

UF1844 - Desarrollo de Aplicaciones Web en el entorno servidor

Capítulo 1. El proceso de desarrollo del software	1
Introducción	1
Modelos del ciclo de vida del software	1
Análisis y especificación de requisitos	8
Diseño	16
Implementación. Conceptos generales de desarrollo de software	26
Validación y verificación de sistemas	42
Pruebas de software	44
Calidad del software	57
Herramientas de uso común para el desarrollo de software	63
Gestión de proyectos de desarrollo de software	70
Resumen	74
Capítulo 2. La orientación a objetos	78
Introducción	78
Principios de la orientación a objetos. Comparación con la programación estructurada	78
Clases de objetos	80
Objetos	85
Herencia	98
Modularidad	107
Genericidad y sobrecarga	110
Desarrollo orientado a objetos	113
Lenguajes de modelización en el desarrollo orientado a objetos	115
Resumen	118
Capítulo 3. Arquitecturas web	122
Introducción	122
Concepto de arquitectura web	122
El modelo de capas	123
Plataformas para el desarrollo en las capas del servidor	125
Herramientas de desarrollo orientadas a servidor de aplicaciones web	125
Resumen	133
Capítulo 4. Lenguajes de programación de aplicaciones web en el lado del servidor	138
Introducción	138
Características de los lenguajes de programación web en el servidor	138
Tipos y características de los lenguajes de uso común	139
Criterios en la elección de un lenguajes de programación web en el servidor. Ventajas e inconvenientes	140
Características generales	141

Gestión de la configuración	148
Gestión de la seguridad	157
Gestión de errores	160
Transacciones y persistencia	164
Componentes en servidor. Ventajas e inconvenientes en el uso de contenedores	172
Modelos de desarrollo. El modelo vista controlador (MVC)	172
Documentación del software. Inclusión en código fuente. Generadores de documentación	178
Resumen	179

Capítulo 1. El proceso de desarrollo del software

Introducción

Un ordenador está compuesto por dos partes claramente diferenciadas: el hardware es el componente físico, mientras que el software es la parte del ordenador que abarca toda la lógica encargada de realizar una tarea de complejidad variable.

El software no es tangible, no es físico, más allá del soporte que lo almacene o donde esté instalado. Su misión es proporcionar al usuario la capacidad de interactuar con el ordenador de muy diferentes maneras. Con Word, por ejemplo, se dispone de la capacidad de procesar y editar texto; Internet Explorer permite navegar por Internet. Un sistema operativo como Windows es también software, pero en este caso sirviendo de plataforma para otros programas.

¿Qué tienen en común todos los programas de software mencionados? Para su desarrollo se ha debido seguir un proceso de varias fases en el que, partiendo de unas necesidades, se ha generado un producto operativo que satisface los requisitos iniciales.

Este proceso de desarrollo es la materia de estudio de la Ingeniería del software, una disciplina perteneciente a las ciencias de la Computación. En este capítulo, se dará una visión global de los aspectos relativos a dicha ingeniería, haciendo especial hincapié en las diferentes fases que conforman el proceso de desarrollo.

Modelos del ciclo de vida del software

Los modelos fueron surgiendo a partir de la década de los 70, fruto de la creciente complejidad asociada al objetivo buscado. Las técnicas clásicas estaban quedándose desfasadas, desperdiando muchos recursos, corrigiendo cosas sobre la marcha y volviendo a plantear requerimientos.



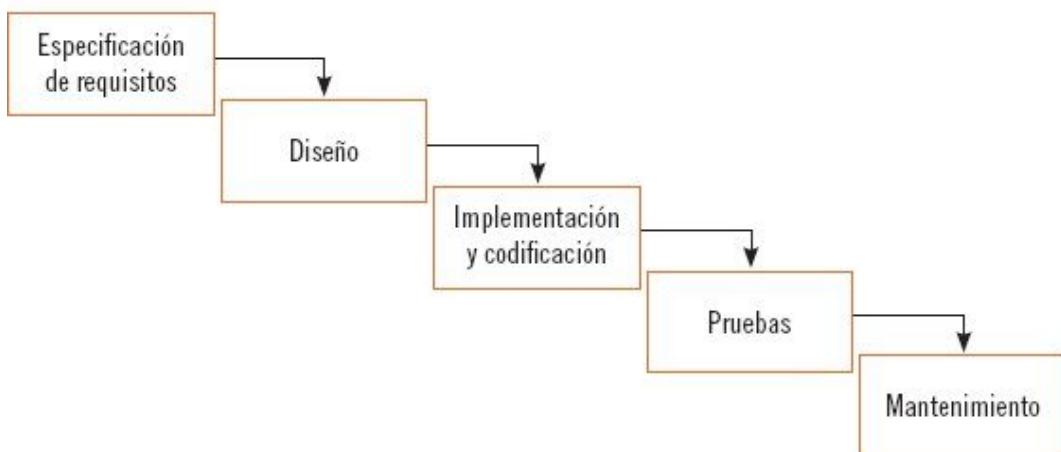
Sabías que...

La técnica “codificar y corregir” consiste en, teniendo una idea general de lo que hay que desarrollar, realizar una combinación de diseño, codificación y prueba hasta llegar a un producto satisfactorio. Se corren muchísimos riesgos, ya que, aunque el producto esté bien realizado, puede terminar siendo rechazado por no haber entendido correctamente los requisitos.

En cascada (waterfall)

Fue descrito formalmente en 1970 por Winston Royce, siendo también conocido como modelo secuencial. Fue el primer modelo definido y sirvió como base para los demás. Hoy todavía es utilizado.

Modelo en cascada clásico



El modelo clásico sigue la secuencia indicada en la imagen, implementando como fases las actividades fundamentales del desarrollo de un *software*. Estas fases son las siguientes:

1. **Especificación de requisitos:** objetivo buscado, necesidades a cubrir o problema a solucionar mediante el software.
2. **Diseño:** se establece cómo va a ser construido el programa, dividiéndolo si es necesario y fijando cómo se va a representar la información con la que se trabajará.
3. **Implementación y codificación:** trasladar el diseño a algo entendible por la máquina mediante un lenguaje de programación.
4. **Pruebas:** realización de pruebas para ver si el producto está libre de errores y cumple con los requisitos.
5. **Mantenimiento:** realización de posibles mejoras en un producto ya funcional en manos del cliente/usuario.

A la vista, puede parecer un modelo estrictamente lineal, pero se pueden realizar las iteraciones necesarias sobre la etapa actual antes de pasar a la siguiente fase.

Las ventajas son las siguientes:

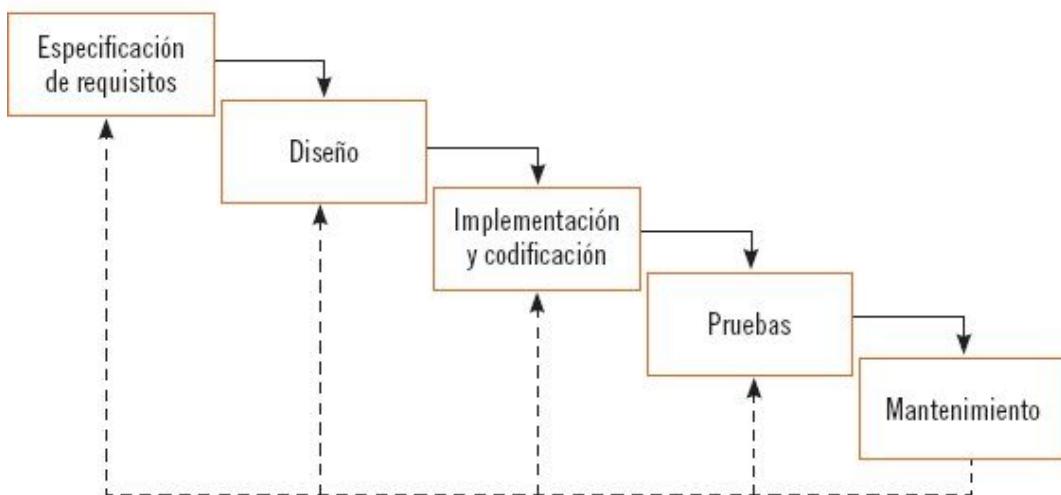
- Planificación sencilla.
- Gran calidad del producto final.
- Todavía es válido para pequeños/medianos desarrollos.

Y los inconvenientes:

- Los requerimientos son necesarios al comienzo del proyecto.
- Los errores son altamente penalizados.
- No se ven resultados hasta una fase avanzada del ciclo.

Una variante más práctica recibe el nombre de modelo en cascada realimentado. Esta versión introduce el concepto de realimentación, proporcionando una mejor adaptabilidad a proyectos en los que hay cierta incertidumbre o se prevén ajustes durante su desarrollo, obligando a volver a etapas previas.

Modelo en cascada realimentado



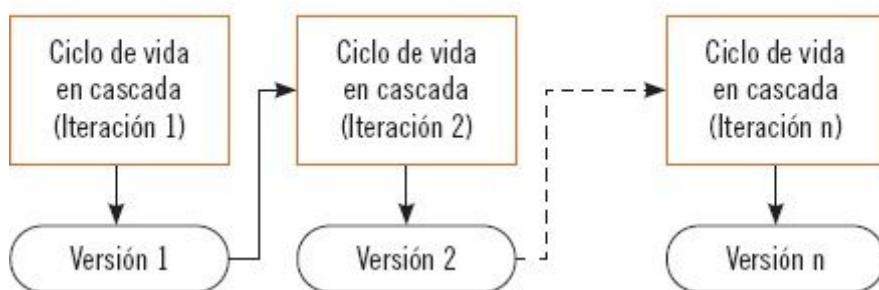
Realimentación (o retroalimentación)

Mecanismo mediante el cual la salida de un sistema sirve como entrada al mismo, regulando su funcionamiento con la nueva información facilitada.

Iterativo

El modelo iterativo consiste en una serie de repeticiones del modelo en cascada previamente comentado. En cada una de las iteraciones se genera una versión del *software*, la cual será evaluada y servirá como punto de partida para la siguiente iteración. Cada versión contendrá las mejoras que se consideren, terminando el proceso cuando el producto sea satisfactorio.

Modelo iterativo



Las ventajas son las siguientes:

- Se ofrecen versiones intermedias, evaluables por el cliente.
- No requiere una alta especificación de requisitos.

Y las desventajas:

- Las numerosas versiones pueden encarecer el desarrollo.
- Si el cliente se involucra excesivamente puede cambiar su idea, modificando drásticamente

un desarrollo avanzado.

- Es difícil establecer un tiempo total de desarrollo.

Incremental

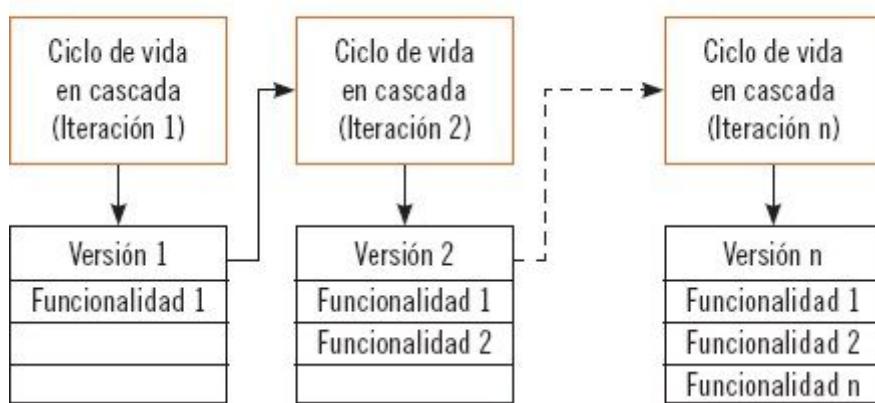
Es muy parecido al modelo iterativo. Se sigue generando una versión en cada iteración, que servirá de entrada para la siguiente una vez realizada la correspondiente evaluación. La diferencia radica en que cada nueva versión conlleva una nueva funcionalidad en forma de módulo.



Módulo

Parte del software que se encarga de una función específica del mismo. Interactúa con otros módulos mediante unas entradas y salidas claramente definidas.

Modelo incremental



Las ventajas son las siguientes:

- Entregas rápidas con funcionalidad creciente.

Y las desventajas:

- Es difícil establecer el coste y tiempo finales de desarrollo.
- No es válido para cualquier *software* (*software* con alta funcionalidad inicial, trabajo en sistemas distribuidos, trabajo en tiempo real, altos requerimientos de seguridad, etc.).

En V

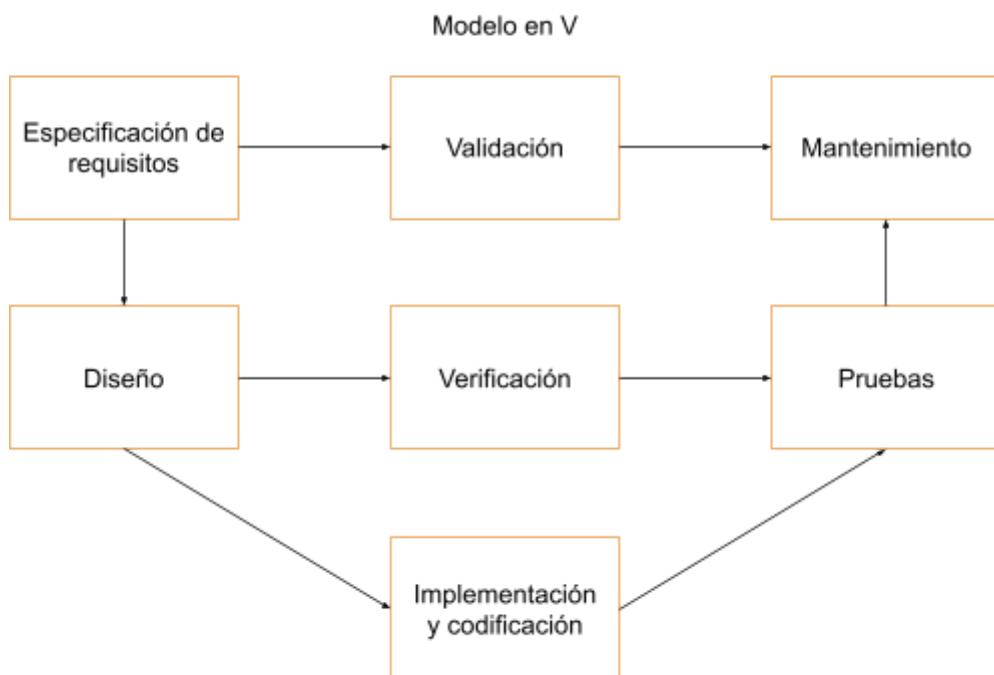
Para cada fase del desarrollo, existe una fase correspondiente o paralela de verificación o validación. Esta estructura obedece al principio de que para cada fase del desarrollo debe existir un resultado verificable.

Las ventajas son las siguientes:

- Las nuevas etapas facilitan la depuración y control de fallos.

Y las desventajas:

- El cliente no recibirá el producto hasta avanzado el desarrollo.
- Las pruebas pueden disparar el coste del producto.

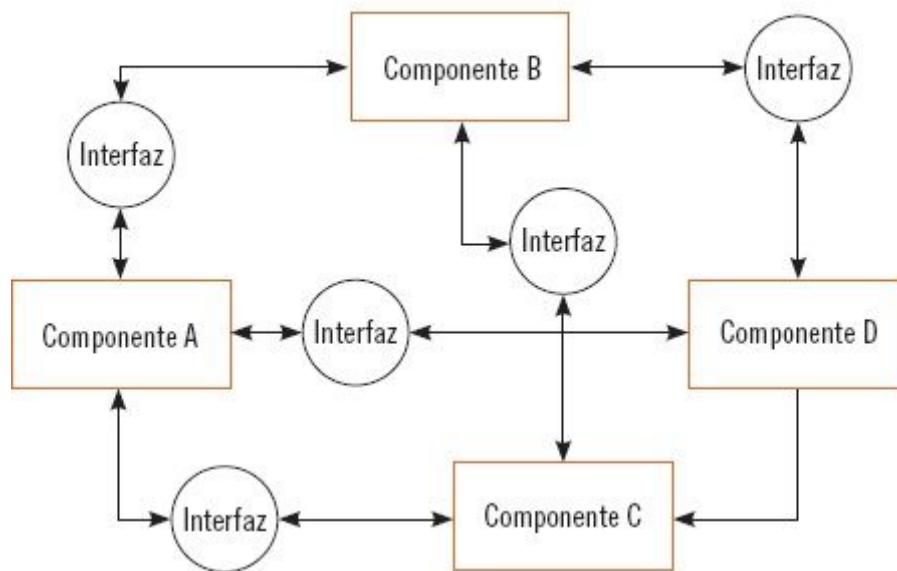


Basado en componentes (CBSE)

Este modelo es una visión diferente. Se construye un nuevo *software* con la ayuda de productos prefabricados. Estos productos reciben el nombre de componentes y generalmente son comerciales.

Cada componente dispone de una funcionalidad y se comunica a través de una interfaz con entradas y salidas claramente definidas. Este modelo incorpora etapas propias, ya que hay que analizar cuidadosamente la aplicación y estudiar los posibles componentes a incorporar, aparte de diseñar una arquitectura que permita la integración de los mismos.

Software basado en componentes



Las ventajas son las siguientes:

- Reutilización de *software*.
- Se puede reducir el tiempo de desarrollo.

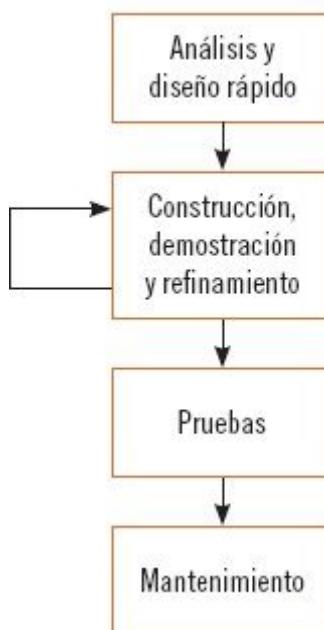
Y los inconvenientes:

- El desarrollo depende de las limitaciones de los componentes.
- Los componentes pueden ser caros.
- El componente, al ser externo, puede no ser actualizado durante la vida del *software*.
- Hay que realizar pruebas exhaustivas.

Desarrollo rápido (RAD)

Un ciclo de desarrollo rápido prácticamente fusiona las fases de análisis y diseño, requiriendo la colaboración absoluta del cliente. Este recibirá muchos prototipos o partes funcionales de sus *software* en cortos espacios de tiempo, evaluando el producto para dar lugar a una retroalimentación en el equipo de desarrollo. De ahí que surja el concepto de **adaptación incremental**, puesto que se espera que los requisitos cambien (y lo harán, sin duda).

Modelo RAD



Las ventajas son las siguientes:

- Resultados rápidamente visibles.
- Permite la reusabilidad de código.
- Participación del usuario.

Y los inconvenientes:

- Exige bastante disciplina y compromiso por todas las partes.
- Gran coste en herramientas y entornos de desarrollo.
- Si el proyecto es grande, son necesarios varios equipos de trabajo.

Ventajas e inconvenientes. Pautas para la selección de la metodología más adecuada

En la elección del ciclo de vida adecuado, hay que considerar cinco factores básicos:

- Tiempo hasta la entrega final.
- Complejidad del problema.
- Necesidad (o no) de entregas parciales.
- Definición y exactitud de los requerimientos.
- Recursos disponibles.

La valoración conjunta de estos cinco factores permite establecer las siguientes preferencias:

- **Modelo de cascada:** desarrollos no excesivamente complejos, sin entregas parciales y requisitos claramente definidos.

- **Modelo iterativo:** desarrollos no excesivamente complejos, sin entregas parciales y requisitos claramente definidos.
- **Modelo incremental:** recomendable para *software* que no requiera toda su funcionalidad de manera inicial y cuando se precisen entregas rápidas.
- **Modelo en V:** desarrollos que requieran gran robustez y confiabilidad. Por ejemplo, *software* que precise de operaciones constantes sobre una base de datos.
- **Modelo basado en componentes (CBSE):** aplicable a desarrollos que han contemplado en su diseño la incorporación de componentes comerciales, siempre que se pueda afrontar su coste.
- **Desarrollo rápido de aplicaciones (RAD):** desarrollos rápidos, siempre que se disponga de suficiente equipo para hacer frente y cumplir los plazos.

Análisis y especificación de requisitos

Según el IEEE (Instituto de Ingenieros Eléctricos y Electrónicos), un requisito encaja en las siguientes definiciones:

- Una condición o necesidad de un usuario para resolver un problema o alcanzar un objetivo.
- Una condición que debe exhibir o poseer un sistema para satisfacer un contrato, estándar, especificación u otra documentación formalmente impuesta.
- Una representación documentada de una condición o capacidad documental como las anteriormente descritas.

Tipos de requisitos

La primera clasificación de requisitos es fijada por el **nivel de especificación** (nivel de detalle) de los mismos:

- **Requisitos de usuario:** descripción en lenguaje natural (o mediante diagramas simples) de servicios y funcionalidades que se esperan del *software*. Por ejemplo: el sistema puede guardar imágenes de archivos.
- **Requisitos de sistema:** especificación completa de los anteriores, que sirve como contrato entre el cliente y el desarrollador. Tomando el ejemplo anterior se puede plantear:
 - El usuario elige la imagen a guardar.
 - El usuario puede elegir el formato de imagen.
 - El usuario puede indicar el nombre del nuevo archivo.
 - El usuario debe confirmar los datos anteriores para el guardado definitivo de la imagen.

Los requisitos del sistema, a su vez, se pueden dividir de la siguiente manera según la **naturaleza del requisito**:

- **Requerimientos funcionales:** descripción de la funcionalidad, del comportamiento del sistema y de su interacción con el entorno. En la medida de lo posible, hay que ceñirse a lo que el sistema debe hacer (o qué no debe hacer). El cómo queda para fases posteriores.
Ejemplos:

- El acceso al sistema requiere darse de alta.
 - Se permite un máximo de dos libros simultáneos en préstamo por cada usuario.
 - El usuario puede consultar su historial de préstamos.
 - Un retraso en la devolución de un libro de más de una semana implica la inhabilitación automática de la cuenta del usuario.
- **Requisitos no funcionales:** restricciones que afectan al sistema (estándares, rendimiento, accesibilidad, interfaz, seguridad, portabilidad, etc.). Ejemplos:
 - Compatible con *Chrome, Firefox y Safari*.
 - La base de datos debe estar implementada con MySQL.
 - Debe cumplir con lo dispuesto en la Ley Orgánica de Protección de Datos.
 - El sistema será accesible a través de *smartphones* y dispositivos móviles, con un interfaz especialmente diseñado para ello.

Modelos para el análisis de requisitos

Una vez los requisitos han sido definidos, se procede a un modelado de los mismos con dos objetivos: delimitar el alcance del sistema y capturar su funcionalidad.

Existen muchos modelos que ayudan al análisis de requisitos. Entre los más importantes se encuentran los diagramas de flujo y los casos de uso.

Diagrama de flujo

Este diagrama representa el flujo de la información desde que entra en un sistema hasta que sale, indicando las transformaciones (burbujas) que sufre dicha información al moverse dentro del sistema. Consta de cuatro componentes básicos:

- **Procesos:** componente que transforma la información. Se representa por un círculo.
- **Flujo de datos:** indica comunicación entre componentes. Se representa por una flecha.
- **Almacenes de datos:** información del sistema. Se representan por dos líneas horizontales paralelas.
- **Entidades externas:** receptores o generadores de información. Se representan por un cuadrado.

Por otra parte, un diagrama de flujo tiene varios niveles, según el grado de detalle del que haga gala:

- **Nivel 0 (diagrama de contexto):** se representa el sistema como un único proceso y las interacciones con el resto de entidades (usando una flecha indicando el sentido de la interacción). Por ejemplo: una salida por pantalla es una interacción en un sentido, mientras que un proceso de lectura/escritura conlleva dos sentidos.
- **Nivel 1 (diagrama de nivel superior):** se indican los procesos que describen el proceso principal, siendo este desglosado en subprocesos. Los subprocesos pueden venir determinados por los requisitos o ser el resultado de la división lógica del proceso padre. En las interacciones, ahora se muestra información complementaria (el dato que se pasa, la función que activa la interacción, etc.)

- **Nivel 2 (diagrama de detalle o expansión):** se aumenta el nivel de detalle indicando excepciones y flujos entre procesos. Básicamente, es un nivel de refinamiento superior considerando todas las posibilidades y alternativas en el flujo.

Actividad 1

En una entrevista de trabajo le proponen realizar un diagrama de flujo de nivel 0 para un sistema de gestión de nóminas. Se le informa que las entidades externas son Impresora, Usuario y Pantalla, mientras que existe un almacén externo que es Base de Datos.

Realizar también el diagrama de flujo de nivel 1 sabiendo que el sistema presenta las siguientes opciones:

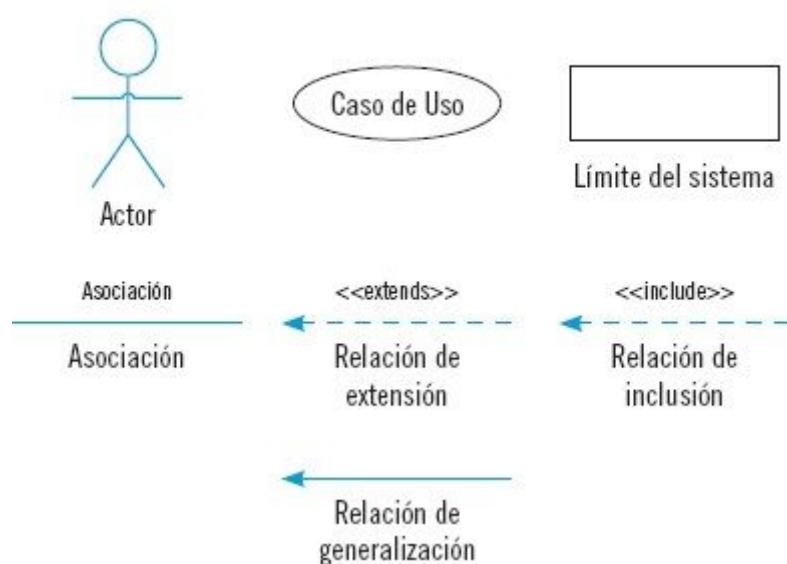
- Imprimir (accede a la entidad base de datos).
- Mostrar por pantalla (accede a la base de datos y muestra información por pantalla) .
- Realizar altas y bajas (accede/modifica a la base de datos).

Diagramas de casos de uso

Un modelo de un caso de uso se puede representar de dos maneras diferentes, según el nivel de detalle que se busque.

Los diagramas de casos de uso muestran la relación entre los casos de uso y los actores, representando una funcionalidad básica del sistema. Los componentes que pueden aparecer son los siguientes:

Componentes de un diagrama de casos de uso



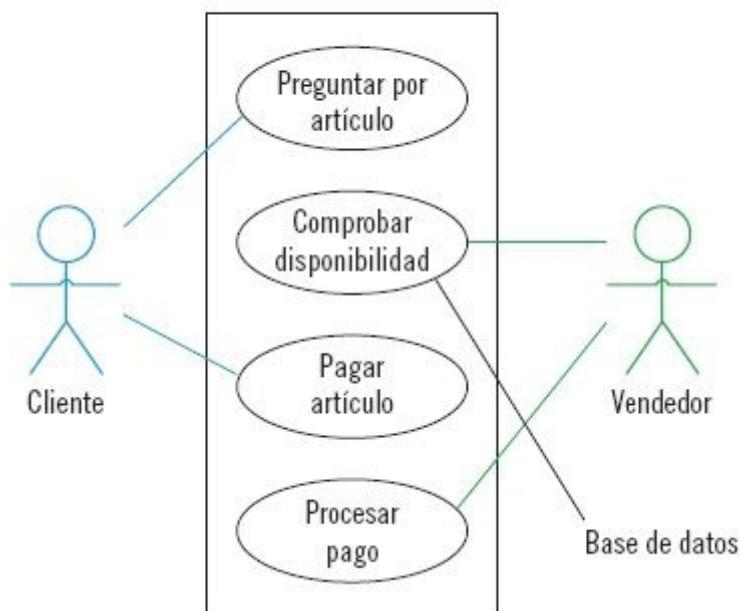
Actor

Entidad externa o interna que interactúa con el sistema, ya sea haciendo uso de una funcionalidad (usuario) o actuando de supervisor o apoyo (base de datos externa).

El proceso para realizar correctamente un diagrama de caso de uso de puede concretar en estos tres puntos:

1. Identificar actor/es principal/es.
2. Identificar el objetivo de cada actor respecto al sistema.
3. Presentar/describir cómo interactúa el actor con el sistema.

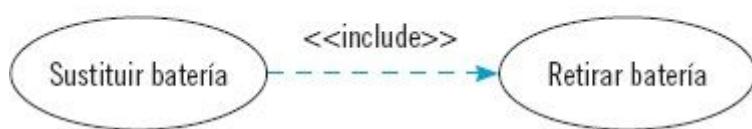
Ejemplo de diagrama de casos de uso



En un mismo diagrama pueden estar presentes varios casos de uso. En el supuesto de dos casos que estén ligados entre sí, se puede establecer una relación de dependencia entre ambos. Se permiten 3 tipos de relación:

- **<<include>>**: un caso de uso está incluido dentro de un caso base, de tal manera que la realización de uno implique necesariamente la realización del otro. Por ejemplo: “Sustituir batería” incluye “Retirar batería antigua” o, lo que es lo mismo, “Retirar batería antigua” está incluido en “Sustituir batería”.

Ejemplo de <<include>>



- **<<extend>>**: un caso de uso puede extender de un caso base bajo ciertas circunstancias.

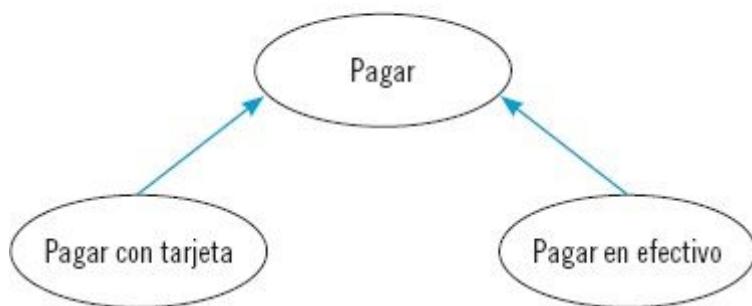
Mientras que <<include>> implicaba la ejecución del caso de uso incluido, ahora esa ejecución depende de una condición. Por ejemplo: el caso de uso “Denegar tarjeta” extenderá de “Pagar con tarjeta” cuando se haya producido un error al efectuar el pago.

Ejemplo de <<extend>>



- **Generalización:** un caso de uso es una especialización de un caso base (o un caso base en una generalización de un determinado caso de uso). Es parecido a la idea de orientación a objetos, que se tratará en capítulos posteriores. Como ejemplo se pueden exponer los casos de uso “Pagar con tarjeta” y “Pagar en efectivo”, cuya generalización es el caso “Pagar”.

Ejemplo de generalización



Recuerda

En el <<include>> hay un comportamiento común, siendo un caso de uso consecuencia o implicación de un caso base. En el <<extend>>, una circunstancia o condición puede implicar que se dispare (o no) el caso de uso que extiende del caso base.

Especificación de casos de uso

Es una descripción en lenguaje fácilmente comprensible, proporcionando información más detallada. Una correcta especificación contendrá:

- Nombre del caso de uso e Identificador.
- Actores.
- Tipo (primario, secundario u opcional).
- Referencias: requisitos que se pueden incluir.
- Precondición: condiciones para que tenga lugar el caso.

- Postcondición: efectos inmediatos sobre el sistema.
- Autor, fecha y versión.
- Propósito: descripción general del caso de uso.
- Flujo normal: curso normal del caso de uso.
- Flujos alternos: cursos alternativos (excepciones).

Especificación de Caso de Uso

Caso de Uso	Compra de Artículo				CU-1
Actores	Cliente (iniciador), Vendedor, Base de Datos del Sistema				
Tipo	Primario				
Referencias	-----				
Precondición	-----				
Postcondición	La venta queda registrada en la Base de Datos del Sistema				
Autor	RGL	Fecha	07/09/2013	Versión	1

Propósito

Registrar la venta de un artículo

Resumen

El Cliente llega a la tienda. Pregunta por un artículo. El Vendedor consulta disponibilidad y le trae el artículo. El Cliente paga, el Vendedor registra la Venta y el Cliente se marcha con su producto.

Flujo normal

1	El Cliente llega al mostrador
2	El Cliente solicita un artículo al Vendedor.
3	El Vendedor se identifica en el Sistema.
4	El Vendedor comprueba disponibilidad.
5	El Vendedor va al almacén por el artículo solicitado.
6	El Vendedor informa del precio al Cliente.
7	El Cliente paga (en efectivo o tarjeta).
8	El Vendedor marca la venta del artículo en el Sistema.
9	El Vendedor genera el ticket.
10	El Cliente toma el ticket y su artículo.

Flujos alternos

4a	El artículo no tiene disponibilidad actualmente. Sugerir su encargo y volver al punto 1.
6a	El Cliente considera que el artículo es muy caro. Cancelar.
7a	El Cliente paga con tarjeta y hay un error durante el proceso. Sugerir pagar en efectivo o Cancelar.

Documentación de requisitos

Los requisitos se deben recoger en el documento ERS (Especificación de Requisitos Software), sin mencionar ningún detalle de diseño. Según el estándar IEEE 830, un documento ERS debe tener las siguientes características:

- Utilizar términos únicos (en caso de que no sea posible indicarlos en un glosario).
- Incluir todos los requisitos significativos.
- Requisitos fáciles de verificar por un procedimiento existente.
- Los requisitos no deben entrar en conflicto.
- Fácil de modificar.
- Facilidad de referencia con otros elementos del ciclo de vida.
- Facilidad de utilización en Explotación y Mantenimiento.

Dicho estándar establece también la estructura recomendada para el documento de ERS, que se puede sintetizar en los siguientes puntos:

1. Introducción:

- a. Objetivo: propósito y audiencia del documento.
- b. Ámbito: qué hace el software.
- c. Definiciones, siglas y abreviaturas.
- d. Referencias a otras partes del documento.
- e. Visión global: organización del resto del documento.

2. Descripción general:

- a. Perspectiva y funciones del producto.
- b. Características del usuario.
- c. Restricciones generales.
- d. Dependencias: factores que afectan a los requisitos.

3. Requisitos específicos: definición de cada requisito, indicando:

- a. Identificador único.
- b. Descripción de la entrada, del proceso y de la salida.
- c. Descripción de las situaciones de error.

4. Apéndices

5. Índice

Validación de requisitos

Los requisitos deben ser validados para asegurar que el documento de ERS define el *software* de manera adecuada. Las principales técnicas de validación son:

- **Revisiones:** revisión de la documentación o del modelado de requisitos por parte de un grupo de personas (con participación de representantes del cliente y de usuarios). Se puede utilizar una lista de comprobación definida al comienzo del proceso con el fin de acotar el ámbito de la revisión.
- **Prototipos:** en ocasiones, durante la definición de requisitos, se opta por desarrollar un prototipo del *software* que no dispone de la funcionalidad completa. Así, el usuario/cliente puede tener una perspectiva del sistema.
- **Pruebas de aceptación:** pruebas previas al paso de producción. Se recomienda que para estas pruebas no formen parte las personas encargadas del desarrollo, con el fin de que sean lo más objetivas posibles y no haya conflicto de intereses.

Gestión de requisitos

Los requisitos pueden cambiar en pleno proceso de desarrollo por diversos factores (cambios de estándar, modificación de la estrategia de negocio, etc.). Estas modificaciones suelen implicar cambios en las relaciones entre requisitos y en las dependencias entre el documento de ERS y otros documentos. El uso de herramientas CASE es muy útil durante este proceso, ya que facilita la generación de documentación y el seguimiento de las modificaciones.



Sabías que...

Las herramientas CASE (Computer Aided Software Engineering) son aplicaciones que ayudan durante el proceso de desarrollo del software, reduciendo el coste de dicho desarrollo y aumentando la productividad.

El proceso de gestión de cambio consta de los siguientes puntos:

1. Comprobar la validez del cambio.
2. Crear una lista de requisitos afectados por la modificación.
3. Proponer cambios de requisitos de la lista anterior.
4. Valorar los costes de los cambios.
5. Valorar la aceptabilidad de los cambios.
6. Implementar los cambios.

Los puntos 3 y 5 requieren de la participación del cliente. Este debe dar el visto bueno a los cambios propuestos y, posteriormente, aceptar el coste de dicha modificación.

Diseño

Una vez definidos los requisitos del sistema, se procede a realizar un modelado del sistema. Aquí se definirán, entre otros aspectos, los componentes que darán cuerpo al sistema, la relación entre los mismos y el interfaz de usuario.

Modelos para el diseño de sistemas

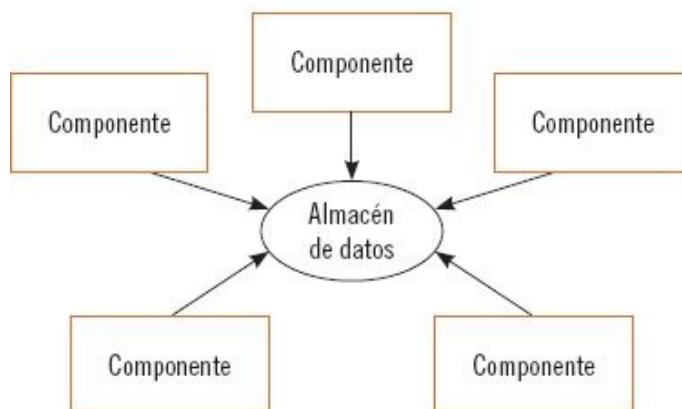
Según Pressman, el objetivo del diseño es realizar un modelo de la entidad que se va a construir. Para llevar a cabo este objetivo, divide el modelo de diseño en otros tres modelos: modelo de arquitectura, de componentes y de interfaz.

Modelo de arquitectura

El modelo de arquitectura define la estructura de los componentes del sistema, sus propiedades externas visibles y las relaciones entre ellos. Entre las arquitecturas clásicas, se encuentran las siguientes:

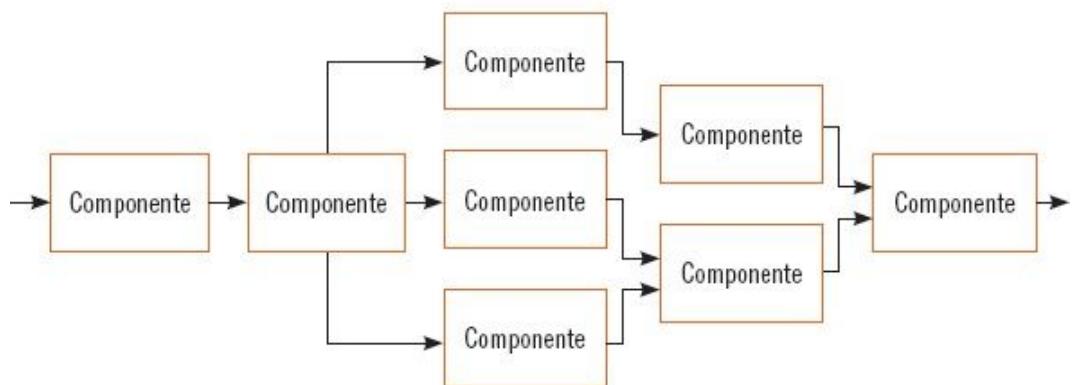
- **Centrada en datos:** existe un almacén central de datos al cual acceden el resto de componentes del sistema.

Arquitectura centrada en datos



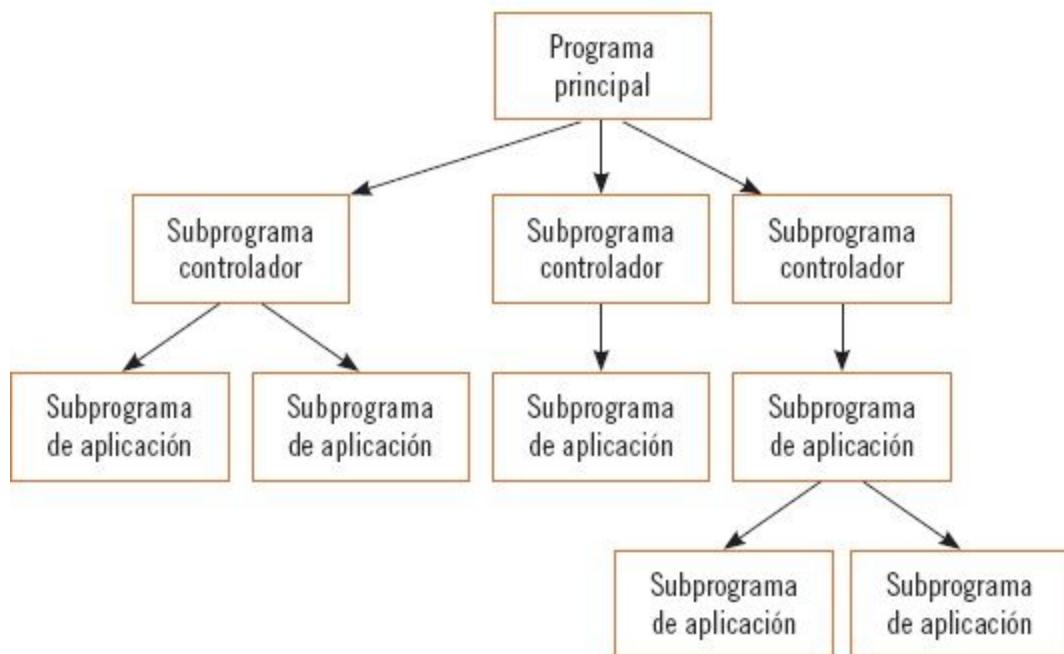
- **De flujo de datos:** los datos se transforman a través de los componentes. Si solo hay un flujo se llama lote secuencial.

