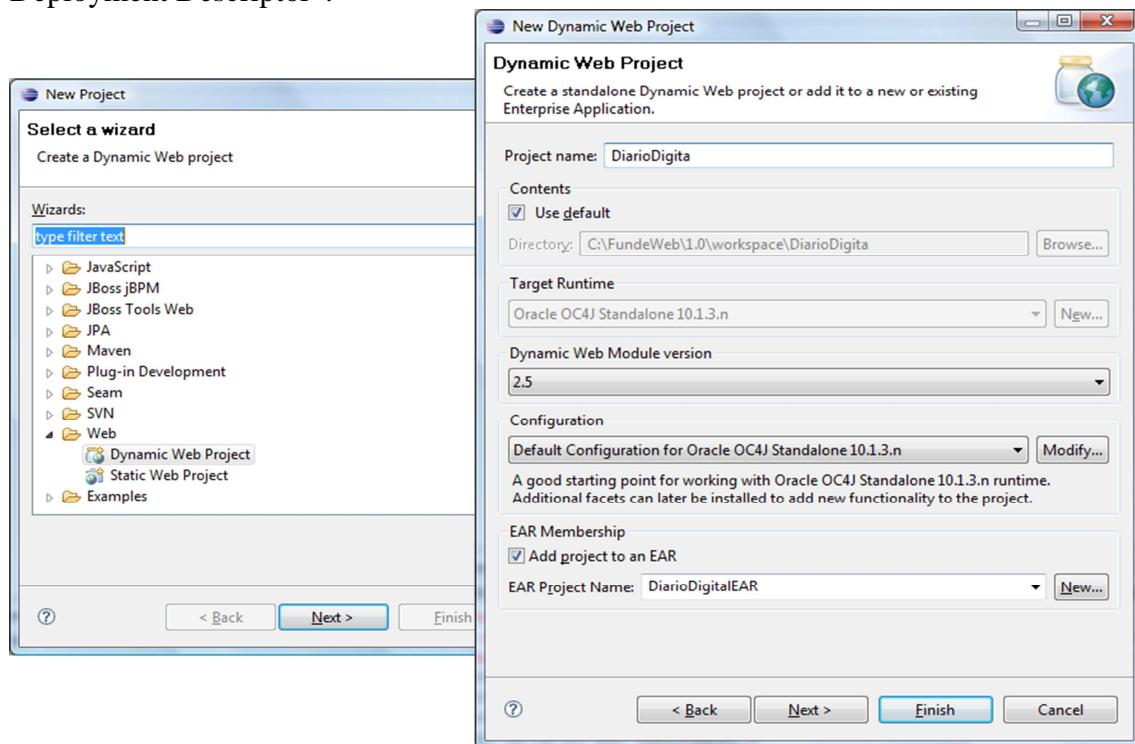
| | |
|--|--|
| Diario Digital <hr/> Portada Sección Uno Sección Dos Sección Tres Editar Noticias <hr/> <p style="text-align: center;">Diario Digital. 2009</p> | Teletipo Digital <hr/> Listado De Noticias del Teletipo <p>Noticia 1 Noticia 2 Noticia 3</p> <p style="text-align: center;">Diario Digital. 2009</p> |
|--|--|

APÉNDICE B: Configurar Proyecto Eclipse

En este apéndice se detallan los pasos para configurar un proyecto con Eclipse para trabajar con las tecnologías vistas en este curso.

B.1.- Crear Proyecto Java EE

El primer paso es crear un nuevo proyecto de tipo “Dynamic Web Project”. En la segunda ventana del asistente marcamos la opción “Add Project to an EAR”, de este modo se generarán dos proyectos, un proyecto con el módulo web y otro con el EAR que será el que usaremos para desplegar en OAS. Debemos asegurarnos que se crea el descriptor de despliegue del EAR, pulsando en “New” y marcando “Generate Deployment Descriptor”.



B.2.- Configurar Facelets y RichFaces

Ya disponemos de un proyecto web, pero ahora le vamos a añadir la característica de Facelets. Es un añadido a JSF, por lo que añadiremos al mismo tiempo las cualidades de JSF. También mostramos los pasos para configurar RichFaces.

Seleccionamos el proyecto web y en sus propiedades el apartado “Java EE Module Dependencies”. En la pestaña “Web Libraries” debemos añadir las siguientes librerías externas:

Para todas ellas {mvn_rep}=C:\FundeWeb\maven_repository\repository

- Librerías JSF -

{mvn_rep}\javax\faces\jsf-api\1.2_12\jsf-api-1.2_12.jar

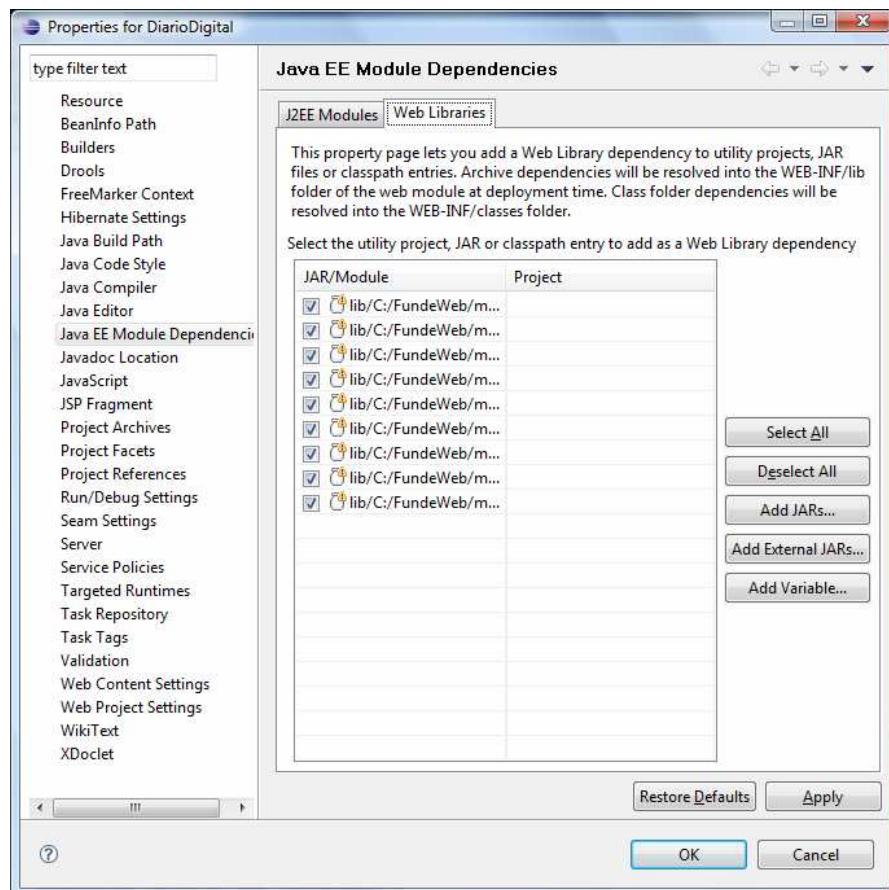
```
{mvn_rep}\com\sun\jsf-impl\1.2_12\jsf-impl-1.2_12.jar
{mvn_rep}\commons-beanutils\commons-beanutils\1.7.0\commons-beanutils-1.7.0.jar
{mvn_rep}\commons-collections\commons-collections\3.2\commons-collections-3.2.jar
{mvn_rep}\commons-digester\commons-digester\1.8\commons-digester-1.8.jar
{mvn_rep}\commons-logging\commons-logging\1.1\commons-logging-1.1.jar
{mvn_rep}\com\sun\el\el-ri\1.0\el-ri-1.0.jar
{mvn_rep}\javax\el\el-api\1.0\el-api-1.0.jar
```

- Librería Facelets -

```
{mvn_rep}\com\sun\facelets\jsf-facelets\1.1.15.B1
```

- Librerías RichFaces -

```
{mvn_rep}\org\richfaces\framework\richfaces-api\3.3.0.GA
{mvn_rep}\org\richfaces\framework\richfaces-impl\3.3.0.GA
{mvn_rep}\org\richfaces\ui\richfaces-ui\3.3.0.GA
```



Una vez que disponemos de las librerías añadidas vamos a configurar el fichero web.xml de nuestra aplicación para añadirle las diferentes características. Como sucedía antes las funcionalidades JSF son añadidas junto con las necesarias para Facelets y RichFaces.

- Fichero Web.xml -

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="Sample Application" version="2.5">

    <display-name>JavaServer Faces Sample Application</display-name>
    <description>JavaServer Faces Guess Sample Application</description>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>

    <!-- JavaServer Faces -->
    <context-param>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <!-- server o client -->
        <param-value>client</param-value>
    </context-param>

    <!-- Faces Servlet -->
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <!-- Faces Servlet Mapping -->
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.jsf</url-pattern>
    </servlet-mapping>

    <!-- Facelets -->
    <context-param>
        <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
        <param-value>.xhtml</param-value>
    </context-param>
    <context-param>
        <param-name>facelets.DEVELOPMENT</param-name>
        <param-value>true</param-value>
    </context-param>

    <!-- RichFaces -->
    <context-param>
        <param-name>org.richfaces.SKIN</param-name>
        <param-value>blueSky</param-value>
    </context-param>
    <context-param>
        <param-name>org.ajax4jsf.VIEW_HANDLERS</param-name>
        <param-value>com.sun.facelets.FaceletViewHandler</param-value>
    </context-param>

    <!-- RichFaces Filter y Listener -->
    <filter>
        <display-name>RichFaces Filter</display-name>
        <filter-name>richfaces</filter-name>
        <filter-class>org.ajax4jsf.Filter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>richfaces</filter-name>
        <servlet-name>Faces Servlet</servlet-name>
        <dispatcher>REQUEST</dispatcher>
        <dispatcher>FORWARD</dispatcher>
        <dispatcher>INCLUDE</dispatcher>
    </filter-mapping>

    <listener>
        <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
    </listener>
</web-app>
```

- Fichero Faces-Config.xml -

En este fichero se incluyen los bean manejados y las reglas de navegación de JavaServer Faces. Además para el caso de facelets es necesario incluir un manejador específico.

```
<?xml version="1.0"?>
<faces-config version="1.2"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xi="http://www.w3.org/2001/XInclude"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">

  <application>
    <!-- tell JSF to use Facelets -->
    <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
  </application>

  <!-- Beans Manejados -->

  <!-- Reglas de Navegación -->

</faces-config>
```

- Fichero Orion-Application.xml -

Este fichero solo es necesario para el caso de utilizar RichFaces con OC4J, ya que existe un conflicto entre el parser xml de los componentes de RichFaces y el empleado por OC4J, de manera que debemos invalidarlo expresamente.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<orion-application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion-
  application-10_0.xsd"
  deployment-version="10.1.3.3.0"
  default-data-source="jdbc/OracleDS"
  schema-major-version="10"
  schema-minor-version="0" >

  <web-module id="Ejemplo" path="Ejemplo.war" />
  <library path="lib" />
  <imported-shared-libraries>
    <remove-inherited name="oracle.xml" />
    <remove-inherited name="oracle.xml.security" />
  </imported-shared-libraries>
  <principals path="principals.xml" />
  <jazn provider="XML" />
  <log>
    <file path="application.log" />
  </log>
</orion-application>
```

- Ejemplo Facelets.xhtml -

Para crear una página facelets debemos por un lado crear la plantilla o layout de nuestra aplicación, sobre la que luego se incrustarán las páginas que desarrollemos. Generalmente este tipo de plantillas definirán todas las partes comunes a todas las páginas de la aplicación, como pueden ser las cabeceras, pies o laterales de exploración.