Arquitectura de flujo de datos



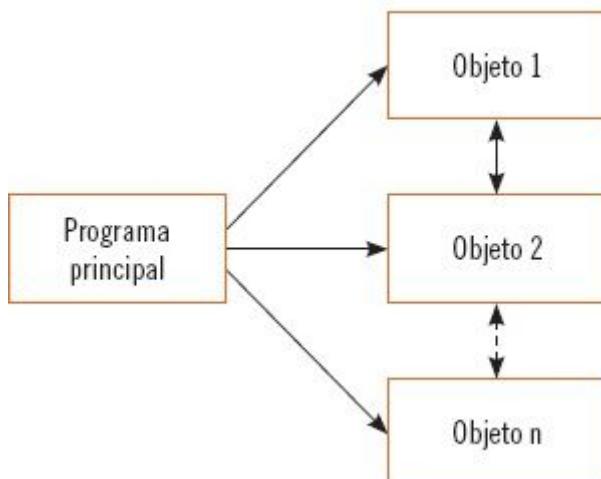
- **De programa principal/subprograma:** un programa principal invoca a otros componentes y estos, a su vez, invocan a otros componentes diferentes. La arquitectura, si presenta una distribución remota, recibe la denominación de llamada de procedimiento remoto.

Arquitectura de programa principal/subprograma



- **Orientada a objetos:** un programa principal se apoya en objetos para llevar a cabo su propósito. Los objetos encapsulan su funcionalidad, comunicándose a través de una interfaz bien definida.

Arquitectura orientada a objetos



Modelo de componentes

Un componente es definido como una parte modular, desplegable y sustituible de un sistema, que incluye la implantación y expone un conjunto de interfaces. El componente tiene matices en su concepción, dependiendo de la orientación dada al software:

- Visión orientada a objetos.
- Visión tradicional.

Visión orientada a objetos

Los componentes implementan el funcionamiento de una entidad llamada **Clase**, la cual consta de unos atributos (para almacenar información) y de unos métodos (que proporcionan funcionalidad). También existen unos métodos especiales que les permiten relacionarse con el exterior. Estas son las llamadas interfaces.

Una clase se representa como un rectángulo dividido en tres partes. La primera es el nombre de la clase, la segunda los atributos y la tercera los métodos. El componente, por otra parte, adopta la forma de un rectángulo, acompañado con otros dos pequeños superpuestos a la izquierda. Las interfaces se identifican con un pequeño círculo unido al componente.

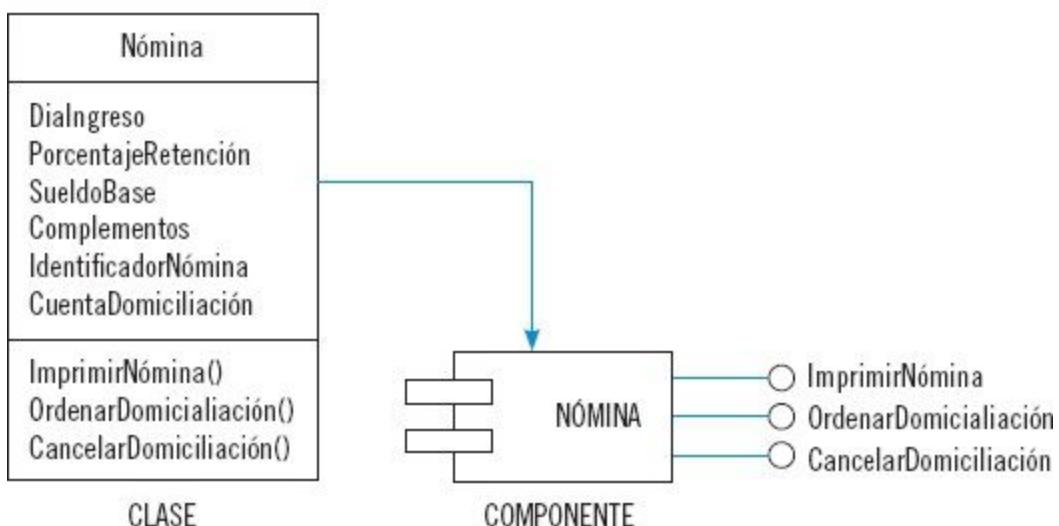
Ejemplo

Crear el componente para una clase Nómina con la siguiente información:

- Atributos:
 - DíaIngreso.

- PorcentajeRetención.
- SueldoBase.
- Complementos.
- IdentificadorNómina.
- CuentaDomiciliación.
- Interfaces:
 - OrdenarDomiciliación.
 - CancelarDomiciliación
 - ImprimirNómina.

Componente orientado a objetos



Orientación a objetos

La orientación a objetos es un enfoque que se fundamenta en la representación de los programas (o cualquiera de sus partes) en forma de entidades, cada una de ellas con un estado definido fruto de sus propiedades y con un comportamiento/funcionalidad derivado de los métodos que implementa.

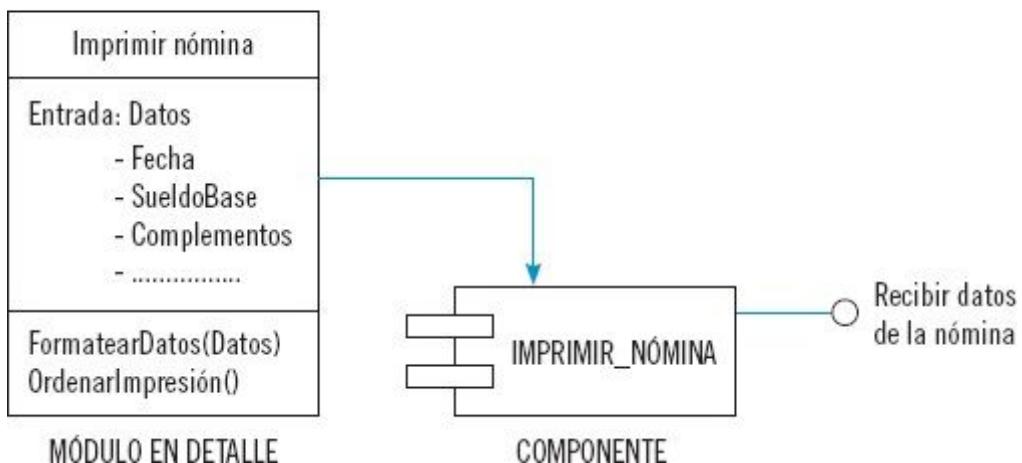
Visión tradicional

El componente es un elemento funcional que incorpora la lógica y las interfaces de entrada/salida de datos. La diferencia respecto a la visión anterior es que aquí se implementa una funcionalidad, mientras que un componente orientado a objetos puede implementar todas las funcionalidades definidas en su clase.

Ejemplo

Componente para la funcionalidad ImprimirNómina. Los datos requeridos, es decir, la información relativa a la nómina, se reciben a través de una interfaz.

Componente tradicional



📎 Recuerda

Un componente de la visión tradicional implementa una funcionalidad. Un componente de la visión orientada a objetos implementa todas las funcionalidades definidas en su clase

Modelo de interfaz

La interfaz define la forma en la que el usuario interactúa con el *software*, tanto proporcionando información como recibiendo resultados. Una mala interfaz (confusa, difícil de aprender) producirá una experiencia frustrante que posiblemente arruine el producto final. Para evitar esto, Theo Mandel definió las tres reglas doradas del diseño de la interfaz:

- Dejar el control al usuario: sin transiciones innecesarias, posibilidad de interrupción, de vuelta atrás, de personalización, etc.
- Reducir la carga de memoria del usuario: atajos intuitivos, organización jerárquica de la interfaz, etc.
- Hacer que la interfaz sea consistente: presentación y obtención de la información según unas reglas de diseño, manteniendo la coherencia entre todas las pantallas. La entrada de datos se debe dar en un contexto identificable, de forma que el usuario siempre sepa dónde está, cómo puede volver y qué debe hacer para continuar.

Para diseñar una interfaz, se puede partir de los casos de uso. El proceso de diseño implica:

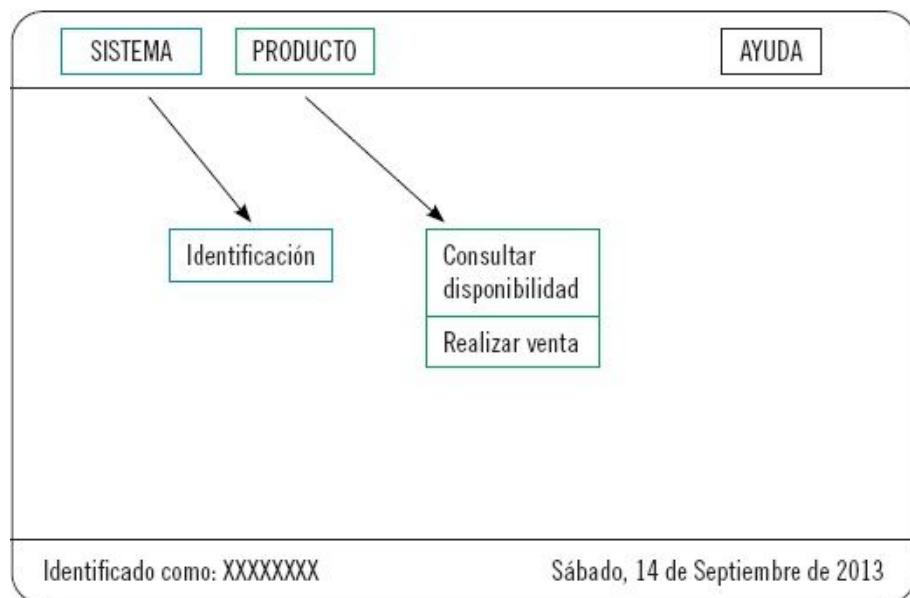
1. Identificar al usuario que interactúa con el sistema.
2. Identificar sus acciones y los objetos que intervienen en dichas acciones.
3. Convertir los objetos en opciones visuales, definiendo, a su vez, las acciones dentro de dichas opciones.

Ejemplo

Definición de interfaz para la especificación de caso de uso usada como ejemplo anteriormente. El caso de uso incluía la siguiente secuencia de acciones:

- El cliente llega al mostrador.
- El cliente solicita un artículo al vendedor.
- El vendedor se identifica en el sistema.
- El vendedor comprueba disponibilidad.
- El vendedor va al almacén a por el artículo.
- El vendedor informa del precio al cliente.
- El cliente paga (en efectivo o con tarjeta).
- El vendedor marca la venta y genera el ticket.
- El cliente toma el ticket y su artículo.

Ejemplo de diseño de interfaz



Diagramas de diseño. El estándar UML

El UML (Lenguaje de Modelado Unificado) se define como un lenguaje estándar para escribir diseños de *software*, creado para ayudar a diseñadores y desarrolladores en el manejo de un lenguaje común. La versión 2.0 proporciona trece diagramas, de los cuales se analizarán los que se describen a continuación.

Diagramas de casos de uso

Los diagramas de casos de uso ya han sido tratados. Su misión es ayudar a determinar la funcionalidad del sistema desde la perspectiva del usuario. Existen dos visiones de los casos de uso:

- Diagramas de casos de uso.

- Especificación de casos de uso (presentan la información de una forma más completa y detallada).

Además, se establecieron tres tipos de relaciones entre casos:

- <<include>>: un caso incluye a otro.
- <<extend>>: un caso extiende de otro.
- Generalización: un caso es la generalización de otro.

Diagramas de implementación

Muestran la distribución física de un sistema de *software* cuando intervienen diferentes componentes de *hardware*. Cada uno de los componentes utilizados contendrá información del mismo, mientras que los flujos de unión pueden especificar la tecnología usada en la comunicación de los componentes.

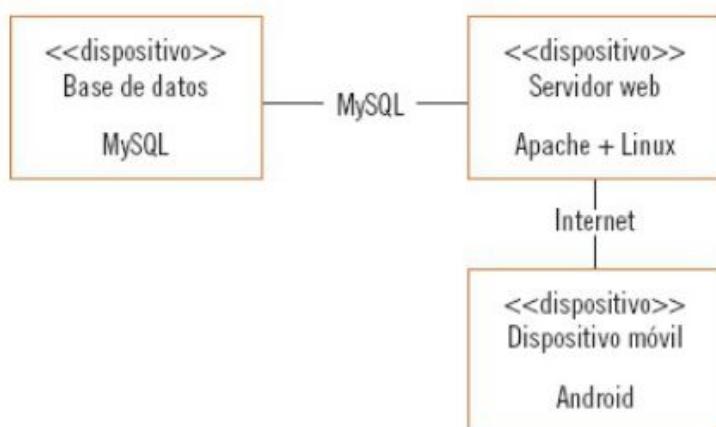
Nota

UML es uno de los “grandes olvidados” durante ciertos desarrollos, prefiriéndose pasar directamente a la programación antes de plantear de manera formal lo que se va a desarrollar. Aunque puede parecer un engorro, el correcto uso de los diagramas UML ahorra a la larga muchísimo tiempo y, sobre todo, evita sorpresas desagradables.

Ejemplo

Representar un diagrama de implementación para un sistema basado en un dispositivo móvil que accede a un servidor y que se nutra de una base de datos externa. Al servidor se accede por *http* y la base de datos se comunica usando MySQL.

Diagrama de implementación



Diagramas de actividad

Un diagrama de actividad muestra el comportamiento dinámico de una actividad que se desea modelar. Es parecido al diagrama de flujo, pero presenta las siguientes diferencias:

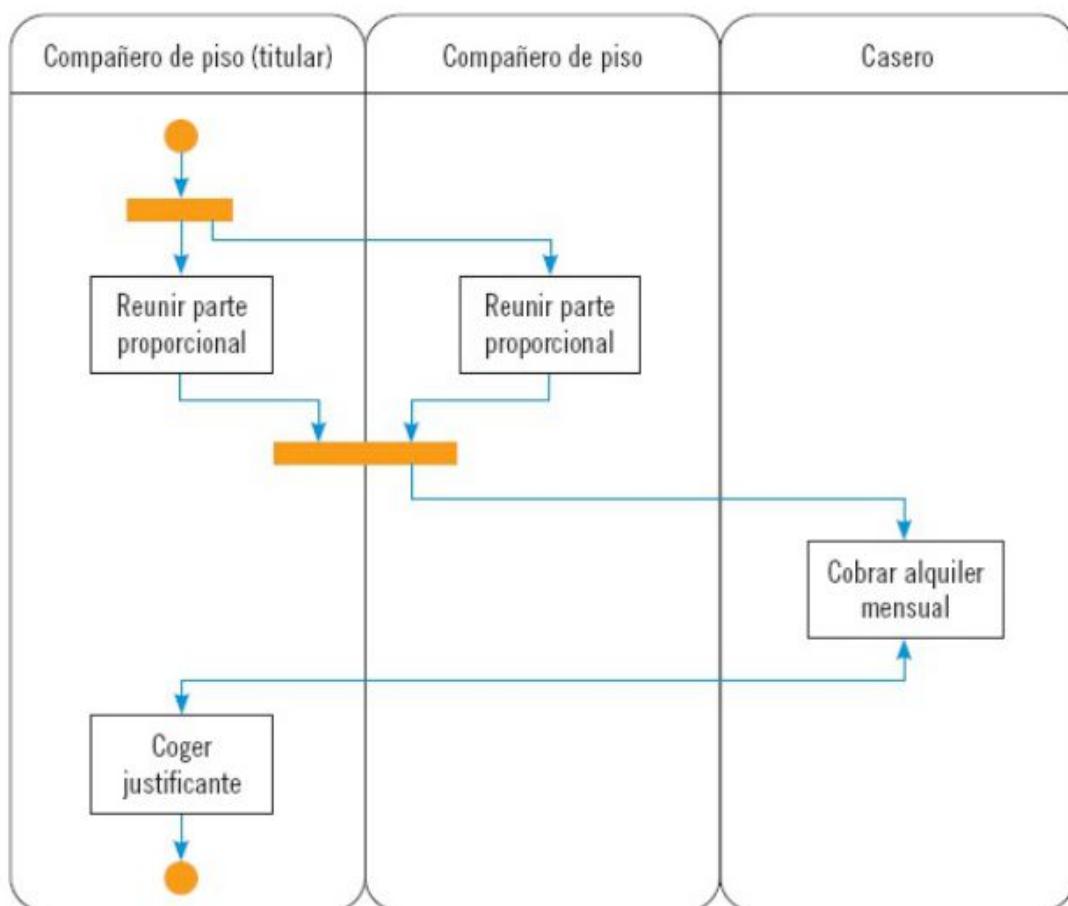
- Se pueden establecer canales para cada participante.
- Se pueden fijar uniones para sincronizar flujos. El flujo de salida no tendrá lugar hasta que se cumplan los correspondientes flujos entrantes.
- El comienzo y final se representan con un punto.

Ejemplo

Modelar una actividad referida a pagar la mensualidad del alquiler del inmueble.

Hay dos compañeros de piso, siendo uno el titular que recibirá el justificante de ingreso.

Diagrama de actividad

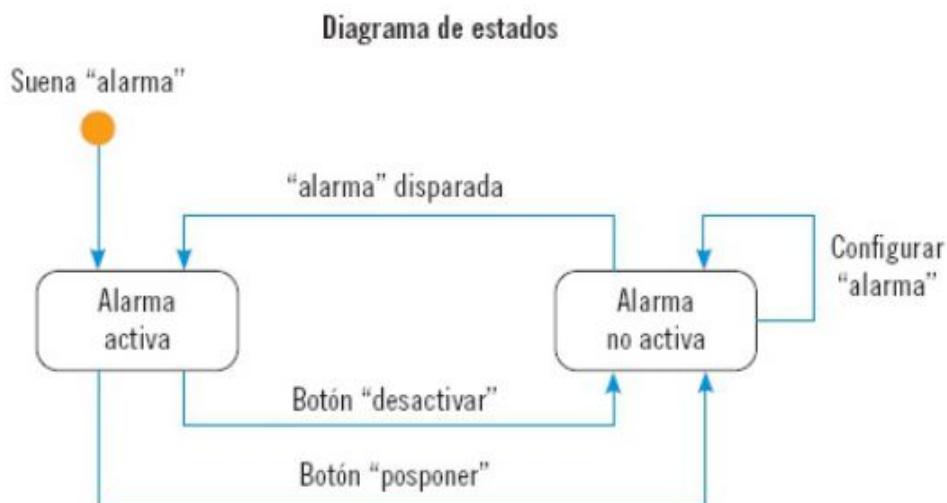


Diagramas de estado

Los diagramas de estado modelan los estados de un objeto, indicando cuáles son las acciones que provocan las transiciones entre los mismo y cuál es el evento disparador que detona el estado inicial.

Ejemplo

Diagrama de estados para una alarma. Se considerarán dos estados: “activa” y “no activa”, siendo el evento disparador la propia alarma.

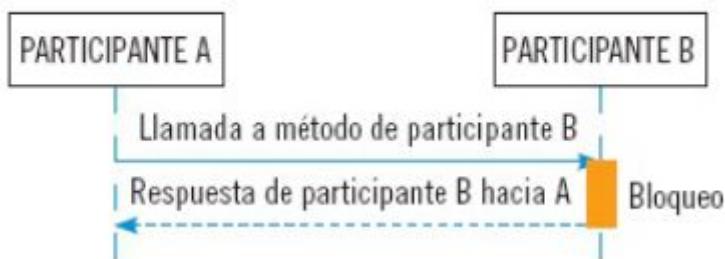


Diagramas de secuencia

Los diagramas de secuencia muestran la comunicación de los objetos durante la ejecución de una tarea. Son de especial aplicación para modelar las interacciones representadas en un caso de uso, creando una especie de línea temporal sobre el mismo. Durante el proceso de representación de un diagrama de secuencia, hay que tener en cuenta los siguientes aspectos:

1. Se parte de una secuencia de acciones en la cual se da una comunicación entre varios objetos (participantes). Estas acciones habrán sido descritas en el caso de uso.
2. Los participantes se representan en una línea vertical continua (un concepto parecido a la idea de canal del diagrama de actividad), siendo las líneas de comunicación flechas horizontales.
3. Los objetos/participantes interactúan entre ellos por medio de llamadas a métodos y mensajes de respuesta. Ambas interacciones toman la forma de flechas, indicando el sentido de la comunicación. Las primeras serán continuas y las segundas discontinuas.
4. Normalmente, entre un objeto B recibe una llamada de un objeto A y emite una respuesta, se da una situación de bloqueo para que dicho objeto B lleve a cabo la funcionalidad implementada en el método. Una vez que el objeto B se desbloquea, puede dar pie a una nueva llamada procedente del objeto A.
5. El tiempo que permanece un objeto bloqueado es escenificado por un rectángulo alargado (a más altura, se entiende que el tramo de tiempo es mayor).

Representación básica de un diagrama de secuencia



Actividad 2

Modele un diagrama de secuencia para una actividad de compra *online*. La secuencia del proceso descrita en el caso de uso es la siguiente:

- El cliente realiza el pedido de un artículo en la tienda *online*.
- El encargado de la tienda prepara el paquete para envío y espera la confirmación de pago.
- El cliente recibe la notificación de que el pedido está listo y para el importe.
- El banco confirma el pago a la tienda *online* y al cliente.
- El encargado envía el paquete.

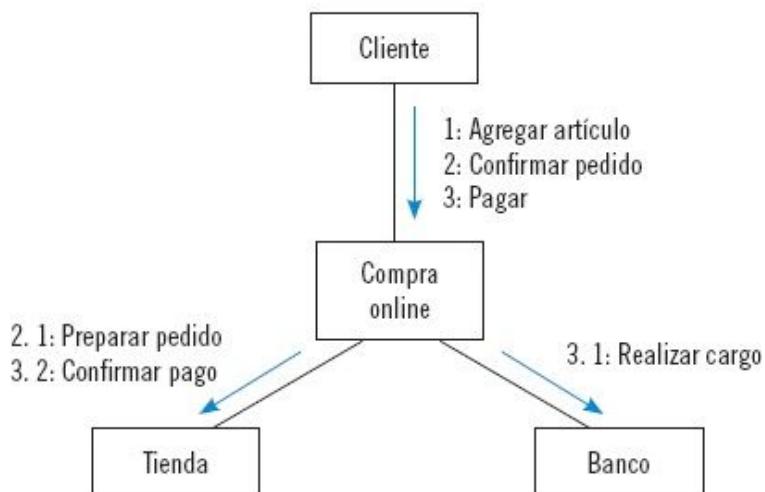
Diagramas de comunicación

La idea es la misma que en los diagramas de secuencia, pero enfatizando la relación entre objetos en lugar del orden temporal. En cada línea de unión se especifican las distintas llamadas, dándoles una numeración que permita establecer el orden de la comunicación.

Ejemplo

Realizar el diagrama de comunicación para la actividad propuesta en la aplicación anterior.

Diagrama de comunicación



Documentación

El Documento de Diseño de Software (SDD) muestra cómo estará estructurado el sistema, de modo que se cumplan los requisitos definidos en la fase anterior. Es la principal referencia del programador para empezar a escribir código. Un SDD deberá contener la siguiente información:

1. Introducción:
 - a. Propósito: propósito del documento y la audiencia a la que va dirigido.
 - b. Ámbito: descripción y ámbito del *software*, destacando metas, objetivos y beneficios.
 - c. Descripción general del documento.
2. Descripción general del sistema: funcionalidad, contexto y diseño del sistema.
3. Arquitectura del sistema:
 - a. Diseño arquitectónico: estructura modular del programa representando las relaciones entre módulos.
 - b. Diseño arquitectónico de los subsistemas: estructura de los subsistemas del programa. Es un nivel de detalle mayor que el punto anterior.
 - c. Criterio de diseño: se justifica la elección de la arquitectura adoptada.
4. Diseño de datos:
 - a. Descripción de datos: cómo los datos van a ser representados, almacenados y procesados.
 - b. Diccionario de datos: lista (y descripción) de las entidades resultantes.
5. Diseño de componentes: diseño de cada componente usado en el sistema.
6. Diseño de interfaz: descripción detallada de la interfaz, mostrando capturas e indicando cuáles son los objetos sobre los que se puede interactuar. Mostrará además la funcionalidad asociada (todo ello desde la perspectiva del usuario).

Implementación. Conceptos generales de desarrollo de software

Una vez que se dispone de la arquitectura del sistema, con la especificación de todas las entidades a desarrollar, entra en juego la codificación. Aquí la idea es hacer una implementación, mediante un lenguaje de programación, para generar un ejecutable.



Sabías que...

La creencia popular es que la fase de codificación ocupa la mayor parte del proceso de desarrollo. Si se parte de buenos diseños previos, esta fase puede ser muy corta.

Técnicas de desarrollo de software

Las técnicas de desarrollo de *software* se definen como el medio escogido para codificar una serie de órdenes lógicas en un determinado lenguaje de programación. Existen bastante técnicas,

agrupándose en una metodología que normalmente deriva en un tipo de programación. En una primera clasificación, se establecen los siguientes tipos:

- **Programación imperativa:** el programa detalla los pasos a seguir, en forma de conjunto de instrucciones, para llevar a cabo una tarea. Es decir, se le dice al programa qué hacer.
- **Programación declarativa:** el programa describe los mecanismos a utilizar y el resultado a obtener, pero no implementa los pasos a seguir. Es el tipo de programación utilizada en lenguajes de Inteligencia Artificial y en lenguajes de consulta sobre base de datos.

Casi todos los lenguajes de referencia son imperativos, así que este manual se centrará en dicha programación. La programación imperativa se puede, a su vez, dividir en diferentes vertientes de desarrollo. A continuación, se presentan dos de las más importantes, la estructurada y la orientada a objetos.

Programación estructurada

Surgió en la década de los 60, fundamentada en el Teorema del programa estructurado de Böhm y Jacopini. Este teorema promulga que cualquier programa puede dividirse en bloques de código que realizan una tarea concreta. Estos bloques de código sólo pueden combinarse mediante las tres estructuras de control básicas (secuencia, condición e iteración).

Una implementación usando esta programación será, generalmente, un programa dividido en bloques (módulos). Seguirá un desarrollo *top-down*, empezando por lo general del problema para irlo refinando poco a poco. El flujo del programa seguirá un orden secuencial, sin saltos, enfocado a la parte de proceso.

Programación orientada a objetos

Surge en la década de los 80, apoyada en la programación estructurada y desarrollando técnicas propias como la herencia, la cohesión, la abstracción, el polimorfismo y el encapsulamiento.

La clave de esta programación es el concepto de objeto. Un objeto representa un concepto del mundo real, albergando tanto datos como procesos, estos últimos en forma de comportamiento.

Esta técnica de programación será tratada en profundidad más adelante.

Principios básicos del desarrollo de software

En este apartado, se utilizarán como referencia los **7 principios básicos del desarrollo software**, que fueron desarrollados por Barry Boehm con el fin de concretar en ellos una larga lista de aforismos y buenas costumbres.

1. Usar un plan de desarrollo basado en un ciclo de vida

Hay que huir de una mala planificación. Una incorrecta definición de requisitos, descartar algunas de las fases del ciclo de vida o no darles la suficiente importancia, etc. son multitud

de factores que pueden arruinar un desarrollo. Alrededor del 50% de los proyectos que fracasan es debido a una mala planificación.

2. Realizar una continua validación

Uno de los problemas más costosos durante el desarrollo de *software* es el descubrimiento de un error en una fase avanzada, error que, normalmente, es anterior a la fase de implementación del código. Boehm aboga por incorporar tareas de validación desde fases tempranas de desarrollo, de tal forma que el error no se extienda y sea menos costoso de reparar.

3. Mantener un control del producto

Es importante mantener un control de versiones y de modificaciones del producto. Durante un desarrollo suelen surgir cambios y, a veces, estos vienen por factores externos que no se pueden evitar. Todos estos detalles deben reflejarse en la documentación. La pérdida de esta disciplina de control puede ocasionar el despilfarro de una valiosa cantidad de tiempo y de recursos.

4. Usar técnicas de programación modernas

Una técnica de programación implica, por norma general, un código más claro y con mayor facilidad de mantenimiento, haciendo la integración con otro *software* más fácil y simplificando la fase de pruebas. Todo esto ayuda también a una más temprana detección de errores.

5. Mantener un control de resultados

El producto *software* es, la mayor parte de las veces, algo que no se puede medir. Un programador puede haber invertido una gran cantidad de tiempo y no tener todavía una determinada funcionalidad del programa lista para enseñar a nivel práctico. Por eso, se recomienda establecer un control de objetivos, empezando por niveles generales (el proyecto en sí), hasta llegar a cada individuo que intervenga en el proyecto.

6. Mucha gente no garantiza un mejor proyecto

Un equipo enorme no implica un desarrollo más rápido o de mayor calidad. Es mejor tener el equipo adecuado, aunque tenga menor número, que un grupo amplio de gente de cuestionable calidad. Por supuesto, hay proyectos y proyectos, pero en líneas generales las comunicaciones son más directas y la coordinación es mejor cuanto menor es el número de individuos en el equipo. Un problema de calendario no se debe solucionar intentando aportar más gente al proyecto.

7. Comprometerse a mejorar el proceso de desarrollo.

Lo que hoy funciona dentro de un tiempo puede no funcionar. No hay que incorporar todas las nuevas técnicas prometedoras que surjan de un día para otro, pero sí es aconsejable ir

evaluando su impacto y analizar su viabilidad, para ir las adaptando poco a poco e incorporarlas al proceso de desarrollo.

Principios básicos de programación

En este apartado, se indicarán los principios básicos de la programación estructurada. Como se dijo anteriormente, muchas de estas ideas son compartidas por la programación orientada a objetos. Inicialmente, se mostrarán las estructuras de control sobre las que se apoya el Teorema del programa estructurado de Böhm y Jacopini. Después se hará un breve recorrido por los tipos de datos, para seguir por los operadores y terminar con las funciones y procedimientos.

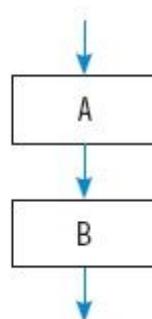
Estructuras de control

Una estructura incide directamente sobre el flujo de ejecución, modificándolo en función de ciertas condiciones. Con la combinación de estas estructuras, se puede desarrollar cualquier programa.

Secuencia

Es, simplemente, la ejecución secuencial de una serie de instrucciones:

Estructura secuencial



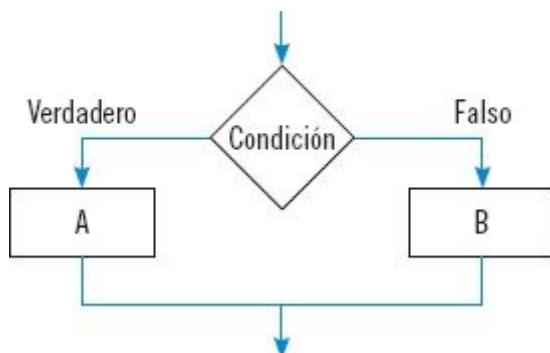
Selección

Se evalúa una condición para determinar por dónde continuará el flujo del programa. Puede ser de dos tipos:

- IF-ELSE

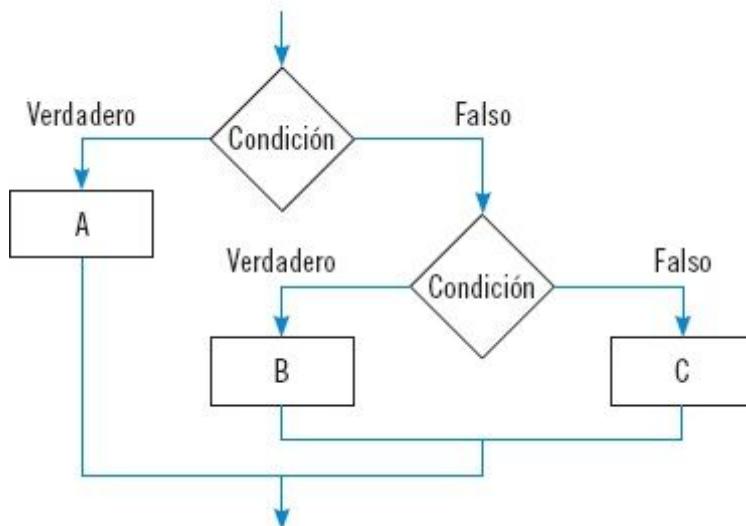
En la imagen siguiente, se aprecia cómo se evalúa una condición y, dependiendo del resultado de la misma, se ejecuta la instrucción A o B.

Estructura IF-ELSE #1



En el siguiente caso, se incluye una nueva estructura IF-ELSE, que se evaluará si no se cumple la primera condición.

Estructura IF-ELSE #2

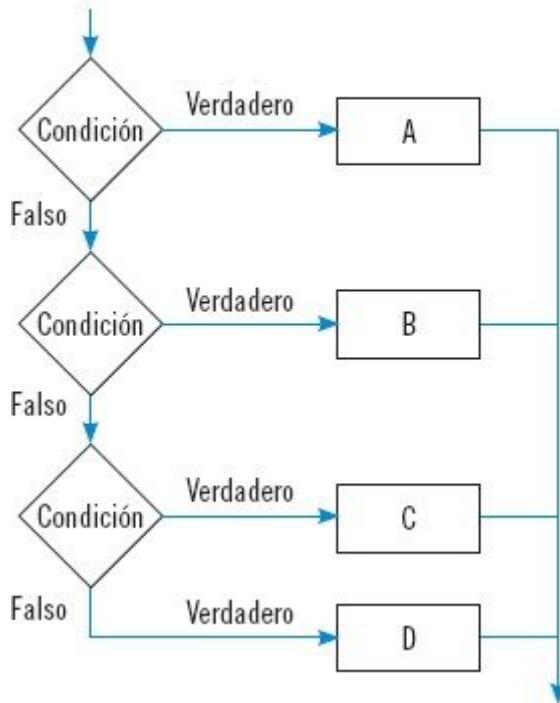


- **SWITCH**

Es una forma simplificada de representar una sucesión IF-ELSE. En la condición se efectúa una comparación y, si coincide, se ejecuta la instrucción correspondiente.

En caso de llegar al final y no encontrar equivalente, tendría lugar la instrucción por defecto. Esta instrucción no es obligatoria pero sí se recomienda su uso. Una vez ejecutada la instrucción que corresponda el programa sigue su curso.

Estructura SWITCH



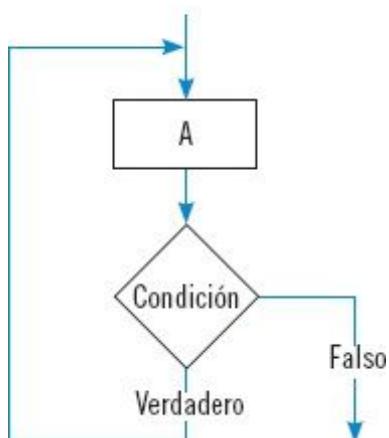
Iteración

En este bloque de control, se repite la ejecución de una instrucción mientras que se cumpla cierta condición. Existen tres variantes:

- DO-WHILE
- WHILE
- FOR

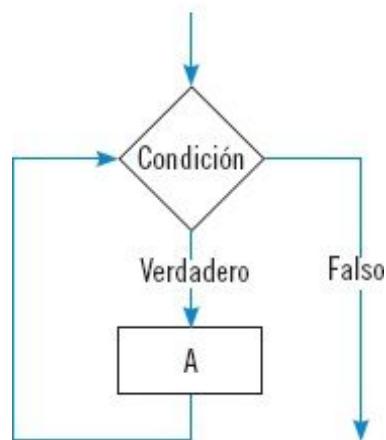
En la imagen, se representa una estructura DO-WHILE. Se ejecuta una instrucción y, según la condición, volverá a repetirse. Así una y otra vez hasta que la condición sea falsa y el programa continúe con el flujo normal.

Estructura DO-WHILE



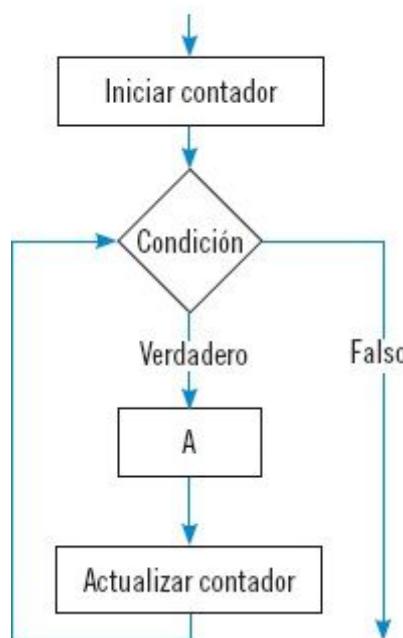
En la siguiente imagen se muestra una estructura WHILE. Es lo mismo que el DO-WHILE, solo que en el caso anterior la instrucción se ejecutaba al menos una vez, ya que la condición se evaluaba al final del bucle. En un WHILE, la condición se evalúa al principio, lo que puede suponer que la instrucción no se llegue a ejecutar ni una sola vez.

Estructura WHILE



La siguiente imagen muestra el bucle FOR. En esta estructura, se inicializa un contador, generalmente un número, fijando un valor a alcanzar y el incremento que sufrirá en cada repetición.

Estructura FOR



Recuerda

Una estructura de control define el flujo de ejecución del programa. Hay tres tipos:

- **Secuencial:** una instrucción después de otra.
- **Selección:** la instrucción a ejecutar depende de una condición a evaluar.
- **Iterativa:** se repite la ejecución de una instrucción mientras se cumpla la determinada condición.

Una vez planteadas y asimiladas las estructuras de control básicas, se procederá a explicar cómo se representa la información que va a ser procesada por el ordenador.

Tipos de datos

Un tipo de datos es un atributo que indica el tipo de dato con el cual se va a trabajar. No es lo mismo trabajar con un simple entero (un número sin parte decimal) que con una cadena de caracteres, un número con decimales o una matriz de enteros.

La distinción clásica establece dos tipos de datos: los datos primitivos y los datos compuestos. Hay que tener en cuenta que esto se refiere a una clasificación teórica. En la práctica, depende muchísimo del tipo de lenguaje de programación utilizado. Lo que en un lenguaje es un tipo primitivo, en otro puede ser compuesto y viceversa o bien puede no tener su correspondiente equivalente.

Tipo de dato primitivo

Son los tipos de datos básicos que proporciona el lenguaje de programación y que pueden ser usados para crear tipos de datos compuestos. Se pueden establecer los siguientes tipos de datos, bastante comunes y que son compartidos por buena parte de los lenguajes:

- **Tipo primitivo carácter (*char*):** sirve para almacenar un carácter. Por carácter se entiende un número, letra, signo de puntuación, etc.
- **Tipo primitivo entero (*int*):** almacena un número entero. El número límite permitido viene definido por la memoria utilizada para almacenar el número y por la posibilidad de representar (o no) números negativos. Si se intenta asignar a un tipo entero un número fuera de rango, se obtiene un desbordamiento (*overflow*).
- **Tipo primitivo real (*float*):** igual que el tipo entero, pero representando un número con decimales. Requiere más memoria que el tipo entero y también sufre problemas de desbordamiento si sobrepasa el límite aceptado.
- **Tipo primitivo booleano (*boolean*):** toma un valor de dos posibles (verdadero o falso). Este valor depende del lenguaje de programación adoptado. En alguno, verdadero se representa con "1" o con la palabra "*true*". Por analogía, el estado falso será "0" o "*false*".

Sabías que...

Durante el desbordamiento, el programa generalmente continúa con su ejecución, pero como el valor almacenado es incorrecto, se terminarán provocando resultados imprevisibles.

Tipo de dato compuesto

Los tipos de datos compuestos son los datos generados a partir de los tipos primitivos. A continuación, se mostrarán tres ejemplos clásicos:

- **Cadena de caracteres (string)**: almacena una serie de caracteres, generalmente uno detrás de otro en posiciones consecutivas de memoria.
- **Vector (array unidimensional)**: igual que la cadena, pero con una serie de números (*ints* o *floats*, dependiendo del tipo definido en el vector).
- **Matriz (array bidimensional)**: un simple vector de vectores, siendo todos ellos del mismo tipo primitivo de datos.

Existen más tipos de datos compuestos, pero avanzar en la materia requiere puntualizar bastante en el lenguaje de programación utilizado. Por ahora, se pretende ver la teoría a nivel genérico.

Los tipos de datos presentados no sirven de nada por sí solos. Para poder trabajar con ellos hay que utilizar variables. Una variable es un contenedor de datos que almacena información de un tipo de dato determinado.

Para definir una variable, se especifica el tipo de dato que se desea utilizar y a continuación el nombre asignado a la misma. Por ejemplo, si se desea definir una variable de tipo entero llamada **contador**, se haría de la siguiente manera:

```
int contador
```

Todas las variables que se definen en un programa deben ser usadas, más tarde o más temprano. Puede que no siempre entren en juego en cada ejecución del programa, pero tiene que existir al menos un flujo alternativo que utilice esa variable. En caso de que no se use una variable definida no supondrá nada, pero dependiendo del lenguaje y del entorno de programación utilizado puede provocar un aviso por pantalla.

Ahora que se dispone de una variable se le puede asignar un valor. Este valor se almacenará en la posición de memoria referida por la variable. Existen dos posibilidades de asignación:

- **Definir la variable e inicializar al mismo tiempo**: se define la variable con el tipo de datos elegido, inicializando un valor mediante el signo “=”. A continuación, se define una variable **contador2** de tipo entero, con un valor de 45.

```
int contador2 = 45
```

- **Asignar un valor después de la definición:** al definir una variable y no inicializarla, esta recibe un valor por defecto (en función al lenguaje de programación). Para modificar el valor por defecto se realiza una asignación. Suponiendo la variable contador anteriormente definida, se procederá a asignarle el entero 45.

```
contador = 45
```

Operadores

Los operadores son elementos que permiten modificar el valor de las variables. Algunos operan con una única variable, mientras que otros necesitan de dos. Existen cuatro tipos:

- Operadores matemáticos.
- Operadores lógicos.
- Operadores relacionales.
- Operadores de asignación.

Operadores matemáticos

Permiten realizar operaciones matemáticas con los valores contenidos en las variables. Se pueden utilizar siempre que estas contengan un valor numérico (ya sea entero o real). Los más usados son los siguientes:

- Suma (+)
- Resta (-)
- Multiplicación (*)
- División (/)
- Módulo (**mod o %**). El módulo es el resto de la división entera.
- Exponenciación (^)

Estos operadores tienen unas reglas de prioridad, las cuales deben ser consideradas para el caso de que se enlacen operaciones con diferentes variables dentro de una misma expresión. El orden es el siguiente:

1. Tiene preferencia lo que haya dentro de un paréntesis.
2. Exponenciación.
3. Multiplicación, división y módulo.
4. Suma y resta.

De esta manera la expresión $4 + 4 * 5$ dará 24, pero $(4 + 4) * 5$ generará 40.

Operadores relacionales

Establecen una relación entre los valores contenidos en dos variables. De esta relación se obtiene un resultado (verdadero o falso). Las dos variables tienen que ser del mismo tipo. En el caso de no especificar con paréntesis, los operadores relacionales tienen menor prioridad que los aritméticos:

- Menor que (<)
- Mayor que (>)
- Menor o igual que (<=)
- Mayor o igual que (>=)
- Diferente (!=)
- Igual (==)

Operadores lógicos

Sirven para combinar condiciones. Indispensables para usar en las estructuras de control vistas anteriormente:

- Operador AND (&&): las dos condiciones deben ser ciertas.
- Operador OR (||): basta con que sea cierta una condición.
- Operador NOT (!): cierta si la condición se cumple.

Ejemplos

Imprimir un número por pantalla 10 veces.

1. FOR 1



```
1 for (int x=0; x<10; x++) {  
2     System.out.println("El número es: " + x);  
3 }
```

2. FOR 2



```
1 for (int x=1; x<=10; x++) {  
2     System.out.println("El número es: " + x);  
3 }
```

3. DO-WHILE



```
1 int x = 1;  
2 do {  
3     System.out.println("El número es: " + x);  
4     x++;  
5 } while (x <= 10)
```

4. WHILE



index.php

```
1 int x = 0;
2 while (x != 10) {
3     System.out.println("El número es: " + x);
4     x++;
5 }
```

Imprimir por pantalla si “c” es mayor que 10 y “d” menor que 0.



```
1 if ((c > 10) && (d < 0)) {
2     System.out.println("c es menor que 10 y d menor que 0.");
3 }
```

Imprimir por pantalla si “m” es mayor que -5 y “n” igual a 3.



```
1 if ((m < -5) && (n == 3)) {
2     System.out.println("m es menor que -5 y n es igual a 3.");
3 }
```

Imprimir el valor de “m” si es igual al de “n”.



```
1 if (m == n) {  
2     System.out.println("El valor de m es: " + m);  
3 }
```

Imprimir el valor de “m” si es distinto a 3 y “n” es igual al carácter ‘m’.



```
1 if ((m != 3) && (n == 'm')) {  
2     System.out.println("El valor de m es: " + m);  
3 }
```

Recorrer de 1 a 100 y mostrar los números divisibles por 2 (un número divisible por 2 es aquel cuyo resto de dividir por 2 es 0).



```
1 for (x=1; x<=100; x++){  
2     if ((x % 2) == 0)  
3         System.out.println(x);  
4 }
```

Mostrar de 1 a 500. Usar un FOR con incrementos de 100, y un WHILE para los intermedios.

```
1 for (x=0; x<500; x+=100){  
2     int y = 1;  
3     while (y != 101)  
4         System.out.println(y + x);  
5     y++;  
6 }
```

Procedimientos y funciones

Una función es una parte de código que realiza una tarea. Puede recibir (o no) parámetros, pero siempre debe devolver algo. Un procedimiento es lo mismo que una función, con la salvedad de que no tiene que devolver un valor.

Esta distinción es la separación histórica entre función y procedimiento, pero existen matices según el lenguaje de programación. Por ejemplo, en C++, solo existe el concepto de función, puesto que siempre hay que devolver algo (aunque sea un tipo especial de datos llamado **void**, que viene siendo el equivalente a no devolver nada).

Ejemplo

Realizar una suma y mostrar el resultado por pantalla. Usar primero un procedimiento y después una función.

- **Procedimiento:**

```
1 void sumarProc(int x, int y)  
2 {  
3     int resultado = x + y;  
4     System.out.println(resultado)  
5 }
```

Se llamará a este procedimiento de la siguiente manera: *sumarProc(5, 6)*.

- **Función:**



```
1 int sumarProc(int x, int y)
2 {
3     int resultado = x + y;
4     return resultado;
5 }
```

La función debe apoyarse en la función `print` del sistema para mostrar el valor que retorna:
`System.out.println(sumarProc)`.

💡 Actividad 3

Su compañero, actualmente de baja por un accidente casero, empezó a desarrollar una pequeña calculadora, por lo que tiene usted que terminar el trabajo. De los documentos que revisa, se extrae la siguiente información de utilidad:

- La calculadora debe presentar un menú central con 5 opciones: Sumar, Restar, Multiplicar, Dividir y Salir.
- Cada operación básica debe implementarse en una función, recibiendo los dos números como parámetro.
- El menú será también implementado como función.
- Se aceptan número decimales.
- La entrada de datos del usuario y mostrar resultados por pantalla ya están implementadas mediante `int entradaUser()`, que devuelve el número introducido por el usuario, y `mostrarResultado(int resultado)`, que muestra por pantalla el número recibido como parámetro.

Para mostrar un mensaje por pantalla use la función `System.out.println()`, indicando entre paréntesis la cadena de caracteres.

Validación y verificación de sistemas

Verificación y validación son dos conceptos parecidos. De hecho, es común encontrarlos referidos a la misma idea. Pero, mientras que la verificación se refiere al conjunto de actividades que aseguran que el *software* realiza una funcionalidad de manera correcta, la validación se encarga de asegurar que el *software* construido cumple con la funcionalidad requerida por el cliente.

A continuación, se tratarán los métodos formales de verificación y los métodos automatizados de análisis. Ambos están englobados dentro de un contexto de análisis estático, ya que no necesitan de la ejecución del programa.

Planificación

Los procesos de validación y verificación son caros, por lo cual se hace recomendable una planificación cuidadosa que permita obtener el máximo provecho y controlar los costes derivados.

En circunstancias ideales, estos procesos deberían ser introducidos en fases tempranas del proceso de desarrollo, estableciendo una lista de comprobación y definiendo los posibles planes de pruebas para tener en cuenta la asignación de recursos. Además, con estas acciones, se podrá estimar el calendario de pruebas que permitirá una mejor visión general durante el proceso de desarrollo.

Recuerda

La validación responde a la pregunta de si se está haciendo el producto correcto, es decir, se refiere a si cumple con los requisitos del cliente.

La verificación responde a si se está haciendo el producto correctamente, en otras palabras, comprobar que el producto es funcionalmente correcto y robusto.

Métodos formales de verificación

Los métodos formales de verificación utilizan técnicas derivadas de las matemáticas y de la lógica para comprobar que un programa cumple con su especificación.

La idea de estos métodos es analizar matemáticamente la especificación para crear otra semánticamente equivalente. De esta manera, si se verifica esta última se da por validada la original.

Aplicada al código del programa, esta metodología permite verificar que el código del programa es coherente con su especificación. Por otra parte, también resulta útil para descubrir errores de programación y de diseño. Sin embargo, en líneas generales, los métodos formales de verificación presentan un gran coste en forma de *software* especializado y expertos matemáticos. Además, son prácticamente imposibles de aplicar en grandes desarrollos, aunque siempre se puede reservar su aplicación a componentes críticos.

Sabías que...

Aparte de la utilidad en la comprobación de si el programa se adapta (o no) a su especificación, los métodos formales pueden tener aplicación en otras fases del ciclo de desarrollo del *software*, tales como análisis de requisitos y pruebas.

Métodos automatizados de análisis

Los métodos automatizados de análisis utilizan herramientas *software* que escanean el código de un programa en busca de posibles errores, comprobando además que el código esté bien construido de acuerdo con la sintaxis del lenguaje de programación utilizado. Pueden implementar tres niveles de comprobación:

1. **Comprobación de errores característicos:** se buscan errores típicos, utilizando patrones que contienen las características de los errores a buscar. El analizador los marca en el código, presentando la causa del error y una posible solución. Los más comunes que se pueden encontrar son:
 - Variables usadas antes de la inicialización.
 - Variables declaradas pero no usadas.
 - Variables asignadas dos veces pero nunca usadas entre ambas asignaciones.
 - Asignaciones fuera de rango.
 - Código inalcanzable.
 - Bucles infinitos.
 - Funciones y procedimientos no usados.
 - Valor de retorno de función no almacenado.
 - Errores de tipo de dato.
2. **Comprobación de errores definidos por el usuario:** aparte de los patrones de error propios del analizador, el usuario puede definir patrones propios. Por ejemplo, se puede dar el caso de que una instrucción A deba ejecutarse siempre que se de una instrucción B. Esto se puede definir como patrón de error.
3. **Comprobación de aserciones:** el desarrollador incluye aserciones formales en el programa. Por ejemplo, se puede indicar que una variable contenga un valor comprendido en un rango. El analizador ejecuta el código de manera interna y comprueba dónde no se cumple la aserción.

El analizador estático es muy efectivo localizando errores, pero también hay que tener en cuenta que puede generar muchos falsos positivos. Estos falsos positivos indican que hay errores en una sección determinada del código, cuando realmente no los hay. Los falsos positivos pueden ser reducidos, incluyendo aserciones formales, pero esto requiere trabajo adicional sobre el código.

Pruebas de software

Las pruebas de *software* son el conjunto de actividades destinadas a verificar de manera objetiva que se ha generado un *software* de calidad, libre de errores y que cumple con lo exigido.

En este capítulo, se recorrerán los diferentes tipos de prueba, abordándose el proceso de diseño de las mismas, su ámbito de aplicación y las herramientas *software* que permiten su automatización.

Tipos

No existe una clasificación oficial de tipos de pruebas *software*, pero se pueden establecer varias clasificaciones en función de la aplicación de diversos criterios:

- Según el enfoque utilizado:
 - Pruebas de tipo caja negra (*black box testing*).
 - Pruebas de tipo caja blanca (*white box testing*).
- Según la necesidad de ejecución del sistema a probar:
 - Pruebas estáticas (sin ejecutar código).
 - Pruebas dinámicas (ejecutando el código).
- Según el tipo de ejecución de las pruebas:
 - Pruebas manuales.
 - Pruebas automáticas.
- Según el ámbito de las pruebas:
 - Pruebas unitarias (*Unit Testing*).
 - Pruebas de integración (*Integration Testing*).
 - Pruebas de sistema (*System Testing*).



Sabías que...

Durante el diseño del programa, se parte de la idea de que el *software* va a tener errores, así que hay que intentar realizar una programación que facilite, en la medida de lo posible, las labores de prueba. Las pruebas ayudan a comprender el sistema, pues hay que diseñarlas para romper el *software*. Una buena prueba sería aquella que encuentre la mayor parte de fallos con el menor esfuerzo.

Ámbitos de aplicación

El ámbito de aplicación de las pruebas es el objetivo a probar durante la realización de las mismas. En función del alcance de dichas pruebas se establecen tres niveles (o ámbitos) de aplicación, partiendo del nivel más específico para llegar al más general.

- Módulo único: pruebas unitarias.
- Grupo de módulos: pruebas de integración.
- Sistema completo: pruebas de sistema.

Pruebas unitarias

Estas pruebas verifican el funcionamiento de una pieza de software. El concepto de pieza abarca los módulos individuales, componentes, subprogramas, etc. De cualquier manera, la idea subyacente es probar la funcionalidad de una parte del sistema.

Pruebas de integración

Un componente puede funcionar correctamente de forma aislada, pero eso no significa que encaje con el resto. Por este motivo, se debe verificar la correcta interacción entre todos los componentes del sistema. Estas pruebas serán tratadas en profundidad en el apartado correspondiente.

Pruebas de sistema

Verifican el funcionamiento del sistema en su conjunto. Las dos pruebas anteriores revelarán la mayor parte de fallos funcionales, así que este nivel es perfecto para probar aspectos globales, tales como la seguridad, la velocidad, etc. También se realizan otras pruebas para medir la integración, la compatibilidad, las interfaces externas, etc.

Pruebas funcionales (BBT)

Estas pruebas verifican un *software* usando su interfaz externa. El sistema es visto como una caja negra de la que no se conoce su funcionamiento interno ni la estructura del sistema. Únicamente basta con saber cuáles son las entradas que debe recibir la aplicación, así como cuáles son las correspondientes salidas. La interfaz tiene que estar muy bien definida para llevar a cabo estas pruebas correctamente. Algunos de los métodos utilizados en el diseño de estas pruebas son:

- Partición de equivalencia.
- Análisis de valores frontera.
- Prueba de arreglo ortogonal.

Pruebas estructurales (WBT)

A diferencia del tipo caja negra, que implicaba una visión externa, estas pruebas conllevan una visión interna. Requieren de conocimiento del código para un correcto análisis de resultados. La persona encargada de efectuar estas pruebas elige diversos valores de entrada, examinando el flujo del programa y comprobando que se devuelven los valores correctos. Los casos de prueba derivados, siempre que se ejecuten de manera exhausta, cumplirán lo siguiente:

- Garantizar que todas las rutas se revisan.
- Revisar todas las revisiones lógicas.
- Ejecutar todos los bucles con sus valores frontera.
- Revisar las estructuras de datos internas.

Diseño de pruebas

A continuación, se presentarán distintos métodos utilizados en el diseño de pruebas de *software*. Se distinguirá entre métodos de caja negra y métodos de caja blanca.

Diseño de pruebas para caja negra

En las pruebas de tipo caja negra se pueden usar los métodos que se describen a continuación.

Partición de equivalencia

Se basa en dividir el dominio posible de datos de entrada de un programa en clases de equivalencia. Una clase de equivalencia representa un conjunto de estados (válidos o inválidos) según una determinada condición de entrada, siendo la condición de entrada los valores que espera el programa. Por ejemplo, en función de la condición de entrada:

- Se requiere un rango: se especifica una clase de equivalencia válida (dentro del rango) y dos inválidas (fuera del rango).
- Se requiere un valor específico: se define una clase válida (valor específico) y dos inválidas (otros valores).
- Se requiere un booleano: se define una clase válida (verdadero) y una inválida (falso).

Una vez definidas las clases, se pueden desarrollar los casos de prueba.

Análisis de valores frontera

Complementa las clases de equivalencia, seleccionando para los casos de prueba clases de equivalencia con valores extremos. Por ejemplo, si se especifica un rango de valores entre a y b como condición de entrada, los casos de prueba se harán con los valores a y b.

Pruebas de arreglo ortogonal

Estas pruebas tienen aplicación cuando se dispone de varias entradas con valores claramente definidos. Un ejemplo sería un programa con tres puntos de entrada que pueden tomar un valor del rango (1, 2, 3). Si se quiere realizar una prueba exhaustiva, habría que probar todas las combinaciones, que son 27 casos de prueba.

Combinaciones de entrada

PRUEBA	ENTRADA 1	ENTRADA 2	ENTRADA 3	PRUEBA	ENTRADA 1	ENTRADA 2	ENTRADA 3
Caso 1	1	1	1	Caso 15	2	2	3
Caso 2	1	1	2	Caso 16	2	3	1
Caso 3	1	1	3	Caso 17	2	3	2
Caso 4	1	2	1	Caso 18	2	3	3
Caso 5	1	2	2	Caso 19	3	1	1
Caso 6	1	2	3	Caso 20	3	1	2
Caso 7	1	3	1	Caso 21	3	1	3
Caso 8	1	3	2	Caso 22	3	2	1
Caso 9	1	3	3	Caso 23	3	2	2
Caso 10	2	1	1	Caso 24	3	2	3
Caso 11	2	1	2	Caso 25	3	3	1
Caso 12	2	1	3	Caso 26	3	3	2
Caso 13	2	2	1	Caso 27	3	3	3
Caso 14	2	2	2				

El arreglo ortogonal contiene un número reducido de combinaciones, además de disponer de una propiedad de equilibrio (que en cada columna los valores tengan la misma frecuencia de aparición) que provoca que los casos se dispersen de manera uniforme. La siguiente imagen establece un arreglo de tipo L9 (donde 9 es el número de casos de prueba).

Arreglo ortogonal L9

PRUEBA	ENTRADA 1	ENTRADA 2	ENTRADA 3
Caso 1	1	1	1
Caso 2	1	2	2
Caso 3	1	3	3
Caso 4	2	1	2
Caso 5	2	2	3
Caso 6	2	3	1
Caso 7	3	1	3
Caso 8	3	2	1
Caso 9	3	3	2

Diseño de pruebas para caja blanca

Se explicarán los siguientes métodos: prueba de la ruta básica y prueba de bucle.

Prueba de la ruta básica

Esta prueba consiste en crear un grado de flujo del programa, que no deja de ser una versión simplificada del diagrama de flujo. Las instrucciones se representan por nodos (las instrucciones secuenciales van dentro de un mismo nodo), mientras que las aristas indican el flujo que puede seguir el programa.

Sobre el grafo se establece la complejidad ciclomática, que indicará el grado de complejidad lógica de dicho grafo. Con este valor, se obtiene el número de rutas independientes. Una ruta independiente es aquella que no se repite, por el simple hecho de no estar incluida en ninguna otra, y servirá como base para los casos de prueba. La complejidad ciclomática $V(G)$ es calculada con la fórmula siguiente:

$$V(G) = E - N + 2$$

Siendo E el número de aristas del grafo y N el número de nodos.

Otra forma de calcular esta complejidad es obteniendo el número de regiones. Por región se entiende el área acotada entre nodos y arista, teniendo siempre presente que todo lo que queda en

el exterior también se considera una región. Es decir, el número de regiones es la suma total de regiones cerradas más uno.

Sabías que...

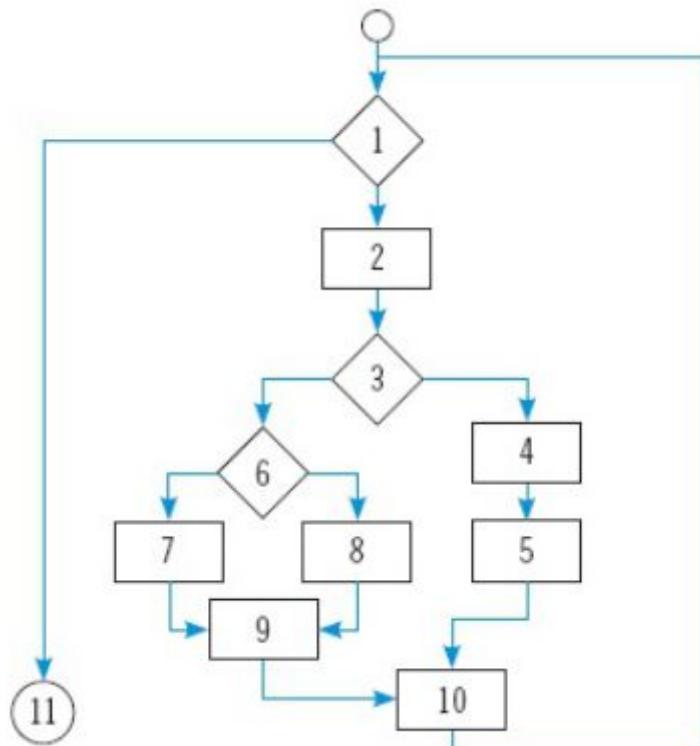
El resultado de la complejidad ciclomática también establece lo siguiente:

- 1-10: Programa de bajo riesgo.
- 11-20: Programa de riesgo moderado.
- 21-50: Programa de alto riesgo.
- Más de 50: Programa de muy alto riesgo.

Actividad 4

Calcular la complejidad ciclomática del diagrama de flujo presentado en la siguiente imagen.
Obtener también las rutas independientes.

Flujo del programa



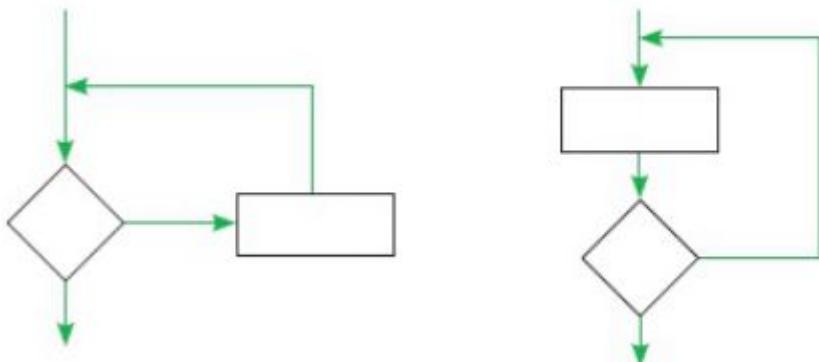
Prueba de bucle

Esta prueba se centra en los bucles, verificándolos. Distingue entre cuatro tipos de bucles:

- **Bucles simples:** En estos bucles se pueden aplicar las siguientes pruebas (siendo n el máximo número de pasadas permitidas):
 1. Saltar el bucle.

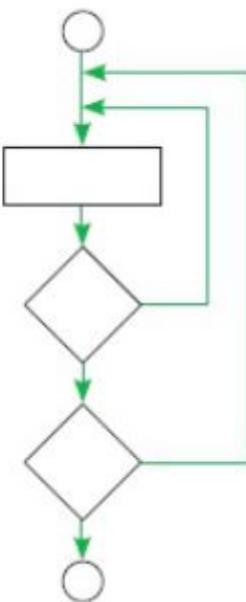
2. Solo una pasada.
3. Dos pasadas.
4. m pasadas, siendo $m < n$.
5. $n - 1, n + 1$, pasadas.

Bucles simples



- **Bucles anidados:** no se puede seguir la misma filosofía de los bucles simples, ya que el número de pruebas crecería exponencialmente.

Bucles anidados

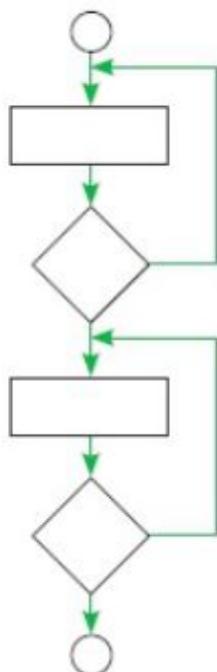


El proceso es el siguiente:

1. Comenzar por el bucle interior. Poner los otros a valores mínimos (por ejemplo, el contador respectivo al valor mínimo).

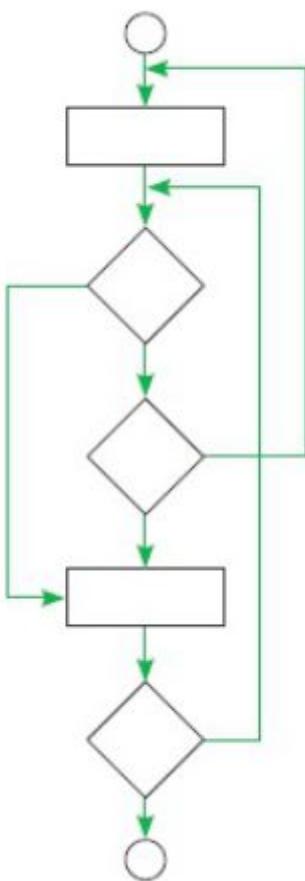
2. Realizar pruebas de bucle simple. Repetir con los bucles exteriores para valores fuera de rango.
 3. Repetir el proceso con los bucles exteriores, uno a uno, manteniendo los interiores a valor normal.
- **Bucles concatenados:** se puede usar el proceso de bucles simples si los dos bucles son independientes. Sin embargo, si los dos bucles comparten algún contador (por ejemplo, el contador del primer bucle inicializa el contador del segundo bucle), hay que aplicar el enfoque visto anteriormente para bucles anidados.

Bucles concatenados



- **Bucles no estructurados:** estos bucles atentan contra los principios de la programación estructurada, así que se recomienda que sean rediseñados para encuadrarlos en alguno de los tipos anteriores.

Bucles no estructurados



Comparativa. Pautas de utilización

Recuérdese la diferencia entre los dos tipos de pruebas. La idea principal es que un producto *software* puede probarse en base a su funcionalidad (realizando pruebas que demuestren que esta es llevada a cabo correctamente) y a su funcionamiento interno (garantizando que las operaciones internas cumplen con las especificaciones). De esta manera, se puede decir que el primer enfoque es una visión interna que conlleva revisar la estructura interna del programa (pruebas de caja blanca).

Ambas pruebas se deben llevar a cabo para garantizar la calidad del producto, utilizando una pauta general que se puede resumir de la siguiente manera:

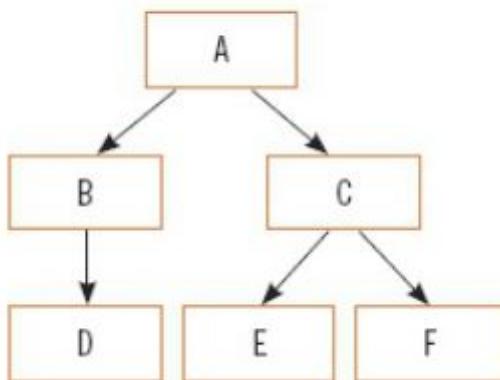
1. **Planificación y control:** establecer un plan, especificando el ámbito, los riesgos e identificando los objetivos de prueba. Fijar un calendario para las actividades relacionadas, determinar los recursos y establecer el criterio de salida (que estará determinado por el porcentaje de *software* que debe ser comprobado, siendo mayor dependiendo del riesgo asociado al *software*).
2. **Análisis y diseño:** revisar la documentación en la que se basan las pruebas (requisitos, arquitectura, interfaces, etc.) para recordar lo que el sistema debe hacer. Posteriormente, se diseñan las pruebas y se prepara el entorno.

3. **Implementación y ejecución:** se intentarán automatizar las pruebas todo lo posible, creándose *suites* de pruebas (colección de pruebas similares) con instrucciones detalladas. Los resultados de las pruebas serán registrados y se comprobarán los resultados obtenidos con los esperados. Si se produce un error, la prueba se repetirá en su momento para confirmar que ha sido arreglado.
4. **Evaluar criterio de salida e informe:** normalmente habrán sido fijadas unas medidas, tales como el porcentaje de casos de prueba satisfactorios o el ratio de errores por debajo de cierto nivel. El criterio de salida será evaluado, registrando conclusiones en un informe, para decidir si se continua o si debe ser modificado.
5. **Cierre de pruebas:** tiene lugar cuando el *software* ha sido entregado, el proyecto es cancelado, los objetivos conseguidos, etc. Se recomienda archivar las pruebas para un posterior uso y evaluar todo el proceso con el fin de mejorar en el futuro.

Pruebas de componentes

Las pruebas de componentes recibían el nombre de pruebas de integración. Hay veces que los componentes superan las pruebas unitarias pero, en el momento de interactuar con otros componentes, se producen errores. De ahí que la misión de estas pruebas sea verificar la correcta interacción entre los componentes del sistema. Para plantear una prueba de componentes, se parte de una representación del sistema en forma de árbol.

Componentes del sistema

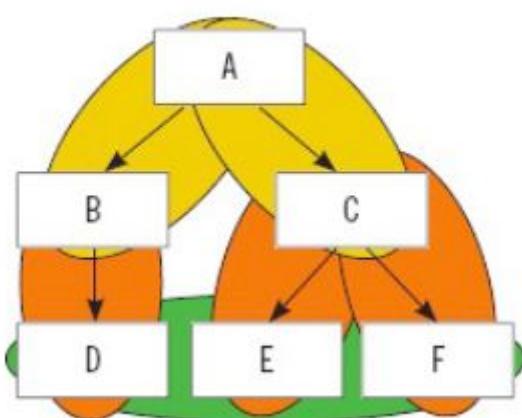


Existen cuatro estrategias para llevarlas a cabo:

- **Big bang:** prueba de funcionamiento global del sistema, así que exige un estado de desarrollo bastante avanzado. Aparte de eso, no resulta muy recomendable (a no ser que se realice sobre un *software* relativamente pequeño), ya que es muy complicado aislar las causas de un posible error. De ahí que se propongan las siguientes estrategias, que probarán la funcionalidad del sistema de manera progresiva.
- **Incremental ascendente:**
 - Se empiezan realizando las pruebas unitarias de los módulos finales.

- Se agrupan módulos por funcionalidad (normalmente el inmediato superior y los módulos hermanos) y se realiza una prueba de regresión programando un módulo especial de prueba (para controlar la entrada y salida de datos durante la prueba).
- Si la prueba es satisfactoria se anula el módulo controlador y el bloque anterior se integra con otro nuevo módulo.
- Se repiten las pruebas de integración, subiendo poco a poco y agrupando funcionalidades hasta el nodo padre.

Prueba incremental ascendente



Las pruebas a realizar para concluir con el proceso incremental serían las siguientes:

1. Pruebas unitarias en D, E y F.
2. Pruebas de regresión de B-D, C-E y C-F-
3. Prueba de regresión de B-D con A y prueba de regresión de C-E-F con A.
4. Pruebas de regresión de B-D-C-E-F con A.

- **Incremental descendente:**

- Primero se realiza un recorrido en profundidad (completando ramas, de arriba hacia abajo), para terminar con un recorrido en anchura (completando niveles de jerarquía, de izquierda a derecha).
- Las pruebas se realizan integrando módulos poco a poco, igual que con la prueba incremental ascendente, actuando el módulo principal como controlador de la prueba.
- Para probar funcionalidades que necesitan interactuar con módulos no afectados todavía por la prueba, se crearán módulos especiales que simularán su funcionalidad.

Los dos recorridos quedarían de la siguiente manera:

1. Recorrido en profundidad: A-B-D-C-E-F.
2. Recorrido en anchura: A-B-C-D-E-F.

- **Sándwich:** es una combinación de las anteriores. Se usa la incremental descendente para módulos superiores, mientras que se reserva la ascendente para los inferiores.

Pruebas de regresión

Pruebas que se realizan cuando se efectúa una modificación en el sistema, se agrega una nueva funcionalidad o se integra un nuevo módulo, con el fin verificar funcionamiento del conjunto.

Pruebas de sistemas

Las pruebas de sistemas se corresponden a la validación del mismo. Recuérdese que la validación respondía a la pregunta de si se está haciendo el producto correcto, así que se comprueba si el programa responde a los requisitos pedidos. Para estas pruebas se necesita:

- El *software* completo.
- Especificación de requisitos.
- Documentación de usuario.

Las pruebas de sistema se realizan en un entorno real. Entre las más importantes, se encuentran las pruebas de validación, realizadas por usuarios finales o por clientes para la aseguración definitiva de que el sistema cumple con los requisitos demandados. Pueden tener lugar en presencia del equipo programador (pruebas alfa) o en ausencia del mismo (pruebas beta).

Entre los aspectos que se trata de verificar por estas pruebas, se encuentran los siguientes:

- Seguridad.
- Recuperación: frente a errores, excepciones.
- Rendimiento en condiciones extremas: muchos usuarios, gran volumen de datos, muchas operaciones con base de datos.
- Eficiencia.
- Interacción con otro *software*.
- Compatibilidad.

Automatización de pruebas. Herramientas

La automatización de pruebas consiste en el uso de un *software* especial que ejecuta pruebas de manera controlada, presentando resultados y comparando con los esperados.

Generalmente, el *software* permite dos tipos de pruebas:

- Pruebas manejadas por el código: automatización de pruebas unitarias mediante casos de prueba.
- Pruebas de interfaz de usuario: permiten grabar acciones que realizaría un usuario sobre la interfaz del producto a evaluar. De esta forma, acciones tediosas y repetitivas pueden ser ejecutadas las veces que haga falta.

Para automatizar el desarrollo de pruebas, se recurre al uso de *frameworks*, tales como JUnit (en entorno Java) y NUnit (en entornos .NET). Ambos forman parte de lo que se conoce como xUnit, que no es más que una agrupación de todos los *frameworks* Unit.

JUnit, al trabajar con un lenguaje gratuito como Java, es uno de los más utilizados. Permite integrarse en entornos de desarrollo tales como NetBeans y Eclipse. Aparte de las mencionadas pruebas unitarias que puede automatizar, es especialmente útil para controlar las pruebas de regresión- Estas pruebas son las encargadas de comprobar la correcta funcionalidad del sistema cuando ha tenido una modificación.

Estándares sobre pruebas de software

Actualmente, se está desarrollando el estándar ISO/IEC 29119, el cual cubrirá áreas no consideradas por estándares anteriores. Entre sus objetivos, se encuentran unificar estándares, abarcar completamente el ciclo de vida y ser consistente con otros estándares ISO. Los estándares que se verán afectados serán los siguientes:

- IEEE 829 Test Documentation.
- IEEE 1008 Unit Testing.
- BS 7925-1 Vocabulary of Terms in Software Testing.
- BS 7925-2 Software Component Testing Standard.

El estándar constará de cinco partes, estando definidas así:

- Parte 1: Definiciones y vocabulario.
- Parte 2: Proceso de pruebas.
- Parte 3: Documentación de pruebas.
- Parte 4: Técnicas de pruebas.
- Parte 5: Pruebas basadas en palabras clave (*keyword-drive testing*).

Como ya se ha mencionado, este nuevo estándar se encuentra en fase de desarrollo, así que este apartado referente a pruebas *software* se centrará en el IEEE 829 Test Documentation. Existen numerosos estándares sobre pruebas *software*, pero este es uno de los más populares.

Su primera versión fue publicada en 1983, mientras que la última apareció en 2008. Actualmente, define los siguientes documentos:

- **Plan de pruebas:** enfoque de las pruebas, recursos y esquema de actividades, así como elementos a probar, sus características y el personal responsable.
- **Especificación de diseño de pruebas:** refinar el enfoque general del documento anterior, haciendo hincapié en las características que se deberían probar y los casos de prueba asociados a esas características.
- **Informe de ejecución:** se compone de los siguientes documentos:
 - Histórico de pruebas: hechos relevantes durante la ejecución de las pruebas.
 - Informe de incidente: incidente ocurrido durante una prueba.

- Informe resumen de pruebas: resume los resultados, efectuando una evaluación del *software*.

Calidad del software

La calidad del *software* es el conjunto de cualidades que lo caracterizan. Pressman la define como “la concordancia con los requisitos funcionales y de rendimiento, con los estándares de desarrollo y con las características implícitas que se espera del *software* desarrollado profesionalmente”.

Esta calidad es medible, dependiendo del programa y del sistema sobre el que se ejecuta. Por ejemplo: no es lo mismo la calidad esperada de un *software* ocasional hecho para una única ejecución que la de un *software* que deba funcionar sin errores durante cinco años.

Principios de calidad del software

Se pueden establecer unos aspectos generales sobre los que apoyar la calidad del *software*:

- **Corrección:** capacidad de realizar las tareas requeridas, es decir, la correcta adaptación de los requisitos.
- **Robustez:** capacidad de reacción ante excepciones e imprevistos durante la ejecución.
- **Eficiencia:** capacidad para hacer uso adecuado de los recursos del sistema.
- **Portabilidad:** facilidad de migración entre diferentes plataformas.
- **Integridad:** capacidad del sistema para proteger sus componentes contra accesos no permitidos.
- **Facilidad de uso:** facilidad de interacción con el usuario.
- **Verificabilidad:** facilidad para la realización de pruebas.
- **Extensibilidad:** facilidad de adaptación ante nuevos requisitos.
- **Reutilización:** capacidad de que el *software*, como bloque, actúa como componente dentro de un nuevo sistema.

Concepto de métrica y su importancia en la medición de la calidad

Una métrica es una medida de alguna propiedad del *software*. Proporciona un dato objetivo de aplicación durante los procesos de evaluación en el ciclo de desarrollo. Las métricas se pueden clasificar según el criterio que se pretenda aplicar, por ejemplo.

- **De complejidad:** definen la complejidad del sistema. Comprenden volumen, tamaño, anidaciones, etc.
- **De calidad:** definen la calidad del sistema. Abarcan exactitud, estructuración o modularidad, pruebas reusabilidad, etc.
- **De desempeño:** miden el rendimiento del sistema. Entre las mismas, se encuentran el tiempo de ejecución del sistema, la complejidad algorítmica, etc.

Para una correcta medición, se deben tener en cuenta ciertos principios. Los objetivos de la medición tienen que estar lo suficientemente claros antes de empezar al proceso de recogida de

datos, recomendándose la automatización del proceso de captura de datos y análisis. Además, unas buenas métricas deben presentar las siguientes características:

- Simples.
- Facilidad de cálculo.
- De naturaleza empírica.
- Objetivas.
- Independientes del lenguaje de programación elegido.
- Facilidad de uso para realimentación.

Sabías que...

La característica más importante de una métrica es la realimentación. De nada sirve obtener una medición sobre el aspecto del programa si no se pueden obtener conclusiones que permitan realizar ajustes o modificaciones en el caso de ser necesarias.

Métricas y calidad del software

Dado que la finalidad general del *software* es ser funcional y de calidad, la métrica proporciona medidas que permiten evaluar esa calidad de manera objetiva.

Para controlar estas medidas, se establecen modelos de calidad. Según el ISO/IEC 8402, un modelo de calidad se define como “el conjunto de características y las relaciones entre ellas que proveen la base para la especificación de los requisitos de calidad y la evaluación de la calidad”.

Un modelo de calidad bien definido permite:

- Una definición estructurada de los criterios usados para la evaluación.
- Una especificación de requisitos con relación a los criterios.
- La descripción de componentes en un marco común.
- La definición de métricas.

Principales métricas en las fases del ciclo de vida software

A continuación, se mostrarán las métricas más utilizadas en las fases del ciclo de vida del *software*.

Métricas en el modelo de análisis

Estas métricas intentan predecir el tamaño del sistema, ya que se puede establecer una relación directa entre el tamaño y la complejidad de diseño. En todas estas métricas se obtiene un valor, partiendo de ciertos datos y aplicando alguna fórmula. El valor obtenido servirá de referencia para sucesivas planificaciones y revisiones:

- **Métricas basadas en la función:** tomando como referencia el diagrama de flujo de datos, se consideran los siguientes aspectos:
 - Nº de entradas de usuario: datos proporcionados por el usuario a la aplicación.

- Nº de salidas de usuario: salidas que proporcionan información al usuario (informes, mensajes de error, etc.).
- Nº de archivos: por archivo se entiende el almacén de información sobre el que accede el sistema para ejecutar su funcionalidad.
- Nº de interfaces externas: interfaces usadas para transmitir información a otro sistema (cintas, archivos de datos, etc.).
- **Métrica bang:** se parte también del diagrama de flujo de datos. En esta métrica, se evalúan primitivas, teniendo en cuenta los siguientes elementos:
 - Primitivas funcionales (PFu): transformaciones que aparecen en el nivel inferior del diagrama.
 - Elementos de datos (ED): atributos de objetos y elementos de datos no compuestos.
 - Objetos (OB): objetos de datos.
 - Relaciones (RE): relaciones entre objetos.
 - Transiciones (TR): número de transiciones de estado del diagrama.
 - Muestras de datos (TCi): son los ED que existen en el límite de la i-ésima primitiva funcional.

Métricas de diseño

Son las métricas utilizadas durante la fase de diseño:

- **Nivel arquitectónico:** miden la arquitectura del sistema, sin importar cómo se comporta el módulo de manera interna. Por ejemplo: si la arquitectura del sistema se representa en forma de árbol, se pueden establecer tres métricas:
 - **Tamaño:** es la suma del número de módulos (nodos) y de flujos (aristas).
 - **Profundidad:** el camino más largo desde el nodo raíz hasta un nodo hoja.
 - **Anchura:** máximo de nodos en cualquier nivel del árbol.
- **Nivel de componentes:** tienen en cuenta la cohesión, el acoplamiento y la complejidad del módulo, de tal forma que, a partir de las mismas, se puede establecer la calidad del diseño. Requieren conocimiento del funcionamiento interno del módulo.
 - **De cohesión:** se mide la cohesión del módulo. A más cohesión, menos interacción con el resto de módulos. El objetivo ideal es conseguir un alto nivel de cohesión.
 - **De acoplamiento:** es una medida de la conectividad del módulo con otros módulos o componentes del sistema. El objetivo es un bajo nivel de acoplamiento.
 - **De complejidad:** cálculo de la complejidad del componente mediante la obtención de la complejidad ciclomática $V(G)$.
- **De diseño de interfaz:** se fundamentan en la asignación de coste a cada secuencia de acciones que el usuario debe realizar para el desarrollo de una tarea específica. Para calcular el coste, se tienen en cuenta, entre otros, factores tales como la frecuencia y la posición en la interfaz del elemento con el cual interactúan.