Mostramos un ejemplo de layout y una página que lo emplea, en este caso el layout es muy sencillo divide una parte superior con la cabecera y una inferior con el pie, mientras que en la parte central de la página se incluye el contenido de cada una de las pantallas de la aplicación.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" >
<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ui="http://java.sun.com/jsf/facelets"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:f="http://java.sun.com/jsf/core">

<!-- Necesario para Internet Explorer 6 y anteriores -->
<f:view contentType="text/html"/>
<head>
    <meta http-equiv="Content-Type" content="text/xhtml+xml; charset=UTF-8" />
    <title>Titulo</title>
    <link href="stylesheet/theme.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <div id="content">
        <div id="header"><ui:include src="header.xhtml"/></div>
        <div id="body">
            <ui:insert name="body">
                Aquí irá el contenido específico de cada página.
            </ui:insert>
        </div>
        <div id="footer"><ui:include src="footer.xhtml"/></div>
    </div>
</body>
</html>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ui="http://java.sun.com/jsf/facelets"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:f="http://java.sun.com/jsf/core">

<body bgcolor="white">
    <ui:composition template="/layout/template.xhtml">
        <ui:define name="body" >
            <h:form id="responseForm" >
                <h:graphicImage id="waveImg" url="/img/wave.med.gif" />
                <h2><h:outputText id="result" value="#{UserNumberBean.response}" /></h2>
                <p><h:commandButton id="back" value="Back" action="success"/></p>
            </h:form>
        </ui:define>
    </ui:composition>
</body>
</html>
```

- Ejemplo RichFaces.xhtml -

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ui="http://java.sun.com/jsf/facelets"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:f="http://java.sun.com/jsf/core"
 xmlns:a4j="http://richfaces.org/a4j"
 xmlns:rich="http://richfaces.org/rich">

<body bgcolor="white">
    <ui:composition template="/layout/template.xhtml">
        <ui:define name="body" >
            <h:form id="responseForm" >
                <rich:panel id="panelRoot" >
                    <rich:separator height="4" lineType="double" /><br/>
                    <h:graphicImage id="waveImg" url="/img/wave.med.gif" />
                    <h2><h:outputText id="result" value="#{UserNumberBean.response}" /></h2>
                    <p><h:commandButton id="back" value="Back" action="success"/></p>
                    <rich:separator height="2" lineType="solid" /><br/>
                </rich:panel>
            </h:form>
        </ui:define>
    </ui:composition>
</body>
</html>
```

B.3.- Configurar Hibernate JPA y HSQLDB

- Fichero Orion-Application.xml -

Como sucedía con RichFaces en OC4J, para emplear Hibernate debemos deshabilitar ciertos atributos empleando el fichero orion-application de OC4J.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<orion-application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion-
application-10_0.xsd"
deployment-version="10.1.3.3.0" default-data-source="jdbc/EjemploDS"
schema-major-version="10" schema-minor-version="0" >
<web-module id="Ejemplo" path="Ejemplo.war" />
<library path="lib" />
<imported-shared-libraries>
<remove-inherited name="oracle.toplink" />
<remove-inherited name="oracle.persistence" />
<remove-inherited name="apache.commons.logging" />
<remove-inherited name="oracle.xml" />
<remove-inherited name="oracle.xml.security" />
</imported-shared-libraries>
<principals path="principals.xml" />
<jazn provider="XML" />
<log>
<file path="application.log" />
</log>
</orion-application>
```

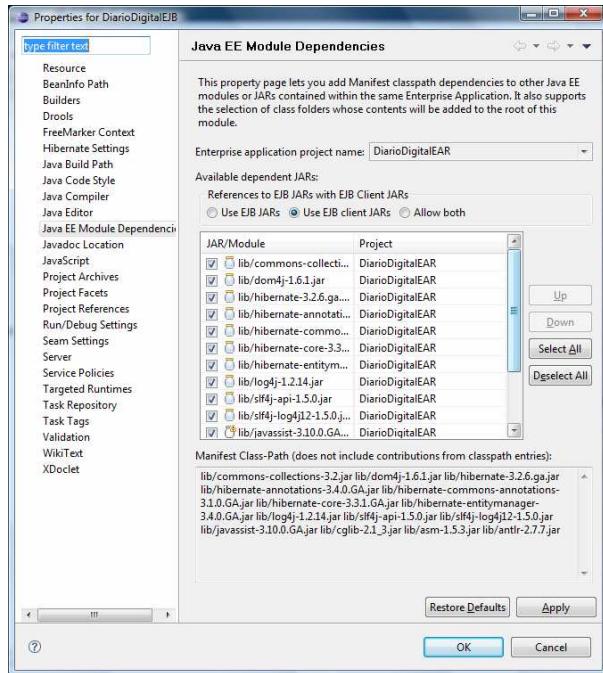
Además debemos incluir en nuestro EAR todas las librerías necesarias para que funcione la conexión con hibernate, para ello añadimos en el directorio /lib del EAR:

```
lib/commons-collections-3.2.jar, lib/dom4j-1.6.1.jar, lib/hibernate-3.2.6.ga.jar,
lib/hibernate-annotations-3.4.0.GA.jar, lib/hibernate-commons-annotations-3.1.0.GA.jar,
lib/hibernate-core-3.3.1.GA.jar, lib/hibernate-entitymanager-3.4.0.GA.jar, lib/log4j-
1.2.14.jar, lib/slf4j-api-1.5.0.jar, lib/slf4j-log4j12-1.5.0.jar, lib/javassist-
3.10.0.GA.jar, lib/cglib-2.1_3.jar, lib/asm-1.5.3.jar, lib/antlr-2.7.7.jar
```

Indicando además en el application.xml la existencia de estos archivos:

```
<?xml version="1.0" encoding="ASCII"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:application="http://java.sun.com/xml/ns/javaee/application_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/application_5.xsd" version="5">
<display-name>EjemploEAR</display-name>
<module>
<web>
<web-uri>Ejemplo.war</web-uri>
<context-root>Ejemplo</context-root>
</web>
</module>
<module>
<ejb>EjemploEJB.jar</ejb>
</module>
<library-directory>lib</library-directory>
</application>
```

Además en el módulo EJB (que es dónde son necesarias las librerías) hay que incluir las librerías, se puede hacer de forma manual incluyendo en el manifest.mf cada uno de los jar o a través de las propiedades del módulo en “Java EE Module Dependencies”.



Ahora sólo queda declarar la unidad de persistencia, en el fichero persistence.xml del módulo EJB, y utilizar dicha unidad desde los EJB de sesión:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">
    <persistence-unit name="ejemploPU" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <non-jta-data-source>jdbc/EjemploDS</non-jta-data-source>
        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.transaction.manager_lookup_class"
                value="org.hibernate.transaction.OC4JTransactionManagerLookup"/>
            <property name="hibernate.query.factory_class"
                value="org.hibernate.hql.classic.ClassicQueryTranslatorFactory" />
            <property name="hibernate.transaction.flush_before_completion" value="true" />
            <property name="hibernate.cache.provider_class"
                value="org.hibernate.cache.HashtableCacheProvider" />
        </properties>
    </persistence-unit>
</persistence>
```

Con update hacemos que si no existe alguna tabla se cree automáticamente.

```
@PersistenceContext(unitName="ejemploPU")
protected EntityManager entityManager;
try {
    // No haría falta si la anotación @PersistenceContext funcionase
    Context ctx = new InitialContext();
    if (entityManager==null) {
        EntityManagerFactory emf = (EntityManagerFactory)ctx.lookup("EjemploEJB/diarioPU");
        entityManager = (EntityManager) emf.createEntityManager();
    }
} catch (Exception ex) { System.out.println("Error: " + ex); }
```

Por último queda configurar el datasource e incluir el driver de conexión a la base de datos en OC4J. El datasource se incluye en el fichero de OC4J data-sources.xml:

```
<managed-data-source
    connection-pool-name="diario-connection-pool"
    jndi-name="jdbc/Diario DataSource"
    name="Diario DataSource" />

<connection-pool name="diario-connection-pool">
    <connection-factory
        factory-class="org.hsqldb.jdbcDriver"
        user="sa"
        password=""
        url="jdbc:hsqldb:hsq1://localhost/ejemplo" />
</connection-pool>
```

Para que todo funcione debemos levantar la base de datos, y si queremos ver los objetos que residen en ella podemos levantar el manejador. Para esto lo más sencillo es crear un proyecto Java con dos clases y un fichero de propiedades, para ejecutar con cada una de ellas el servidor y el manager:

```
public class Servidor {
    public static void main(String[] args) {
        org.hsqldb.Server.main(args);
    }
}
public class Manager {
    public static void main(String[] args) {
        org.hsqldb.util.DatabaseManager.main(args);
    }
}
// Fichero de Propiedades
server.database.0 file:./ejemplo
server.dbname.0 ejemplo
```

B.4.- Configurar Servicios Web

- Fichero Orion-Application.xml -

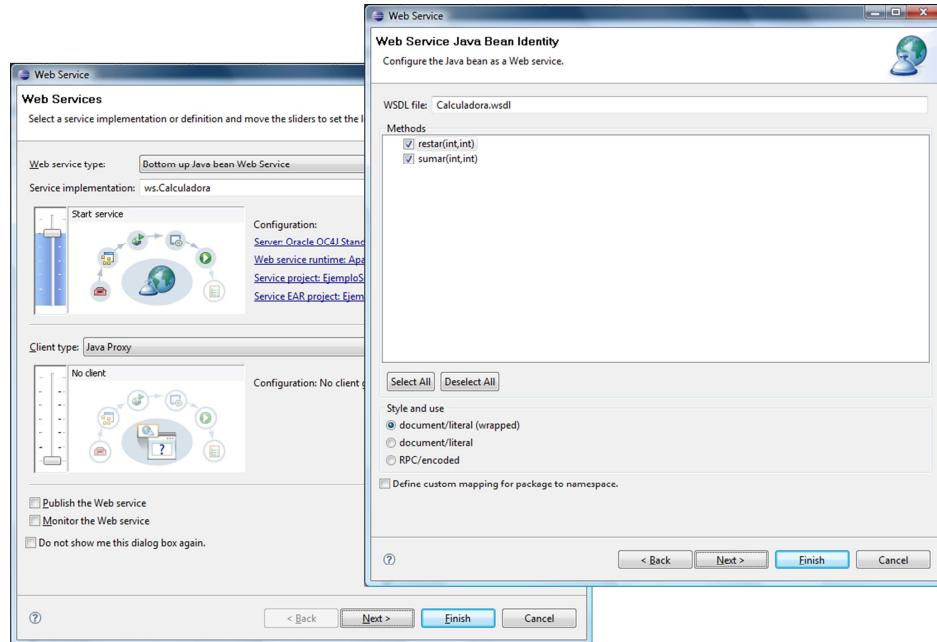
Al igual que en otras ocasiones tenemos que deshabilitar ciertas características propias de OC4J para que funcionen correctamente los servicios web desarrollados mediante Axis o JAX-WS.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<orion-application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/orion-
application-10_0.xsd"
deployment-version="10.1.3.3.0"
default-data-source="jdbc/OracleDS"
schema-major-version="10"
schema-minor-version="0" >