Métricas de codificación

Estas métricas ayudan durante la fase de codificación del programa, principalmente ofreciendo datos sobre la complejidad del código:

- **Complejidad ciclomática**
- **Otras métricas (programación estructurada):**
 - Nº de líneas de código.
 - Nº de líneas en blanco.
 - Nº de comentarios.
 - Nº total de módulos.
- **Otras métricas (programación orientada a objetos):**
 - Nº de métodos de la clase.
 - Nº de atributos de la clase.
 - Nº de atributos accesibles desde el exterior (atributos públicos).
 - Nº de atributos no accesibles desde el exterior (atributos privados).
 - Nº de interfaces de la clase.
 - Cohesión entre métodos de la clase.

Métricas para pruebas

Estas métricas se aplican durante la fase de prueba. Entre los datos que proporcionan, se encuentran medidas sobre el éxito de las pruebas, la cobertura de requisitos y la tasa de eliminación de errores. Estas se explican a continuación:

- **Cobertura de funcionalidad:** mide la cobertura de requisitos durante la realización de las pruebas frente al total de requisitos definidos para el sistema.

$$(Requisitos cubiertos/Requisitos totales) * 100$$

- **Densidad de errores en casos de prueba:** mide el éxito de los casos de prueba (aquellos que localizaron algún error), frente al total de casos de prueba ejecutados.

$$(Casos de prueba que localizaron error/Total casos de prueba) * 100$$

- **Volatilidad de requisitos:** porcentaje de requisitos añadidos, modificados o eliminados frente al número total de requisitos originales.

$$(Requisitos modificados, añadidos o eliminados/Requisitos originales) * 100$$

- **Eficacia de revisión:** mide la eficacia del proceso de revisión, al comparar los errores encontrados durante la revisión con el número total de errores encontrados durante la revisión y prueba.

$$(Errores encontrados durante revisión/Errores encontrados durante revisión y prueba) * 100$$

- **Eficiencia de eliminación de errores:** cuantifica el éxito de eliminación de errores, valorando los errores eliminados durante el proceso de desarrollo frente al número total de errores encontrados en desarrollo y producción.

$$EED/(EDD + EDP) * 100$$

Siendo EED los errores eliminados durante el desarrollo, EDD los errores descubiertos en desarrollo y EDP los errores descubiertos en producción.

Recuerda

Existen numerosas métricas que permiten medir objetivamente distintas características del *software* y la calidad del mismo. Estas métricas son de utilidad durante la fase de desarrollo.

Estándares para la descripción de los factores de Calidad

Los estándares aplicado a la calidad del *software* son una serie de recomendaciones a seguir para garantizar que el *software*, además de tener calidad, cumpla con los requisitos establecidos por el cliente.

La mayor parte de los estándares surgen de la ISO (*International Organization for Standardization*), que es una organización no gubernamental compuesta por representantes de organizaciones nacionales.

Un estándar es de adopción voluntaria. No garantiza el resultado final, pero su seguimiento significa que el proceso sigue unos mínimos.

Sabías que...

Los estándares ISO abarcan muchísimos aspectos, no solo relacionados con el *software*. Incluyen desde el tratamiento de citas bibliográficas hasta sistemas de gestión de energía, pasando por normas relativas a productos sanitarios.

Estándar ISO/IEC 9126

El ISO 9126 es un estándar internacional para la evaluación de la calidad del *software*. Está dividido en cuatro partes:

- **Modelo de calidad:**
 - **Funcionalidad:** satisfacer las necesidades implícitas y explícitas (eficacia, seguridad, interoperabilidad, etc.)
 - **Fiabilidad:** el programa debe tener una buena capacidad de recuperación (madurez, tolerancia a fallos, recuperabilidad, etc.).
 - **Usabilidad:** interfaz intuitiva y con buena línea de aprendizaje (operabilidad, comprensión, etc.).
 - **Eficiencia:** buen consumo de recursos sin degradación con el tiempo (comportamiento en el tiempo, utilización de recursos, etc.).
 - **Mantenibilidad:** facilidad para extender funcionalidad y modificar o corregir errores (estabilidad, testeabilidad, modificabilidad, etc.).

- **Portabilidad:** capacidad del sistema para ser transferido a otra plataforma (adaptabilidad, coexistencia, instalabilidad, etc.).
- **Métricas externas:** encargadas de cuantificar la calidad externa del *software*. Miden las siguientes características:
 - Métricas de funcionalidad.
 - Métricas de fiabilidad.
 - Métricas de usabilidad.
 - Métricas de mantenibilidad.
 - Métricas de portabilidad.
- **Métricas internas:** miden la calidad interna del *software* en base a las siguientes subcaracterísticas:
 - Métricas de eficacia.
 - Métricas de seguridad.
 - Métricas de interoperabilidad.
 - Métricas de madurez.
 - Métricas de tolerancia a fallos.
 - Métricas de recuperabilidad.
 - Métricas de operabilidad.
 - Métricas de comprensión.
 - Métricas de comportamiento en el tiempo.
 - Métricas de utilización de recursos.
 - Métricas de estabilidad.
 - Métricas de testeabilidad.
 - Métricas de modificabilidad.
 - Métricas de instalabilidad.
- **Métricas de calidad de uso:** miden si un producto cumple las necesidades especificadas por el usuario, distinguiendo entre defecto y no conformidad. Estas métricas sólo pueden ser medidas en un entorno real y son:
 - Métricas de efectividad.
 - Métricas de productividad.
 - Métricas de seguridad.
 - Métricas de satisfacción.

No conformidad y defecto

La no conformidad se refiere al incumplimiento de un requisito, mientras que el defecto se refiere a una incorrecta implantación.

Otros estándares. Comparativa

Aparte del ISO 9126, derivado de la familia ISO 9000, existe el estándar ISO/IEC 25000, de muy reciente creación y fruto de la evolución del anterior (al cual irá reemplazando poco a poco). Está compuesto por cinco divisiones:

1. ISO/IEC 2500n - División de gestión de calidad: define los modelos y términos comunes a toda la familia 25000.
2. ISO/IEC 2501n - División de modelo de calidad: representación de modelos detallados para calidad interna, externa y en el uso del producto software.
3. ISO/IEC 2502n - División de medición de calidad: definición de métricas para medir la calidad interna, externa y de uso.
4. ISO/IEC 2503n - División de requisitos de calidad: especificación de requisitos de calidad válidos durante el proceso de desarrollo.
5. ISO/IEC 2504n - División de evaluación de calidad: recomendaciones y guías válidas para el proceso de evaluación del software.

Sabías que...

El ISO 2500 recibe el sobrenombre de SQaRE (*Software Product Quality Requirements and Evaluation*)

Herramientas de uso común para el desarrollo de software

En este punto se hará un repaso general a las diversas herramientas que tienen utilidad durante el desarrollo del *software*. Muchas de ellas agrupan varias funcionalidades en la misma herramienta, de complejidad más o menos variables, pudiendo ser catalogadas como imprescindibles en función al desarrollo que se quiera realizar.

Editores orientados a lenguajes de programación

Un editor es un programa, generalmente de naturaleza ligera, que permite la creación (y edición) de archivos de texto. Para desarrollar un programa puede bastar un simple editor de texto como el *Bloc de notas* de *Windows*. No obstante, resulta recomendable desechar esta opción (salvo para ediciones puntuales rápidas). Existen muchas más alternativas, gratuitas, que facilitan enormemente el trabajo del programador. Entre alguna de las características generales se encuentran:

- **Coloreado de sintaxis:** reconoce el lenguaje de programación usado en el archivo, coloreando el código según determine.
- **Edición de múltiples archivos:** despliegue en pestanas independiente. Muchas veces se ofrece la posibilidad de comparar dos archivos, indicando las diferencias.
- **Formateo de texto,** con sangrados y saltos de línea.
- **Conexiones a repositorio o FTP**
- **Enlaces a otras herramientas** (compiladores, repositorios, etc.).

Entre los más populares se encuentran *Atom*, *Visual Studio Code*, *Sublime Text*, *Visual Studio*, *Eclipse*, *Netbeans*, *Xcode*, *Android Studio*...

Compiladores y enlazadores

Un programa puede ser escrito en muchos lenguajes de programación, pero para su ejecución debe ser traducido a un lenguaje entendible por el ordenador. El primer paso de la traducción lo realiza el compilador, transformando el código en lo que se conoce como código de ensamblador.

Este último código debe ser interpretado por el ensamblador, con el fin de generar archivos objeto. Estos archivos, generalmente, no son ejecutables y requieren de la actuación del enlazador.

El enlazador se encarga de generar el ejecutable, ligando los archivos objeto con otros archivos fuente o con funciones de una biblioteca (siempre que sea necesario).

El ejemplo más clásico de compilación lo ofrece el lenguaje de programación C, cuya secuencia se muestra en la siguiente imagen:



Depuradores

Los depuradores (*debuggers*) son una de las herramientas más útiles para un programador. Integrados frecuentemente dentro de un entorno de desarrollo, un depurador permite la ejecución del código línea a línea, presentando en todo momento información del estado de las variables usadas en el programa y más datos de interés.

También ofrecen la posibilidad de indicar puntos de interrupción, que son lugares señalados dentro del código a partir del cual el programa entrará en modo depuración. Esto resulta especialmente útil para evitar partes del código que funcionan de manera correcta, yendo directamente a las líneas que interesa depurar.

Generadores de programas

Los generadores de programas son herramientas que crean *software* basado en algún lenguaje de programación sin necesidad de tener nociones del mismo. El concepto es muy amplio, al igual que la forma de llevar a cabo esta generación de código. Por ejemplo, *PHPRunner* permite generar una interfaz plenamente funcional en PHP sobre una base de datos, partiendo de la misma. Otros, como *Adobe Muse*, habilitan al usuario para desarrollar un sitio web basado en la combinación de HTML5, CSS3 y JavaScript.

Sabías que...

Muchos IDE permiten la generación automática de código, sobre todo en lo referido a la interfaz. Si se desea, por ejemplo, agregar un botón a una ventana, simplemente arrastrando el componente sobre la ventana contenedora se implementará todo el código (con el consiguiente ahorro de tiempo por parte del programador).

De pruebas y validación de software

Estas herramientas se encargan de automatizar pruebas unitarias y de otro tipo (tales como pruebas de regresión, con el fin de comprobar si el programa se ajusta a los requisitos). Completan su funcionalidad facilitando la labor de creación de informes y cotejamiento entre resultados obtenidos y resultados esperados.

Recuerda

Las pruebas unitarias eran aquellas que se realizaban sobre un componente para verificar su funcionalidad. El *framework JUnit* permite la integración dentro de entornos de desarrollo con el fin de realizar estas pruebas.

Optimizadores de código

Los optimizadores de código actúan sobre el código objeto generado por el ensamblador, realizando una optimización del mismo. La forma en que llevan a cabo su propósito es variada:

- Reordenación de código.
- Reducción de tamaño.
- Optimización de bucles.
- Eliminación de redundancias.
- Reordenación de operaciones.
- Eliminación de asignaciones muertas (no utilizadas).

El código resultado será equivalente al original, pero llevará a cabo su misión de manera más eficiente y ocupará menos espacio.

Nota

El optimizador de código forma parte del compilador y se ejecuta automáticamente durante el proceso de compilación. Por lo general, no es necesaria ninguna acción por parte del programador.

Empaquetadores

Los empaquetadores son herramientas que crean un paquete instalable a partir de un código. Un paquete *software* lleva, aparte del código compilado, cierta información adicional que puede ser la descripción del paquete, su versión y sus dependencias (otros paquetes de los que depende).

Los paquetes están muy arraigados en la filosofía *Linux*, pues buena parte de la funcionalidad de este se obtiene instalando paquetes a través del potente gestor que incluyen las distintas *distros*.

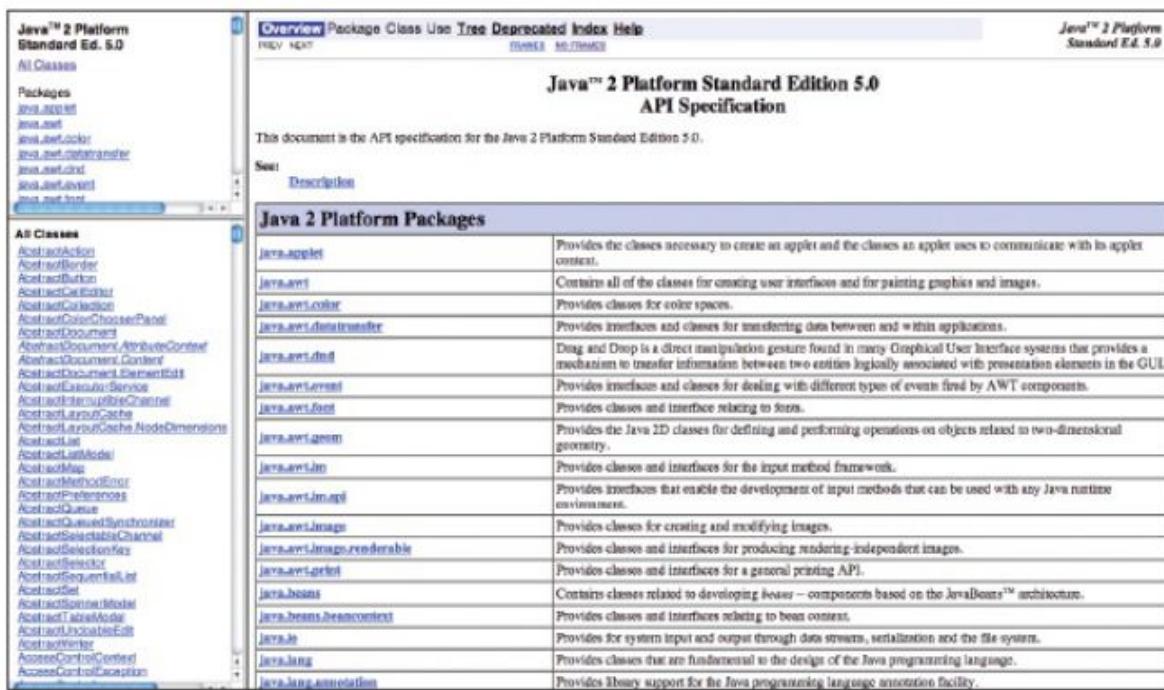
Sabías que...

El concepto de empaquetador también existe en *Windows*, pero quizás desde una perspectiva un poco más clásica. Normalmente, es una herramienta que permite crear menús de instalación (como el típico único archivo *setup* o *install* para realizar la instalación de un programa).

Generadores de documentación de software

Un generador de *software* es una herramienta destinada a la generación automática de documentación, ya sea para el usuario final o para el programador.

A continuación, se expondrá la utilidad *JavaDoc*, desarrollada por Oracle, que se utiliza para generar documentación Java (usando HTML) a partir de comentarios en el código fuente.



Java 2 Platform Packages	
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.image	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.
java.awt.print	Provides classes and interfaces for a general printing API.
java.beans	Contains classes related to developing beans - components based on the JavaBeans™ architecture.
java.beans.beancontext	Provides classes and interfaces relating to bean context.
java.io	Provides for system input and output through disk streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.lang.annotation	Provides library support for the Java programming language annotation facility.

Documentación generada por JavaDoc

La información a documentar se inserta dentro de los comentarios, usando tags especiales. Algunas de estas etiquetas son:

- @author: nombre del desarrollador.
- @param: descripción para cada uno de los parámetros.
- @return: lo que devuelve el método.
- @version: versión.

Así, por ejemplo, dentro del código para una función que realice una suma, podríamos encontrar los siguientes:

```

1 /**
2 * Esta función realiza una suma de dos números.
3 * @param num1 Primer número a sumar.
4 * @param num2 Segundo número a sumar.
5 * @return La suma de los dos números.
6 * @author Pablo Tamayo.* @version 1.0
7 */

```

De distribución de software

La distribución de *software* consiste en proporcionar *software* al usuario final. Estas herramientas, dependiendo de la configuración adoptada, pueden llegar a automatizar el proceso de instalación del nuevo *software*, llegando a estar completamente operativo en el ordenador con una mínima participación del usuario.

Nota

La tendencia actual es la distribución de *software* en formato digital. El proveedor proporciona su *software* (gratuito o de pago) a través de una plataforma (suya o de terceros), ofreciendo además servicio de soporte técnico hacia el usuario final.

Gestores y repositorios de paquetes. Versionado y control de dependencias

Los gestores son una colección de herramientas que abarcan todo lo relacionado con la vida de un paquete *software*, desde su instalación hasta su posible eliminación. Un gestor de paquetes normalmente va asociado a un repositorio, que es un sitio especial en Internet que contiene las últimas versiones de una serie de paquetes.

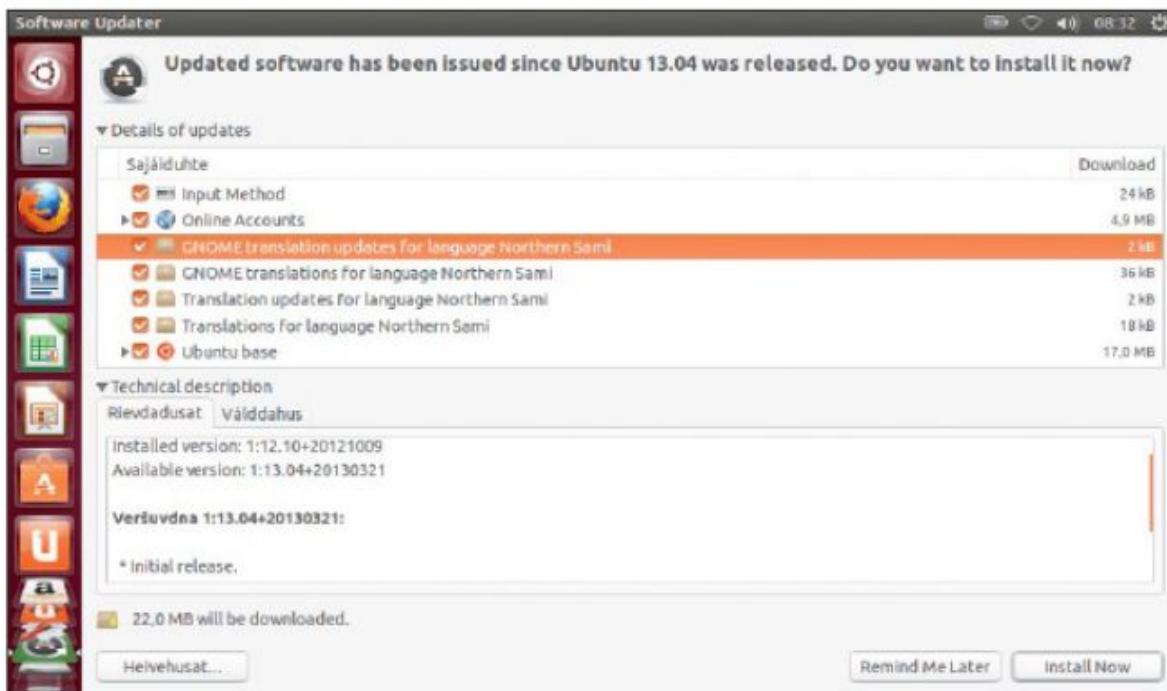
El gestor de paquetes verifica la versión de los paquetes que el usuario tiene instalados en el sistema, ofreciendo la posibilidad de descargarlos (o haciéndolo automáticamente si ha sido programado para ello). Al mismo tiempo, mantiene un control de dependencias, con el fin de indicar los paquetes adicionales que hacen falta para el funcionamiento de uno en concreto.

Gestores de actualización de software

Los gestores de actualización son pequeños programas que se instalan de manera complementaria a uno mayor. Su misión es monitorizar la versión del *software* actual, realizando cierta acción cuando encuentren una versión superior del programa al cual están asociados.

Como norma general, el comportamiento del gestor de actualización se puede definir de tres maneras diferentes:

- Permisos absolutos para instalar la última versión sin preguntar al usuario.
- Preguntar al usuario antes de iniciar una actualización.
- No comprobar nunca actualizaciones.

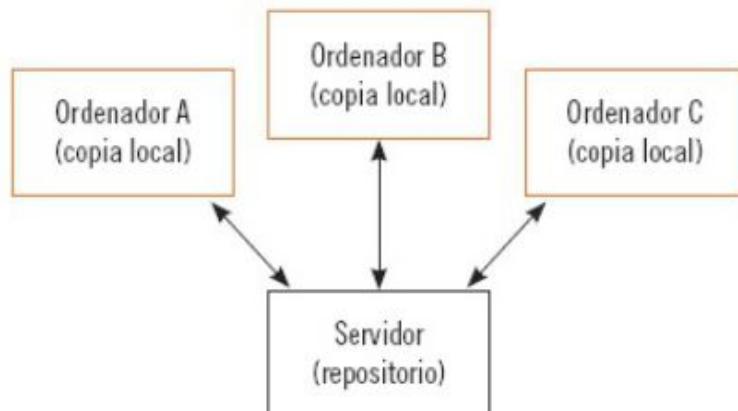


De control de versiones

Las herramientas de control de versiones se utilizan para llevar el registro de todas las modificaciones realizadas en el código de un proyecto *software*, permitiendo restaurar cualquier versión anterior. Actualmente, la filosofía es tener un repositorio en un servidor, al cual pueden acceder los desarrolladores para efectuar la edición del código del programa.

El gestor de versiones más utilizado actualmente es *Git*. Se trata de un sistema de control de versiones distribuido, gratis y *open source*. Resulta muy fácil de aprender y proporciona al desarrollador multitud de herramientas para gestionar sus proyectos.

Filosofía general del control de versiones



Gestión de proyectos de desarrollo de software

En este punto, se ofrecerá una visión general sobre la administración de proyectos de desarrollo, además de mostrar algunas herramientas útiles que facilitan esta labor.

Planificación de proyectos

La planificación del proyecto es un aspecto de suma importancia, muchas veces descuidado. Muchos estudios demuestran que una mala planificación desemboca, muy frecuentemente, en un proyecto fracasado. La planificación no es una ciencia exacta, incurriendo en ella multitud de variables. Esto no debe hacer que se menosprecie, ya que continúa siendo sumamente importante, pero tampoco hay que obsesionarse con ella.

El proceso para realizar una correcta planificación es el siguiente:

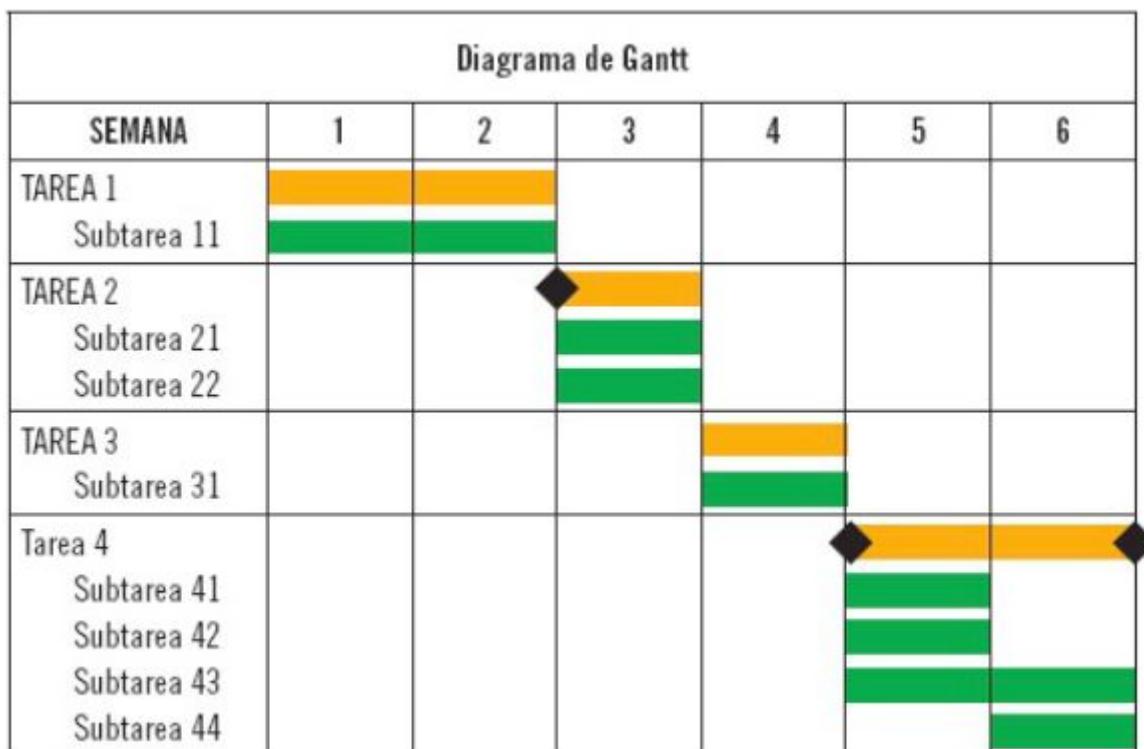
1. Definir el ámbito del *software*: situar el *software* en el contexto, indicando cómo se produce la entrada/salida de información y funcionalidad que presenta.
2. Descomposición del problema en subproblemas: aplicación del “divide y vencerás”, creando problemas más manejables.
3. Hacer estimaciones para cada subproblema: aquí juega un papel importantísimo la experiencia en desarrollos previos, aparte de los datos históricos.
4. Realizar la estimación global, considerando un factor de riesgo.

La estimación requiere especificar los recursos humanos y las herramientas *software* y *hardware* de las que se dispone. También se indicará el *software* reutilizable que pueda ser de aplicación (componentes adquiridos o previamente desarrollados, módulos de otros proyectos, etc).

Una vez realizada la estimación, hay que considerar una planificación temporal con lo siguiente:

- Definir todas las tareas.
- Marcar las tareas que sean consideradas como críticas.
- Identificar el camino crítico.

La representación gráfica de las tareas se realiza con Diagramas de Gant, mientras que el cálculo del camino crítico se deriva de los Diagramas de Pert.



Los diamantes que se presentan en la imagen son hitos. Estos son unos puntos específicos del calendario que tiene utilidad diversa. Una de las más típicas, es fijar una revisión de todo lo que se ha hecho en el proyecto hasta ese momento.

Para calcular el camino crítico y poder hacer cálculos, se recurre a los Diagramas de Pert. Esta es otra forma de representación de las tareas, pero, en vez de usar barras, se opta por un diagrama de flujos. Cada nodo es un instante de un proyecto y los instantes se relacionan entre ellos mediante las flechas (que corresponden a las tareas). La información que presenta un nodo es la siguiente:

- Número de nodo.
- Inicio mínimo: de todas las tareas que parten de ese nodo.
- Fin máximo: de todas las tareas que lleguen a ese nodo.

El proceso de creación del Diagrama de Pert, partiendo de una tabla con las dependencias de las tareas y su duración, es el siguiente:

1. Se empieza desde un nodo 1, cuyo Inicio mínimo y Fin máximo estarán a 0.
2. Se representa una flecha por cada tarea inicial, partiendo del nodo 1. Una tarea inicial es aquella que no tiene antecedente en la tabla.
3. Al final de cada flecha, se crea un nuevo nodo, cuyo Inicio mínimo será el Inicio mínimo del nodo anterior más la duración de la actividad indicada por la flecha. En caso de más de una actividad enlazada en el nodo, se toma el valor máximo de todas las candidatas.
4. Se repite el proceso hasta llegar al nodo final, que tendrá el mismo valor para Inicio mínimo y Fin máximo.

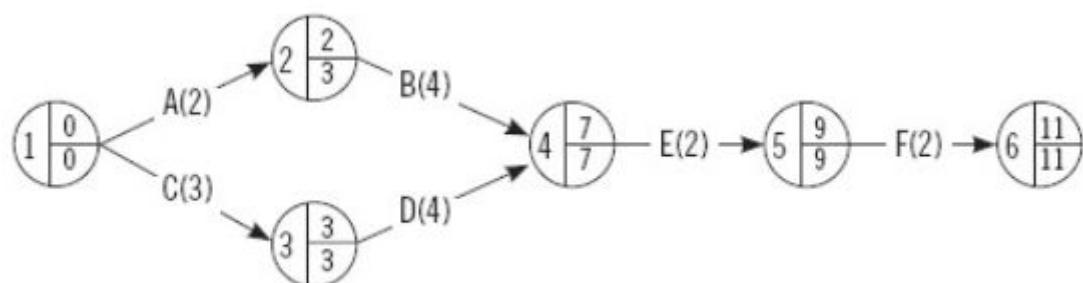
5. Empezando desde el nodo final, se va retrocediendo, haciendo que el Fin máximo sea el Fin máximo del nodo posterior menos la duración de la actividad. En caso de más de una actividad, se tomará el valor mínimo de las candidatas.
6. El camino crítico estará formado por aquellas tareas que no tengan margen alguno. Para este cálculo, hacen falta el Inicio máximo y el Fin mínimo.
 - $Fin \text{ } mínimo = Inicio \text{ } mínimo + \text{duración}$
 - $Inicio \text{ } máximo = Fin \text{ } máximo - \text{duración}$
 - $Margen = Fin \text{ } máximo - Fin \text{ } mínimo$

Ejemplo

Diagrama de Pert y camino mínimo para las siguientes tareas.

Diagrama de Pert

TAREA	PRECEDENTE	DURACIÓN
A	-	2
B	A	4
C	-	3
D	C	4
E	D, B	2
F	E	1



Camino crítico

TAREA	PRECEDENTE	DURACIÓN	INICIO MIN	FIN MIN	INICIO MAX	FIN MAX	MARGEN
A	-	2	0	2	1	3	1
B	A	4	2	6	3	7	1
C	-	3	0	3	0	3	0
D	C	4	3	7	3	7	0
E	D, B	2	7	9	7	9	0
F	E	1	9	10	10	11	1

Control de proyectos

El control del avance del proyecto se realiza en los puntos marcados como hito en el calendario. En estas revisiones, se repasa el estado del proceso y pueden resultar muy útiles para encontrar problemas en fases tempranas del desarrollo.

La revisión debe comprobar que se está desarrollando un producto de calidad, ajustándose a los requisitos y cumpliendo lo indicado en la planificación. También se puede establecer el uso de métricas, con el fin de obtener mediciones objetivas sobre la situación actual del proyecto.



Sabías que...

Los hitos reciben el nombre de *milestones*. No solo sirven para indicar fechas de revisión en el calendario, sino que también pueden referirse a acontecimientos importantes, como una entrega parcial.

Ejecución de proyectos

La ejecución del proyecto es el paso posterior a la planificación, una vez conseguidos los recursos *software* necesarios. Esta fase engloba prácticamente todo el ciclo de vida del *software*, aunque el análisis de requisitos se ha debido solapar ligeramente con la planificación del proyecto con el fin de poder plantear el producto.



Recuerda

Para el desarrollo de un *software* se usaba el modelo de ciclo de vida que definía las etapas y la interacción entre ellas con el fin de obtener un producto de calidad.

Herramientas de uso común para la gestión de proyectos

Existen multitud de herramientas *software* que pueden ayudar al equipo de desarrollo durante un proyecto *software*. Entre las funciones más típicas, se encuentran la gestión de un calendario, con completas administraciones de tareas y subtareas y definiciones de tiempo en las mismas. Algunas herramientas, incluso, disponen de lugares de encuentro para compartir documentación o mantener una videoconferencia.

Hace unos años casi todas esas herramientas eran locales, pero con el auge de la red y el trabajo remoto muchísimas de ellas se han ido enfocando al trabajo *online*. Entre las más valoradas se pueden citar *Slack* o *TeamWork* pero la oferta es amplísima.

Resumen

Durante el proceso de creación de un producto *software*, conviene adoptar un modelo de desarrollo, ya que marcará las pautas a seguir para obtener un *software* de calidad. Existen muchos modelos, surgidos de la necesidad de tener metodologías que encaminasen unos desarrollos *software* que iban creciendo en complejidad. La elección entre un modelo y otro depende de ciertos factores (límite de tiempo, necesidad de entregas parciales, recursos, complejidad del sistema, etc).

El ciclo de vida clásico, que sirvió como inspiración para el resto de modelos, se divide en las siguientes fases:

- Análisis de requisitos: captura de los requisitos para la realización del sistema.
- Diseño: diseño arquitectónico, de componentes y de interfaz en base a los requisitos anteriores.
- Implementación y codificación: implementación del diseño en un lenguaje de programación. Normalmente, se puede seguir una metodología de programación estructurada o de programación orientada a objetos.
- Pruebas: existiendo un especial interés en el capítulo de implementación, al introducir unas nociones básicas de programación estructurada. También se ha tratado la calidad del *software* y la verificación y validación del sistema, para establecer lo que se espera de un *software* de calidad y los diferentes estándares de aplicación.
- Mantenimiento: diseño arquitectónico, de componentes y de interfaz en base a los requisitos anteriores.

Para llevar a cabo el desarrollo del *software*, generalmente englobado dentro de un proyecto, hay herramientas, tanto gratuitas como de pago, que ofrecen la funcionalidad necesaria sobre la que apoyar una correcta planificación y ejecución del desarrollo.

Ejercicios de repaso y autoevaluación

1. De las siguientes afirmaciones, indique cuál es verdadera o falsa.

- a. El primer modelo definido del ciclo de vida del software fue el modelo iterativo.
 - Verdadero
 - Falso
- b. El modelo en cascada es el más recomendado para grandes desarrollos.
 - Verdadero
 - Falso
- c. El modelo de desarrollo rápido de aplicaciones permite un desarrollo rápido del software sin pagar un precio mayor.
 - Verdadero
 - Falso
- d. El modelo basado en componentes permite la reutilización del software.
 - Verdadero
 - Falso

2. Complete la siguiente oración

Un requisito es una _____ de un usuario para resolver un _____.

3. ¿En cuántos niveles se puede dividir un diagrama de flujo?

4. Relacione los siguientes elementos.

- | | |
|-----------------------|-------------------------|
| a. Documento ERS | ● Pruebas |
| b. Documento SSD | ● Diseño |
| c. Informes Ejecución | ● Análisis de requisito |

5. Los requisitos del sistema...

- a. definen las características técnicas del sistema.
- b. describen de manera completa la funcionalidad del software.
- c. describen de manera científica la funcionalidad del software.
- d. Todas las respuestas anteriores son verdaderas.

6. El modelo de interfaz...

- a. define la forma en que el usuario interactúa con el sistema.
- b. define la forma en que comunican los componentes del sistema.
- c. Las dos respuestas anteriores son correctas.
- d. Ninguna de las respuestas anteriores es correcta.

7. Son relaciones entre casos de uso...

- a. la extensión.
- b. la inclusión.
- c. la generalización.
- d. Todas las respuestas anteriores son correctas.

8. De las siguientes afirmaciones, indique cuál es verdadera o falsa

- a. Los diagramas de implementación pertenecen a UML.
 - Verdadero
 - Falso
- b. Los diagramas de actividad definen la transición entre los posibles estados del sistema, incluido el evento disparador.
 - Verdadero
 - Falso
- c. Los diagramas de secuencia muestran la comunicación de los objetos durante la ejecución de una tarea.
 - Verdadero
 - Falso
- d. El Documento de Diseño de Software (SDD) recoge como está estructurado el sistema para cumplir los requisitos.
 - Verdadero
 - Falso

9. La programación estructurada...

- a. se fundamenta en el Teorema del Programa Estructurado.
- b. permite el uso de herencia, cohesión y abstracción.
- c. describe los mecanismos a utilizar, pero no implementa los pasos a seguir para solucionar un problema.
- d. Todas las respuestas anteriores son correctas.

10. La planificación de proyectos...

- a. utiliza Diagramas de Boehm.
- b. no depende de proyectos anteriores.
- c. exige una serie de estimaciones.
- d. Todas las respuestas anteriores son correctas.

11. De las siguientes afirmaciones, indique cuál es verdadera o falsa.

- a. La validación es diferente a la verificación.
 - Verdadero
 - Falso

- b. Un método formal de verificación utiliza técnicas matemáticas.
 - Verdadero
 - Falso
- c. Un método automatizado de análisis implica un software especial.
 - Verdadero
 - Falso
- d. Un bucle infinito no es un error característico.
 - Verdadero
 - Falso

12. Una prueba unitaria es:

- a. Una prueba aplicada al sistema.
- b. Una prueba aplicada a un grupo de módulos.
- c. Una prueba aplicada a un módulo.
- d. Todas las respuestas anteriores son correctas.

13. Relacione los siguientes elementos.

- | | |
|----------------------------------|----------------|
| a. Prueba de la ruta básica. | ● Caja blanca. |
| b. Prueba de bucle. | ● Caja negra. |
| c. Prueba de arreglo ortogonal | |
| d. Análisis de valores frontera. | |

14. Las métricas...

- a. no sirven para medir la calidad del software.
- b. pueden predecir el tamaño del sistema.
- c. tienen aplicación en momentos puntuales del ciclo de vida del software.
- d. Ninguna de las respuestas anteriores es correcta.

15. Complete la siguiente oración.

El _____ es una herramienta que transforma el _____ en un lenguaje entendible por el _____.

Capítulo 2. La orientación a objetos

Introducción

La programación orientada a objetos es un paradigma mediante el cual se intentan representar los objetos (y las interacciones entre ellos) de manera cercana al mundo real. El método para conseguir esto es realizar un proceso de abstracción, intentando captar la esencia de los objetos para materializar la representación de los mismos.

En este tema, se presentarán los nuevos conceptos asociados a este paradigma (herencia, polimorfismo, genericidad, sobrecarga de métodos, etc) y los principios que establecen sus bases, determinando las diferencias con la programación estructurada.

El capítulo concluirá con una enumeración de los lenguajes más comunes que hacen uso de este paradigma y con una ampliación de lo visto sobre UML, agregando los diagramas relativos a la programación orientada a objetos.

Principios de la orientación a objetos. Comparación con la programación estructurada

Los conceptos y principios derivados de la programación orientada a objetos pueden condensarse en los siguientes puntos:

- Ocultación de la información.
- El tipo abstracto de datos (ADT).
- Paso de mensajes.

Ocultación de información (information hiding)

La información que describe el estado de un objeto se almacena en forma de atributos, siendo estas variables cuya visibilidad externa puede definirse como pública o privada. Si un atributo es público es accesible desde el exterior del objeto, pero si es privado únicamente es alcanzable directamente desde el interior del mismo.



Ejemplo

Definir interfaces para acceso y modificación de atributos de una clase Persona. La información que se desea almacenar es Edad, Nombre y Sexo. Indicar también la visibilidad (público o privado) y el tipo de dato del atributo.



```
1 // Atributos, especificando tipo de datos y la visibilidad
2 private string nombre;
3 private string sexo;
4 private int edad;
5
6 // Métodos de modificación y acceso para los atributos
   anteriores (setters and getters)
7 public string getNombre();
8 public void setNombre(string nNombre);
9 public string getSexo();
10 public void setSexo(string nSexo);
11 public int getEdad();
12 public void setEdad(int nEdad);
13
```

El tipo abstracto de datos (ADT). Encapsulados de datos

El tipo abstracto de datos es un tipo de dato definido por el programador, especificando un conjunto de valores permitidos y una serie de operaciones aceptadas sobre dicho conjunto de valores. El tipo abstracto de datos debe ser definido con claridad y precisión, ya que es independiente de cualquier implementación. Dicho con otras palabras, la definición de un tipo abstracto de dato no va asociada a un lenguaje de programación en particular.

Abstracción

Consiste en quedarse con las características esenciales que definen un objeto, prescindiendo del resto de información (que es considerada como no relevante).

El encapsulado de datos consiste en representar en una clase el tipo abstracto de datos. Al terminar el proceso de encapsulación, se logra que la clase actúe como una caja negra: los detalles internos no son visibles al usuario, de tal manera que se sepa lo que hace la clase pero no cómo lo hace.

Un ejemplo de encapsulación podría ser el funcionamiento de la llave a distancia de un vehículo. Esta llave dispone de dos botones (apertura y cierre). Imagine que estos dos botones actúan como interfaz. A través de ellos, se puede realizar toda la funcionalidad que ofrece el objeto (abrir y cerrar el vehículo) sin necesidad de entender cómo funciona por dentro o de saber cómo realiza la comunicación con el coche.

Paso de mensajes

El paso de mensajes define la manera en que se comunican dos objetos. Esta comunicación puede ser para realizar una acción determinada o modificar el estado de un objeto. La forma es que esta comunicación se lleva a cabo en programación orientada a objetos es a través de las llamadas a métodos que forman parte de la interfaz de un objeto.

Recuerda

En programación orientada a objetos, un objeto está definido por la suma de los atributos y de sus métodos. Los atributos establecen el estado actual del objeto, mientras que los métodos abarcan a su funcionalidad.

Clases de objetos

En este apartado, se avanzará un poco en la definición de atributos y métodos, además de establecer una diferencia entre clase y objeto (conceptos que tienden a confusión).

Atributos, variables de estado y variables de clase

En el punto anterior, se expuso el concepto de atributo aplicado a un objeto o instancia de una clase. Los atributos están definidos en el tipo abstracto de datos y están representados por una clase, de tal manera que todas las instancias de esa clase presentan los mismos atributos. Estos pueden recibir el nombre de variables de estado (o variables de instancia), puesto que su valor define el estado del objeto.

Recuerda

Las postcondiciones y precondiciones están asociadas a los métodos, estableciendo lo que se espera de ellos y lo que ellos esperan recibir. La invariante, por otro lado, es una condición especial que deben cumplir todos los objetos de una clase.