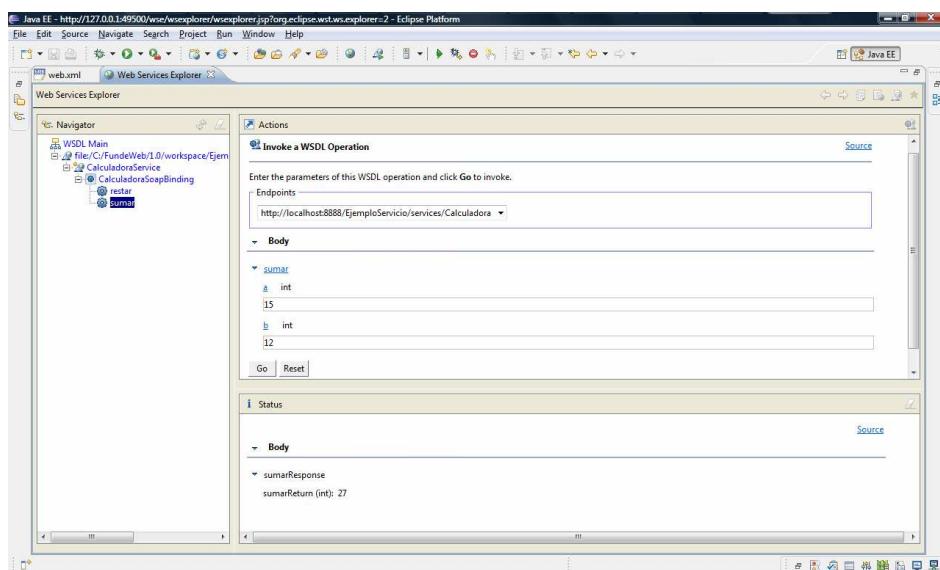
<web-module id="JaxwsService" path="JaxwsService.war" />
<library path="lib" />
<imported-shared-libraries>
    <remove-inherited name="apache.commons.logging" />
    <remove-inherited name="oracle.jwsdl" />
    <remove-inherited name="oracle.ws.core" />
    <remove-inherited name="oracle.ws.jaxrpc" />
    <remove-inherited name="oracle.ws.client" />
    <remove-inherited name="oracle.ws.reliability" />
    <remove-inherited name="oracle.ws.security" />
    <remove-inherited name="oracle.xml" />
    <remove-inherited name="oracle.xml.security" />
</imported-shared-libraries>
<principals path="principals.xml" />
<jazn provider="XML" />
<log>
    <file path="application.log" />
</log>
</orion-application>
```

- Crear un Servicio Web -

Podemos emplear dos estrategias, utilizando eclipse con Axis o JAX-WS con anotaciones. En el primer caso tras crear la clase Java en la que se incluyan los métodos que queramos publicar mediante un servicio web, lanzaremos el asistente a través del menú contextual.



Tras el asistente se generan una serie de archivos que permiten desplegar el servicio y testarlo a través del *Web Services Explorer* de Eclipse. Se generan el fichero WSDL, así como las entradas en el web.xml necesarias para que funcione el servicio web mediante Axis.



La otra alternativa es utilizar JAXWS. En ese caso se anotan las clases para indicar que servicios web se despliegan, y posteriormente se genera el resto del código empleando wsigen. Para emplear esta solución se requiere el uso de JAX-WS y sus librerías, que se pueden descargar de: <https://jax-ws.dev.java.net/2.1.3/>

```

package ws;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
public class Calculadora
{
    @WebMethod
    public int sumar(int x, int y) { return x + y; }
}

```

Una vez que la clase está anotada basta con ejecutar wsgen para generar el WSDL y las clases Java necesarias. En este caso la inclusión de las entradas en el web.xml hay que hacerla de forma manual:

```

<listener>
    <listener-class>
        com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
</listener>
<servlet>
    <description>Calculadora</description>
    <display-name>Calculadora</display-name>
    <servlet-name>Calculadora</servlet-name>
    <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Calculadora</servlet-name>
    <url-pattern>/calculadora</url-pattern>
</servlet-mapping>

```

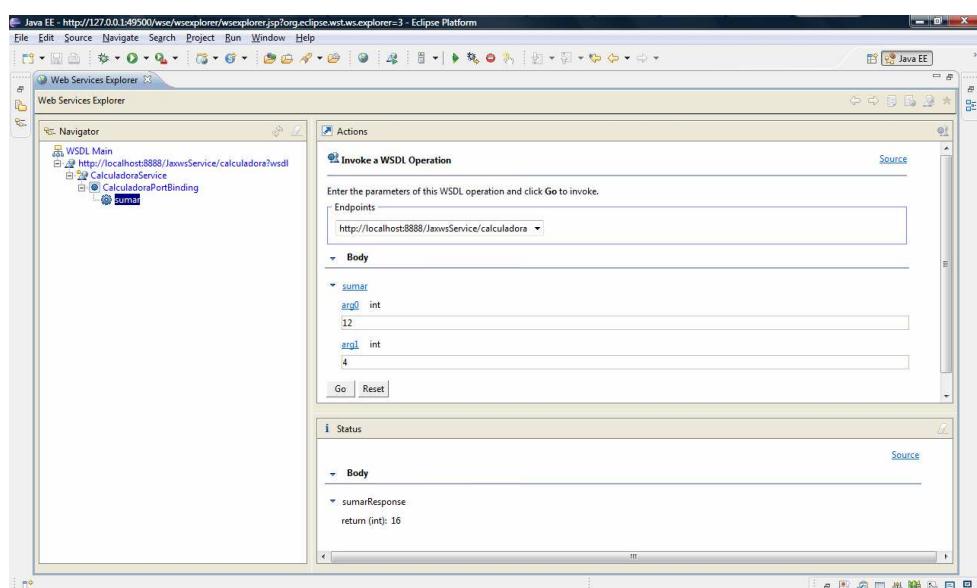
Además hay que crear un fichero describiendo un end-point, y que se sitúa junto al web.xml y se denomina *sun-jaxws.xml*:

```

<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
    <endpoint name="calculadora"
              implementation="ws.Calculadora"
              url-pattern="/calculadora" />
</endpoints>

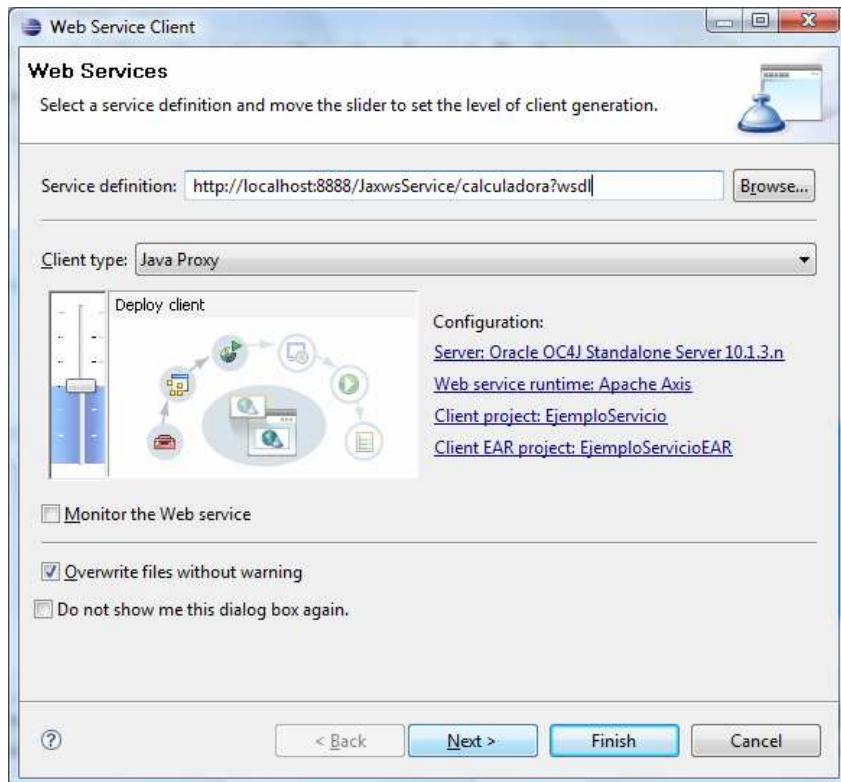
```

Y podemos testar el resultado como antes:



- Crear un Cliente -

Una vez creado el servicio web podemos crear un cliente para invocarlo. El cliente puede hacerse también mediante eclipse, que nos generará a partir del WSDL una clase Proxy adecuada para invocar al servicio definido utilizando Axis.



Este asistente genera una serie de clases que permiten invocar a los métodos de nuestro servicio como si fuesen objetos normales.

```
public static void main(String[] args) {
    try {
        CalculadoraProxy cp = new CalculadoraProxy();
        System.out.println(cp.sumar(10, 10));
    } catch (Exception ex) {
        System.out.println("Err: " + ex);
    }
}
```

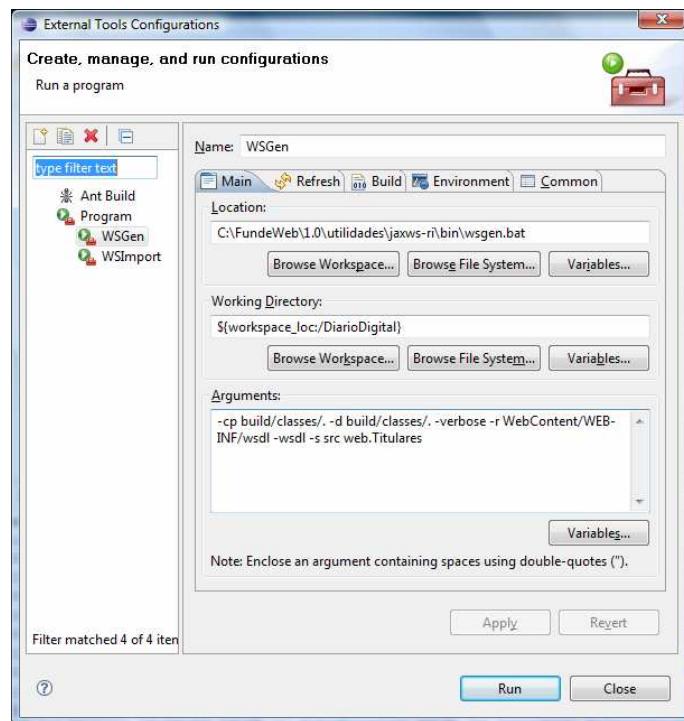
También se puede crear el cliente con JAXWS, empleando la herramienta wsimport, que genera una clase Service que nos sirve para invocar al servicio de forma similar.

```
public static void main(String[] args) {
    CalculadoraService service = new CalculadoraService();
    System.out.println(service.getCalculadoraPort().sumar(10, 15));
}
```

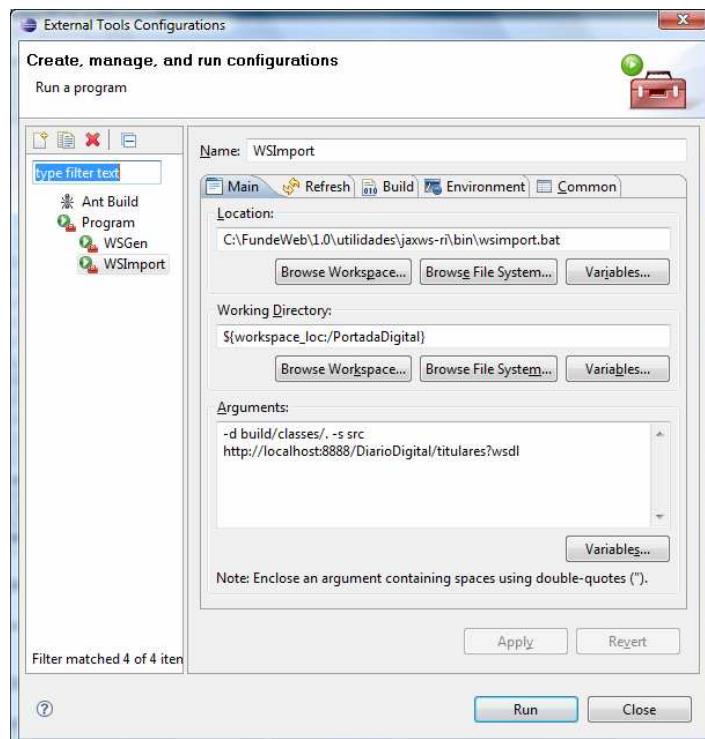
- Configurar JAXWS -

Para crear las clases Java necesarias para que funcione el servicio dentro de nuestro aplicativo, necesitamos descargar e instalar jax-ws. Una vez hecho incluimos las librerías de su directorio lib en el WEB-INF/lib de nuestra aplicación.

Después de anotar las clases de nuestra aplicación que vayan a implementar el servicio web, debemos generar los artefactos Java con WSGEN. Para ello podemos crear una herramienta externa:



Cambiando el “Working Directory” y la clase de implementación, podemos generar los artefactos Java que necesitemos para nuestro servicio web. Del mismo modo crearemos otra herramienta externa para generar el cliente una vez que hayamos desplegado el servicio web.



APÉNDICE C: Transparencias del Curso



Contenidos I

- Objetivos del Curso
 - Curso de introducción al mundo de Java EE.
 - Dirigido a:
 - Desarrolladores sin experiencia en Java EE.
 - Analistas/Jefes de Proyecto inmersos en proyectos Java EE.
 - Cualquier profesional con ánimo de conocer Java EE.
 - Para saber más: FundeWeb.
- Herramientas de Desarrollo
 - Nociones básicas de Eclipse.
 - Mención especial a otras herramientas importantes.
- Lenguaje Java
 - Nociones básicas de POO.
 - Nociones básicas de POA.
 - Nociones básicas de Java y novedades de Java 1.5.
- Patrones de Diseño
 - Ideas generales de patrones de diseño.
 - Patrones de diseño Web.
 - El Patrón MVC.

Contenidos II

- Arquitectura Java EE
 - Modelo de Capas
 - Contenedores y Servicios Java EE
 - Ensamblado y Empaquetado Java EE.
 - Eclipse y Java EE.
- Tecnologías Java EE
 - Tecnologías de la Vista: JSF y Facelets
 - Tecnologías de Control: EJB
 - Tecnologías del Modelo: JPA
- Tecnologías Avanzadas Java EE
 - Servicios WEB: JAXWS
 - Autenticación Java EE
 - Portlets
- El Proyecto del Curso
 - Un periódico digital con teletipo.

I. Herramientas de Desarrollo

I. Herramientas de Desarrollo

■ Multitud de Herramientas

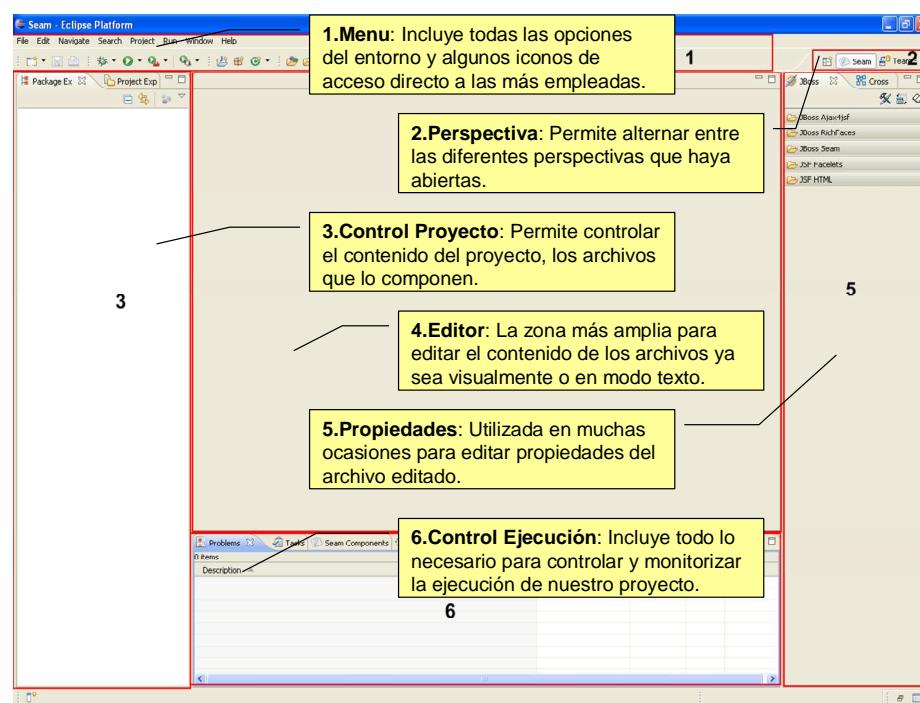
□ Jdeveloper, NetBeans, IntelliJ, Eclipse,...

□ La decisión es clara: Eclipse.

- Modular y ampliable mediante plugins.
- Muy utilizada y en constante desarrollo.
- En realidad todas las citadas tienen gran calidad.
- Es la que se emplea en FundeWeb.

□ No sólo sirve para desarrollar Java.

I. Herramientas de Desarrollo



I. Herramientas de Desarrollo

- Terminología de Eclipse
 - Espacio de Trabajo: Workspace
 - Directorio en el que se almacenan los proyectos y configuraciones específicas.
 - Se puede cambiar de espacio de trabajo de forma sencilla.
 - Recomendación: No tener todos los proyectos en un ET.
 - Perspectivas
 - Configuración predeterminada de las zonas de trabajo adecuadas para un tipo de tarea concreta: Proyectos Web, Swing, BBDD, etc...
 - Vistas
 - Pestañas que se ubican en las zonas de trabajo y que permiten hacer tareas concretas: Editor, Console, Navigator, Server,...
 - Artefactos de desarrollo: Vistas que nos ayudan a editar el proyecto.
 - Artefactos de runtime: Vistas que nos ayudan a monitorizar y testar el proyecto.
 - Configuración y Actualización de Eclipse
 - Window >> Preferences.
 - Podemos ampliar las funcionalidades del IDE según nuestras necesidades.
 - Help >> Software Updates.

I. Herramientas de Desarrollo

- Subversión.
 - Eclipse incluye su control de versiones local por sesión, pero no es suficiente.
 - Mediante el plugin “Subclipse”, se incorporan a nuestros proyectos todas las posibilidades del control de versiones.
 - Importante emplear Subversión de forma correcta:
 - Liberar versiones.
 - Emplear comentarios adecuados.
 - Crear ramas.

I. Herramientas de Desarrollo

■ Maven.

- Gestor de proyectos.
- Forma de compartir el conocimiento de la metainformación de un proyecto.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">  
    <modelVersion>4.0.0</modelVersion>  
    <groupId>org.um.atica</groupId>  
    <artifactId>Prueba_Maven</artifactId>  
    <name>Prueba_Maven</name>  
    <version>0.0.1-SNAPSHOT</version>  
    <description>Ejemplo de archivo POM de Maven.</description>  
  
    <dependencies>  
        <dependency>  
            <groupId>junit</groupId>  
            <artifactId>junit</artifactId>  
            <version>4.4</version>  
            <scope>test</scope>  
        </dependency>  
    </dependencies>  
</project>
```

I. Herramientas de Desarrollo

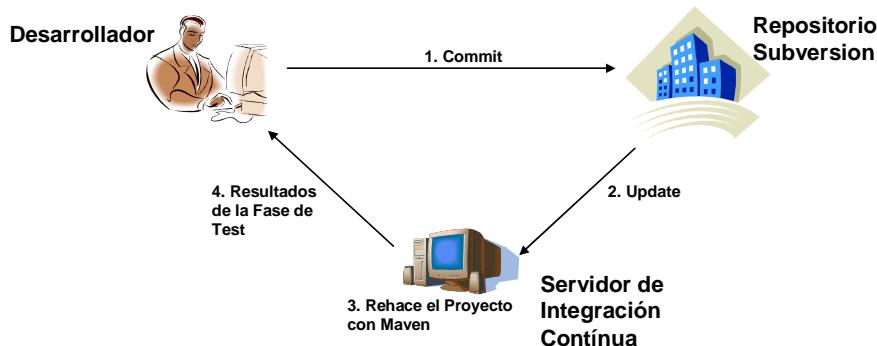
■ Maven.

- Incorporar alguien a un proyecto.
- Sin Maven
 - Instalación del software.
 - Descarga de fuentes.
 - Descarga de librerías.
 - **Configurar la herramienta para compilar y ejecutar.**
- Con Maven
 - Instalación del software.
 - Descarga de fuentes.
 - Ejecución de tarea maven.

I. Herramientas de Desarrollo

■ Hudson.

- Servidores de integración continua.
- Todas las fases de generación de mi proyecto automatizadas.
- No necesito personas supervisando tareas automáticas.



I. Herramientas de Desarrollo

■ Prácticas I

- Introducción a Eclipse
 - Crea un proyecto Java.
 - Crea dos clases.
 - Cambia la perspectiva entre Java y Resource.
- Perspectivas
 - Modifica las vistas que aparecen en la perspectiva Resource.
 - Elimina la vista Project Explorer.
 - Incluye la vista Navigator y Package Explorer.
 - Incluye la vista Problemas.
 - Restaura el estado original de la perspectiva.
- Configuración
 - Modifica aspectos generales del editor, colores, tipos de letra, etc...
 - Amplia la funcionalidad de eclipse con algún plugin.

I. Herramientas de Desarrollo

■ Prácticas II

□ Gestión de Proyectos

- Modifica las clases Java: Crea un “Hola Mundo”.
- Compara ambas clases.
- Incluye tareas pendientes en ambas clases.
- Visualiza las tareas pendientes.

□ Ejecución de Proyectos

- Ejecuta alguna de las clases creadas.
- Incluye puntos de ruptura.
- Observa las posibilidades del Debug.

□ Compartir Proyectos

- Comparte tu proyecto en un repositorio.
- Descarga un proyecto desde el repositorio.

II. Lenguaje Java

II. Lenguaje Java

■ Definición:

"Lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 90. El lenguaje toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria."

II. Lenguaje Java

■ Programación Orientada a Objetos

- Conceptos de Diseño
 - Clase
 - Definición de propiedades y comportamiento de un tipo.
 - Objeto
 - Instancia de una clase, dispone de unas propiedades concretas.
 - Método
 - Algoritmo asociado a un objeto que se lanza tras recibir un mensaje.
 - Atributo
 - Contenedor de un tipo de dato asociado a un objeto. Notación punto.
 - Mensaje
 - Comunicación dirigida a un objeto. Notación punto.

II. Lenguaje Java

- Programación Orientada a Objetos
 - Sintaxis en Java

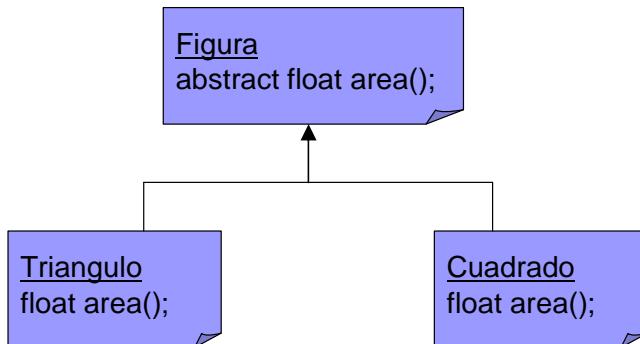
```
public class Triangulo {  
    // Propiedades del triángulo  
    public int base;  
    public int altura;  
    // Métodos del triángulo  
    public float area() { return (base*altura)/2; }  
}  
// Ejemplo de Uso  
Triangulo t = new Triangulo();  
t.altura = 10; t.base = 2;  
System.out.println("Area: "+t.area());
```

II. Lenguaje Java

- Programación Orientada a Objetos
 - Conceptos de Análisis
 - Abstracción
 - Características esenciales del objeto.
 - Ocultación
 - Exponer una interface.
 - Encapsulamiento
 - Aumentar la cohesión.
 - Polimorfismo
 - Comportamientos diferentes asociados a objetos diferentes pero con igual nombre.
 - Herencia
 - Las clases se relacionan a través de una jerarquía.

II. Lenguaje Java

- Programación Orientada a Objetos
 - Análisis en Java



II. Lenguaje Java

- Programación Orientada a Aspectos
 - Conceptos Básicos
 - Aspect
 - Funcionalidad transversal a implementar.
 - Join Point
 - Punto de la ejecución donde puede conectarse un aspecto.
 - Advice
 - Implementación del aspecto.
 - Pointcut
 - Define los aspectos a aplicar en cada Join Point.
 - Introduction
 - Permite añadir métodos o atributos a clases ya existentes.
 - Target
 - Clase a la que se añade el aspecto.
 - Proxy
 - Objeto creado después de aplicar el aspecto.
 - Weaving
 - Proceso de aplicar aspectos a los objetos.