Queda claro que las variables de estado contienen un valor para cada objeto (o instancia). Evidentemente, pueden coincidir, pero dicho valor no es compartido por todas las instancias de clase. Para ello están las variables de clase, de tal forma que puedan contener un valor común a todas las instancias de la clase.

Métodos. Requisitos e invariantes

Los métodos, de manera general, definen algo que el objeto puede hacer. Dicho con otras palabras: albergan toda la funcionalidad del objeto. Según el fin por el cual un método ha sido creado, se puede establecer una clasificación:

- **Constructores:** se llaman cuando se crea la instancia de una clase. Inicializan atributos o realizan operaciones previas a la creación de dicha instancia.
- **Destructores:** se llaman cuando se destruye el objeto. Normalmente liberan memoria.
- **Métodos selectores (*getters*):** acceden al valor de un atributo.
- **Métodos modificadores (*setters*):** modifican el valor de un atributo.
- **Métodos visualizadores:** muestran el estado del objeto (todos los atributos del mismo).

Los conceptos de requisitos e invariantes surgen del lenguaje Eiffel, creado por Bertrand Meyer y enfocado a la programación orientada a objetos. Este lenguaje es anterior a C# y Java, por ejemplo, y muchos de sus conceptos fueron introducidos en dichos lenguajes. También popularizó una metodología llamada Diseño por contrato, en la cual cada elemento es considerado como un participante en un contrato, de tal manera que tiene ciertas obligaciones y tareas que cumplir. Las restricciones que se pueden establecer son las siguientes:

- **Invariante:** condición que debe ser verdadera para todas las instancias de la clase (es decir, debe cumplirse para todos los objetos definidos a partir de ella).
- **Precondición:** algo que debe ser verdadero antes de realizar una operación.
- **Postcondición:** algo que debe ser verdadero después de ejecutar una operación.

Las precondiciones y postcondiciones ya se consideran al definir un método en una clase. Por ejemplo, `string getNombre()` no recibe ningún parámetro, pero sabemos que devuelve un tipo string. Lo mismo pasa con `void setNombre(string nuevoNombre)`, no devuelve absolutamente nada pero necesita un string para ejecutar su funcionalidad.

El caso de las invariantes es un poco más complejo. En muchos lenguajes no tienen implementación de manera nativa (como es el caso de Java y C#), aunque se puede hacer con herramientas externas o simular su comportamiento mediante el uso de aserciones (*assert*).

Recuerda

Las postcondiciones y precondiciones están asociadas a los métodos, estableciendo lo que se espera de ellos y lo que ellos esperan recibir. La invariante, por otro lado, es una condición especial que deben cumplir todos los objetos de una clase.

En el caso de que una invariante no se cumpla, se lanzará una excepción, que será el objeto de estudio del apartado siguiente.

Ejemplo

Definición de **assert** para evitar que la edad sea menor que 0 y mayor que 99.



```
1 public void verificarInvariant() {  
2     assert(edad < 0) : "Edad menor que 0";  
3     assert(edad > 99) : "Edad mayor que 0";  
4 }
```

Gestión de excepciones

Una excepción es un mecanismo de control para una situación de error. Resulta de gran utilidad, puesto que permite seguir con la ejecución del programa y tratar el error de una forma previamente implementada.

En el apartado anterior, se mencionó que una excepción se dispara en el caso de no cumplirse con un assert. Esta es una de las múltiples situaciones que puede provocar una excepción. Dentro de las excepciones disponibles en Java, se van a mencionar las siguientes:

- **ArithmaticException**: se realiza una operación aritmética no permitida.
- **ArrayIndexOutOfBoundsException**: desbordamiento durante el uso de un array. Se accede a posiciones no permitidas.
- **ClassCastException**: se quiere convertir un objeto de una clase en otra. Sin entrar en el uso de clases, es un comportamiento análogo a intentar meter una cadena de caracteres en un tipo entero.
- **ClassNotFoundException**: se pretende acceder a una clase que no existe o no está implementada todavía.
- **FileNotFoundException**: se quiere usar un fichero que no existe.
- **IOException**: error de entrada/salida (por ejemplo: un error recibiendo los datos de un fichero de texto o escribiendo información en el mismo).

Sabías que...

Aparte de las excepciones definidas por Java, existe la posibilidad de que el programador cree sus propias excepciones. Esto permitirá cubrir situaciones que no estén contempladas entre las propuestas por Java. Dicha opción será tratada en un punto avanzado del manual.

La estructura de una excepción en Java es la siguiente:



```
1 try {
2     // código problemático
3 } catch(tipoExcepcion1 e) {
4     // Tratamiento del tipoExcepcion1
5 } catch(tipoExcepcion2 e) {
6     // Tratamiento del tipoExcepcion1
7
8 ...
9
10 } catch(tipoExcepcionN e) {
11     // Tratamiento del tipoExcepcionN
12 } finally {
13     // código que se ejecutará siempre, se haya producido o
14     // no excepción
}
```

En caso de que haya más de un catch para atrapar una excepción, esta será recogida por el primero con el que empareje. Por eso, se recomienda ir poniendo los catch desde el más específico al más general, con el fin de evitar que una excepción sea tratada incorrectamente.

La inclusión del bloque *finally* no es obligatoria.



Sabías que...

Si existen dudas sobre la obligación o no de colocar una excepción y se está usando un entorno de desarrollo, este advierte inmediatamente al respecto.

Agregación de clases

Antes de hablar de la agregación de clases, hay que parar un momento en el concepto de **asociación**. En una asociación, una clase A tiene un atributo que hace referencia a una clase B (por ejemplo: un alumno puede tener varios profesores o un profesor puede tener a varios alumnos).

La **agregación** es igual que la asociación, pero con un pequeño matiz. Los dos objetos mantienen una existencia independiente; la no existencia de uno no implica que no exista el otro. Por ejemplo, supongamos una clase Cliente y una clase Empresa. La primera, en sus atributos, requiere de un array que hace referencia a todos sus clientes. Si no existiese la clase Empresa, todos los objetos de la clase Cliente seguirían existiendo.

Como contrapunto a la agregación existe la **composición**. En este caso, sí hay una existencia dependiente entre ambas clases. Si se parte de una clase Libro y de una clase Página (queda claro que un libro tiene entre sus atributos un array de páginas), no resulta difícil llegar a la conclusión de que una página no existirá en el momento en que desaparezca un objeto derivado de la clase Libro.

Este tipo de relaciones son conceptos puramente teóricos que pueden ser representados mediante diagramas de UML. No tienen implementación directa en el lenguaje de programación Java, ya que no distingue entre relaciones, al tratarlas todas de igual manera.

Recuerda

Una relación de asociación implica a una clase haciendo uso de otra para llegar a un objetivo. La variación “relación de agregación” indica que los objetos de las clases son independientes, mientras que en la “relación de composición” un objeto es siempre dependiente de otro (no existe sin él).

Diferencia entre clases y objetos

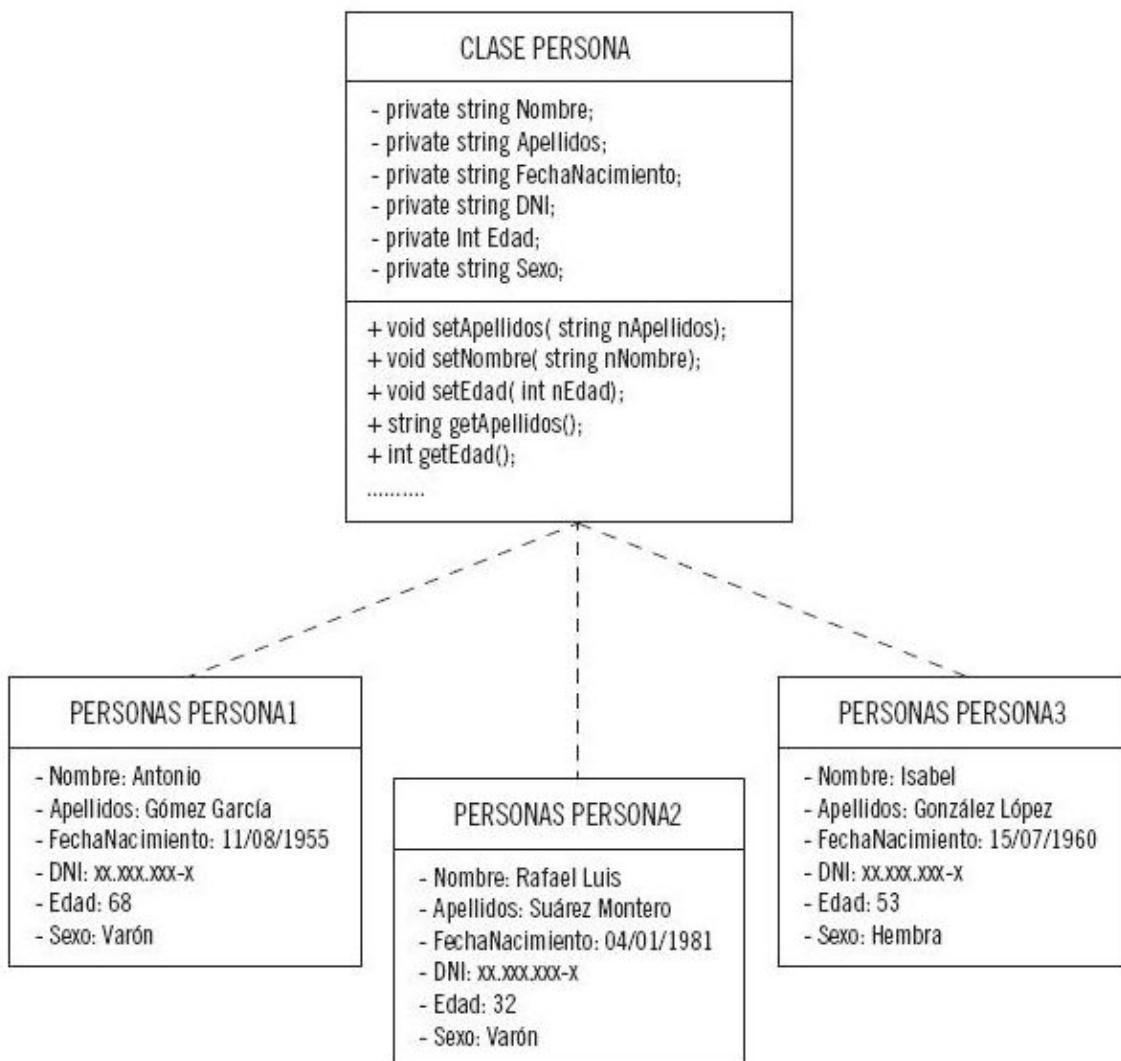
Clase

Es la implementación (en un lenguaje de programación) de la especificación (o tipo abstracto de datos) de todo aquello que define algo del mundo real. Si se pretende definir a una **Persona**, por ejemplo, se necesita su nombre, apellidos, fecha de nacimiento, DNI, sexo, etc., aparte de los métodos de acceso, modificación, constructores, etc.

Objeto

Es una instancia creada a partir de una clase. Una clase dice qué tienen que tener y cómo deben funcionar los objetos que se crean a partir de ella. Así, se pueden crear una serie de objetos (*Persona1*, *Persona2*, *Persona3* ... *PersonaN*), cada uno con sus propias características, pero respetando las reglas del juego que ofrece la clase **Persona**.

Ejemplo de clases y objetos



En la imagen, se aprecia cómo se han creado tres personas diferentes a partir de una única clase **Persona**. Cada objeto tiene sus propios valores para los atributos, pudiendo hacer uso de toda la funcionalidad implementada por la clase **Persona**.

Objetos

Se ha visto que un objeto es una instancia de una clase. A continuación, se verá cómo se realiza esta instancia y cómo se llaman los métodos implementados en la clase.

Creación y destrucción de objetos

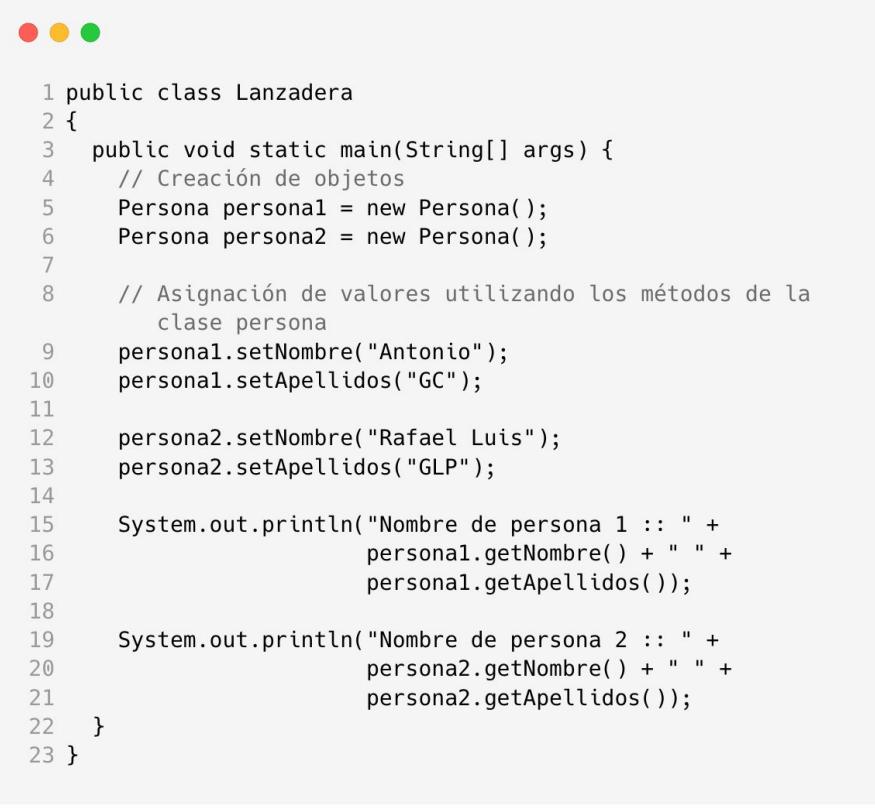
Para poder crear un objeto de una clase, primero hay que implementar la clase. En el siguiente código, se puede ver la implementación en Java de la clase **Persona**:

```
 1 public class Persona
 2 {
 3     private String Nombre;
 4     private String Apellidos;
 5     private String FechaNacimiento;
 6     private String DNI;
 7     private int Edad;
 8     private String Sexo;
 9
10    public String getNombre() {
11        return Nombre;
12    }
13
14    public void setNombre(String nombre) {
15        Nombre = nombre;
16    }
17
18    public String getApellidos() {
19        return Apellidos;
20    }
21
22    public void setApellidos(String apellidos) {
23        Apellidos = apellidos;
24    }
25
26    public String getFechaNacimiento() {
27        return FechaNacimiento;
28    }
29
30    public void setFechaNacimiento(String fechaNacimiento) {
31        FechaNacimiento = fechaNacimiento;
32    }
33
34    public String getDNI() {
35        return DNI;
36    }
37
38    public void setDNI(String dni) {
39        DNI = dni;
40    }
41
42    public int getEdad() {
43        return Edad;
44    }
45
46    public void setEdad(int edad) {
47        Edad = edad;
48    }
```

```
49
50     public String getSexo() {
51         return Sexo;
52     }
53
54     public void setSexo(String sexo) {
55         Sexo = sexo;
56     }
57
58 }
```

Se recomienda que cada clase en Java vaya en un fichero independiente. Así que este código se guardará en un fichero llamado “*Persona.java*”. Para poder hacer uso de esta clase y crear objetos de la misma hay que crear un pequeño código que tenga un método **main**.

Este método es el punto de entrada del programa, el que siempre busca Java para iniciar la aplicación que se está desarrollando, así que se creará un nuevo fichero, llamado “*Lanzadera.java*”, que contendrá el siguiente código:



```
1 public class Lanzadera
2 {
3     public void static main(String[] args) {
4         // Creación de objetos
5         Persona personal1 = new Persona();
6         Persona persona2 = new Persona();
7
8         // Asignación de valores utilizando los métodos de la
9         // clase persona
10        personal1.setNombre("Antonio");
11        personal1.setApellidos("GC");
12
13        persona2.setNombre("Rafael Luis");
14        persona2.setApellidos("GLP");
15
16        System.out.println("Nombre de persona 1 :: " +
17                            personal1.getNombre() + " " +
18                            personal1.getApellidos());
19
20        System.out.println("Nombre de persona 2 :: " +
21                            persona2.getNombre() + " " +
22                            persona2.getApellidos());
23    }
}
```

Al ejecutar los códigos anteriores en cualquier IDE (*NetBeans* o *Eclipse*, por ejemplo) o mediante la compilación manual a través de línea de comandos, se generará la siguiente salida por la consola:

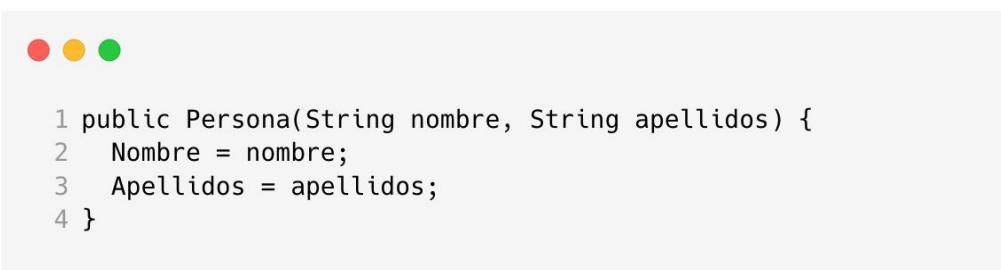


A screenshot of a Java application's console window. The tabs at the top are 'Problems', 'Javadoc', 'Declaration', and 'Console'. The 'Console' tab is selected. The output text is:
<terminated> Lanzadera [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (12/10/2013 20:12:20)
Nombre de persona 1 :: Antonio GC
Nombre de persona 2 :: Rafael Luis GLP

Observe que para crear un objeto de la clase **Persona** se ha tenido que usar la palabra reservada **new** e invocar un método que se llama como la clase, sin ningún parámetro. Este método es el constructor, absolutamente necesario cuando se pretende instanciar una clase.

¿De dónde sale ese método **Persona()**? No se encuentra definido dentro de la clase **Persona**. La respuesta es sencilla: si al definir una clase no se crea ningún constructor, automáticamente se invoca a un constructor por defecto que devolverá un objeto de esa clase.

Aquí entra en juego el concepto de sobrecarga de métodos. Este permite que se puedan definir tantos métodos con igual nombre como se consideren (siempre que tengan distintos parámetros). Tal idea es aplicable también a los constructores, puesto que no dejan de ser métodos. Por ejemplo, supóngase que se define el siguiente constructor en la clase **Persona**:



```
1 public Persona(String nombre, String apellidos) {  
2     Nombre = nombre;  
3     Apellidos = apellidos;  
4 }
```

El constructor es un método especial que se llama como la clase y es de tipo público (lógico, ya que se invoca fuera de la misma). Una vez que se define un constructor no vale seguir usando el constructor por defecto, así que si se mantiene el código que se tenía inicialmente en el método main de la clase Lanzadera dará error. Después de la modificación, el código de Lanzadera quedará así:



```
1 public class Lanzadera
2 {
3     public void static main(String[] args) {
4         // Creación de objetos
5         Persona personal1 = new Persona("Antonio", "GC");
6         Persona persona2 = new Persona("Rafael Luis", "GLP");
7
8         System.out.println("Nombre de persona 1 :: " +
9                             personal1.getNombre() + " " +
10                            persona1.getApellidos());
11
12        System.out.println("Nombre de persona 2 :: " +
13                             persona2.getNombre() + " " +
14                            persona2.getApellidos());
15    }
16 }
```

A continuación, se realizarán otras consideraciones sobre el código:

- El método **main** de la clase **Lanzadera** debe ser estático. Un método estático es aquel que se puede invocar sin necesidad de instanciar a la clase. Como **main** es el punto de entrada del programa y la clase **Lanzadera** no se instancia en ningún momento, para poder llamarlo se define estático.
- Para acceder a un determinado método de un objeto se usa el punto, indicando a continuación el nombre del método y los parámetros respectivos dentro de un paréntesis. En caso de que no tenga parámetros, el paréntesis va vacío.
- Para mostrar información por pantalla se usa el método **System.out.println**. Obsérvese que **out** es un atributo de la clase **System**. Este atributo es de la clase **PrintStream**, está definido como estático y contiene un método **println**. Idéntica situación a lo anteriormente comentado para el método **main**, no se realiza instanciación: se accede directamente al método.
- Las cadenas de caracteres literales que se van a mostrar por pantalla van entre comillas dobles. Hay que puntualizar que si son obtenidas a través de un método no lo llevan, ya que en caso contrario serían interpretadas como una cadena literal. Para enlazar cadenas se usa el carácter **+**.

Igual que existe el concepto de constructor, también existe el de destructor. Mientras que el constructor es el método que se invoca para crear un objeto, el destructor es llamado explícitamente cuando se desea terminar con un objeto.

Hay que aclarar que en Java no hay destructores. No son necesarios, puesto que su misión se lleva a cabo por el *Garbage Collector*, que será abordado en próximos puntos.

Llamada a métodos de un objeto

En el punto anterior se ha visto cómo se llamaba a los métodos **getNombre()** y **getApellidos()** para acceder a los atributos respectivos del objeto.

Recuérdese que hay que usar un punto inmediatamente después del nombre del objeto, para a continuación escribir el nombre del método. A continuación, se presentan otros ejemplos (respecto a persona1):

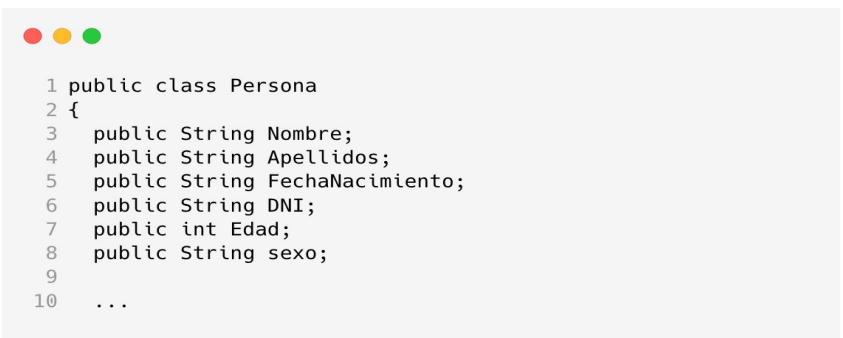
- Acceder a edad: **persona1.getEdad();**
- Modificar edad: **persona1.setEdad(39);**
- Modificar DNI: **persona1.setDNI("33333333-C");**
- Acceder a DNI: **persona1.getDNI();**

Nota

Los paréntesis tienen que ir siempre acompañando al nombre del método, independientemente de que tenga o no parámetros. En caso contrario, será tratado como un atributo público, dando error de compilación, puesto que no existirá.

Visibilidad y uso de las variables de estado

Al desarrollarse el principio de ocultación de información, quedó establecido que las variables de estado (o atributos) podían ser públicas o privadas. Si eran privadas, solo podían ser accedidas desde el interior de la clase mientras que, siendo públicas, son accesibles desde el exterior. En el ejemplo actual de la clase **Persona** todas las variables de estado son privadas y necesitan de un método para su acceso o modificación desde la clase **Lanzadera** (la que contiene el método **main**). Ahora se va a efectuar la siguiente modificación en la clase, poniendo todos los atributos como público (el resto continúa igual):



```
1 public class Persona
2 {
3     public String Nombre;
4     public String Apellidos;
5     public String FechaNacimiento;
6     public String DNI;
7     public int Edad;
8     public String sexo;
9
10    ...
}
```

Al final del método **main** de la clase **Lanzadera** deben añadirse las siguientes líneas.

```
1 ...
2
3 Persona persona3 = new Persona("Isabel", "LPP");
4
5 System.out.println("Nombre de persona 3 :: " +
6           persona3.Nombre + " " +
7           persona3.Apellidos);
8
9 ...
```

La salida es exactamente igual a las anteriores, solo que ahora se ha incluido a una persona más. En cambio, la forma de acceder al atributo **Nombre** y **Apellidos** ha sido diferente. Mientras que con **persona1** y **persona2** se usaron los métodos **getNombre()** y **getApellidos()** para obtener los valores de los atributos del objeto, con **persona3** se accede directamente, puesto que las variables de estado son públicas.

Sin embargo, este comportamiento es muy peligroso. En el ejemplo propuesto poco importa, pero imagínese que se trabaja con un atributo que almacena información trascendental (por ejemplo, un *password* o un número de cuenta) y ha sido definido como público. Estará completamente expuesto al exterior y podrá ser accedido/modificado, perdiéndose todo el control sobre los datos.

Referencias a objetos

Una referencia a un objeto consiste en usar el objeto de otra clase como atributo. Un ejemplo válido con la información de la que se dispone hasta ahora sería crear una nueva clase Java llamada **Familia**, la cual dispusiera de un array de elementos de la clase **Persona**. Esta clase sería definida así:



```
1 import java.util.ArrayList
2
3 public class Familia
4 {
5     private ArrayList<Persona> miembros;
6
7     public Familia() {
8         miembros = new ArrayList<Persona>();
9     }
10
11    public void agregar(Persona p) {
12        miembros.add(p);
13    }
14
15    public void mostrarFamiliar() {
16        for(int i = 0; i < miembros.size(); i++)
17            miembros.get(i).mostrar();
18    }
19 }
```

Aclaraciones sobre el código:

- En vez de usar un **array** (que se define poniendo corchetes después del nombre de variable usado para el atributo), se ha optado por aprovechar la clase **ArrayList** de Java. Esta clase encapsula la funcionalidad de una lista de elementos y debe importarse con la línea **import java.util.ArrayList**. De no existir la importación, se producirá un error de compilación.
- Este **ArrayList** debe inicializarse antes de su utilización. Por eso se implementa un constructor que crea un nuevo objeto de la clase **ArrayList**, llamando al constructor de la misma (sin parámetros) y usando la palabra reservada **new**.
- En el **ArrayList**, entre “<” y “>”, se especifica el tipo de dato que va a contener. En este caso, se pretende que la lista contenga objetos de la clase **Persona**.
- El método **add** de **ArrayList** añade un elemento al final de la lista. El método **get** hace lo contrario, extrae el elemento indicado en la posición pasada como parámetro; **size**, por otra parte, devuelve el tamaño del **ArrayList** (muy útil para recorrerlo con un bucle FOR).
- El método **mostrarFamilia** recorre la lista de miembros de la familia, invocando un nuevo método **mostrar** que será comentado a continuación.

La clase **Persona** debe disponer de un método **mostrar**. La función de este método es muy simple: mostrar toda la información relacionada con el objeto que lo invoca.



```
1 public void mostrar() {  
2     System.out.println();  
3     System.out.println("Nombre :: " + Nombre);  
4     System.out.println("Apellidos :: " + Apellidos);  
5     System.out.println("DNI :: " + DNI);  
6     System.out.println("Fecha de nacimiento :: " +  
7                         FechaNacimiento);  
8     System.out.println("Edad :: " + Edad);  
9     System.out.println("Sexo :: " + sexo);  
10 }
```

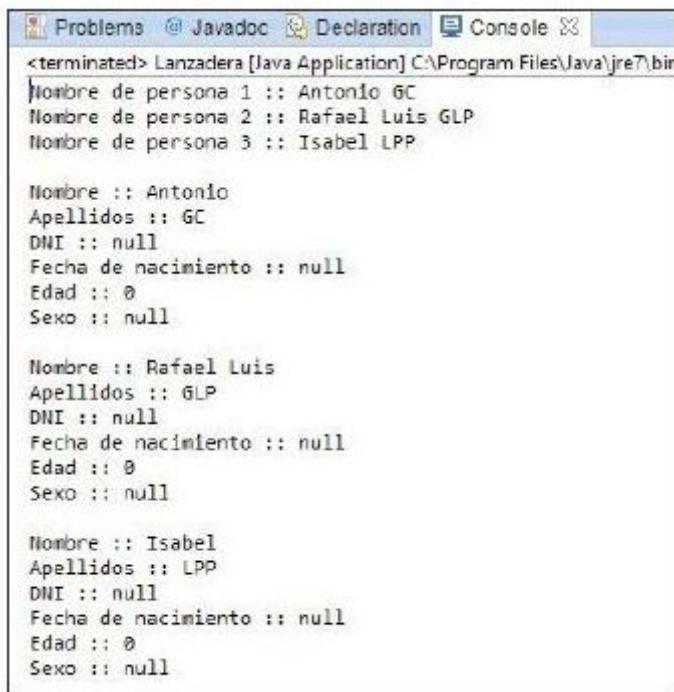
Por último, se modifica la clase **Lanzadera** con el fin de reflejar los cambios en **Persona** y la inclusión de la nueva clase **Familia**. El siguiente código debe agregarse al final del método **main**.

En este código, se crea un objeto de la clase **Familia**. El constructor inicializa el atributo **miembros**, que es un objeto de la clase **ArrayList**. A continuación, se añaden los tres objetos de la clase **Persona** creados previamente y se muestra toda la información de los mismos por la consola. El resultado será el siguiente:



```
1 ...  
2  
3 Familia familia = new Familia();  
4  
5 familia.agregar(personal1);  
6 familia.agregar(persona2);  
7 familia.agregar(persona3);  
8  
9 familia.mostrarFamilia();  
10  
11 ...
```

En este código, se crea un objeto de la clase **Familia**. El constructor inicializa el atributo **miembros**, que es un objeto de la clase **ArrayList**. A continuación, se añaden los tres objetos de la clase **Persona** creados previamente y se muestra toda la información de los mismos por la consola. El resultado será el siguiente:



```
Problems Javadoc Declaration Console
<terminated> Lanzadera [Java Application] C:\Program Files\Java\jre7\bin
Nombre de persona 1 :: Antonio GC
Nombre de persona 2 :: Rafael Luis GLP
Nombre de persona 3 :: Isabel LPP

Nombre :: Antonio
Apellidos :: GC
DNI :: null
Fecha de nacimiento :: null
Edad :: 0
Sexo :: null

Nombre :: Rafael Luis
Apellidos :: GLP
DNI :: null
Fecha de nacimiento :: null
Edad :: 0
Sexo :: null

Nombre :: Isabel
Apellidos :: LPP
DNI :: null
Fecha de nacimiento :: null
Edad :: 0
Sexo :: null
```

Persistencia de objetos

La persistencia de objetos se refiere a que, una vez concluida la ejecución del programa que los creó, el estado de estos objetos siga estando disponible para ser usado en el futuro. Es decir, se pretende que la información sobreviva a la ejecución del programa.

Esto, en Java, se puede lograr de varias maneras. Entre las más importantes, se encuentran el uso de base de datos y la serialización. Las bases de datos se tratarán más adelante, así que este punto se centrará en la serialización.

El mecanismo de serialización de Java se encarga de reflejar el estado de un objeto sobre una secuencia de bytes. El uso que se le pretenda dar depende del programador, pero si se quiere almacenar el estado del objeto hay que guardar esa secuencia en, por ejemplo, un fichero.

Ejemplo sobre persistencia de objetos

En el ejemplo siguiente, se serializará un objeto de la clase **Coche** (esta clase solo tiene tres atributos: **matrícula**, **modelo** y **motor**). El funcionamiento del ejemplo se representará en una clase **Lanzadera**. En esta clase, se creará un objeto de la clase **Coche**, con el fin de serializarlo en un fichero para posteriormente mostrarlo por pantalla (una vez recuperado):



```
1 import java.io.Serializable;
2
3 public class Coche implements Serializable
4 {
5     // Este campo es necesario cuando se serializa un
6     // objeto.
7     private static final long serialVersionUID = 1L;
8
9     private String Modelo;
10    private String Matricula;
11    private String Motor;
12
13    // Constructor con tres parámetros.
14    public Coche(String modelo, String matricula, String
15                  motor) {
16        Modelo = modelo;
17        Matricula = matricula;
18        Motor = motor;
19    }
20
21    // Método que muestra la información del vehículo
22    public void Mostrar() {
23        System.out.println();
24        System.out.println("Modelo :: " + Modelo);
25        System.out.println("Matricula :: " + Matricula);
26        System.out.println("Motor :: " + Motor);
27    }
28 }
```

```
1 public class Lanzadera {  
2     public static void main(String[] args) {  
3         // Se instancia la clase Coche con tres parámetros  
4         Coche = new Coche("Seat León", "xxx-XXX", "1.4 Gasolina");  
5  
6         try {  
7             // Flujo de escritura sobre el fichero "coche.ser"  
8             FileOutputStream fileOut = new FileOutputStream("coche.ser");  
9             ObjectOutputStream out = new ObjectOutputStream(fileOut);  
10              
11            // Se escribe el objeto  
12            out.writeObject(coche);  
13  
14            // Se cierra el flujo de escritura y el fichero-  
15            out.close();  
16            fileOut.close();  
17            System.out.println("Objeto serializado en \"coche.ser\"");  
18        }  
19        catch(IOException i) {  
20            i.printStackTrace();  
21        }  
22    }  
23  
24    try {  
25        // Flujo de entrada sobre el fichero "coche.ser".  
26        FileInputStream fileIn = new FileInputStream("coche.ser");  
27        ObjectInputStream in = new ObjectInputStream(fileIn);  
28  
29        // Se lee el objeto, convirtiéndolo a tipo Coche.  
30        Coche cocheLeido = (Coche) in.readObject();  
31  
32        // Cierra el flujo de entrada de datos y fichero.  
33        in.close();  
34        fileIn.close();  
35  
36        // Se muestra la información del objeto  
37        cocheLeido.Mostrar();  
38    }  
39    catch(IOException i) {  
40        i.printStackTrace();  
41    }  
42    catch(ClassNotFoundException e) {  
43        e.printStackTrace();  
44    }  
45 }  
46 }
```

En el ejemplo se ha realizado un *casting*, que consiste en convertir el tipo de un objeto. Por ejemplo, un número se puede convertir a tipo cadena de caracteres. En el código, se lee un objeto genérico con la función `in.readObject()`, y este, queriendo almacenarse como tipo **Coche**, debe ser convertido anteponiendo (**Coche**) después del operador de asignación.

Otros detalles (el uso de la palabra clave **implements**, la escritura y lectura en ficheros...) serán comentados en próximos puntos.

Optimización de memoria y recolección de basura (garbage collection)

En un lenguaje de programación como C la memoria se divide en tres segmentos, dispuestos de la siguiente manera:

- **Segmento 1:** almacena las variables globales (aquellas que son accesibles por todo el código del programa).
- **Segmento 2 (stack):** argumentos de funciones y variables definidas dentro de ellas.
- **Segmento 3 (heap):** variables creadas dinámicamente.

El sistema se encarga de gestionar automáticamente las partes correspondientes al segmento 1 y al segmento 2. Lo que sí requiere atención por parte del programador es la parte referida al segmento 3 (heap).

El **heap** (montón) almacena las variables creadas dinámicamente. Una variable dinámica es aquella cuyo tamaño puede cambiar en tiempo de ejecución del programa. El tamaño se refiere a las posiciones de memoria que ocupa. A más tamaño, más posiciones de memoria. Como ejemplo típico, se encuentra una cadena de caracteres, cuyo número exacto de elementos se sabrá en tiempo de ejecución gracias a una entrada de usuario (el usuario necesita una cadena de 20 caracteres, por ejemplo). Mediante funciones especiales, se reserva una posición en memoria para almacenar 20 caracteres.

Las variables que no son dinámicas, por el contrario, tienen un tamaño fijo definido en tiempo de compilación (antes de la ejecución del programa). Si se pretende usar un entero para almacenar una edad, se reserva un tamaño en memoria para almacenar un número y ese espacio reservado nunca cambiará.

Sabías que...

En lenguajes como C y C++ se pueden definir variables dinámicas modificando en tiempo de ejecución el espacio que ocupan en memoria mediante el uso de funciones especiales. El resto de variables ocuparán el espacio de memoria asignado antes de la ejecución del programa.

Las variables dinámicas son también usadas cuando se trabaja con objetos y clases. Al crearse un objeto, se produce una reserva de memoria en tiempo de ejecución en el área correspondiente. Un

detalle importante respecto a esta gestión de memoria es que, cuando se termine de usar la variable dinámica, hay que efectuar una liberación de memoria con el fin de que ese espacio esté disponible para otras variables.

La gestión de memoria en Java es bastante más sencilla. Siguen existiendo los tres segmentos de memoria, pero toda la parte referida a reserva y liberación de memoria es ejecutada de manera automática.

La liberación de memoria se realiza por medio del Garbage Collector. Este es un proceso de baja prioridad (es decir, se ejecuta cuando el procesador no tiene apenas carga de trabajo), que efectúa un barrido sobre la memoria dejando libre los espacios que ya no se necesitan.

Nota

Realmente, una variable dinámica es una variable (conocida como puntero) que se sitúa en el segmento 2 (stack), pero que contiene una referencia a una posición de memoria situada en el segmento 3 (heap).

El Garbage Collector de Java sabe los datos que tiene que eliminar, ya que comprueba que no existe ningún puntero desde el stack que los refiriérase.

Actividad 5

Desarrolle una clase que recoja las características de un ordenador, estructurado este en cuatro grandes bloques: Pantalla, Disco Duro, Memoria y Procesador. La información que se desea almacenar para cada uno de los bloques es:

- Pantalla: Resolución, Pulgadas y Píxeles.
- Disco duro: Velocidad, Capacidad y Tiempo medio de acceso.
- Procesador: Número de núcleos, Velocidad y Bits.
- Memoria: Tamaño, Tecnología y Velocidad.

La clase estará acompañada por un método mostrar, que imprimirá toda la información en la consola. El código será estructurado como se considere, pero intentando aprovechar el concepto de asociación de clases.

Herencia

Este capítulo definirá todos los conceptos asociados a la herencia, mostrando las posibilidades que implica esta idea para la reusabilidad y el mantenimiento de código.

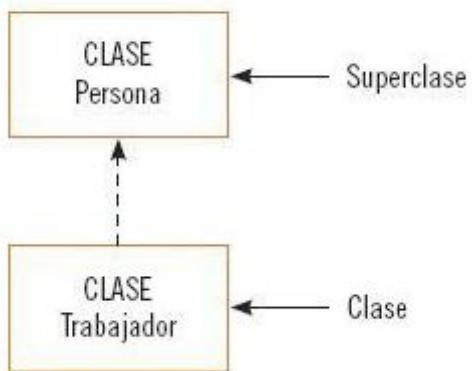
Concepto de herencia. Superclases y subclases

La herencia es un mecanismo mediante el cual una clase puede heredar la funcionalidad y atributos de otra.

La clase fue definida como una especificación para un tipo abstracto de datos en la cual se indicaban los atributos y la funcionalidad que debía disponer cualquier implementación hecha en un lenguaje de programación.

Como se dijo anteriormente, una clase puede heredar de otra para tomar su funcionalidad y características. Pues bien, si existe una clase A de la cual hereda una clase B, se dice que la clase A es la superclase de la clase B.

Ejemplo de clase y superclase



La siguiente clase, **Trabajador**, hereda de **Persona**. Es decir, un trabajador es todo lo que es una persona pero con los atributos propios que define la clase **Trabajador**. Un detalle muy importante es el constructor de la clase que hereda, que siempre debe llamar al constructor de la superclase para inicializar sus atributos.



```
1 public class Trabajador extends Persona{
2     private String LugarTrabajo;
3     private String NumSS;
4
5     public Trabajador(String lugarTrabajo, String numSS, String nombre, String
6                         apellidos){
7         super(nombre, apellidos);
8         LugarTrabajo = lugarTrabajo;
9         NumSS = numSS;
10    }
11
12    public String getLugarTrabajo() {
13        return LugarTrabajo;
14    }
15
16    public void setLugarTrabajo(Stringf lugarTrabajo){
17        LugarTrabajo = lugarTrabajo;
18    }
19
20    public String getNumSS() {
21        return NumSS;
22    }
23
24    public void setNumSS(String numSS){
25        NumSS = numSS;
26    }
27 }
```

En el constructor de **Trabajador** se aprecia cómo se invoca al constructor de **Persona** usando la palabra reservada **super**. Esta invocación al constructor de la superclase debe ser siempre la primera línea dentro del cuerpo del constructor de la clase que hereda.

Tipo de herencia

En programación orientada a objetos se consideran dos tipos de herencia. La herencia simple es la contemplada en el punto anterior, mediante la cual una clase heredera únicamente de otra clase. También existe la herencia múltiple, que será tratada en el siguiente apartado.



En la herencia, además de los métodos, también se heredan los atributos de la superclase.

Herencia múltiple

La herencia múltiple es un concepto de programación orientada a objetos que sugiere que una misma clase puede heredar de dos o más clases al mismo tiempo, tomando la funcionalidad de

ambas. La herencia simple, en contraste, solo habla de heredar de una clase. Teóricamente, es perfectamente posible, pero, a nivel práctico, existe un problema.

Ejemplo de clase y superclase



El problema derivado de la imagen propuesta es que se puede producir ambigüedad. Supóngase que existen métodos o atributos con el mismo nombre, como por ejemplo un método que se llame **ObtenerUbicacion()**. ¿Con qué ubicación se queda el programa, con la del lugar de estudio o la del lugar de trabajo? El programador sabe lo que quiere hacer mientras está implementando el código, pero el programa no tiene forma de conocer con qué funcionalidad heredada debe quedarse en el caso de existir ambigüedad.

Esta situación la resuelve C++ indicando a qué clase pertenece un método antes de usarlo. Pero en el caso de Java no se permite la herencia múltiple, al menos su aplicación directa. No obstante, hay una manera de implementar la idea de manera muy parecida, mediante el uso de una clase especial llamada Interfaz. Esta clase será tratada más adelante.