II. Lenguaje Java

■ Programación Orientada a Aspectos: AspectJ

```
public class ColaCircular {
    private Object[] array;
    private int ptrCola = 0, ptrCabeza = 0;

    public ColaCircular (int capacidad) {
        array = new Object [capacidad];
    }

    public void Insertar (Object o) {
        array[ptrCola] = o;
        ptrCola = (ptrCola + 1) % array.length;
    }

    public Object Extraer () {
        Object obj = array[ptrCabeza];

        array[ptrCabeza] = null;
        ptrCabeza = (ptrCabeza + 1) % array.length;

        return obj;
    }

    public int getCapacidad(){
        return capacidad;
    }
}
```

```
aspect ColaCirSincro{
    private int eltosRellenos = 0;

    pointcut insertar(ColaCircular c):
        instanceof (c) && receptions<void Insertar(Object)>;
    pointcut extraer(ColaCircular c):
        instanceof (c) && receptions<Object Extraer()>;

    before(ColaCircular c):insertar(c) {
        antesInsertar(c);
    }

    protected synchronized void antesInsertar
        (ColaCircular c){
        while (eltosRellenos == c.getCapacidad()) {
            try { wait(); } catch (InterruptedException ex) {};
        }
    }

    after(ColaCircular c):insertar(c) { despuesInsertar();}

    protected synchronized void despuesInsertar (){
        eltosRellenos++;
        notifyAll();
    }

    before(ColaCircular c):extraer(c) { antesExtraer();}

    protected synchronized void antesExtraer () {
        while (eltosRellenos == 0) {
            try { wait(); } catch (InterruptedException ex) {};
        }
    }

    after(ColaCircular c):extraer(c) {
        despuesExtraer();
    }

    protected synchronized void despuesExtraer (){
        eltosRellenos--;
        notifyAll();
    }
}
```

II. Lenguaje Java

■ Programación Orientada a Aspectos

□ Interceptores en Java

- Una de las formas en las que podemos encontrar este tipo de programación es con los interceptores.
- Ciertos objetos disponen de un ciclo de vida.
- Interceptando este ciclo podemos añadir funcionalidad sin modificar el código del objeto.
- Ejemplo: Auditar los accesos a una aplicación.
 - Se puede hacer interceptando el ciclo de vida de las sesiones del servidor.
 - Añadiendo esta funcionalidad en el evento de creación.

II. Lenguaje Java

- Plataforma Java
 - Lenguaje: Veremos sus características.
 - JVM: Máquina virtual.
 - API: Biblioteca de librerías.
- Ediciones
 - Java ME: Micro Edition.
 - Java SE: Standart Edition.
 - Java EE: Enterprise Edition.
- Desarrollo vs Ejecución
 - JDK: Kit de desarrollo.
 - JRE: Kit de runtime.

II. Lenguaje Java

■ Elementos del Lenguaje

```
package curso.ejemplos;  
import curso.interfaces.Dibujable;  
public class Cuadrado extends Figura implements Dibujable {  
    private int lado;  
    public void dibujar() { ... }  
}
```

- Package, Clase
- Herencia, Implementación

II. Lenguaje Java

■ Elementos del Lenguaje

□ Variables

- Tipos primitivos: char, byte, short, int, long, float, double y boolean.
- Objetos y Arrays.
- El ámbito marcado por los bloques { ... }.
- Accesibilidad: public, protected, private.
- Ejemplos de declaración:
 - int x; int y=1;
 - Clase c; Clase d = new Clase();
 - int [] array; int [] array = {1,2,3,4};
 - Clase [] array = new Clase[] {c,d};

II. Lenguaje Java

■ Elementos del Lenguaje

□ Operadores

- Aritméticos: +, -, *, /, %
- Asignación: =, += , -=, *=, /=, %=.
- Incrementales: ++, --.
- Relacionales: >, >=, <, <=, ==, !=.
- Lógicos: &&, ||, !.
- Cadenas: +.
- Bits: >>, <<, &, |, ^, ~.

□ Estructuras de Control

- Comentarios: // y /* ... */
- Bifurcaciones: if (cond) { ... } else { ... }
- Bucles: while (cond) { ... }, for (init; cond; inc) { ... }

II. Lenguaje Java

■ Elementos del Lenguaje

□ Excepciones

- Control de errores dentro del propio lenguaje.
- Implícitas: RuntimeException.
- Generadas por errores de programación.
- Explícitas: Resto de Exception.
- El compilador obliga a chequear estas excepciones.

```
public void ejemplo (String n, String c) throws Exception {
    try {
        int num = Integer.parseInt(n);
        Class.forName(c);
    } catch (NumberFormatException nfe) { throw nfe; }
    catch (ClassNotFoundException cnfe) { throw cnfe; }
    finally {}
}
```

II. Lenguaje Java

■ Elementos del Lenguaje: Novedades Java 1.5

- Tipos Parametrizados
 - Vector<String> v = new Vector<String>();
 - String s = v.elementAt(0); // No requiere casting.
- Autoboxing
 - Vector<Integer> v = new Vector<Integer>();
 - v.addElement(30); // No requiere conversión int – Integer.
- Bucles Simples
 - Vector<String> v = new Vector<String>();
 - for (String c: v) System.out.println(c);
- Tipo “enum”
 - enum EstadoCivil { soltero, casado, divorciado };
 - EstadoCivil ec = EstadoCivil.casado;
- Import Estático
 - import static java.lang.Math;
 - double raiz = sqrt(1245);
- Argumentos Variables
 - public void miFunc(String p1,int ... args) { for (int i:args) { ... } }
- Metainformación
 - Anotaciones p.e. @Override.

II. Lenguaje Java

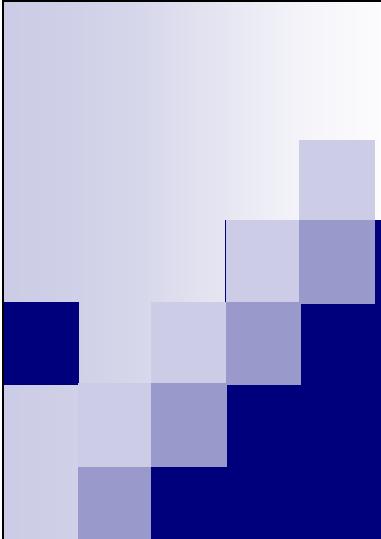
■ Prácticas I

- Crea un proyecto Java: Calculadora.
- Crea una calculadora para las 4 operaciones básicas.
- Como interface usaremos los argumentos de entrada.
 - `java Calculadora 3 * 5`
- Maneja las excepciones oportunas.

II. Lenguaje Java

■ Prácticas II: El diario digital.

- La práctica global del curso consiste en la creación de un Diario Digital.
- El Diario es un simple listado de noticias, y cada noticia pertenece a una categoría.
- La portada del Diario estará formada por las noticias de última hora (novedades).
- Las noticias se componen de un título, una fecha y el contenido.
- Crea las clases que creas necesarias y una sencilla interface para testarlas.



III. Patrones de Diseño



III. Patrones de Diseño

■ Definición:

- “Esquemas predefinidos aplicables en diversas situaciones que garantizan ciertas cualidades al diseño obtenido.”

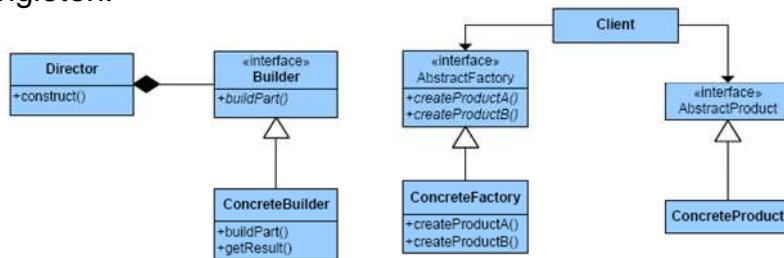
■ Motivación:

- Un porcentaje muy elevado de los problemas a los que nos enfrentamos ya han sido resueltos anteriormente por otras personas.

III. Patrones de Diseño

■ División GoF

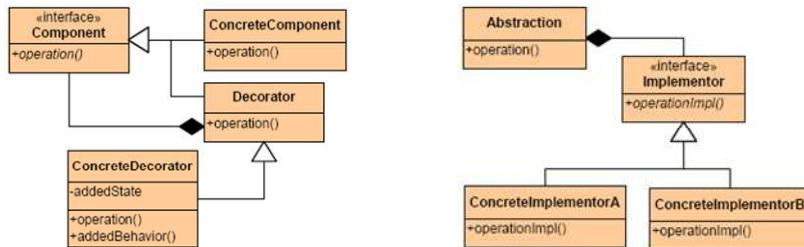
- Patrones Creacionales: Abstraer el proceso de creación de los objetos.
 - Abstract Factory, Builder, Factory Method, Prototype y Singleton.



III. Patrones de Diseño

■ División GoF

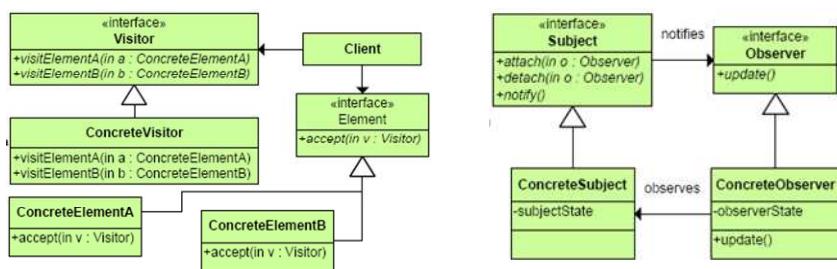
- Patrones Estructurales: Combinar objetos para crear estructuras de mayor tamaño.
 - Adapter, Bridge, Composite, Decorator, Facade, Flyweight y Proxy.



III. Patrones de Diseño

■ División GoF

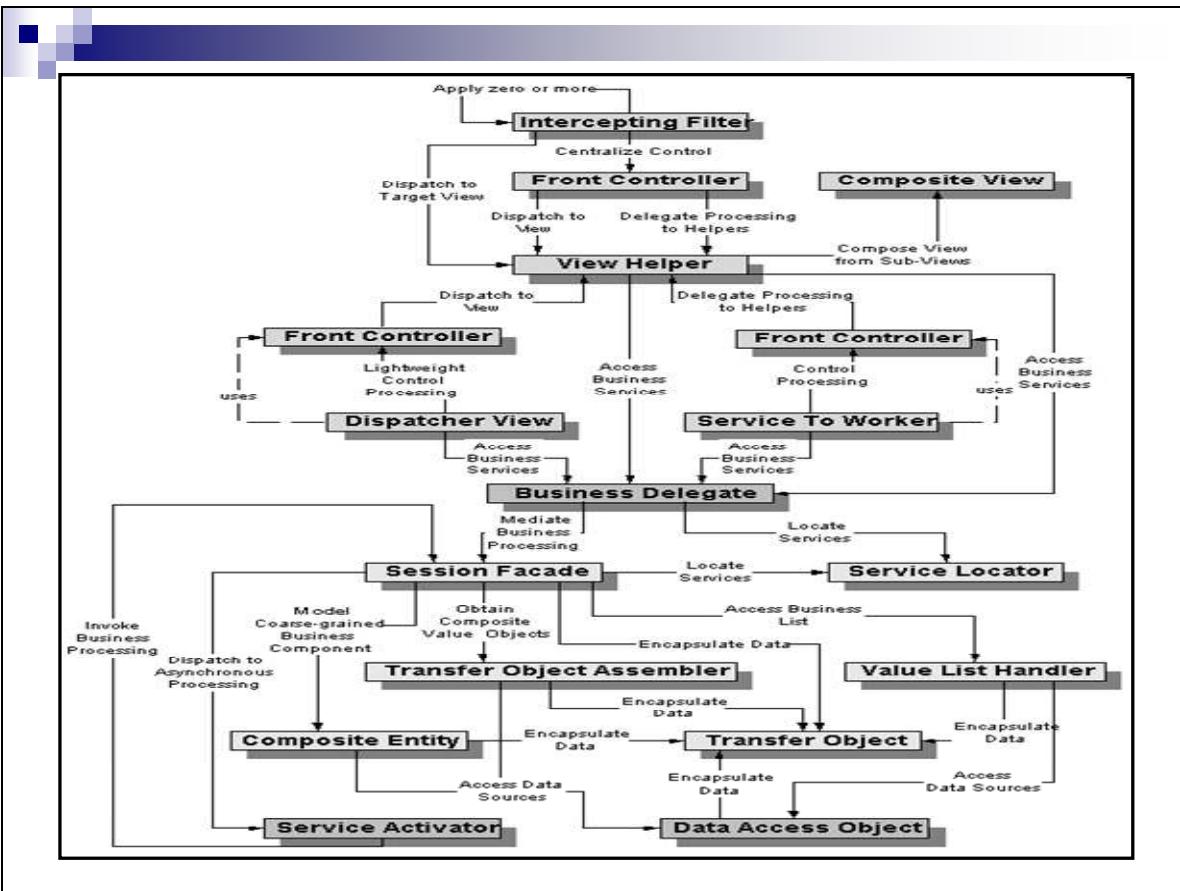
- Patrones Comportamiento: Definir interacción entre objetos reduciendo el acoplamiento.
 - Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method y Visitor.



III. Patrones de Diseño

■ Patrones Web

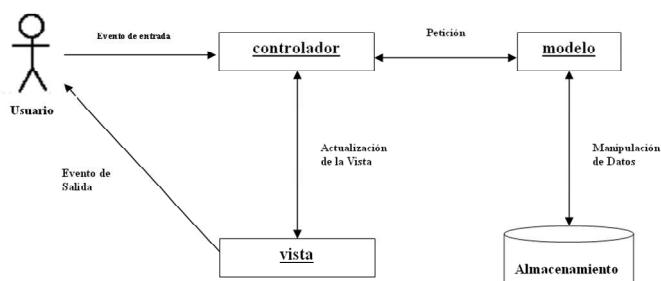
- Al amparo de los patrones de diseño del GoF aparecen otra serie de patrones específicos del mundo Web.
 - <http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html>
- Se distribuyen en tres capas: Presentación, Negocio e Integración.
- De entre todos los patrones Web existentes el más conocido y empleado es el MVC.



III. Patrones de Diseño

■ Modelo Vista Control

- **Modelo:** Representación específica de la información.
- **Vista:** Representación del modelo en un formato adecuado para que el usuario interactúe.
- **Control:** Responde a la interacción del usuario con la vista provocando cambios en el modelo.



III. Patrones de Diseño

■ Modelo Vista Control

- Unidos a este modelo aparecen otras dos estrategias.

- Inversión de Control (IoC)

- Empleada en todos los frameworks de desarrollo.
 - Principio Hollywood: "No me llames, ya te llamo yo".

- Inyección de Dependencias (DI)

- Configurar vs Programar.
 - Simplifica la obtención de recursos comunes.
 - Utilización de las anotaciones.

III. Patrones de Diseño

■ Prácticas

- Repasa los diferentes patrones del GoF y localiza los más adecuados para estos problemas:

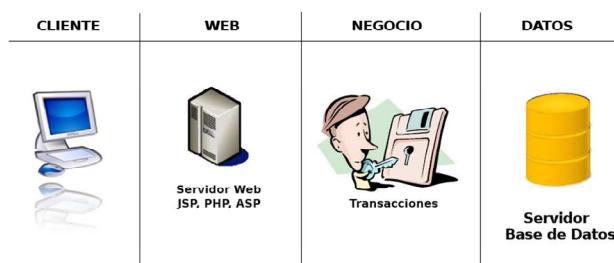
- Dotar de un sistema de undo/redo a un editor.
 - Crear un sistema gestor de ventanas que permita crear ventanas con diferentes características.
 - Crear un sistema de actualizaciones automáticas en función de la variación de diversos objetos.
 - Crear números aleatorios permitiendo emplear diversas API's de generadores de números aleatorios.
 - Asegurar la aleatoriedad de dichos números.

IV. Arquitectura Java EE

IV. Arquitectura Java EE

■ Modelo de Capas

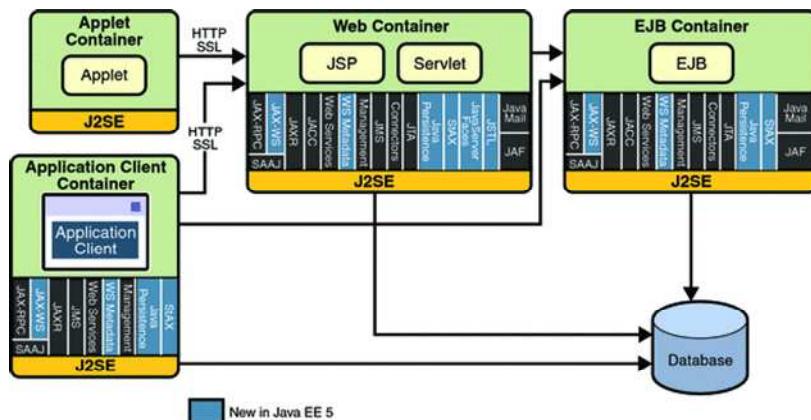
- A medida que evolucionan las aplicaciones Web, surge la necesidad de organizarlas.
- Inicialmente se emplean 3 capas.
- Actualmente es más adecuado el uso de 4.



IV. Arquitectura Java EE

■ Contenedores Java EE

- Entorno de ejecución específico para un conjunto de objetos de un determinado tipo y con unos fines concretos.



IV. Arquitectura Java EE

■ Servicios Java EE

- Para cada contenedor Java EE proporciona una serie de servicios, como por ejemplo:
 - Java Transaction API (JTA)
 - Java Persistence API (JPA)
 - Java Message Service (JMS)
 - Java Naming Direct Interface (JNDI)
 - JavaMail
 - Java Beans Active Framework (JAF)
 - Java API for XML Procesing (JAXP)
 - Java EE Connector Arquitecture
 - Java Authentication and Authorization Service (JAAS)
 - Servicios Web (JAXWS)

IV. Arquitectura Java EE

■ Ensamblado y Empaquetado

□ Módulo EAR

- Contienen una aplicación completa.
 - /*.war
 - /*.jar
 - /META-INF/application.xml

```
<?xml version="1.0" encoding="ASCII"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee" xmlns:application="http://java.sun.com/xml/ns/javaee/application_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/application_5.xsd" version="5">
  <display-name>EjemploEAR</display-name>
  <module>
    <web>
      <web-uri>Ejemplo.war</web-uri>
      <context-root>Ejemplo</context-root>
    </web>
  </module>
</application>
```

IV. Arquitectura Java EE

■ Ensamblado y Empaquetado

□ Módulo WEB

- Contienen un módulo web.
 - /*.*
 - /WEB-INF/web.xml
 - /WEB-INF/classes/*.class
 - /WEB-INF/lib/*.jar

- Dentro del módulo web:

- Servlets
 - Filters
 - Listeners

- El contenido varia mucho en función del tipo de desarrollo utilizado.

IV. Arquitectura Java EE

■ Ensamblado y Empaquetado: Módulo WEB

```
<web-app ... id="DiarioDigital" version="2.5">
<display-name>DiarioDigital</display-name>
<description>DiarioDigital</description>
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
<filter>
    <display-name>RichFaces Filter</display-name>
    <filter-name>richfaces</filter-name>
    <filter-class>org.ajax4jsf.Filter</filter-class>
</filter>
<filter-mapping>
    <filter-name>richfaces</filter-name>
    <servlet-name>Faces Servlet</servlet-name>
</filter-mapping>
<listener><listener-class>com.sun.faces.config.ConfigureListener</listener-class></listener>
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
</web-app>
```

IV. Arquitectura Java EE

■ Ensamblado y Empaquetado

□ Módulo EJB

- Contienen un módulo EJB.
 - /*.class
 - /META-INF/ejb-jar.xml
- El uso de anotaciones simplifica el contenido del descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
    <display-name>SimpleEJB</display-name>
</ejb-jar>
```

IV. Arquitectura Java EE

■ Prácticas

- Crea un Dynamic Web Project usando J2EE Preview.
- Añade un Servlet.
- Añade un Filtro.
- Añade un Listener.
- Añade una página index con un enlace al servlet.
- Crea un servidor para probar la aplicación.
- Incluye la vista “TCP/IP Monitor” y úsala.

V. Tecnologías Java EE

V. Tecnologías Java EE

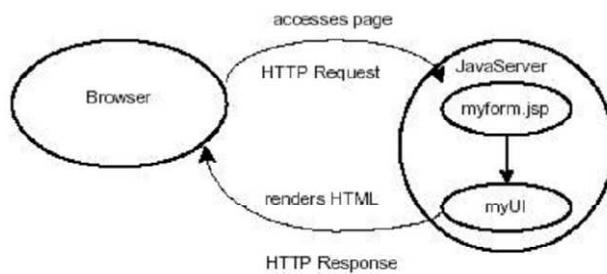
■ Tecnologías Vista: JSF

- Estándar SUN: Existen muchas alternativas.
 - Comunidad de desarrollo amplia.
 - Apoyo tecnológico de las principales compañías.
 - Adaptación de las mejores ideas de otros.
 - Lentitud en asimilar nuevas tecnologías.
 - Modificaciones o mejoras lentas.
 - Dependencia de implementaciones de terceros.

V. Tecnologías Java EE

■ Tecnologías Vista: JSF

- Componentes de JSF:
 - API + Implementación de Referencia.
 - Representan componentes UI y manejan su estado, eventos, validaciones, navegación, etc...
 - Librería de Etiquetas.
 - Etiquetas personalizadas de JSP para dibujar los componentes UI dentro de las páginas JSP.



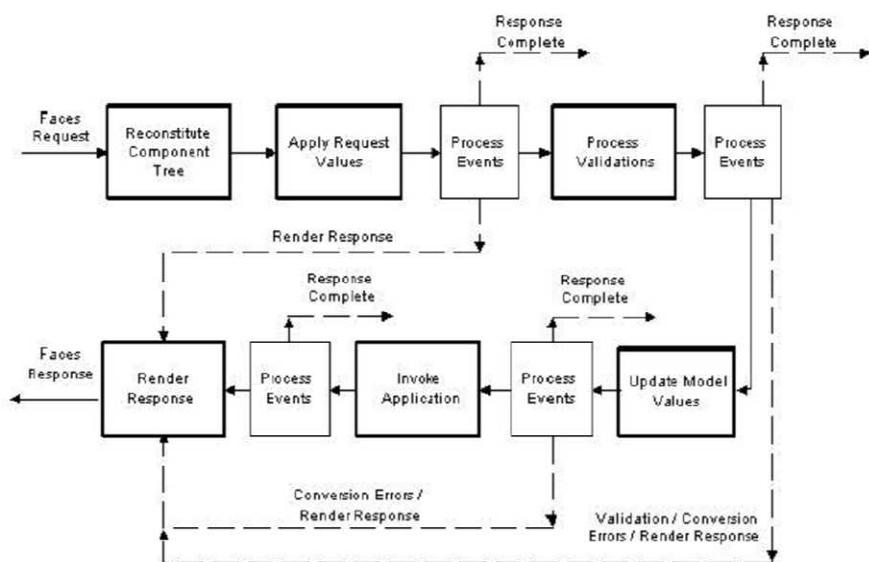
V. Tecnologías Java EE

■ Ciclo de Vida JSF

- Las peticiones Faces no se limitan a petición-respuesta, disponen de un ciclo de vida.
- El ciclo de vida depende del tipo de petición.
 - Respuesta No-Faces: Respuesta generada al margen de la fase de renderizar respuesta de faces.
 - Respuesta Faces: Respuesta generada en la fase de renderizar respuesta de faces.
 - Petición No-Faces: Petición enviada a un componente no faces.
 - Petición Faces: Petición enviada desde una respuesta faces previamente generada.
- El escenario normal Petición faces/Respuesta faces.

V. Tecnologías Java EE

■ Ciclo de Vida JSF



V. Tecnologías Java EE

- Ciclo de Vida JSF
 - Reconstruir el árbol de componentes.
 - Se construye el árbol de componentes faces.
 - Aplicar valores a la petición.
 - Se asocian a los componentes los nuevos valores desde los parámetros de la petición.
 - Procesar validaciones.
 - Se procesan las validaciones para los componentes.
 - Actualizar los valores del modelo.
 - Una vez es válido se actualizan los valores del modelo.
 - Invocar aplicación.
 - En este punto se manejan los eventos a nivel de aplicación.
 - Renderizar respuesta.
 - Por último se dibujan los componentes del árbol.

V. Tecnologías Java EE

- Componentes JSF
 - Conjunto de clases UIComponent.
 - Representan los componentes.
 - Modelo de renderizado.
 - Forma de visualizar el componente.
 - Modelo de eventos.
 - Forma de manejar los eventos lanzados.
 - Modelo de conversión.
 - Conectar conversores de datos al componente.
 - Modelo de validación.
 - Forma de registrar validadores para el componente.
- Se emplean las etiquetas.
- RichFaces, ICEFaces: Librerías de etiquetas.

V. Tecnologías Java EE

■ Componentes JSF