Clases abstractas

Una clase abstracta es aquella cuya implementación está incompleta, ya que alguno de sus métodos es abstracto. Para que un método sea abstracto no debe tener cuerpo. En otras palabras, se sabe que el método se llama X y que necesita (o no) una serie de parámetros definidos, pero no implementa la funcionalidad. Un detalle importante es que una clase abstracta no puede ser instanciada; siempre se debe heredar para transmitir su funcionalidad.

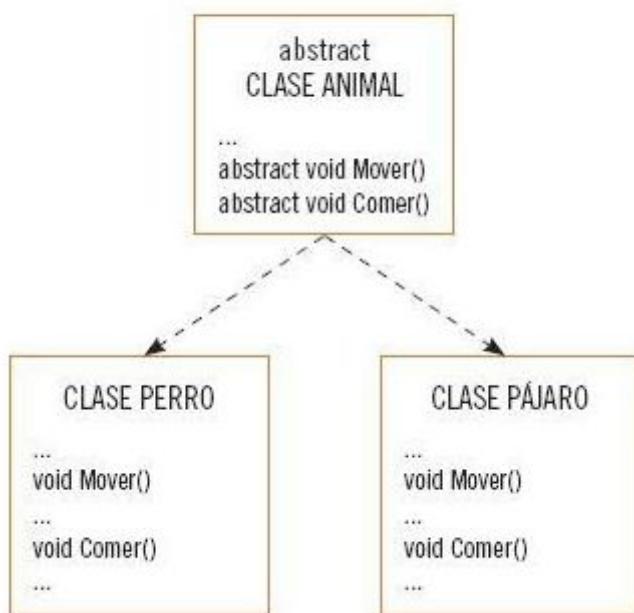
```
 1 abstract public class Animal{  
 2     private String Nombre;  
 3     private String Habitat;  
 4     private String Reino;  
 5  
 6     abstract void Mover();  
 7     abstract void Comer();  
 8  
 9     public String getNombre() {  
10         return Nombre;  
11     }  
12  
13     public void setNombre(String nombre){  
14         Nombre = nombre;  
15     }  
16  
17     public String getHabitat() {  
18         return Habitat;  
19     }  
20  
21     public void setHabitat(String habitat){  
22         Habitat = habitat;  
23     }  
24  
25     public String getReino() {  
26         return Reino;  
27     }  
28  
29     public void setReino(String reino){  
30         Reino = reino;  
31     }  
32  
33     void mostrarInfo() {  
34         System.out.println();  
35         System.out.println("Nombre :: " + Nombre);  
36         System.out.println("Habitat :: " + Habitat);  
37         System.out.println("Reino :: " + Reino);  
38     }  
39 }
```

En el código se aprecia que la clase tiene la palabra reservada **abstract** en su definición. Los métodos de acceso y de modificación están definidos como siempre, pero hay dos que no tienen cuerpo: **void Mover()** y **void Comer()**, estando acompañados también por **abstract**.

¿Qué implica esto? Que cuando una clase herede de una clase abstracta debe implementar forzosamente los métodos definidos como abstractos. A primera vista, puede no tener utilidad práctica, pero obsérvese la clase abstracta **Animal**. ¿Por qué los métodos **Mover** y **Comer** se han definido como abstractos? La respuesta a esta última pregunta es muy sencilla.

La clase **Animal** intenta implementar una especificación común que abarca a todos los animales. Un animal tiene un nombre, un reino y un hábitat, y estos atributos tienen los clásicos métodos de acceso y modificación. También está claro que un animal come y se mueve, pero no todos los animales lo hacen de la misma manera. Así que hubiese sido un gran error establecer una implementación de estos métodos a un nivel tan general como puede ser animal, ya que eso hubiese ocasionado que todos los animales tuvieran un comportamiento común. Como ejemplo muy ilustrativo, piénsese en que un perro y un pájaro se mueven de manera muy diferente.

Ejemplo de clase abstracta



Sabías que...

Mientras que los métodos abstractos se deben implementar, los que no lo son se pueden reimplementar. Si es necesario invocar el método original de la superclase para extender su funcionalidad, se usará **super** (igual que con el constructor).

Interfaces

Llegado este momento, se puede presentar el concepto de interfaz. Una interfaz es una especie de clase abstracta en la que todos los métodos son abstractos. Es decir, una clase abstracta debe ser implementada al completo, pero respetando las reglas que propone: todos los métodos se tienen que llamar de la manera indicada, debiendo recibir los tipos de parámetros descritos. La mayor utilidad de las interfaces es que permiten simular la idea de herencia múltiple. Solo se puede heredar de una clase, pero se pueden implementar todas las interfaces que se quiera.

Las interfaces se definen con la palabra reservada **interface** y se implementan en una clase con **implements**. No pueden llevar constructor, ni tampoco atributos, puesto que una interfaz no

conlleva ningún comportamiento asociado. Como mucho, pueden usar variables estáticas, que se comportan como constantes. Una interfaz **IAnimal** sería definida así:

```
● ● ●  
1 public interface IAnimal{  
2     abstract void Mover();  
3     abstract void Comer();  
4 }
```

Una clase **Perro** que implementa la interfaz **IAnimal** quedaría de la siguiente forma:

```
● ● ●  
1 public class Perro implements IAnimal{  
2     private String Raza;  
3     private int Peso;  
4     private int Edad;  
5  
6     public Perro(String raza, int peso, int edad){  
7         Raza = raza;  
8         Peso = peso;  
9         Edad = edad;  
10    }  
11  
12    public void Mover() {  
13        // Implementación de este método para la clase Perro  
14    }  
15  
16    public void Comer() {  
17        // Implementación de este método para la clase Perro  
18    }  
19 }
```

📎 Recuerda

1. Una clase abstracta no se puede instanciar.
2. Los métodos abstractos deben ser implementados en las clases que hereden de una clase abstracta.
3. Si todos los métodos de una clase abstracta son abstractos, se considera una interfaz.
4. Una clase puede implementar todas las interfaces que quiera.
5. Las interfaces definidas con interfaces no llevan constructor ni atributos. Las constantes son válidas.

Polimorfismo y enlace dinámico (dynamic binding)

El polimorfismo es una característica que permite a un objeto adoptar diversas formas, quedando establecido en tiempo de ejecución (mediante el enlace dinámico) el método que debe usar. El polimorfismo surge cuando dos clases implementan el mismo método. No es una situación mediante la cual una clase implementa un método abstracto de su superclase, ahora se sobrescribe un método que ya ha sido implementado. Se muestra un ejemplo partiendo de una clase **Animal** y de una clase **Perro**.

```
● ● ●

1 public class Perro {
2     private String Raza;
3
4     public Perro(String raza){
5         Raza = raza;
6     }
7
8     public void Alimentar() {
9         System.out.println("Un perro " + Raza + " está comiendo!");
10    }
11 }
```

```
● ● ●

1 public class Mascota extends Perro {
2     private String Nombre;
3
4     public Mascota(String raza){
5         super(raza);
6         Nombre = nombre;
7     }
8
9     public void Alimentar() {
10        System.out.println("La mascota " + Nombre + " está comiendo!");
11    }
12 }
```



```
1 public class Lanzadera {  
2     public static void main(String[] args) {  
3         Perro p1 = new Perro("Labrador");  
4         Perro p2 = new Mascota("Labrador", "Muri");  
5  
6         p1.Alimentar();  
7         p2.Alimentar();  
8     }  
9 }
```

Como se observa en el código, se han definido dos objetos de la clase **Perro**. ¿No sería lógico que el método **Alimentar** de ambos mostrase la misma salida por pantalla? En eso consiste el enlace dinámico. Al instanciarse **p2** como **Mascota**, se decide en tiempo de ejecución que sea el método **Alimentar** de **Mascota** (y no el método **Alimentar** de **Perro**) el que se ejecute.

Directrices para el uso correcto de la herencia

Un uso correcto de la herencia en programación orientada a objetos se puede concretar en los siguientes puntos:

- Se pretende representar una relación del tipo “es-un”. Esto es básicamente lo visto hasta ahora: una persona “es-un” trabajador, un coche “es-un” vehículo, un perro “es-un” animal, etc.
- Se quiere reutilizar el código de la clase base para futuras clases. Si este código es ya plenamente funcional y está depurado ahorra muchísimo trabajo al programador.
- Los cambios globales que afecten a todas las clases se harán en la clase base. Por ejemplo: si todas las clases heredan un método que utiliza una fórmula para calcular un valor, en el caso de que esta fórmula sufra modificación, bastará solo con modificar el método en la superclase.

Consejo

Cuando se descubre el concepto de herencia se corre el riesgo de establecer un árbol de jerarquía demasiado grande. Se terminan viendo clases innecesarias, de tal manera que lo que se puede hacer en dos-tres niveles de jerarquía se va de las manos. La herencia es una posibilidad, no una obligación.

Actividad 6

Desarrolle una estructura de clases que sirva como base para una tienda de animales. La tienda va a trabajar con animales, en concreto con perros y peces. Las consignas son las siguientes:

- Se desea almacenar su nombre común, su precio y especie.
- Para cada animal se podrá imprimir una ficha por pantalla con toda la información.
- Los perros pueden presentar certificado de pedigrí.
- Para los peces se especificarán las clases de peces con las que no son compatibles, aparte de las unidades que se den de alta en el sistema.

Modularidad

La modularidad es una metodología de diseño que se basa en dividir el programa en módulos más pequeños con una funcionalidad bien definida. La aplicación de este concepto a la programación orientada a objetos implica que los módulos se representan en la forma de clases, paquetes o bibliotecas.

Librerías de clases. Ámbito de utilización de paquetes

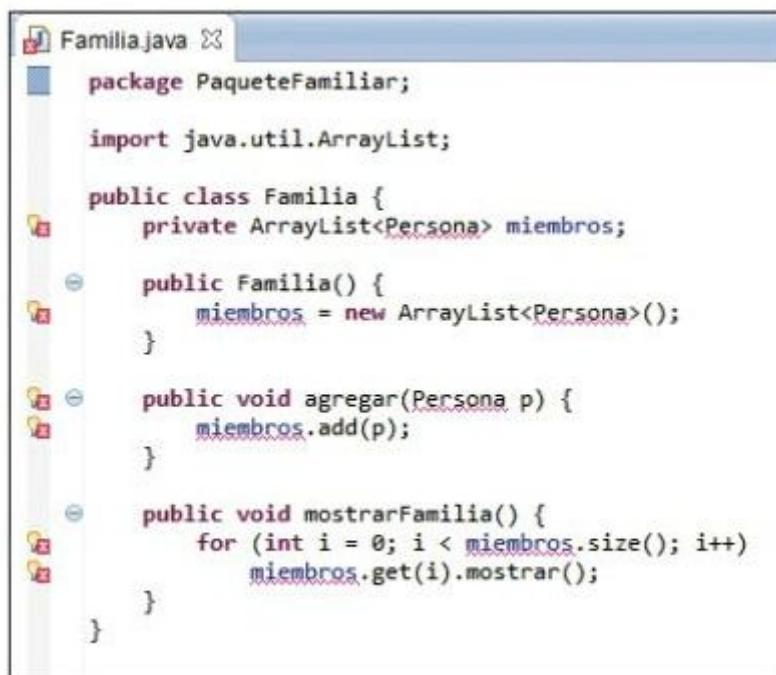
Las librerías en Java reciben el nombre de paquetes (*packages*). El sentido de una librería es agrupar varias clases que comportan una funcionalidad o utilidad común.

Java tiene a disposición del programador muchísimas clases y la mayor parte tienen que ser importadas antes de su uso. En anteriores apartados, se ha realizado la importación de clases. Recuérdese como, por ejemplo, la línea ***import java.util.ArrayList;*** permitía el uso de la clase **ArrayList** en el código.

Si no se especifica nada, las clases que se agreguen a un proyecto no estarán referidas a ningún paquete. Este es el caso de la siguiente imagen. Se crearon tres clases para el proyecto **EjemploPersona** y todas forman parte del paquete por defecto.



Para modificar el paquete al cual pertenece una clase, se utiliza la opción **package** más el nombre del paquete. A continuación, se agrega la opción **package PaqueteFamiliar** a la clase **Familia**, de tal forma que quede así:



```

Familia.java ×
package PaqueteFamiliar;

import java.util.ArrayList;

public class Familia {
    private ArrayList<Persona> miembros;

    public Familia() {
        miembros = new ArrayList<Persona>();
    }

    public void agregar(Persona p) {
        miembros.add(p);
    }

    public void mostrarFamilia() {
        for (int i = 0; i < miembros.size(); i++)
            miembros.get(i).mostrar();
    }
}

```

Si se modifica el paquete al cual pertenece una clase, hay que tener especial cuidado con el resto de clases que interactúan con ella. Por ejemplo, respondiendo a la actividad planteada antes, la clase **Persona** produce un error al querer usarla dentro de la clase **Familia**, puesto que no se puede acceder a ella tal cual están planteados los paquetes en el proyecto.

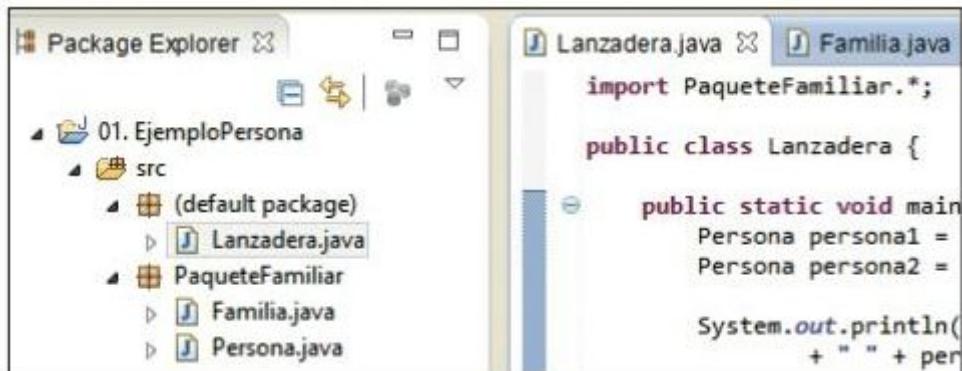


Este mismo ejemplo sirve para establecer el ámbito de visibilidad. Las clases dentro de un paquete solo son visibles entre ellas. Así, modificando la clase **Persona** para que forme parte del mismo paquete **PaqueteFamiliar** se solucionarán los problemas de visibilidad en la clase **Familia** para hacer referencia a **Persona**.



Sin embargo, ahora surge un nuevo problema. ¿Cómo se puede hacer que **Lanzadera** pueda disponer del contenido de **PaqueteFamiliar** sin necesidad de estar incluida dentro de dicho paquete?

La respuesta se vio al principio del apartado: simplemente, agregando la línea `import PaqueteFamiliar.*;` al principio de la clase **Lanzadera**.



Recuerda

Import sirve para importar un paquete dentro de una clase, mientras que *package* permite definir el paquete al cual pertenece una clase.

Ventajas de la utilización de módulos o paquetes

Las ventajas de la utilización de módulos o paquetes son:

- **Simplificación del diseño:** al aplicarse el “divide y vencerás” se descompone el problema en subproblemas más sencillos. Esto implica también una menor complejidad algorítmica.
- **Reusabilidad de código:** los módulos o paquetes generados pueden reutilizarse en otros programas, simplemente realizando la importación correspondiente.
- **Mayor facilidad de depuración:** al estar el programa estructurado en módulos o clases, es más fácil realizar pruebas, simplificando la acotación del error en el caso de que se generen.
- **Mayor facilidad de mantenimiento:** lógicamente, si hay que realizar extensiones o modificaciones resulta un proceso mucho menos traumático.

Divide y vencerás

Técnica que consiste en dividir un problema en problemas más pequeños. Es la base de muchos algoritmos, aplicando una solución al problema inicial que lo irá descomponiendo hasta que los problemas resultantes sean lo suficientemente pequeños como para poder resolverlos.

Genericidad y sobrecarga

Con la genericidad y la sobrecarga se termina el estudio de los conceptos asociados a programación orientada a objetos contemplados en este manual.

Concepto de genericidad

La aplicación del concepto de genericidad permite que un método acepte objetos de diferente tipo, para los cuales tendrá un mismo comportamiento. Por ejemplo: supóngase un método que muestre los elementos de un **ArrayList** (independientemente que sean enteros, caracteres o decimales):

```
1 import java.util.ArrayList;
2
3 public class ElementosGenerico<T> {
4     private ArrayList<T> Lista;
5
6     public ElementosGenerico(ArrayList<T> lista){
7         Lista = lista;
8     }
9
10    public ArrayList<T> getData() {
11        return Lista;
12    }
13
14    public void setData(T elemento, int pos){
15        Lista.add(pos, elemento);
16    }
17
18    public void mostrar() {
19        for(int i = 0; i < Lista.size(); i++)
20            System.out.println(Lista.get(i) + " ");
21
22        System.out.println();
23    }
24 }
```

Obsérvese que la clase **ElementosGenerico** está parametrizada, llevando un elemento **T** que acompaña al nombre. Al instanciar esa clase hay que definir la clase del tipo de dato que va a soportar, lo cual no afecta en absoluto a la funcionalidad interna de la clase, pues trabaja con el

elemento genérico **T**. Es importante el matiz clase del tipo de dato (de ahí que se haya utilizado la clase **Integer**, que representa a los enteros de tipo int).

Recuerda

<T> actúa como una especie de comodín, adaptándose a la clase que representa el tipo de dato utilizado al crear la instancia.

Ahora se mostrará una clase **Lanzadera**, la cual creará dos objetos de la clase **ElementosGenerico**, siendo uno de tipo entero (clase **Integer**) y otro de tipo cadena (clase **String**):

```
● ● ●  
1 import java.util.ArrayList;  
2 import java.util.Arrays;  
3  
4 public class Lanzadera {  
5     public static void main(String[] args) {  
6         ArrayList<Integer> enteros =  
7             new ArrayList<Integer>(Arrays.asList(1, 2, 3));  
8  
9         ArrayList<String> cadenas =  
10            new ArrayList<String>(Arrays.asList("AA", "BB", "CC"));  
11  
12         ElementosGenerico<Integer> listaEnteros =  
13             new ElementosGenerico<Integer>(enteros);  
14  
15         ElementosGenerico<String> listaCadenas =  
16             new ElementosGenerico<String>(cadenas);  
17  
18         listaEnteros.mostrar();  
19         listaCadenas.mostrar();  
20     }  
21 }
```

En esta clase, primero se definen dos **ArrayList** (especificando el tipo correspondiente, **Integer** o **String**), cada uno con tres elementos. Para que el constructor de la clase **ArrayList** acepte una lista de parámetros, hay que usar el método **Arrays.asList**, para lo cual hay que importar la clase **Arrays**. De esta manera, se inicializa el **ArrayList** cuando se crea la instancia, evitando tener que agregar los elementos uno a uno.

A continuación, se crean los objetos de la clase **ElementosGenerico**, (indicando de nuevo el tipo de elemento). Acto seguido se invocan los métodos **mostrar** de cada uno de los objetos, generando la siguiente salida:



The screenshot shows a Java IDE's console window. The title bar says "Problems @ Javadoc Declaration Search Console". The main area displays the output of a Java application named "Lanzadera". The output shows three lines of text: "1 2 3" and "AA BB CC", which are the elements of the two ArrayLists. The console window has a standard OS X style with red, yellow, and green close buttons at the top left.

Como se puede apreciar en la imagen, la salida por consola da el resultado esperado. Se recorren los dos **ArrayList** y se muestran sus elementos. Únicamente ha sido necesaria una clase para recorrer los dos objetos, independientemente del tipo de dato que contienen.

Concepto de sobrecarga. Tipos de sobrecarga

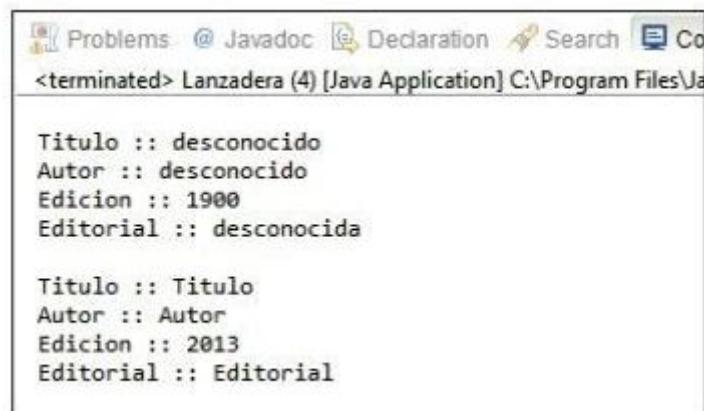
La sobrecarga consiste en definir métodos con el mismo nombre, pero con diferentes parámetros. Este concepto ya ha sido aplicado durante el capítulo, al definir diferentes constructores de una clase según los parámetros que se van a recibir al instanciar la clase. He aquí otro ejemplo:

En el código anterior, hay dos ejemplos de sobrecarga. El primero es el constructor: existe un constructor sin parámetros y otro que recibe tres. El otro ejemplo es el método **setEditorial**. Existe una versión que no recibe parámetro y asigna a **Editorial** el valor “desconocido”, mientras que la segunda versión recibe un parámetro (que será el que asignará a **Editorial**). A continuación, se muestra el código de la clase **Lanzadera**:



```
1 public class Lanzadera {  
2     public static void main(String[] args) {  
3         Libro libro1 = new Libro();  
4         Libro libro2 = new Libro("Titulo", "Autor", 2013);  
5  
6         libro1.setEditorial();  
7         libro2.setEditorial("Editorial");  
8  
9         libro1.mostrar();  
10        libro2.mostrar();  
11    }  
12 }
```

Se crean dos objetos de la clase **Libro**, el primero con el constructor por defecto y el segundo con el parametrizado. A continuación, se invoca el **setEditorial** sin parámetro para el primer objeto y la versión con parámetro para el segundo. La salida por consola se muestra a continuación:



```
Problems @ Javadoc Declaration Search Co
<terminated> Lanzadera (4) [Java Application] C:\Program Files\Ja
Titulo :: desconocido
Autor :: desconocido
Edicion :: 1900
Editorial :: desconocida

Titulo :: Titulo
Autor :: Autor
Edicion :: 2013
Editorial :: Editorial
```

Los resultados obtenidos son los lógicos. El primer libro tiene un **Titulo**, un **Autor** y una **Edicion** desconocida debido al constructor, y una **Editorial** también desconocida fruto del método **setEditorial**. El segundo libro tiene valores asignados, debido a que ha usado las otras versiones del constructor y del método.

Nota

El tipo de sobrecarga vista anteriormente se llama sobrecarga de métodos. Existe otra variante, la sobrecarga de operadores, mediante la cual el tipo de dato de los operadores determina la operación a aplicar. Esto permite, por ejemplo, sumar dos matrices de la misma forma que se suman números enteros. No tiene implementación en Java, pero si en otros lenguajes como C++.

Comparación entre genericidad y sobrecarga

Cuando exista la duda entre usar genericidad o sobrecarga, la clave es la siguiente: Si para todos los tipos de datos se requiere una acción común, hay que usar genericidad. En caso contrario, se recomienda el uso de sobrecarga.

Si se prefiere usar sobrecarga, se cae en el riesgo de describir bastantes métodos que realizan exactamente lo mismo, pero utilizando tipos de datos diferentes. El problema es que en cualquier momento se puede requerir modificar el método, lo cual implica retocar todas las implicaciones.

Desarrollo orientado a objetos

Java no es el único lenguaje orientado a objetos, pero sí se puede decir que es de los más populares. Existen multitud de herramientas y lenguajes que permiten trabajar con esta filosofía.

Lenguajes de desarrollo orientado a objetos de uso común

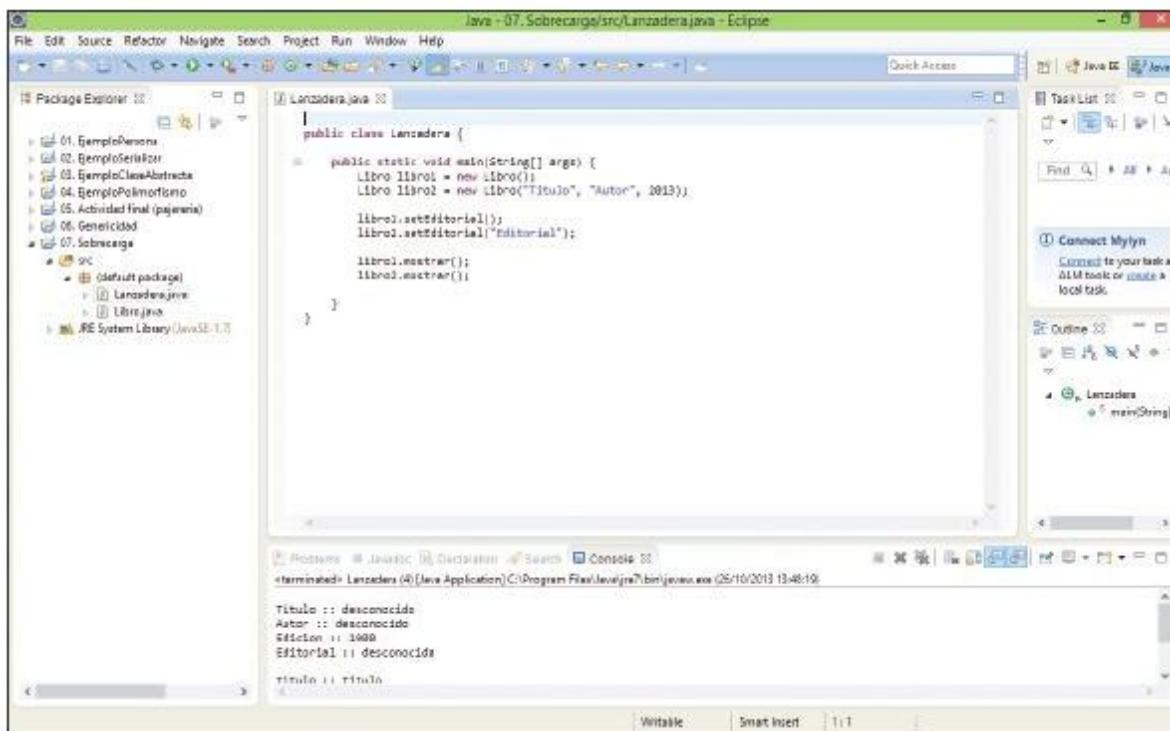
Se puede elaborar la siguiente lista de lenguajes de desarrollo orientado a objetos:

- **Java:** desarrollado en 1995. Sus desarrolladores se inspiraron mucho en C y C++. Entre sus ventajas, se encuentra que el código puede ejecutarse en cualquier máquina que tenga instalada una máquina virtual de Java.
- **C++:** es la versión extendida de C, surgida en 1983 para aplicar la orientación a objetos. El código en C++ debe ser recompilado en cada máquina para generar el ejecutable. Inicialmente, tenía más desempeño que Java, pero con el paso de los años el tiempo de ejecución de ambos se ha acercado mucho.
- **Python:** es un lenguaje de programación muy sencillo de usar para profanos en la materia, puesto que usa tipado dinámico. El tipo de la variable puede cambiar el tiempo de ejecución, mientras que en Java y C, por ejemplo, el tipo de dato que tenga una variable será inmodificable. Este lenguaje aparece en 1991 y debe de ser compilado para generar un ejecutable dependiente de la máquina (igual que C).
- **Eiffel:** máximo exponente de los lenguajes orientados a objetos. Su primera versión data de 1985 y está muy asociado al principio de diseño por contrato (la relación entre los elementos se trata de manera similar a un contrato de negocios).
- **C#:** el equivalente a Java, pero desarrollado por Microsoft. Utiliza el framework .NET, también de Microsoft.

Herramientas de desarrollo

La herramienta de desarrollo por excelencia para Java es *Eclipse*, seguida muy de cerca por *NetBeans*. *Eclipse* surgió en el año 2004 y actualmente va por la versión 2020-12 R. Entre sus características más importantes se pueden nombrar las siguientes:

- Está gestionado por la Fundación Eclipse, que es un consorcio de empresas lideradas por IBM.
- Entorno de trabajo dividido en paneles, cada uno con una funcionalidad. Estos paneles se pueden ocultar, redimensionar o distribuir a gusto del usuario. Asistente para la creación de proyectos y elementos de los mismos, automatizando ciertos aspectos que de otra manera pueden ser muy repetitivos. Generación automática de métodos getters y setters (estos eran los métodos que accedían al valor de un atributo y que permitían su modificación).
- Integración con otros lenguajes de programación, tales como C++, PHP, Python, etc.
- Completo editor, que incluye coloreado de sintaxis y formato automático. Posibilidad de autocompletado de código, con documentación de referencia integrada.
- Integración con los principales gestores de base de datos.
- Control de pruebas unitarias con JUnit.
- Control de versiones con CVS.
- Integración con los frameworks más populares de Java, como *Hibernate* y *Spring*.
- Instalación de plugins, que le confieren al IDE más características y funcionalidad.



Lenguajes de modelización en el desarrollo orientado a objetos

Anteriormente, se presentaron una serie de diagramas de UML, gran parte de los cuales son de aplicación a programación orientada a objetos. En este tema, se hará un repaso de estos diagramas, aparte de presentar algunos exclusivos.

Uso del lenguaje unificado de modelado (UML) en el desarrollo orientado a objetos

UML es un lenguaje que define un estándar para modelar aspectos tales como el comportamiento y la arquitectura del sistema. Consta de un total de trece diagramas, siendo los más importantes los siguientes:

- **Diagramas de casos de uso:** sirven para representar la funcionalidad del sistema desde la perspectiva del usuario.
- **Diagramas de implementación:** muestran la distribución física del sistema software, incluyendo la relación con componentes hardware.
- **Diagramas de actividad:** representan el comportamiento dinámico del sistema durante una actividad.
- **Diagramas de estado:** representan los diferentes estados de un objeto, indicando las transiciones entre los mismos.
- **Diagramas de secuencia:** representan el comportamiento del sistema, enfatizando en la comunicación entre los objetos.

- **Diagramas de comunicación:** representan el comportamiento del sistema, haciendo hincapié en la relación entre objetos por encima de la comunicación.

Diagramas para la modelización de sistemas orientados a objetos

A continuación, se analizarán los dos diagramas exclusivos de UML para la programación orientada a objetos.

Diagramas de clase

Modelan una clase, indicando sus atributos, métodos y relaciones con otras clases.



Recuerda

Una clase es una especificación de un concepto del mundo real. Consta de dos partes claramente diferencias: los atributos (que caracterizan el estado) y los métodos (que definen la funcionalidad).

En un diagrama, una clase se representa por un bloque dividido en tres niveles: el primero el nombre de la clase, el segundo sus atributos y el tercero sus métodos. Hay que especificar el tipo de dato en los atributos, en los parámetros de los métodos y en el dato que devuelvan.

Para indicar una relación entre clases en forma de herencia se utiliza una flecha discontinua, apuntando una clase hacia la otra de la cual hereda.



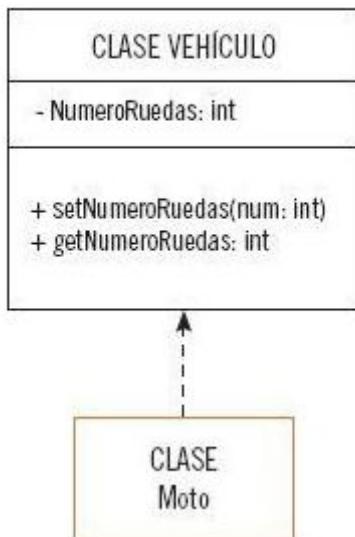
Ejemplo

Representar el diagrama de una clase Vehículo y su relación con la clase Moto.

La clase Vehículo presenta las siguientes características:

- Atributos:
 - *NumeroRuedas*: int.
- Métodos:
 - *setNumeroRuedas(int num)*
 - *getNumeroRuedas() : int*

Ejemplo de diagrama de clases



Diagramas de objetos

Un diagrama de objetos muestra el estado (parcial o completo) de un sistema en un momento dado, representando a un conjunto de objetos y los valores de sus atributos.



Ejemplo

Representar tres instancias de la clase Perro (p1, p2, y p3). Los datos de las tres instancias son las siguientes:

- p1:
 - Raza: San Bernardo.
 - Peso: 70 kilos.
 - Edad: 5 años.
- p2:
 - Raza: Labrador.
 - Peso: 42 kilos.
 - Edad: 6 años.
- p3:
 - Raza: Pastor Alemán.
 - Peso: 45 kilos.
 - Edad: 4 años.

Ejemplo de diagrama de objetos

p1: Perro	p2: Perro	p3: Perro
- Raza: San Bernardo - Peso: 70 - Edad: 5	- Raza: Labrador - Peso: 42 - Edad: 6	- Raza: Pastor Alemán - Peso: 70 - Edad: 5

Resumen

La programación orientada a objetos presenta muchas características que permiten un mayor acercamiento al mundo real, trabajando con clases que se instancian como objetos. Las clases especifican cómo tiene que ser un objeto y qué tiene que llevar para comportarse como tal. Esta especificación es independiente del lenguaje de programación utilizado.

Se puede establecer una jerarquía de clases cuando se observe que un concepto del mundo real con el que se pretende trabajar requiere un comportamiento ya definido en una clase, pero agregando funcionalidad propia. En ese caso, se creará una nueva clase que herede de la que se estime y especifique los nuevos atributos y métodos requeridos.

Los objetos representados presentan una serie de atributos que determinan su estado, los cuales, idealmente, no deben ser visibles al exterior de manera directa. Se utilizarán métodos especialmente definidos para ello, tanto para modificar como para acceder a su valor.

Para implementar este paradigma de programación existen muchos lenguajes, siendo los más utilizados Java y C++. Como entorno de desarrollo se recomienda Eclipse, aunque existen alternativas.

Para modelar y representar un sistema orientado a objetos existen los diagramas de UML, siendo todos perfectamente válidos, con mención especial a los diagramas de clases y a los de objetos.

Ejercicios de repaso y autoevaluación

- De las siguientes afirmaciones, indique cuál es verdadera o falsa.
 - El estado de un objeto es descrito por los métodos que implementa.
 - Verdadero
 - Falso
 - Se recomienda que los atributos de un objeto deben ser directamente accesibles desde el exterior.
 - Verdadero
 - Falso
 - La abstracción consiste en quedarse con las características esenciales que definen un objeto.
 - Verdadero
 - Falso
 - La manera en que se comunican dos objetos se denomina sobrecarga.
 - Verdadero
 - Falso

2. Complete la siguiente oración

La encapsulación provoca que los detalles _____ de la _____ no sean visibles al _____.

3. ¿Cuál es la diferencia entre variables de estado y variables de clase?

4. ¿Qué afirmaciones respecto a las excepciones son ciertas?

- a. En Java se puede introducir un bloque opcional Finally después de la excepción.
 - b. Las excepciones son un mecanismo de control de errores.
 - c. Una excepción es capturada por el primer catch con el que empareje.
 - d. Todas las respuestas anteriores son correctas.

5. El destructor debe ser implementado en Java...

- a. siempre.
 - b. únicamente si se dispone de poca memoria en el ordenador.
 - c. cuando se use programación orientada a objetos.
 - d. Ninguna de las respuestas anteriores es correcta.

6. En caso de no definirse el constructor...

- a. se crea uno por defecto.
- b. el constructor es obligatorio.
- c. no pasa nada, pero esa clase no se podrá instanciar nunca.
- d. Ninguna de las respuestas anteriores es correcta.

7. Establezca la diferencia entre clase y objeto

8. De las siguientes afirmaciones, indique cuál es verdadera o falsa.

- a. En la agregación, los dos objetos mantienen una existencia independiente.
 - Verdadero
 - Falso
- b. En la composición, los dos objetos mantienen una existencia dependiente.
 - Verdadero
 - Falso
- c. La asociación de clases implica que un atributo de una clase A hace referencia a una clase B.
 - Verdadero
 - Falso
- d. Las diferentes relaciones entre clases no tienen implementación directa en Java.
 - Verdadero
 - Falso

9. La herencia múltiple...

- a. permite heredar de dos o más clases al mismo tiempo.
- b. no tiene implementación directa en Java.
- c. puede ocasionar problemas si dos métodos tienen el mismo nombre en las superclases.
- d. Todas las respuestas anteriores son correctas.

10. Una clase abstracta...

- a. tiene todos sus métodos implementados.
- b. no se puede instanciar.
- c. tiene que heredar obligatoriamente de una superclase.
- d. Todas las respuestas anteriores son correctas.

11. De las siguientes afirmaciones, indique cuál es verdadera o falsa.

- a. El polimorfismo permite definir objetos de diferentes clases con el mismo nombre.
 - Verdadero

- Falso
- b. El enlace dinámico (dynamic binding) tiene lugar en tiempo de compilación.
 - Verdadero
 - Falso
- c. Hay dos tipos de herencia: simple y múltiple.
 - Verdadero
 - Falso
- d. Las clases de un package son visibles desde fuera sin necesidad de importación
 - Verdadero
 - Falso

12. La sobrecarga...

- a. implica crear muchos métodos con igual nombre y parámetros, pero con distinta implementación.
- b. se puede aplicar a constructores.
- c. tiene solo aplicación en clases abstractas.
- d. Ninguna de las respuestas anteriores es correcta.

13. Complete la siguiente oración

La _____ permite definir clases parametrizadas para implementar una funcionalidad _____.

14. ¿Todos los diagramas UML son de aplicación a orientación a objetos?

- a. No, cada uno tiene su campo específico.
- b. Sólo los diagramas de clase y los diagramas de objeto.
- c. Todos los diagramas son aplicables a programación orientada a objetos.
- d. Únicamente los diagramas de clase, objeto, secuencia, actividad y eventos.

15. El método super()...

- a. llama al constructor de la superclase.
- b. puede incluirse en cualquier parte del código.
- c. nunca acepta parámetros.
- d. Ninguna de las respuestas anteriores es correcta.

Capítulo 3. Arquitecturas web

Introducción

Hoy en día, muchísimos programas ofrecen su funcionalidad a través de la red. El usuario contacta con una aplicación disponible en un servidor, a través de un cliente que, por lo general, es un navegador web.

En este capítulo, se contemplará la estructura básica de una aplicación web y el modelo de capas que propone la base teórica en la cual se apoya lo que hoy se conoce como Internet, para terminar con un recorrido sobre plataformas y herramientas orientadas al desarrollo en el servidor de aplicaciones web.

Concepto de arquitectura web

En el sentido más general, la arquitectura web abarca toda la tecnología utilizada para poner en marcha un servidor que permita a un usuario determinado visualizar contenidos a través de Internet.

En el contexto de este manual, la arquitectura web se refiere a la programación de una aplicación web, lo cual incluye tener un servidor operativo (Apache, por ejemplo) y una base de datos (en MySQL o cualquier otro lenguaje de base de datos con el cual se disponga de conector).

El núcleo de la aplicación se desarrollará, básicamente, en un lenguaje como PHP o Java (mediante JSP), estando acompañado por código HTML y por JavaScript. Más adelante, se ampliará este último punto, pues el código implementado se puede ejecutar en el cliente o en el servidor dependiendo del lenguaje utilizado.

Estructura básica de una aplicación web



La estructura básica de una aplicación web se ha representado en la imagen. A grandes rasgos, el funcionamiento del sistema será el siguiente:

1. El usuario realiza una petición a través del cliente (por ejemplo, un navegador web como *Chrome* o *Firefox*) sobre el servidor que contiene la aplicación web.
2. Este servidor, por lo general, dispone de un acceso a una base de datos que contiene información necesaria para la ejecución de la aplicación.
3. Despues de obtener/modificar la información de la base de datos, el servidor envía la respuesta al cliente.
4. El cliente renderiza la respuesta suministrada por el servidor y se la presenta al usuario.

El modelo de capas

En este apartado, se estudiará el modelo OSI (Open System Interconnection), el cual describe la interconexión entre sistemas de comunicaciones. Este modelo está recogido por el ISO/IEC 7498-1 y estructura las funciones que se llevan a cabo durante la comunicación en siete capas lógicas.

Capa física

Define los medios físicos que van a posibilitar la comunicación, especificando sus características materiales y eléctricas e indicando cómo se manejan las señales eléctricas y electromagnéticas.

Esta capa codifica una serie de bits en señales, para posteriormente enviarlas a través del medio de transmisión.

Capa de enlace de datos

Proporciona un medio de transferencia entre dos nodos que están conectados directamente, detectando y corrigiendo posibles errores de la capa física. La información se estructura en bloques, en los cuales se incluye una dirección (la dirección MAC).

Capa de red

Se encarga de la comunicación entre dos nodos que no tienen que estar conectados directamente. Los datos llegarán desde un origen a un destino, según la ruta seleccionada por esta capa. La capa de red puede funcionar de dos maneras diferentes:

- **Mediante datagramas:** los paquetes de datos son independientes, sin que haya un establecimiento previo de comunicación.
- **Mediante circuitos virtuales:** se establecerá una conexión entre los dos nodos que se quieren comunicar, de tal forma que los paquetes sigan el circuito establecido.

Por otra parte, la capa de red puede ofrecer dos tipos de servicios:

- **Servicios orientados:** el primer paquete lleva la dirección de destino, estableciendo la ruta de los paquetes de la misma conexión.
- **Servicios no orientados:** cada paquete lleva la dirección de destino, eligiéndose la ruta según una técnica de encaminamiento.