```
<h:dataTable id="noticias" value="#{Noticias.listadoCategoria}" var="noti">
<h:column>
    <f:facet name="header"><h:outputText value="Titular"/></f:facet>
    <h:outputText value="#{noti.titulo}" />
</h:column>
<h:column>
    <f:facet name="header"><h:outputText value="Contenido"/></f:facet>
    <h:outputText value="#{noti.contenido}" />
</h:column>
</h:column>
</h:dataTable>
-----
<h:form id="NoticiaForm">
<h:outputText value="Código:"/>
<h:inputText id="codigo" value="#{GestorNoticias.noticia.codigo}" required="true" /><br/>
<h:outputText value="Título:"/>
<h:inputText id="titulo" value="#{GestorNoticias.noticia.titulo}" required="true" /><br/>
<h:outputText value="Contenido:"/>
<h:inputText id="contenido" value="#{GestorNoticias.noticia.contenido}" required="true" /><br/>
<h:outputText value="Fecha:"/>
<h:inputText id="fecha" value="#{GestorNoticias.noticia.fecha}" required="true">
    <f:convertDateTime pattern="dd/MM/yyyy"/>
</h:inputText><br/>
<h:outputText value="Portada:"/>
<h:selectBooleanCheckbox id="portada" value="#{GestorNoticias.noticia.portada}" required="true" /><br/>
<h:outputText value="Categoria:"/>
<h:selectOneMenu id="categoria" value="#{GestorNoticias.categoriaId}">
    <f:selectItems value="#{GestorNoticias.selectCategorias}" />
</h:selectOneMenu><br/>
<h:commandButton value="Guardar" action="#{GestorNoticias.saveNoticia}" />
</h:form>
```

| Titular | Contenido |
|------------------|--|
| Nuevo Diario | Este es un diario digital de nuevo estilo. |
| Titular | Esta noticia es de titular |
| Título Extremo | Contenido extremo |
| Título Deportivo | Este es el contenido |

| | |
|--|-------------------------------------|
| Código: | 0 |
| Título: | <input type="text"/> |
| Contenido: | <input type="text"/> |
| Fecha: | <input type="text"/> |
| Portada: | <input checked="" type="checkbox"/> |
| Categoría: | Sociedad |
| <input type="button" value="Guardar"/> | |

V. Tecnologías Java EE

■ Faces-Config.xml

- Archivo de configuración principal.

- Describe los bean manejados.

```
<managed-bean>
    <description>Noticiero</description>
    <managed-bean-name>GestorNoticias</managed-bean-name>
    <managed-bean-class>web.GestorNoticias</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

- Describe las reglas de navegación.

```
<navigation-rule>
    <from-view-id>/editar/editar.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>nuevaCategoria</from-outcome>
        <to-view-id>/editar/new/categoria.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>nuevaNoticia</from-outcome>
        <to-view-id>/editar/new/noticia.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

V. Tecnologías Java EE

■ Facelets

- Complemento ideal para JSF.
- Definir una plantilla para tu portal y emplearla en todas tus páginas.

```
<ui:include src="cabecera.xhtml"/>
    <ui:insert name="body"/>
        /pagina.xhtml
        <ui:composition template="/plantilla.xhtml">
            <ui:define name="body">
                ...
            </ui:define>
        </ui:composition>
<ui:include src="pie.xhtml"/>
```

V. Tecnologías Java EE

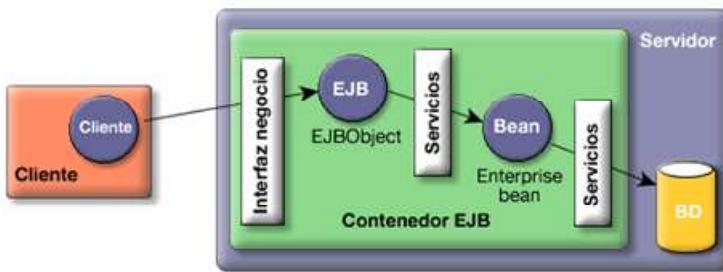
■ Prácticas I

- Crea un Hola Mundo JSF.
- Crea una plantilla facelets.
- Crea la estructura necesaria para el Diario Digital.

V. Tecnologías Java EE

■ Tecnologías Control: EJB

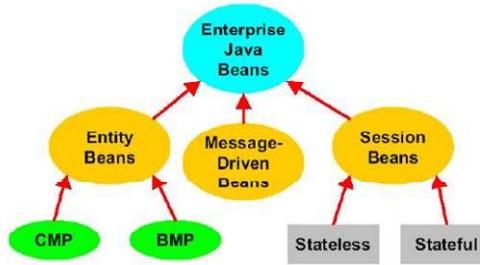
- Dar más servicios a los objetos empleados en las aplicaciones web.
- Contenedor específico para desplegar y ejecutar este tipo de objetos.
- Posibilidad de compartir lógica a través de estos objetos.
- Necesario un Contenedor de EJB. Servidor JEE.



V. Tecnologías Java EE

■ Tecnologías Control: EJB

- Tipos de EJB
 - Session Beans: Sitos en la lógica de negocio.
 - Stateless: Sin información de estado.
 - Stateful: Mantienen el estado entre peticiones.
 - Message Driven Beans: Utilizados para invocar métodos de forma asíncrona.
 - Entity Beans: Empleados en la capa de persistencia para representar datos del modelo.

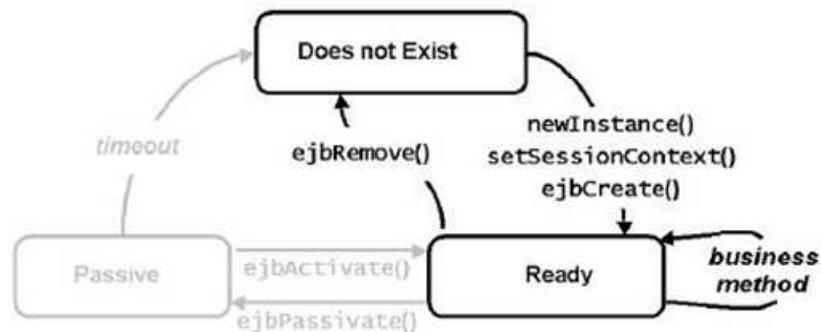


V. Tecnologías Java EE

■ Tecnologías Control: EJB

□ Ciclo de Vida

- Creación, Destrucción, Pasivación (Stateful).



V. Tecnologías Java EE

■ Tecnologías Control: EJB

□ Callbacks

- Siempre que tenemos un ciclo de vida.
- Posibilidad de emplear AOP.
 - PostConstruct
 - PreDestroy
 - PreActivate
 - PostActivate

□ Interceptores

- Siempre que empleamos contenedores IoC y Ciclos de Vida.
- Posibilidad de emplear AOP.
 - Default
 - Clase
 - Método

V. Tecnologías Java EE

■ Tecnologías Control: EJB

□ Anotaciones.

- Forma de simplificar la definición del EJB.
 - @Stateful
 - @Stateless

```
@Stateless
public class PlaceBidBean implements PlaceBid {
    @Interceptors(ActionBazaarLogger.class)
    public void addBid(Bid bid) {
        ...
    }
}

public class ActionBazaarLogger {
    @AroundInvoke
    public Object logMethodEntry(InvocationContext invocationContext) throws Exception {
        System.out.println("Entering: " + invocationContext.getMethod().getName());
        return invocationContext.proceed();
    }
}
```

V. Tecnologías Java EE

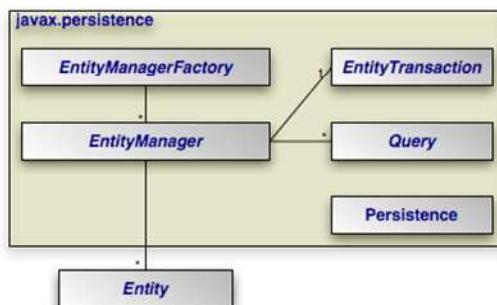
■ Prácticas II

- Crea un proyecto EJB.
- Liga tu proyecto EJB a una aplicación Web.
- Emplea los EJB's creados desde la aplicación Web.
- Crea los EJB necesarios para Diario Digital.

V. Tecnologías Java EE

■ Tecnologías Modelo: JPA

- Muchos proyectos diferentes ORM.
 - iBatis, Hibernate, JDO, TopLink,...
- Necesario unificar: JPA.



V. Tecnologías Java EE

■ Tecnologías Modelo: JPA

- Contextos de Persistencia

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence_1_0.xsd"
version="1.0">
  <persistence-unit name="defaultPU" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/NombreDataSource</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="validate"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
      <property name="hibernate.default_schema" value="NOMBRE"/>
      <property name="hibernate.dialect" value="org.hibernate.dialect.Oracle10gDialect"/>
      <property name="hibernate.transaction.manager_lookup_class"
value="org.hibernate.transaction.OC4JTransactionManagerLookup"/>
      <property name="hibernate.query.factory_class" value="org.hibernate.hql.classic.ClassicQueryTranslatorFactory"/>
      <property name="hibernate.transaction.flush_before_completion" value="true"/>
      <property name="hibernate.cache.provider_class" value="org.hibernate.cache.HashtableCacheProvider"/>
    </properties>
  </persistence-unit>
</persistence>
  
```

V. Tecnologías Java EE

■ Tecnologías Modelo: JPA

□ Empleo de Persistencia

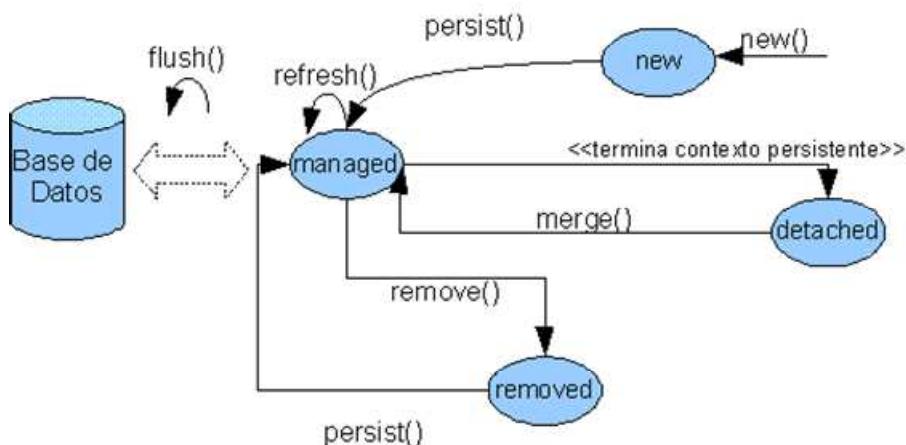
```
public @Stateful class NoticiasBean implements Noticias,Serializable {
    @PersistenceContext(unitName="diarioPU")
    protected EntityManager entityManager;
    private List<Noticia> listado;

    public NoticiasBean() {
        try {
            // En principio no haría falta si la anotación @PersistenceContext funcionase
            Context ctx = new InitialContext();
            if (entityManager==null) {
                EntityManagerFactory emf = (EntityManagerFactory) ctx.lookup("DiarioDigitalEJB/diarioPU");
                entityManager = (EntityManager) emf.createEntityManager();
            }
        }catch (Exception ex) { System.out.println("NoticiasBean: " + ex); }
    }
    public List<Noticia> getListado() {
        listado = entityManager.createQuery("from noticias.Noticia noti").getResultList();
        return listado;
    }
    public void nuevaNoticia(Noticia not) { entityManager.persist(not); }
}
```

V. Tecnologías Java EE

■ Tecnologías Modelo: JPA

□ Ciclo de Vida: Entity Beans



V. Tecnologías Java EE

■ Tecnologías Modelo: JPA

□ Anotaciones Básicas: Entidades y Claves

```
@Entity
public class Entidad {
    @EmbeddedId
    @AttributeOverrides({
        @AttributeOverride(name = "id", column = @Column(name = "ID",
            nullable = false, precision = 5, scale = 0)),
        @AttributeOverride(name = "nombre", column = @Column(name =
            "NOMBRE", nullable = false, length = 50)),
    })
    private EntidadId id;
    public Entidad() { }
    // Getters y Setters
}
@Entity
public class EntidadId {
    int id;
    String nombre;
    public EntidadId() { }
    public boolean equals(Object o) /* Comprueba si son iguales */ {
    public int hashCode() { /* Buenas practicas equals() -> hashCode() */ }
}
```

Clave Simple

```
@Entity
@Table(name="USUARIOS")
public class Usuario {
    @Id
    private String nick;
    ...
    public Usuario() { }
    // Getters y Setters
}
```

Clave Compuesta

V. Tecnologías Java EE

■ Tecnologías Modelo: JPA

□ Anotaciones Básicas: Atributos

```
@Entity
@Table(name="USUARIOS")
public class Usuario {
    @Id
    private String nick;
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="codigoPostal",column=@Column(name="COD_POS")),
        @AttributeOverride(name="direccionPostal",column=@Column(name="DIR_POS"))
    })
    private Direccion direccion;
    public Usuario() { }
    // Getters y Setters
}
@Entity
@Table(name="USUARIOS")
public class Direccion implements Serializable {
    private String direccionPostal;
    private String ciudad;
    private int codigoPostal;
    private String pais;
    public Direccion() { }
    public boolean equals(Object o) /* Comprueba si las dos entidades son iguales */ {
    public int hashCode() { /* Buenas practicas equals() -> hashCode() */ }
    // Getters y Setters
}
```

Atributo Simple

```
@Entity
@Table(name="USUARIOS")
public class Usuario {
    @Id
    private String nick;
    @Column(name="PASSWORD", nullable=false)
    private String pass;
    @Column(name="E-MAIL", nullable=false)
    private String mail;
    @Lob
    @Basic(fetch=FetchType.LAZY)
    private byte[] imagen;
    ...
    public Usuario() { }
    // Getters y Setters
}
```

Atributo Compuesto

V. Tecnologías Java EE

■ Tecnologías Modelo: JPA

□ Anotaciones Básicas: Relaciones [OneToOne](#)

```
@Entity
@Table(name="USUARIOS")
public class Usuario {
    @Id
    private String nick;
    @Column(name="PASSWORD", nullable=false)
    private String pass;
    @OneToOne
    @JoinColumn(name="PERFIL_USUARIO_ID", referencedColumnName="PERFIL_ID", updatable=false)
    private Perfil perfil;
    private Set<Noticia> noticias = new HashSet<Noticia>(0);
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy = "usuario")
    public Set<Noticia> getNoticias() {
        return this.noticias;
    }
    public void setNoticias(Set<Noticia> noticias) {
        this.noticias = noticias;
    }
    ...
}
```

```
@Entity
@Table(name="PERFILES")
public class Perfil {
    @Id
    @Column(name="PERFIL_ID")
    private int id;
    ...
}
```

ManyToMany: Muy similar a ManyToOne pero simétrica en ambas clases.

[ManyToOne](#)

```
@Entity
@Table(name = "NOTICIA")
public class Noticia implements java.io.Serializable {
    ...
    private Usuario usuario;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "USUARIO", nullable = false)
    @NotNull
    public Usuario getUsuario() { return this.usuario; }
    public void setUsuario(Usuario usuario) { this.usuario = usuario; }
    ...
}
```

V. Tecnologías Java EE

■ Tecnologías Modelo: JPA

□ Anotaciones Avanzadas: Querys Personalizadas

```
@NamedNativeQuery(
    name="nativeResult",
    query="SELECT USUARIO_ID,NOMBRE,APELLIDOS FROM USUARIOS WHERE USUARIO_ID= 123",
    resultSetMapping = "usuarioNamedMapping")

@SqlResultSetMapping (
    name="usuarioNamedMapping",
    entities = { @EntityResult (
        entityClass = mi.clase.Usuario.class,
        fields = {@FieldResult (
            name="usuarioid",
            column="USUARIO_ID"),
        @FieldResult (
            name="nombre",
            column="NOMBRE"),
        @FieldResult (
            name="apellidos",
            column="APELIDOS")})})
```

V. Tecnologías Java EE

■ Prácticas III

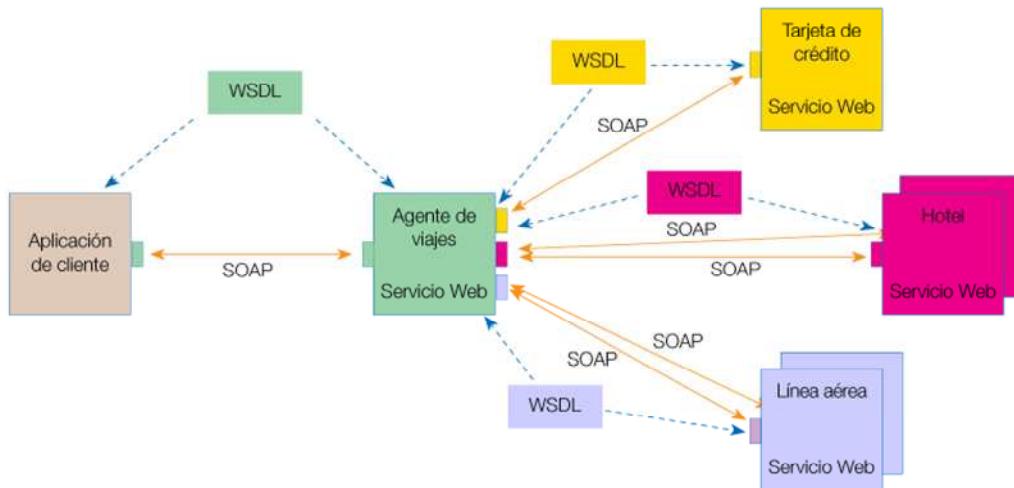
- Crea un ejemplo de Bean de Entidad.
- Incluye atributos de diferentes tipos.
- Relaciona varias entidades.
- Convierte los objetos del Diario Digital en entidades JPA para dotarlas de persistencia.

VI. Tecnologías Avanzadas

VI. Tecnologías Avanzadas

■ Servicios WEB

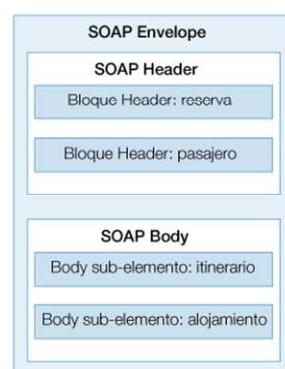
- Forma de interactuar.



VI. Tecnologías Avanzadas

■ Servicios WEB

- Buscar servicios Web: UDDI.
 - Catálogo de Servicios.
- Invocar Servicios Web: WSDL.
 - Descriptor del Servicio.
- Intercambio de Información: SOAP.
 - Documentos XML.
- Las herramientas nos simplifican el trabajo.
 - A partir de un método podemos crear un servicio.
 - A partir de un WSDL podemos crear un cliente.



VI. Tecnologías Avanzadas

■ Servicios WEB

- Implementaciones Diversas: Tratan de automatizar la creación de servicios y clientes.
 - Axis2
 - JAXWS
 - CXF
- Usaremos JAXWS.
- En FundeWeb se usará CXF, pero el curso se centra en la interacción y no en las implementaciones.

VI. Tecnologías Avanzadas

■ Prácticas I

- Crea un Sencillo Servicio Web.
- Pruébalo con el “Web Service Explorer”
- Genera un cliente a partir del WSDL.
- Crea el Servicio Web “Teletipo”.
- Incluye en el Diario Digital las noticias del “Teletipo”.

VI. Tecnologías Avanzadas

■ Autenticación JAAS

- Java EE permite emplear roles para securizar recursos de una aplicación.

```
<security-role>
    <role-name>administrador</role-name>
</security-role>
<security-role>
    <role-name>usuario</role-name>
</security-role>
```

```
<orion-application ...>
...
<jazn provider="XML" location=".jazn-data.xml" default-realm="example.com">
    <property name="jaas.username.simple"value="false"/>
</jazn>
</orion-application>
```

```
<jazn-data
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/jazn-data-10_0.xsd" filepath="" OC4J_INSTANCE_ID="">
<jazn-realm>
<realm>
<name>example.com</name>
<users><user><name>admin</name><credentials>!admin</credentials></user></users>
<roles>
    <role><name>administrador</name><members><member><type>user</type><name>admin</name></member></members></role>
</roles>
</realm>
</jazn-realm>
</jazn-data>
```

VI. Tecnologías Avanzadas

■ Autenticación JAAS

- Java EE permite emplear roles para securizar recursos de una aplicación.

```
@DeclareRoles({"administrador", "usuario"})
public class Ejemplo extends HttpServlet {
    protected void service(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
        if (req.isUserInRole("administrador")) {
            // El usuario Autenticado tiene el rol administrador
        }
    }
}
```

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Permiso Ejemplo</web-resource-name>
        <url-pattern>/Ejemplo</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>administrador</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
</security-constraint>
<!-- LOGIN CONFIGURATION--&gt;
&lt;login-config&gt;
    &lt;auth-method&gt;BASIC&lt;/auth-method&gt;
&lt;/login-config&gt;</pre>

```

VI. Tecnologías Avanzadas

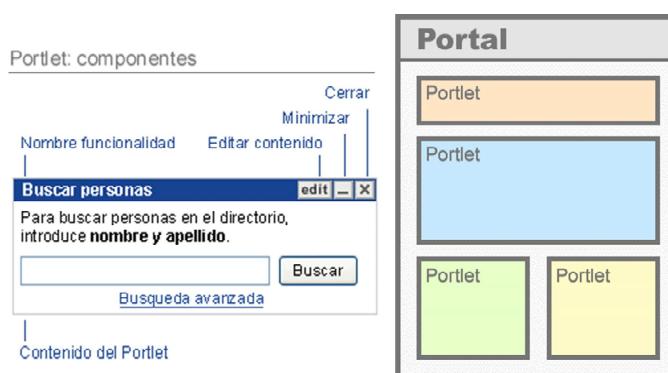
■ Prácticas II

- Crea roles en una aplicación.
- Añade seguridad a diferentes recursos.
- Securiza la creación de noticias en el Diario Digital.

VI. Tecnologías Avanzadas

■ Portales y Portlets

- Idea similar a la de los servlets.
- Componentes configurables y reubicables.
- Pensados para su uso en portales.
- Especificación JSR 168.



VI. Tecnologías Avanzadas

■ Portales y Portlets

□ Ventajas

- Desarrollo independiente y reutilizable.
- Personalización dinámica.
- Seguridad ante fallos. El fallo de un portlet no afecta al resto del portal.
- Adoptado por otras tecnologías. PHP Portlet.

□ Inconvenientes

- Tecnología relativamente nueva.
- Genera portales con poca personalidad.

A. Diario Digital

A. Diario Digital

■ Práctica Global

- Muestra las noticias de un diario y un teletipo.