Respecto a las técnicas de encaminamiento, hay que decir que estas dependen, generalmente, del estado de la red. El camino se elige de manera dinámica, según diversos criterios. Los dispositivos que realizan el encaminamiento son los *routers*. Entre los protocolos que se aplican a esta capa, se encuentra el IP.

IP (*Internet Protocol*)

Principal protocolo de Internet. Su tarea es entregar paquetes desde una máquina origen a una máquina destino determinada por la dirección IP contenida en los paquetes.

Capa de transporte

Es la capa encargada de transportar los datos desde la máquina origen a la máquina destino (independientemente de su localización) haciendo que esta transferencia esté libre de errores. Existen dos protocolos en la capa de transporte, dependiendo de la orientación a conexión:

- **UDP:** protocolo no orientado a conexión. Cada datagrama contiene la información necesaria para llegar a su destino, pero no existe confirmación de entrega ni control de errores. Además, los paquetes pueden no llegar en el mismo orden en que son enviados.
- **TCP:** protocolo orientado a conexión. Los paquetes son entregados en orden, existiendo control de errores. También facilita la distinción entre distintas aplicaciones que actúan sobre una misma máquina mediante el uso de puertos.

Capa de sesión

Esta capa mantiene el enlace entre dos máquinas durante la transmisión de archivos. Para ello, la capa de sesión ofrece los siguientes servicios:

- **Control de sesión:** conocimiento del receptor, del transmisor y de la sesión en sí.
- **Control de concurrencia:** evitar que dos operaciones sobre el mismo recurso se ejecuten al mismo tiempo.
- **Mantenimiento de puntos de reanudación:** para reanudar la comunicación en el caso de que la transmisión caiga.

Capa de presentación

La finalidad de esta capa es representar la información de una manera que permita que los datos sean legibles (ya sean imágenes, sonido, caracteres, vídeo, etc.). Permite, entre otras funcionalidades, el cifrado de información.

Capa de aplicación

Permite a las aplicaciones acceder a los servicios proporcionados por las otras capas. Esta es la capa sobre la que se construyen aplicaciones de correo electrónico, navegadores, clientes de FTP, etc. Un detalle muy importante de esta capa es que contiene el protocolo HTTP (*Hipertext Transfer Protocol*), fundamentado en un modelo cliente-servidor mediante petición y respuesta.

Plataformas para el desarrollo en las capas del servidor

Para el desarrollo en el lado del servidor, existen muchísimas plataformas, de las cuales se enumerarán tres de las más populares. La elección entre una u otra dependerá de diversos criterios (curva de aprendizaje, entornos de programación, si es o no lenguaje propietario, etc.):

- **PHP:** son las siglas de PHP HiperText Preprocessor (originalmente Personal Homes Pages). Este popular lenguaje se presenta en forma de scripts embebidos dentro de páginas HTML. Es abierto, gratuito y relativamente fácil de usar con las nociones clásicas de programación, soportando actualmente orientación a objetos. Un código en PHP generalmente es interpretado, aunque también se puede compilar. Entre sitios web desarrollados con PHP, se encuentra Facebook.
- **ASP.NET:** Plataforma propietaria de Microsoft, haciendo uso del framework .NET. La presencia de este framework le da una gran integración con las aplicaciones de escritorio desarrolladas con este. La diferencia respecto a PHP es que el código en ASP debe ser compilado antes de su utilización en el servidor. Esto acarrea ventajas, ya que la ejecución será más rápida y no se podrá ver el código embebido en el HTML.
- **JSP:** la plataforma Java Server Pages es desarrollada por Sun Microsystems (actualmente Oracle). A igual que ASP, un código en JSP debe ser compilado y ejecutado en el servidor para que sea accesible por el cliente. Aquí también se incluyen tecnologías derivadas de Java, como los servlets.

Código embebido

Aquel que se incluye dentro de otro usando etiquetas especiales, estando ambos códigos escritos en diferentes lenguajes de programación.

Herramientas de desarrollo orientadas a servidor de aplicaciones web

Una aplicación web necesita de varios componentes funcionando en el servidor. Algunos son necesarios, pero otros se pueden instalar como módulos opciones. En este punto se avanzará sobre estos detalles.

Tipos de herramientas

A continuación, se repasarán los componentes de la aplicación web, indicando qué tipos de aplicaciones pueden ser útiles para su configuración:

Infraestructura de red

Es todo el hardware sobre el cual se va a apoyar el servidor. Básicamente consiste en un ordenador (especialmente configurado) y una conexión a Internet. No se avanzará más en este aspecto ni se comentarán herramientas, ya que sobrepasa el alcance de este manual.

Servidor web

Como se ha dicho, la infraestructura de red abarca la parte hardware. El servidor web es el componente software que hay que instalar y configurar para que el servidor sea operativo. Existen varios programas que permiten a un ordenador actuar como servidor, siendo los más importantes *Apache* e *IIS (Internet Information Server)*.

Apache

Creado originalmente para *Linux*, pero también existen versiones para *Windows*. Es multiplataforma, gratuito y de código abierto. La configuración por defecto es muy válida si se pretende trabajar en un entorno de desarrollo en modo local, pero para realizar ajustes avanzados hay que editar en archivos de configuración presentes en una carpeta especial dentro del directorio de instalación del servidor. Una herramienta simple (tipo Bloc de Notas) es suficiente para realizar la edición.

Nota

Un servidor web tiene que tener acceso al exterior, pero tampoco es estrictamente necesario. Por ejemplo, un mismo equipo puede tener instalado el servidor web, la base de datos y actuar también como cliente de acceso a través del navegador de Internet.

Para acceder al servidor web instalado en un mismo equipo, normalmente se pone <<http://localhost>> en la barra de direcciones del navegador.

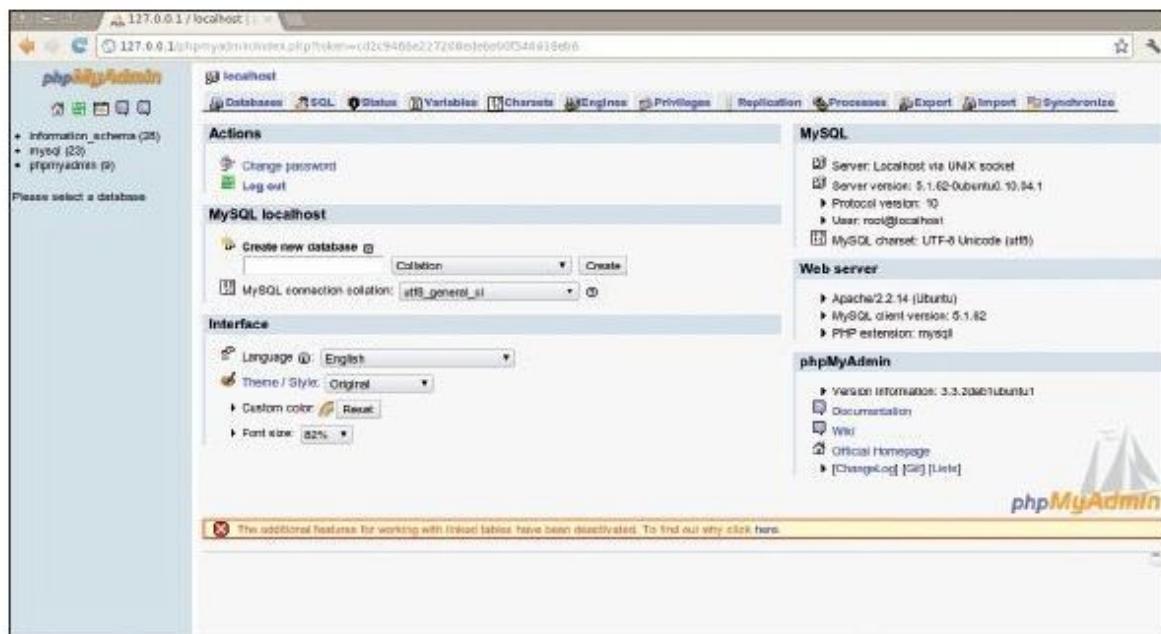
Internet Information Server

Disponible únicamente para *Windows*, con lo cual ya restringe bastante las opciones. Además, el código del servidor no es abierto, perteneciendo a Microsoft. La configuración de un sitio web IIS se realiza a través de un panel de control, accesible desde el propio servidor.

Servidor de base de datos

Una aplicación web requiere de una base de datos para funcionar. Al igual que los lenguajes, las tecnologías son muchas, aunque MySQL suele ser una elección perfectamente válida para la mayoría de las elecciones. Si se instala MySQL junto con Apache, se dispone de una herramienta llamada *PhpMyAdmin*.

Esta herramienta permite gestionar las bases de datos de una manera más sencilla, siendo una alternativa al modo consola. *PhpMyAdmin* es accesible a través de un navegador, por ejemplo, lo cual permite gestionar la base de datos remotamente.



Nota

La gestión de la base de datos también se puede realizar a través de un modo consola. Esta, a veces, es la única opción, puesto que no todos los servidores son accesibles a través de la página web.

Aplicación web

Es el programa que está instalado en el servidor, “a la escucha” de peticiones por parte del cliente. Esta aplicación está desarrollada teniendo como base algunas de las plataformas comentadas en el apartado anterior o bien una combinación de ambas. Para su codificación se usará uno de los IDE comentados anteriormente, como *NetBeans* o *Eclipse*.

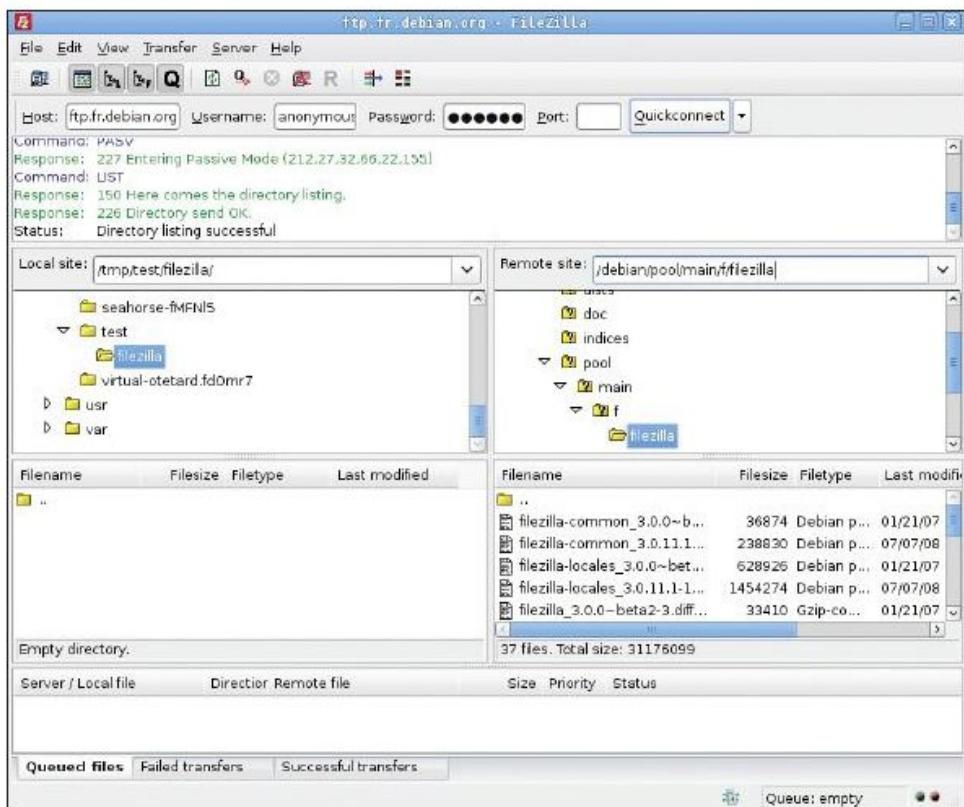
Otras herramientas/programas

Apache Tomcat

Es una especie de servidor complementario a *Apache*, aunque realmente se trata de un contenedor de aplicaciones. Es requerido para realizar aplicaciones web en el servidor con Java. Será tratado más adelante.

Cliente/Servidor FTP

Normalmente, una aplicación web se desarrolla en un sitio diferente al que la contendrá en última instancia. Para subir los archivos es común que el servidor web definitivo tenga configurado un servidor FTP, al cual se accederá con programas tales como *Filezilla*.



Servidor de correo

Es una buena idea que se disponga de un servidor de correo, con el fin de proporcionar un soporte que permita realizar envíos de correo a través de la aplicación web.

Depurador

Normalmente, se incluye dentro del propio IDE, aunque es posible que haya que realizar algunas modificaciones dentro de la configuración del servidor para que esté plenamente activo. Los errores en una aplicación web pueden ser muy difíciles de localizar, así que se hace casi imprescindible el uso de un depurador para la ejecución de código paso a paso.



El servidor proporciona el servicio, mientras que el cliente accede al mismo.

Extensibilidad. Instalación de módulos

Los programas instalables que hacen que un ordenador actúe como servidor vienen con una serie de módulos instalados por defecto que proporcionan la funcionalidad básica. Sin embargo, se dispone también de otros módulos, desactivados inicialmente, que proporcionan nuevas características.

El usuario puede acceder a <http://httpd.apache.org/docs/2.4/es/mod/> para ver una lista completa de todos los módulos disponibles en la última versión de Apache.

El fichero que se encarga de cargar los módulos cada vez que se ejecuta el servidor de *Apache* es “*httpd.conf*”. Dependiendo del sistema operativo (*Windows* o *Linux*) y de su distribución, se puede encontrar localizado en un sitio diferente:

- **Linux**: “/etc/httpd/conf/httpd.conf”.
 - **Windows**: disponible en la carpeta **conf**, dentro del directorio de instalación del servidor. Suponiendo que se tenga instalado Apache (el proceso se comentará en un próximo apartado), el directorio de instalación por defecto será “c:\xampp\apache\conf\httpd.conf”.

```
C:\xampp\apache\conf\httpd.conf - Notepad++  
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?  
  
httpd.conf  
61 # Dynamic Shared Object (DSO) Support  
62 #  
63 # To be able to use the functionality of a module which was built as a DSO  
64 # have to place corresponding 'LoadModule' lines at this location so the  
65 # directives contained in it are actually available _before_ they are used  
66 # Statically compiled modules (those listed by 'httpd -l') do not need  
67 # to be loaded here.  
68 #  
69 # Example:  
70 # LoadModule foo_module modules/mod_foo.so  
71 #  
72 LoadModule access_compat_module modules/mod_access_compat.so  
73 LoadModule actions_module modules/mod_actions.so  
74 LoadModule alias_module modules/mod_alias.so  
75 LoadModule allowmethods_module modules/mod_allowmethods.so  
76 LoadModule asis_module modules/mod_asis.so  
77 LoadModule auth_basic_module modules/mod_auth_basic.so  
78 #LoadModule auth_digest_module modules/mod_auth_digest.so  
79 #LoadModule authn_anon_module modules/mod_authn_anon.so  
80 LoadModule authn_core_module modules/mod_authn_core.so  
81 #LoadModule authn_dbd_module modules/mod_authn_dbd.so  
82 #LoadModule authn_dbm_module modules/mod_authn_dbm.so  
83 LoadModule authn_file_module modules/mod_authn_file.so  
84 #LoadModule authn_socache_module modules/mod_authn_socache.so  
85 #LoadModule authnz_ldap_module modules/mod_authnz_ldap.so  
86 LoadModule authz_core_module modules/mod_authz_core.so  
87 #include "conf.d/*.conf"
```

Como se puede apreciar, los módulos se cargan con la directiva **LoadModule**. En este fichero, los comentarios empiezan con el carácter almohadilla #, así que si se quiere que el sistema cargue un módulo simplemente hay que descomentar la línea.

No se extenderá más la explicación de este tema, ya que forma parte más de la teoría y práctica de administración de servidores que de la programación de una aplicación web. Lo único con lo que se debe quedar el lector es con la idea de que los módulos se pueden activar y desactivar según las necesidades y la funcionalidad buscada.

Sabías que...

Aparte de los módulos propios, existen módulos desarrollados por terceros que extienden la funcionalidad del servidor.

Técnicas de configuración de los entornos de desarrollo, preproducción y producción

Antes de comentar las técnicas de configuración, se establecerá la diferencia entre estos tres entornos:

- **Entorno de desarrollo:** entorno sobre el cual se desarrolla la aplicación, realizándose sobre el mismo la mayor parte de las pruebas.
- **Entorno de preproducción:** entorno muy parecido a producción. Resulta esencial cuando no se tiene la certeza de que una aplicación funcione en producción, por circunstancias diversas (por ejemplo: el hardware puede cambiar sustancialmente).
- **Entorno de producción:** entorno en el cual el software/aplicación se pone a disposición del usuario final. Un producto erróneo que alcance un entorno de producción puede tener drásticas consecuencias.

Nota

En el mundo de las aplicaciones web, existe el término, sandbox, que se refiere a copias de sitios determinados (*PayPal*, por ejemplo). Estas copias son proporcionadas por las propias compañías para facilitar un entorno lo más parecido a producción que sirva para el desarrollo por parte de terceros.

Una secuencia válida para realizar un desarrollo para servidor sería la siguiente:

1. La aplicación se desarrolla en una máquina local.
2. Esta aplicación es probada de manera local.
3. La aplicación sube a un servidor de preproducción y se vuelve a probar.
4. La aplicación, una vez recibe el visto bueno, pasa a producción.

Como la aplicación web va a necesitar de una base de datos, es una buena idea realizar una copia de la base de datos original sobre la que va a trabajar y apoyar todo el desarrollo sobre la duplicada. Nunca se debe desarrollar directamente con la base de datos de producción (a no ser que sea estrictamente necesario y extremando las precauciones).

Funcionalidades de depuración

Una de las características más potentes de un servidor consiste en llevar a cabo un registro de la actividad y eventos. Los archivos que recogen esta información reciben el nombre de *logs*.

Existen muchísimos tipos de *logs* con información diversa. Entre los más útiles, se encuentran los de errores, que recogen cualquier error que se presente al realizar peticiones sobre el servidor. Por eso, cuando haya un error en apariencia desconocido, se recomienda mirar en dichos logs de errores para obtener información sobre el mismo.

Error Log	
Last 200 Error Log Messages in reverse order:	
[Sat Aug 6 00:31:16 2005]	[error] [client 207.46.98.53] client denied by server configuration: /home/nicesoda/public_html/laishiu/life/live
[Sat Aug 6 00:28:53 2005]	[error] [client 207.46.98.53] client denied by server configuration: /home/nicesoda/public_html/laishiu/life/live
[Sat Aug 6 00:28:05 2005]	[error] [client 207.46.98.53] client denied by server configuration: /home/nicesoda/public_html/laishiu/life/live
[Sat Aug 6 00:09:44 2005]	[error] [client 66.249.65.52] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Sat Aug 6 00:09:44 2005]	[error] [client 66.249.65.52] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Sat Aug 6 00:02:42 2005]	[error] [client 66.249.64.27] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Sat Aug 6 00:02:42 2005]	[error] [client 66.249.64.27] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Fri Aug 5 23:59:54 2005]	[error] [client 66.142.249.156] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Fri Aug 5 23:59:53 2005]	[error] [client 66.142.251.123] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Fri Aug 5 23:18:27 2005]	[error] [client 66.142.251.123] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Fri Aug 5 23:12:10 2005]	[error] [client 66.142.251.126] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Fri Aug 5 23:08:15 2005]	[error] [client 207.46.98.53] client denied by server configuration: /home/nicesoda/public_html/laishiu/life/live
[Fri Aug 5 23:08:15 2005]	[error] [client 207.46.98.53] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Fri Aug 5 22:48:26 2005]	[error] [client 66.196.101.36] client denied by server configuration: /home/nicesoda/public_html/laishiu/life/live
[Fri Aug 5 22:48:26 2005]	[error] [client 66.196.101.36] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Fri Aug 5 22:41:32 2005]	[error] [client 66.142.249.138] client denied by server configuration: /home/nicesoda/public_html/laishiu/life/live
[Fri Aug 5 22:27:27 2005]	[error] [client 66.142.249.77] client denied by server configuration: /home/nicesoda/public_html/laishiu/life/live
[Fri Aug 5 22:27:26 2005]	[error] [client 66.142.251.123] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Fri Aug 5 21:56:43 2005]	[error] [client 66.142.138.82] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Fri Aug 5 21:21:35 2005]	[error] [client 66.196.91.118] client denied by server configuration: /home/nicesoda/public_html/laishiu/Blurp.txt
[Fri Aug 5 21:21:34 2005]	[error] [client 66.196.91.118] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Fri Aug 5 20:47:01 2005]	[error] [client 66.142.250.62] client denied by server configuration: /home/nicesoda/public_html/laishiu/life/live
[Fri Aug 5 20:47:01 2005]	[error] [client 66.142.251.123] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Fri Aug 5 19:18:49 2005]	[error] [client 66.142.250.121] client denied by server configuration: /home/nicesoda/public_html/laishiu/life/live
[Fri Aug 5 19:05:22 2005]	[error] [client 66.142.251.39] client denied by server configuration: /home/nicesoda/public_html/laishiu/life/live
[Fri Aug 5 19:05:20 2005]	[error] [client 66.142.251.128] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Fri Aug 5 18:45:40 2005]	[error] [client 207.46.98.53] client denied by server configuration: /home/nicesoda/public_html/laishiu/life/live
[Fri Aug 5 18:44:24 2005]	[error] [client 207.46.98.53] client denied by server configuration: /home/nicesoda/public_html/laishiu/life/live
[Fri Aug 5 18:34:35 2005]	[error] [client 66.142.250.163] client denied by server configuration: /home/nicesoda/public_html/laishiu/life/live
[Fri Aug 5 18:02:58 2005]	[error] [client 66.196.91.124] client denied by server configuration: /home/nicesoda/public_html/laishiu/Blurp.txt
[Fri Aug 5 18:02:58 2005]	[error] [client 66.196.91.124] client denied by server configuration: /home/nicesoda/public_html/laishiu/Blurp.txt
[Fri Aug 5 18:02:57 2005]	[error] [client 66.196.91.124] client denied by server configuration: /home/nicesoda/public_html/laishiu/Blurp.txt
[Fri Aug 5 18:02:57 2005]	[error] [client 66.196.91.118] client denied by server configuration: /home/nicesoda/public_html/laishiu/robots.txt
[Fri Aug 5 18:02:56 2005]	[error] [client 66.142.249.211] client denied by server configuration: /home/nicesoda/public_html/laishiu/life/live

Los archivos *logs* pueden abrirse con cualquier editor de texto, mostrando entradas ordenadas según fecha y hora. Cada software que instale un servidor dispondrá de su propio archivo *log* en la carpeta que haya sido dispuesta para ello. De esta manera, una base de datos MySQL tendrá sus logs correspondientes, mientras que un servidor FTP dispondrá de otros.



Los *logs* no están asociados únicamente al campo de los servidores. Un antivirus, por ejemplo, puede generar un log al realizar un análisis del sistema.

Instalación de Apache

En las siguientes líneas, se indicará brevemente la instalación del servidor Apache, tanto en Linux como en Windows. También se expondrá dónde situar los archivos para acceder a ellos como si el equipo fuese un servidor.

Linux

La instalación en *Linux* se lleva a cabo de la siguiente forma:

1. Instalar el paquete de *Apache* con la siguiente línea (teniendo en cuenta que hay que realizarlo con credenciales de administrador):

```
apt-get install apache2
```

Otra opción más sencilla es descargar el paquete XAMPP en el siguiente enlace: <<http://www.apachefriends.org/en/xampp-linux.html>>. Este instalador contiene, entre otros, *Apache*, *MySQL* y *PhpMyAdmin*.

2. Escribir <<http://localhost>> en un navegador para comprobar que el servidor local funciona.
3. Los archivos a incluir en el servidor deberán ir dentro de una carpeta en la ruta /var/www/ (en algunas distribuciones de Linux puede cambiar). Esta dirección se puede modificar dentro del fichero “*httpd.conf*”.

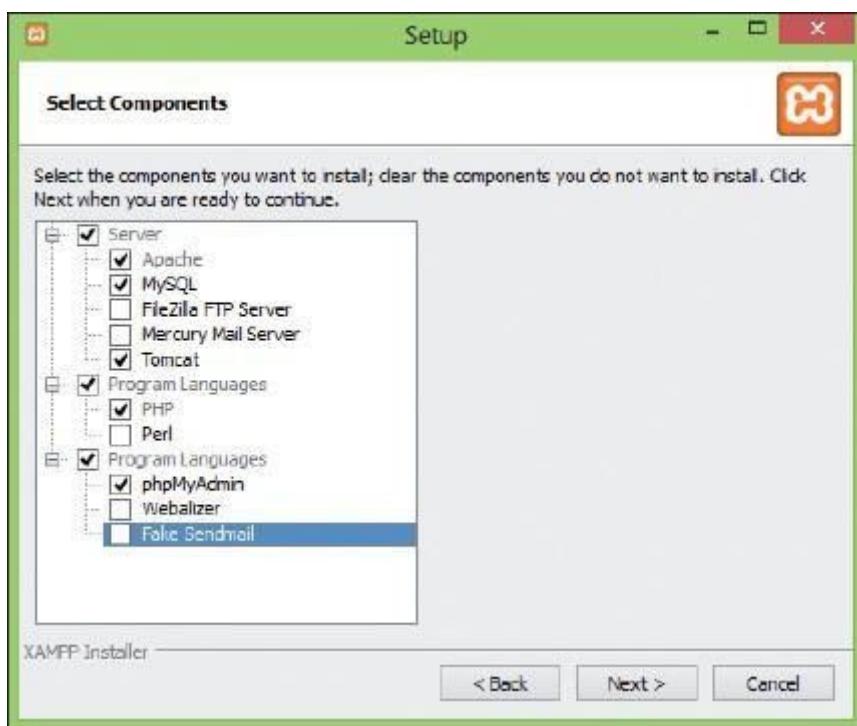
Windows

La instalación en *Windows* se lleva a cabo de la siguiente forma:

1. Para *Windows* se puede recurrir nuevamente al paquete XAMPP. Como en *Linux*, este software instala en el sistema varios programas y utilidades necesarios para poner en marcha un servidor.

Se puede descargar aquí: <<http://www.apachefriends.org/en/xampp-windows.html>>.

Se recomienda instalar *Apache Tomcat*, el cual será necesario en el próximo tema.



2. Acceder a <<http://localhost>> en un navegador para comprobar que el servidor local funciona.
3. Los archivos a incluir en el servidor deberán ir dentro de un directorio en la ruta **C:\xampp\htdocs** (dando por supuesto que XAMPP haya sido instalado en el disco duro C dentro de la carpeta **xampp**).

Resumen

Una aplicación web es una aplicación disponible en un servidor, la cual se encuentra a la espera, pendiente de recibir peticiones por parte de un cliente para generar una respuesta y enviarla. Este cliente accede a la aplicación del servidor de diversas maneras, siendo la más común a través de un navegador web.

Las interconexiones entre equipos se basan en un modelo de siete capas (llamado modelo OSI), que especifica las funciones y protocolos que dan lugar a la comunicación.

La preparación de un servidor es un proceso que requiere la instalación de varios programas que proporcionan la funcionalidad básica (como puede ser un servidor de base de datos, por ejemplo), disponiéndose también de módulos accesorios (inicialmente deshabilitados o de terceros) que amplían las posibilidades y características.

Para llevar a cabo un desarrollo, es muy normal tener instalado un servidor de manera local, trabajando codo con codo con una base de datos (también instalada en local). Para acceder a dicho servidor local, hay que escribir <http://localhost> en la barra de direcciones del navegador.

Ejercicios de repaso y autoevaluación

1. Una aplicación web...

- a. requiere de un servidor a la escucha de solicitudes.
- b. se puede desarrollar en varios lenguajes de programación.
- c. normalmente hace uso de una base de datos.
- d. Todas las respuestas anteriores son correctas.

2. Complete la siguiente oración.

El modelo _____ es un modelo de _____ capas que define la _____ entre equipos.

3. Defina brevemente la función de la capa de red.

4. ¿A qué capa pertenece el protocolo HTTP?

- a. A la capa de red.
- b. A la capa de sesión.
- c. A la capa de enlace de datos.
- d. Ninguna de las respuestas anteriores es correcta.

5. ¿Cómo se accede al servidor local?

- a. www.localhost.
- b. http://localhost.es.
- c. http://localhost.
- d. Las respuestas b. y c. son correctas.

6. ¿Quién renderiza la respuesta de un servidor para que sea mostrada de manera legible?

- a. Ya viene renderizada desde el servidor.
- b. El cliente que la solicitó.
- c. Depende de la información que haya sido solicitada.
- d. Ninguna de las respuestas anteriores es correcta.

7. Son plataformas de desarrollo en el servidor...

- a. JSP (Java Server Pages).
- b. PHP (PHP Hipertext Preprocessor).
- c. ASP.NET.
- d. Todas las respuestas anteriores son correctas.

8. UDP y TCP son protocolos de la capa de transporte...

- a. no orientado a conexión y orientado a conexión, respectivamente.
- b. orientado a conexión y no orientado a conexión, respectivamente.
- c. ambos son orientados a conexión.
- d. en la definición de los protocolos no se especifica su orientación.

9. Apache es, aparte de un servidor, una base de datos.

- a. Apache es únicamente un servidor.
- b. Apache es una base de datos.
- c. Apache es un instalador que permite seleccionar los componentes a instalar, ya sea servidor o base de datos.
- d. Ninguna de las respuestas anteriores es correcta.

10. Internet Information Server...

- a. es un servidor disponible para Linux.
- b. es un servidor disponible para Windows.
- c. depende de Microsoft.
- d. Las respuestas b. y c. son correctas.

11. PHPMyAdmin...

- a. es un programa para configurar PHP.
- b. es un gestor de base de datos.
- c. es un administrador de cuentas de servidor.
- d. Las respuestas a. y c. son correctas.

12. Un servidor FTP...

- a. sirve para acceder a otros servidores y subir archivos.
- b. se instala para convertir el sistema en un servidor de aplicaciones web.
- c. permite que un cliente FTP conecte con él para gestionar archivos.
- d. Todas las respuestas anteriores son correctas.

13. Los tres entornos básicos son:

- a. Desarrollo, producción y postproducción.
- b. Desarrollo, prueba y final.
- c. Prueba, preproducción y producción.
- d. Desarrollo, preproducción y producción.

14. Los logs...

- a. muestran información de interés relacionada con la actividad del servidor.
- b. como concepto no están restringidos a los servidores.
- c. se pueden abrir con un simple editor de texto.
- d. Todas las respuestas anteriores son correctas.

15. Complete la siguiente oración.

Para que un _____ sea mostrado a través de una petición al _____
hay que colocarlo en una _____ dentro del directorio correspondiente.

Capítulo 4. Lenguajes de programación de aplicaciones web en el lado del servidor

Introducción

No cabe duda de que el hecho de disponer de aplicaciones en un servidor (permanentemente disponibles y accesibles por el cliente de diversas formas) es una idea atractiva para cualquier programador y, sin duda, ofrece un gran abanico de posibilidades.

En este último capítulo, se ampliarán todas las ideas recogidas en los tres capítulos anteriores, con el fin de establecer unos conocimientos mínimos que sienten las bases que permitan afrontar el desarrollo de una aplicación web perfectamente operativa.

Conceptos como servidor, lenguajes de programación en el servidor, bases de datos, documentación de código, etc., adquirirán una dimensión práctica para llenar el hueco dejado en temas anteriores, más enfocados hacia un aspecto teórico.

Características de los lenguajes de programación web en el servidor

Un lenguaje de programación web de lado servidor es aquel que se ejecuta en el servidor. Un programa creado con uno de estos lenguajes (PHP, JSP, ASP. NET) puede acceder a recursos (una base de datos, por ejemplo) antes de crear la página que recibirá el cliente como respuesta a su petición.

Frente a los lenguajes del lado del servidor, están los lenguajes de programación del lado del cliente. Estos son independientes del servidor y se descargan e interpretan a través del cliente que accede a ellos.

Entre algunos ejemplos del lado del cliente, se puede encontrar JavaScript o Applets. El primero requiere del complemento JavaScript habilitado en el navegador, mientras que el segundo necesita de la máquina virtual Java.

Por razones obvias, un código del lado de servidor es más fácil de controlar al ejecutarse en dicho servidor. El del lado del cliente, por analogía, se ejecuta en el cliente, pudiendo existir limitaciones debidas a la gran cantidad de navegadores y equipos que pueden acceder. También hay que decir que un código ejecutado en el lado del cliente es más vulnerable.

El párrafo anterior no se debe interpretar como una crítica, ya que estos lenguajes son necesarios. Un código ejecutado en el lado del cliente puede realizar acciones interactivas con el usuario sin

necesidad de recargar la página realizando una nueva petición al servidor. Esto no es posible con lenguajes de servidor, a no ser que se refresque la página completamente.

Así que, a modo de resumen, las características de un lenguaje de programación del lado servidor son:

- Se ejecutan en el servidor, enviando un contenido al cliente.
- Son más seguros que los lenguajes del lado del cliente.
- Una página enviada al cliente no se puede modificar.
- Es más fácil depurar y seguir el rastro de un error.
- No puede ser bloqueado por el navegador.

Tipos y características de los lenguajes de uso común

Dentro de los lenguajes de servidor, hay variantes. Estas se describen a continuación.

Interpretados orientados a servidor

Los lenguajes interpretados son aquellos que no necesitan de compilación, siendo interpretados al mismo tiempo que se ejecutan y generando una salida hacia el cliente. El ejemplo más claro es PHP.

Por normal general, un lenguaje interpretado es más lento que uno compilado, pero, al mismo tiempo, es menos dependiente de la máquina que lo ejecuta. También se reitera el detalle de que la salida que recibirá el cliente, una vez generada y enviada, no puede ser modificada por el servidor.

Nota

PHP se acepta como lenguaje interpretado, por regla general. Sin embargo, también puede ser compilado. Tal vez no sea el mejor ejemplo, pero Facebook, por ejemplo, utiliza una suerte de compilador que transforma PHP en código C y procede a su compilación.

Lenguajes de cliente interpretados en servidor

Los lenguajes del lado de cliente como HTML, JavaScript y CSS son interpretados por el cliente. Esto permite ejecutar código en tiempo real fruto de interactividad con el usuario, sin necesidad de refrescar la página, pero sus posibilidades de interacción con el servidor son prácticamente nulas. Para paliar esta carencia, surgieron tecnologías como AJAX (Asynchronous JavaScript And XML), que permite seguir ejecutando código en el lado del cliente, pero manteniendo conectividad con el servidor en un segundo plano.

De esta manera, por ejemplo, una selección del usuario sobre un desplegable en un formulario puede disparar un script PHP en el servidor. Este script recibirá el valor seleccionado por el usuario y hará una consulta en la base de datos, devolviendo nuevos datos que serán cargados en otros puntos del formulario sin necesidad de refrescar la página y perder lo ya introducido.

Recuerda

- **Lenguajes de servidor:** se ejecutan en el servidor. Una vez enviada la respuesta no se puede modificar.
- **Lenguajes de cliente:** se ejecutan en el cliente. Pueden interactuar con la página actual (de manera limitada) sin necesidad de realizar una nueva petición al servidor.
- **Lenguajes de cliente interpretados en servidor:** ejecutan scripts almacenados en el servidor, provocando modificaciones en la página sin necesidad de recarga.

Lenguajes compilados

El código escrito en estos lenguajes debe ser compilado con vistas a su ejecución. JSP y ASP.NET son dos de sus máximos exponentes. Esto provoca un mayor desempeño que un código equivalente en lenguaje interpretado.

JSP requiere un servidor adicional aparte de *Apache*, como *Apache Tomcat*, mientras que ASP.NET necesita del *framework* .NET.

Criterios en la elección de un lenguajes de programación web en el servidor. Ventajas e inconvenientes

El principal criterio de elección viene determinado por el servidor que acogerá la aplicación web. Si el servidor es *Internet Information Server* (recordemos, exclusivo de *Windows*) está claro que la plataforma de desarrollo debe ser ASP.NET (aunque también puede acoger PHP).

Si, por el contrario, el servidor es *Linux*, hay más alternativas. Las principales, como se ha ido comentando hasta ahora, son PHP y *Java Server Pages* (JSP). La decisión entre uno y otro es más personal que otra cosa, pero, a grandes rasgos, se pueden establecer los siguientes criterios:

- **Separación de capas:** consiste en diferenciar entre la capa de presentación (cómo se muestra la información), la capa de negocio (la lógica de la aplicación) y la capa de datos (donde residen los datos). Esto incide en una mayor extensibilidad y mantenibilidad. JSP lleva todo esto de serie, mientras que con PHP hay que recurrir a frameworks externos. En el apartado dedicado al modelo vista controlador se ampliará este aspecto.
- **Mantenibilidad:** derivada de la anterior. Mientras mejor estructurada esté la aplicación, más fácil será su mantenimiento.
- **Curva de aprendizaje:** PHP es más fácil de aprender que Java o, al menos, no requiere tanto para dominarlo.
- **Integración externa:** que JSP utilice Java tiene sus ventajas. Esta tecnología tiene muchísimos complementos externos, aparte de que permite aprovechar clases utilizadas en desarrollos para escritorio.
- **Coste de desarrollo:** PHP es más inmediato, pues, como se ha dicho, es más fácil de aprender. Además, no requiere de un servidor complementario, como puede ser Apache Tomcat.

Las conclusiones finales son que PHP es perfectamente válido para pequeños/medios desarrollos, pero si ya se desea una aplicación de gran complejidad/nivel empresarial, la elección de JSP es la recomendada.

Nota

PHP y Java disponen de frameworks que amplían bastante las posibilidades. Por ejemplo, para PHP existen Zend o CodeIgniter, que facilitan enormemente los desarrollos implementando la separación de capas.

En este manual, se ha venido usando Java en prácticamente todos los ejemplos. Entre los objetivos de este texto está saber desarrollar componentes basados en tecnologías *Servlet* y *Applet*. La primera se ejecuta en el lado del servidor, requiriendo (al igual que JSP) del servidor *Apache Tomcat*, mientras que la segunda se ejecuta en el lado del cliente. Dado que estas dos tecnologías pertenecen a la versión *Enterprise* de Java, a partir de este punto se da por hecho que todo el código de los ejemplos se escribirá en Java.

Características generales

En este tema, se hará un resumen de todo lo comentado en capítulos anteriores relativo a la programación bajo lenguaje Java.

Tipos de datos

Un tipo de datos es un atributo que indica el tipo de dato con el cual se va a trabajar. Se puede establecer una clasificación, distinguiendo entre tipo de dato primitivo y tipo de dato compuesto. El primero es el tipo básico que proporciona el lenguaje, mientras que el segundo es el generado a partir del primero. Entre los tipos de datos primitivos de Java están los siguientes:

- **Carácter (char):** sirve para almacenar un carácter. Por carácter se entiende un número, una letra, un signo de puntuación, etc.
- **Entero (int):** almacena un número entero. Este tipo permite representar números con una precisión de 32 bits (es decir, se pueden representar $2^{32} = 4294967296$ valores).
- **Real (float):** representa un número decimal. Utiliza 32 bits.
- **Doble (double):** número decimal de doble precisión. Usa 64 bits.
- **Cadena (string):** representa una cadena de caracteres. En Java se considera un tipo primitivo, pero hay que usar la clase String con su correspondiente constructor.
- **Booleano (boolean):** toma un valor de dos posibles (verdadero o falso). Verdadero se representa con la palabra reservada “true”, mientras que falso usará “false”.

Clases

Una clase es la implementación de la especificación (o tipo abstracto de datos) de un concepto del mundo real. Los objetos que se crean a partir de una clase son instancias de la misma. Se puede ver

la especificación que implementa la clase como una especie de contrato, gracias al cual los objetos que deriven de la clase se comprometen a cumplir dicha especificación.

La definición de una clase incluye los atributos y métodos que tendrán las instancias de la misma. Recuérdese que los atributos debían ir acompañados con el modificador **private**, con el fin de que no fuesen accesibles desde el exterior con fines maliciosos. Cualquier acceso para consulta de valor o modificación debía hacerse con métodos definidos con tal propósito (*los getters y los setters*).

Para instanciar una clase en Java se utiliza la palabra reservada **new**, con el fin de invocar al constructor de la clase. Se podían implementar tantos constructores como fuesen necesarios (siempre que tuviesen diferentes parámetros), puesto que la elección se lleva a cabo en tiempo de ejecución.

Teniendo en cuenta lo anterior y partiendo de una clase **Persona** (previamente definida) con un constructor que recibiese dos **String** conteniendo el nombre y el apellido, la creación de un objeto se hará de la siguiente forma:

```
Persona p = new Persona( "Rafael", "Granados");
```

Operadores básicos. Manipulación de cadenas de caracteres

Los operadores básicos de Java (y otros lenguajes) son los que se describen a continuación.

Operadores matemáticos

Realizan operaciones matemáticas con los valores de las variables:

- Suma (+).
- Resta (-).
- Multiplicación (*).
- División (/).
- Módulo (mod). El módulo es el resto de la división entera.
- Exponenciación (^).

Operadores relacionales

Establecen una relación entre los valores contenidos en dos variables (que deben ser del mismo tipo). El resultado de la relación será verdadero o falso:

- Menor que (<).
- Mayor que (>).
- Menor o igual que (<=).
- Mayor o igual que (>=).
- Diferente (!=).

- Igual (==).

Operadores lógicos

Sirven para combinar condiciones:

- Operador AND (&&): las dos condiciones deben ser ciertas.
- Operador OR (||): basta con que sea cierta una condición.
- Operador NOT (!): cierta si la condición no se cumple.

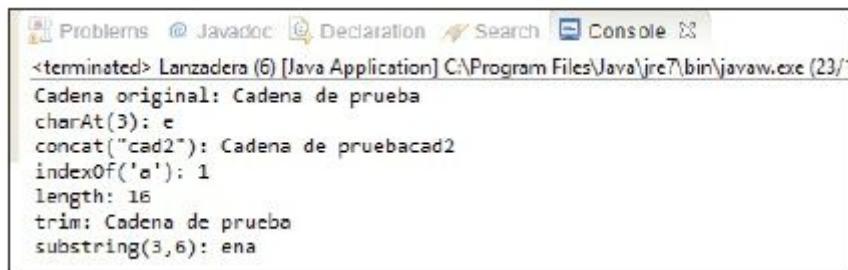
Manipulación de cadenas de caracteres

Respecto a la manipulación de cadenas de caracteres en Java, se puede hacer uso de los métodos de la clase String. Se listan los más importantes:

- **char charAt(int index)**: obtiene el carácter de la posición index.
- **String concat(String str)**: concatena dos cadenas.
- **int indexOf(int ch)**: primera posición de un carácter.
- **int length()**: longitud de la cadena.
- **String trim()**: elimina los espacios en blanco.
- **String substring(int begin, int end)**: obtiene la subcadena delimitada por los índices que se reciben como parámetro.
- **bool equals(Object obj)**: devuelve true si la cadena pasada como parámetro coincide con la que invoca al método.

A continuación, se muestra la salida por consola para el siguiente código:

```
 1 public class Lanzadera {  
 2     public static void main(String[] args) {  
 3         String cad = "Cadena de prueba";  
 4  
 5         System.out.println("Cadena original: " + cad);  
 6         System.out.println("charAt(3): " + cad.charAt(3));  
 7         System.out.println("concat(\"cad2\"): " + cad.concat("cad2"));  
 8         System.out.println("indexOf('a'): " + cad.indexOf('a'));  
 9         System.out.println("length: " + cad.length());  
10         System.out.println("trim: " + cad.trim());  
11         System.out.println("substring(3, 6): " + cad.substring(3, 6));  
12     }  
13 }
```



```
Problems Declaration Search Console
<terminated> Lanzadera (6) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (23)
Cadena original: Cadena de prueba
charAt(3): e
concat("cad2"): Cadena de pruebacad2
indexOf('a'): 1
length: 16
trim: Cadena de prueba
substring(3,6): ena
```

 **Sabías que...**

La posición en las cadenas (y también en los arrays) empieza a contar desde la posición 0. Así, para una cadena de 10 caracteres, se podrá acceder a estos mediante un índice de valor comprendido entre 0 y 9.

Estructuras de control. Bucles y condicionales

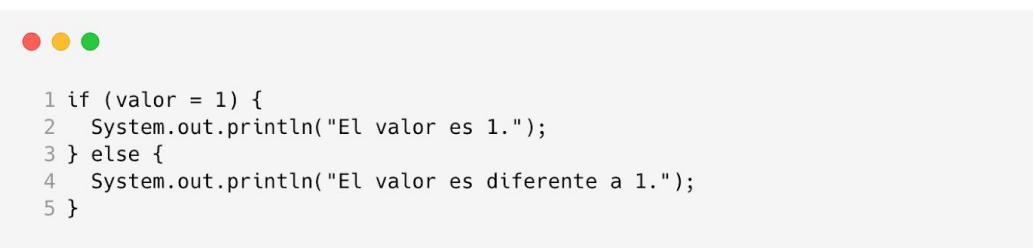
Una estructura de control determina el flujo de ejecución que seguirá un programa. Las estructuras de control básicas son las que se describen a continuación.

Selección

Se evalúa una condición para determinar por dónde continuará el flujo del programa.

Puede ser de dos tipos:

- **If-else** (si... sino): se elige una alternativa de dos posibles. Se pueden hacer anidamientos para controlar más condiciones.



```
1 if (valor = 1) {
2     System.out.println("El valor es 1.");
3 } else {
4     System.out.println("El valor es diferente a 1.");
5 }
```

- **Switch** (según): es la forma más elegante de controlar varias alternativas, con el fin de no caer en muchos IF-ELSE anidados que dificultan la lectura del código. Cada alternativa recibirá un tratamiento dentro de un case.

```
● ● ●  
1 switch (valor) {  
2     case 1:  
3         System.out.println("El valor es 1.");  
4         break;  
5     default:  
6         System.out.println("El valor es diferente a 1.");  
7         break;  
8 }
```

Iteración

En este bloque de control se repite la ejecución de una instrucción mientras se cumpla cierta condición. Existen tres variantes:

- **Do-while** (hacer... mientras): se ejecuta lo que hay dentro del bloque de control y se evalúa la condición. En el ejemplo, contador está inicializado a 0. En cada pasada incrementa su valor en uno, terminando las iteraciones si alcanza el valor 10.

```
● ● ●  
1 int contador = 0;  
2  
3 do{  
4     contador++;  
5     System.out.println(contador);  
6 } while(contador != 10);
```

- **While** (mientras...): es lo mismo que el anterior, pero la condición se evalúa al principio. De esta manera, según el ejemplo propuesto, no se mostrará el valor de contador.



```
1 int contador = 10;
2
3 while (contador < 10){
4     contador++;
5     System.out.println(contador);
6 }
```

- **For** (durante...): el While, por lo general, tiene que hacer uso de un contador previamente inicializado en la condición a evaluar. El bloque de control For integra el contador dentro del propio bucle.



```
1 for (int contador = 0; contador <= 10; contador++) {
2     System.out.println(contador);
3 }
```

Recuerda

Una estructura de control define el flujo de ejecución del programa. Hay tres tipos:

- Secuencial: una instrucción se ejecuta después de otra.
- Selección: la instrucción a ejecutar depende de una condición a evaluar.
- Iterativa: se repite la ejecución de una instrucción mientras se cumpla una determinada condición.

Módulos o paquetes

Las librerías en Java reciben el nombre de paquetes (packages). El sentido de una librería es agrupar varias clases que comportan una funcionalidad o utilidad común.

Java tiene a disposición del programador muchísimas clases y la mayor parte tienen que ser importadas antes de su uso. A continuación, se realizará la importación de la clase **ArrayList** con el fin de poder utilizarla. Recuérdese que **ArrayList** era extremadamente útil para tratar con **arrays**, implementando ya métodos con funcionalidades bastante comunes:

```

1 import java.util.ArrayList;
2
3 public class Lanzadera {
4     public static void main(String[] args) {
5         ArrayList<Integer> array = new ArrayList<Integer>();
6
7         array.add(4);
8         array.add(5);
9         array.add(7);
10
11        System.out.println("Longitud: " + array.size());
12        System.out.println("Contiene 4?: " + array.contains(4));
13        System.out.println("Borrar 4: " + array.remove((Integer) 4));
14        System.out.println("Contiene 4?: " + array.contains(4));
15        System.out.println("Longitud: " + array.size());
16    }
17 }
```



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the following text:

```

Problems @ Javadoc Declaration Search Console
<terminated> Lanzadera (7) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (23)
Longitud: 3
Contiene 4?: true
Borrar 4: true
Contiene 4?: false
Longitud: 2

```

Un detalle muy importante, ya mencionado en capítulos anteriores, es el concepto de casting. El casting es una especie de transformación, a través de la cual un objeto cambia la clase a la que pertenece dentro de ciertos límites (una cadena que contenga letras no se puede transformar en un número).

En el código anterior se observa, en el método ***array.remove((Integer)4)***, que se ha aplicado un *casting* a **Integer**. El método remove tiene dos variaciones:

- Una recibe la posición del elemento a eliminar.
- La otra recibe el elemento directamente, el cual será eliminado si es localizado.

Recuérdese que el **ArrayList** contiene elementos de tipo **Integer**, pues así se ha especificado en su definición. De no haber usado el casting, el método **remove** interpretaría que se quiere eliminar el elemento de la posición 4 y devolvería un error, puesto que únicamente existen tres elementos en ese momento.

De ahí que se deba convertir el número 4 en objeto, con el fin de indicar que se quiere eliminar el objeto de tipo **Integer** que contiene el valor 4 (independientemente de su posición).

Recuerda

Un *casting* modifica la clase a la que pertenece un objeto, siempre que la conversión esté permitida. El *casting* se aplica entre clases, de ahí que se use, por ejemplo, la clase **Integer** en vez del tipo de dato **int**.

Herencia

La herencia es un mecanismo mediante el cual una clase puede heredar la funcionalidad y los atributos de otra. A continuación, se comentan los puntos clave de la herencia en Java:

- Una clase sólo puede heredar de una única clase.
- La herencia se indica con la palabra reservada `extends`.
- Para implementar el concepto de multiherencia, se utilizan interfaces. Recuérdese que una interfaz era una clase que no se podía instanciar y que todos sus métodos eran abstractos (un método abstracto es aquel que requiere implementación obligatoria).
- En el constructor de la clase que hereda hay que referenciar al constructor de la superclase mediante `super`. Con esta palabra también se puede hacer referencia a cualquier otro método de la superclase.

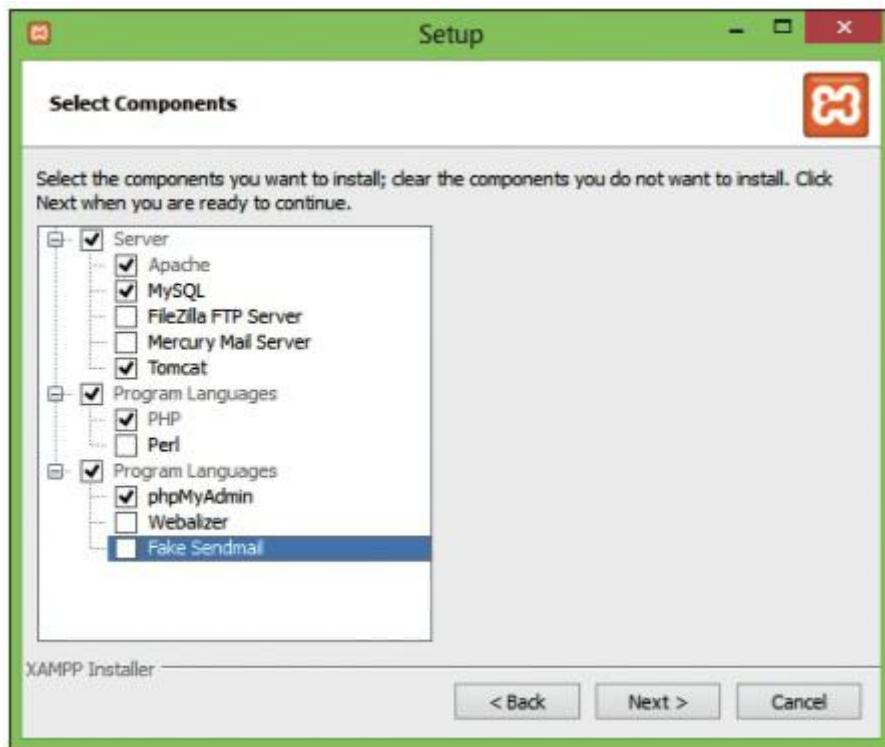
Gestión de la configuración

Para que una aplicación web funcione desde el servidor, se requiere una configuración previa, la cual se detallará en los siguientes puntos.

Configuración de descriptores

Un descriptor es un fichero XML, que contiene información necesaria para que una aplicación web funcione en el lado del servidor. Como se ha comentando durante el capítulo, la tecnología Java necesita de un servidor complementario funcionando. Este servidor puede ser Apache Tomcat, el cual se puede instalar junto con el paquete XAMPP.

En la siguiente imagen, se muestran las opciones de instalación del paquete XAMPP. Obsérvese que se ha seleccionado *Apache Tomcat*.



Nota

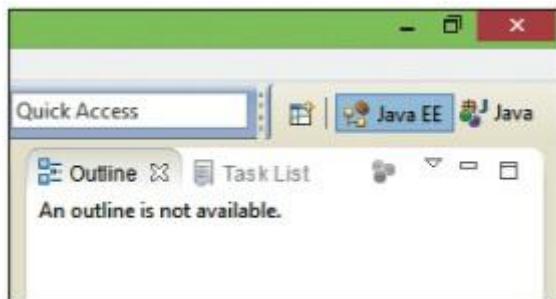
Realmente, más que un servidor, *Apache Tomcat* es un contenedor de aplicaciones que implementa la especificación de las tecnologías Servlet y JSP.

A continuación, se detallará el proceso para crear un proyecto web que contenga un *servlet*. Se parte del IDE de *Eclipse* para Java EE (Enterprise Edition), el cual permite la creación de proyectos web. Se puede descargar en:

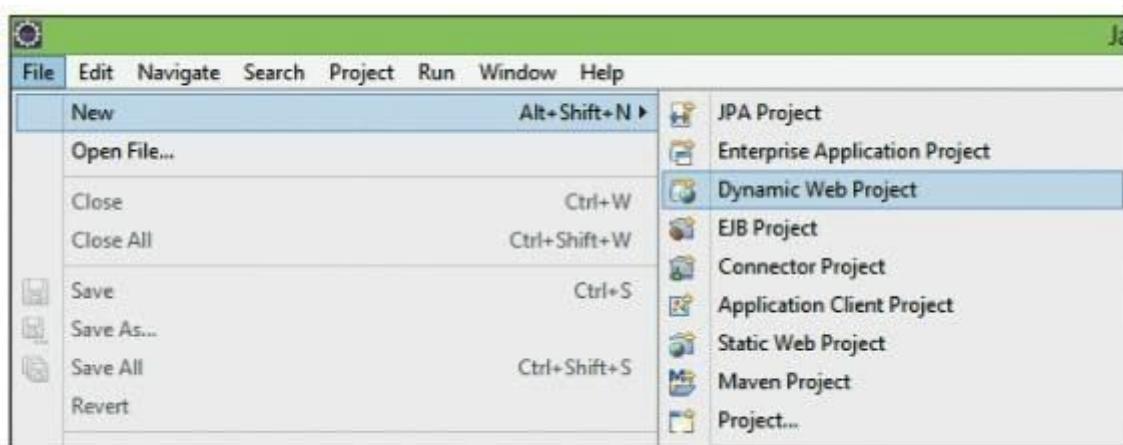
<https://www.eclipse.org/downloads/>

Una vez instalado, los pasos a seguir son:

1. Seleccionar la perspectiva de Java EE: por defecto sale la perspectiva de Java SE (Standard Edition), la cual es más simple (no tiene tantas opciones por defecto) y no muestra la opción de crear proyectos web.

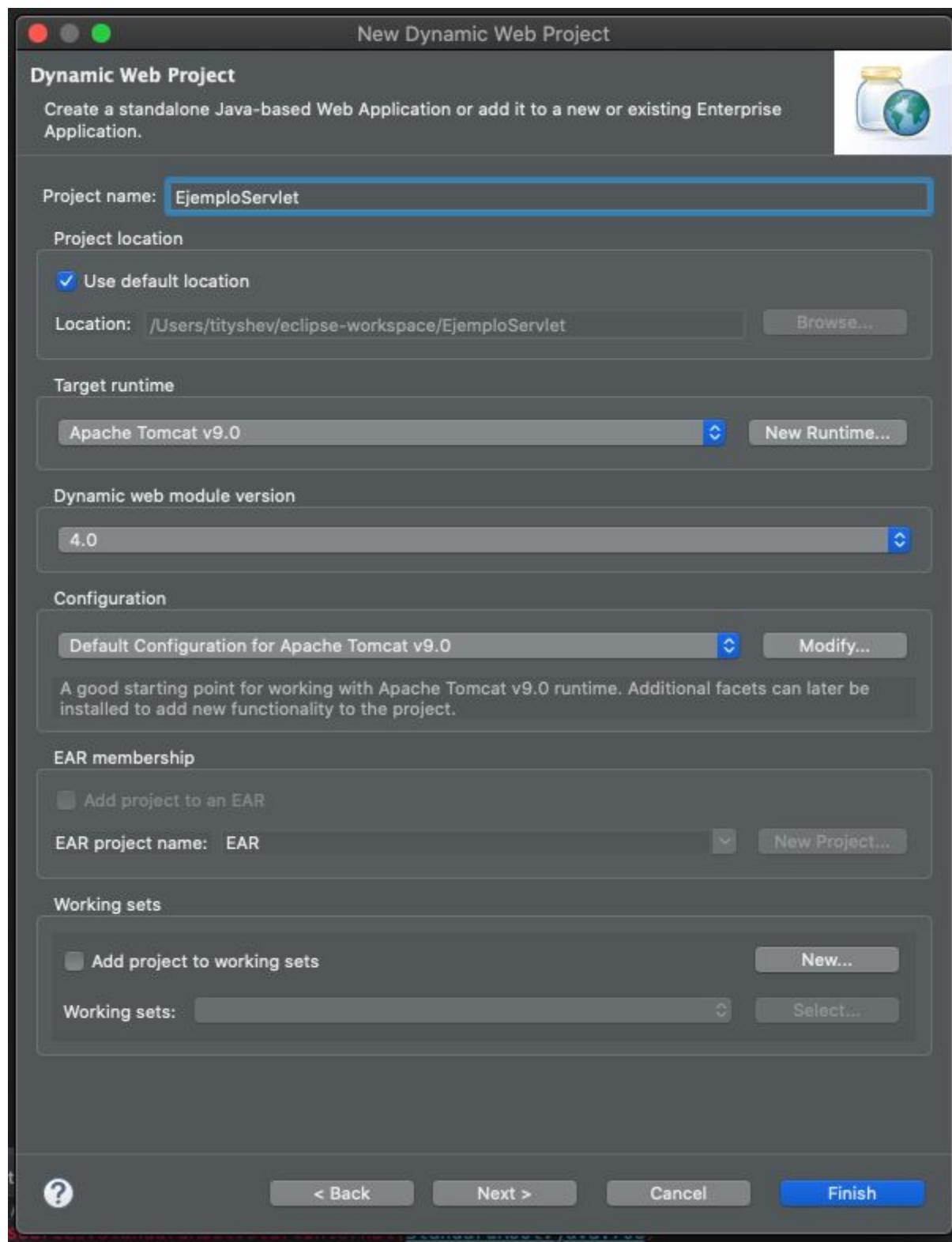


2. Seleccionar la opción de crear un proyecto web dinámico. Este menú aparecerá haciendo click en **File/New**.

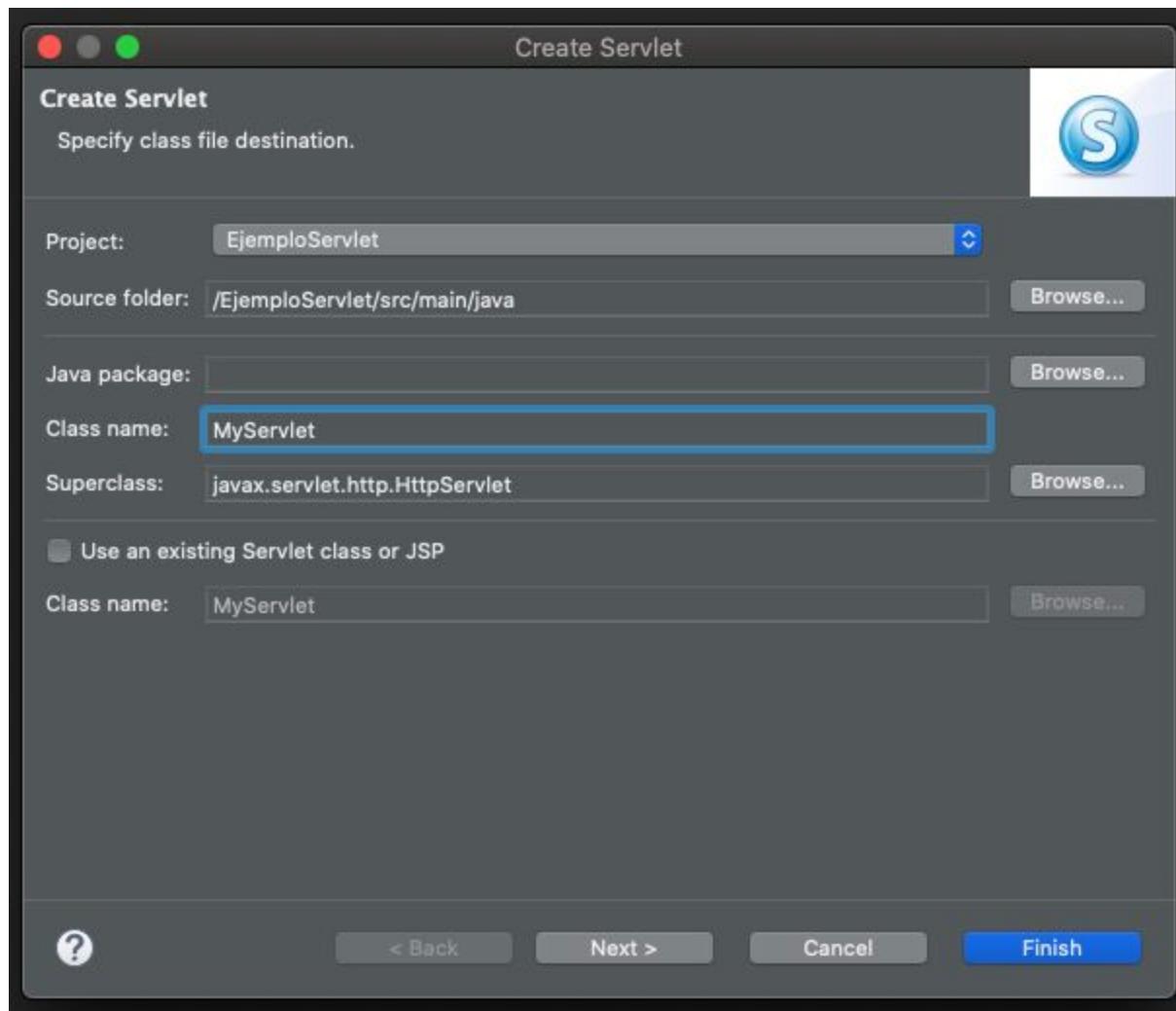


3. Aparecerá una nueva ventana. Los datos a rellenar son:

- **Project name:** EjemploServlet.
- **Target Runtime:** Apache Tomcat v9.0
- **Dynamic web module version:** 4.0



4. Agregar un *Servlet*. Se selecciona **New/Servlet**.

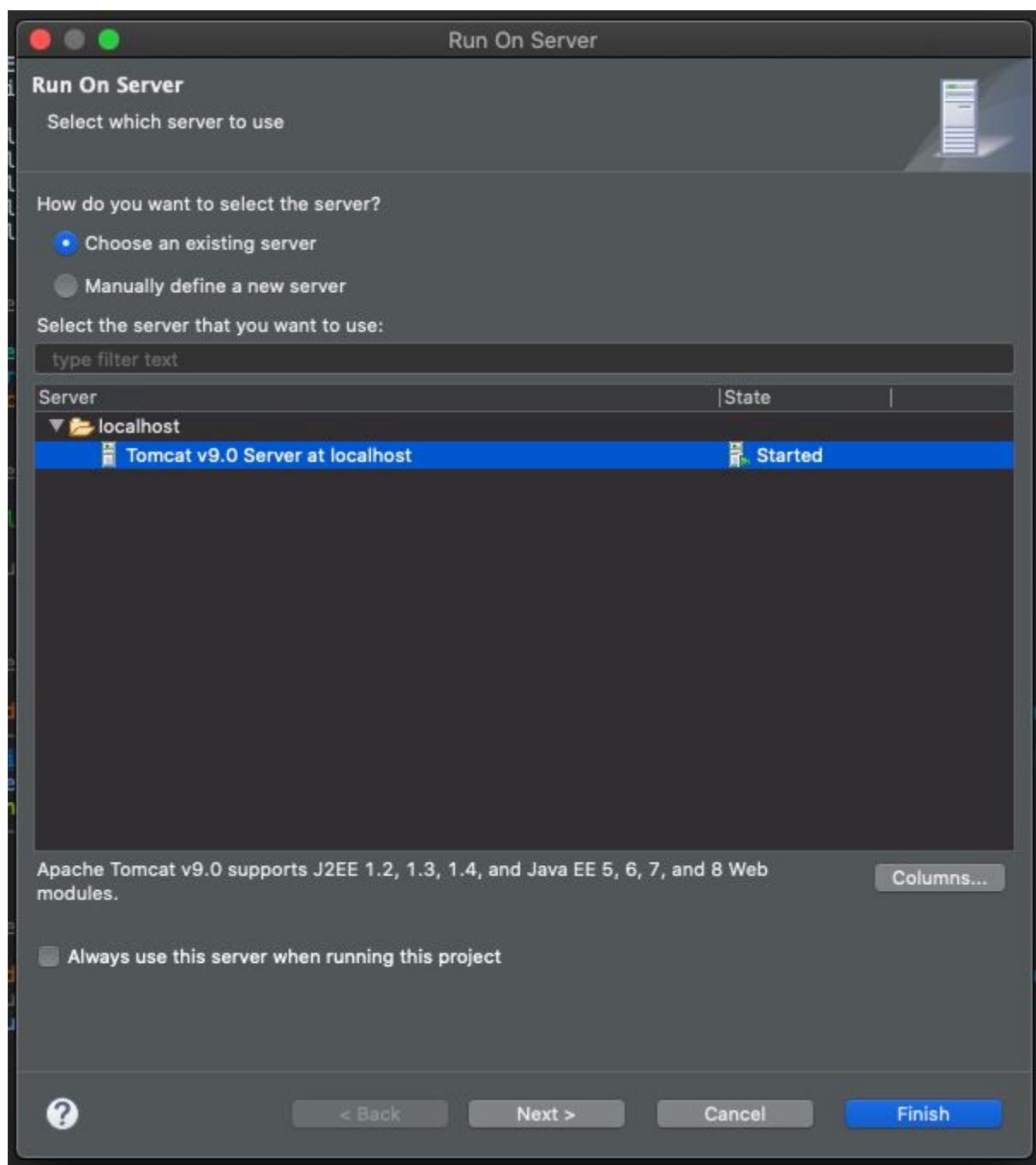


5. Eclipse generará automáticamente el esqueleto del *Servlet*.
6. A continuación se agregará código al método ***doGet***; La clase completa quedará de la siguiente manera:

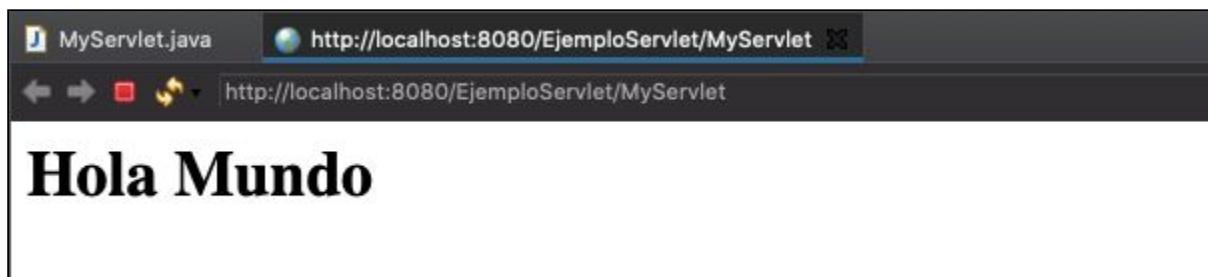
```
1
2
3 import java.io.IOException;
4 import java.io.PrintWriter;
5
6 import javax.servlet.ServletException;
7 import javax.servlet.annotation.WebServlet;
8 import javax.servlet.http.HttpServlet;
9 import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
11
12 /**
13  * Servlet implementation class MyServlet
14 */
15 @WebServlet("/MyServlet")
16 public class MyServlet extends HttpServlet {
17     private static final long serialVersionUID = 1L;
18
19     /**
20      * @see HttpServlet#HttpServlet()
21     */
22     public MyServlet() {
23         super();
24         // TODO Auto-generated constructor stub
25     }
26
27     /**
28      * @see HttpServlet#doGet(HttpServletRequest request,
29      HttpServletResponse response)
30     */
31     protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
32
33         // -----
34         PrintWriter out = response.getWriter();
35         response.setContentType("text/html");
36         out.println("<h1>Hola Mundo</h1>");
37         // -----
38     }
39
40     /**
41      * @see HttpServlet#doPost(HttpServletRequest request,
42      HttpServletResponse response)
43     */
44     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
45         // TODO Auto-generated method stub
46         doGet(request, response);
47     }
48
49 }
50
```

En este código, se crea un flujo de escritura sobre la respuesta que ofrecerá el *Servlet* (el paquete **PrintWriter** se deberá importar, como bien se encargará Eclipse de avisar). En esta respuesta, se indicará que la salida a generar es un documento HTML, con el texto “Hola mundo” entre etiquetas HTML del tipo **<h1></h1>**.

7. Ahora se ejecutará el proyecto a través de **Run As** (aparece con clic derecho). Si hay varias alternativas, se selecciona **Run on server**.



8. El resultado de la ejecución será el siguiente:



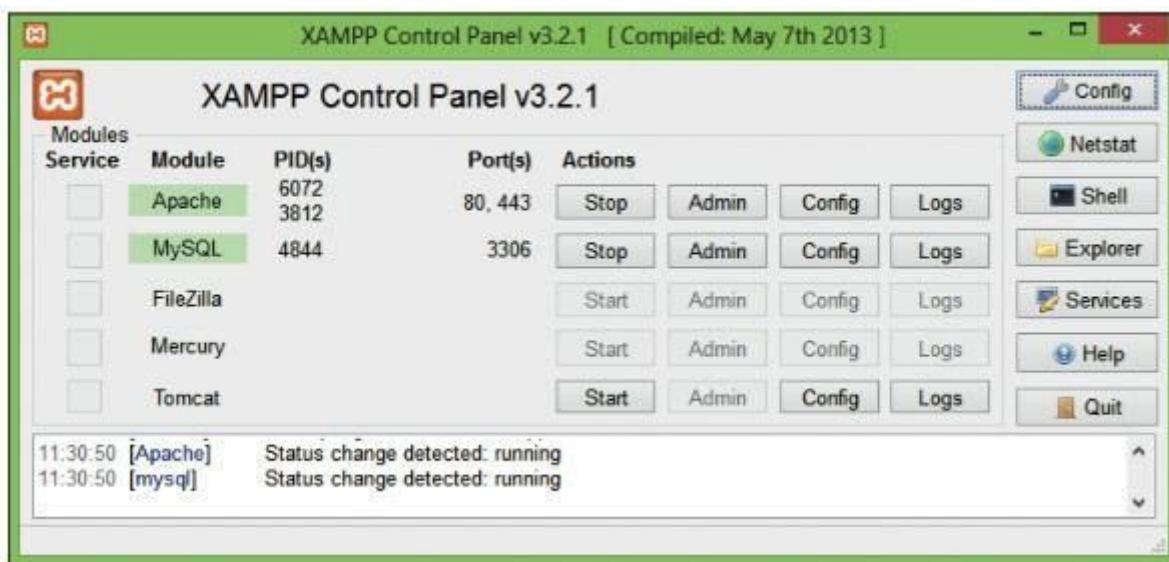
Respecto al fichero descriptor, este se encuentra disponible dentro del proyecto, en el apartado **Deployment Descriptor**. Aquí se presenta la estructura del fichero XML que contiene información necesaria para que el servicio web funcione. Los campos más relevantes en este fichero son:

- **Web-app**: define los esquemas utilizados y la versión de la aplicación. Generalmente, se dejan los valores por defecto.
- **Welcome-file-list**: páginas iniciales por defecto.
- **Display-name**: título de la página al ser mostrada.
- **Servlet**: servlets que forman parte del servicio web. Para cada entrada se especifica el nombre a mostrar, el nombre del servlet y la clase en la que se basa.
- **Servlet-mapping**: Especifica la forma en que se accede al servlet. Hay que indicar la dirección url de acceso y el servlet al que se refiere.

Node	Content
?? xml	version="1.0" encoding="UTF-8"
web-app	<pre><{description*, display-name*, icon*}> distributable context-param fil...</pre> <p>@ xmlns xsi="http://www.w3.org/2001/XMLSchema-instance" @ xmlns="http://java.sun.com/xml/ns/javaee" @ xs:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/w...</p>
id	WebApp_ID
version	2.5
display-name	EjemploServlet
welcome-file-list	<pre>(welcome-file+)</pre> <p>welcome-file index.html index.htm index.jsp default.html default.htm default.jsp</p>
servlet	<pre>(({description*, display-name*, icon*}), servlet-name, (servlet-class jsp-f...</pre> <p>description display-name servlet-name servlet-class</p>
servlet-mapping	(servlet-name, url-pattern+)
servlet-name	MyServlet
url-pattern	/MyServlet

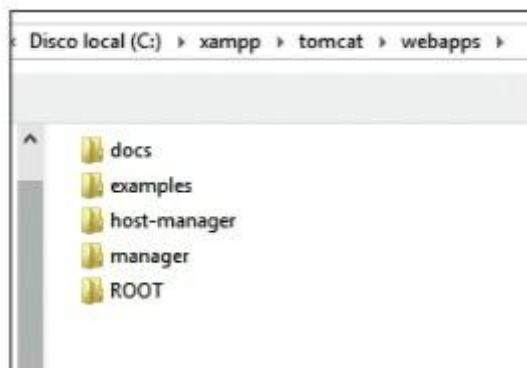
Configuración de ficheros

Todo lo dicho anteriormente es perfectamente válido durante el desarrollo de la aplicación web a través de *Eclipse*. Sin embargo, para que el servicio esté disponible en producción, hay que colocar sus archivos dentro del contenedor de *Apache Tomcat*. Además, el módulo de *Tomcat* debe estar iniciado; en caso de no estarlo hay que arrancarlo desde el Panel de XAMPP.



El punto de partida serán los datos creados en el proyecto anterior. Hay que copiar los siguientes ficheros:

- El “.class” de la clase que se ha creado para el *servlet*. Para el ejemplo, el fichero a copiar será “MyServlet.class”.
- El descriptor “web.xml”.
- El directorio donde está el *Apache Tomcat* aparece en la siguiente imagen. Hay que tener en cuenta que depende del directorio de instalación de XAMPP.



Lo único que hay que hacer es crear una nueva carpeta dentro de webapps que contenga la siguiente estructura:

- Una nueva carpeta **WEB-INF**.
- Dentro de **WEB-INF** se copia el archivo “web.xml”.
- En **WEB-INF** se crea otra carpeta, llamada **classes**, que contendrá a “MyServlet.class”.

El servicio web se invocará introduciendo la siguiente dirección en el navegador:

<http://localhost:8080/EjemploServlet/MyServlet>

Recuerda

Un servlet es una aplicación integrada dentro de un servicio web y se ejecuta en el servidor. Está programado en Java y requiere de *Apache Tomcat*. Las interacciones con el servicio web (forma de invocarlo, integración con otros servlets, etc.) están especificadas en un archivo XML llamado descriptor.

Gestión de la seguridad

La seguridad es uno de los aspectos más importantes a controlar en una aplicación web, sobre todo si se están manejando datos importantes o se requiere que el servicio no sea accesible para todo el mundo.

Conceptos de la identificación, autenticación y autorización

Estos tres conceptos son realmente los tres pasos naturales que hay que dar para poder acceder a un sistema. Sus definiciones son:

- **Identificación:** el usuario introduce, por ejemplo, un identificador. De esta manera, el sistema puede saber quién pretende acceder.
- **Autenticación:** la autenticación se produce al facilitar una contraseña, con el fin de demostrar que el usuario previamente introducido es quien dice ser.
- **Autorización:** el sistema comprueba la autenticación realizada por el usuario y, en caso de ser aceptada, le autoriza el acceso para realizar todo aquello que le esté permitido.
Evidentemente, no todos los usuarios deben tener los mismos privilegios (no es lo mismo un administrador que un usuario estándar), pero esto ya depende de cómo esté programado el sistema.

Técnicas para la gestión de sesiones

La idea subyacente de este apartado es controlar cuándo un usuario vuelve a usar una aplicación web en cierto lapso de tiempo. Por ejemplo, imagínese que alguien ingresa en la aplicación con un usuario y una contraseña, cierra la pestaña/navegador y la vuelve a abrir. ¿Debería solicitar de nuevo las credenciales de acceso? Existen dos posibilidades para establecer este control:

- **Uso de cookies:** almacenadas en el navegador del usuario. Permanecen allí, aunque el navegador se cierre. La cantidad de información que permiten almacenar es limitada. Además, debido a que se almacenan en el cliente, pueden ser manipuladas.
- **Uso de sesiones:** son almacenadas en el servidor. No permanecen para siempre, pues se pierden una vez se ha cerrado el navegador. Al almacenarse en el servidor no pueden ser manipuladas, permitiendo almacenar una mayor cantidad de información que las cookies.

La combinación ideal sería una aplicación web que manejase tanto cookies como sesiones. Las cookies recordarán el acceso del usuario si este apagase el ordenador y volviese a acceder y las sesiones actuarían en defecto de las anteriores (no permitirían recordar al usuario si este cerrase el navegador, pero sí en el resto de circunstancias).

Hoy en día, hay una corriente bastante crítica respecto a las *cookies*, ya que se consideran intrusivas y que no respetan la privacidad del usuario. Prácticamente todos los navegadores permiten su deshabilitación, así que el siguiente ejemplo se centrará únicamente en el uso de sesiones. Para ver de manera práctica la gestión de sesiones se agregará un nuevo *Servlet* al ejemplo anterior. Dicho servlet se llamará *ServletSesion*. No olvidar propagar los cambios a los directorios correspondientes de Apache Tomcat, con el fin de que refleje el nuevo servlet.

```

1
2
3 import java.io.IOException;
4 import java.io.PrintWriter;
5
6 import javax.servlet.ServletException;
7 import javax.servlet.annotation.WebServlet;
8 import javax.servlet.http.HttpServlet;
9 import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
11 import javax.servlet.http.HttpSession;
12
13 /**
14  * Servlet implementation class ServletSesion
15 */
16 @WebServlet("/ServletSesion")
17 public class ServletSesion extends HttpServlet {
18     private static final long serialVersionUID = 1L;
19     // -----
20     private HttpSession session = null;
21     // -----
22
23     /**
24      * @see HttpServlet#HttpServlet()
25      */
26     public ServletSesion() {
27         super();
28         // TODO Auto-generated constructor stub
29     }

```

```

31   /**
32    * @see HttpServlet#doGet(HttpServletRequest request,
33    *      HttpServletResponse response)
34    */
35   protected void doGet(HttpServletRequest request, HttpServletResponse
36    *                      response) throws ServletException, IOException {
37    // -----
38    PrintWriter out = response.getWriter();
39    response.setContentType("text/html");
40
41    sesion = request.getSession(false);
42
43    if (sesion == null) {
44        sesion = request.getSession();
45        out.println("<p>Nueva sesión " + sesion.getId() + "</p>");
46    } else {
47        out.println("<p>Sesión existente (ID: " + sesion.getId() +
48                    "</p>");
```

Lo único que se hace con este código es controlar si la sesión está creada o no. Para ello, se utiliza el método `getSession()` del objeto `request` de dos maneras diferentes:

- **`request.getSession(false)`**: si este método devuelve `null` significa que la sesión no ha sido creada. Es decir, se trata del primer acceso y debe crearse.
- **`request.getSession()`**: si el método no recibe ningún parámetro, simplemente se crea la sesión.

Otro método importante es **`sesion.getId()`**. Con él, se obtiene el identificador asociado a la sesión actual. Gracias a este método, se podrá ver, efectivamente, si se está en una nueva sesión. Con una primera llamada a través del navegador a la siguiente URL se ve que se crea la primera sesión:

<http://localhost:8080/EjemploServlet/ServletSesion>



Si refrescamos la página:



Si cerramos el navegador y volvemos a cargar la URL, comprobaremos que se crea una nueva sesión.

Recuerda

Las sesiones tienen persistencia mientras el navegador esté abierto, controlándose a través del servidor. Las *cookies* se mantienen aunque se cierre el navegador. Se almacenan en el cliente, pero son más intrusivas.

Gestión de errores

La gestión de errores es uno más de los aspectos a considerar al desarrollar una aplicación web. Además de que la aplicación esté bien desarrollada y responda a las expectativas, está tiene que controlar que los errores no sean fatales y que los datos proporcionados por los usuarios estén dentro de lo esperado.

Técnicas de recuperación de errores

La principal técnica de recuperación de errores disponible en Java es la excepción. Como ya se trató, este mecanismo está compuesto por dos bloques de código (y un tercero opcional):

- **Bloque try:** contiene el código conflictivo.
- **Bloque catch:** contiene el bloque que se ejecuta si se dispara la excepción.
- **Bloque finally:** es un bloque que se ejecuta siempre, independientemente de que se dispare la excepción.



```
1 try {
2     // Código problemático
3 }
4 catch {
5     // Tratamiento de la excepción
6 }
7 finally {
8     // Este código que se ejecutará siempre
9 }
```

Java trae multitud de excepciones predefinidas, pero en el siguiente bloque se aprenderá a definir excepciones propias.

Programación de excepciones

Las excepciones propias son sumamente útiles, pues permiten tratar de una manera elegante aquellas situaciones que se consideren y que no estén contempladas por las excepciones predefinidas de Java.

A continuación, se presentará un ejemplo en el cual se ha definido una excepción que se dispara si el nombre de un fichero que se solicita no coincide con “prueba.txt”.

Las excepciones propias, en su mínima expresión, heredan de la superclase **Exception** y solo disponen de un constructor, el cual recibe un *string* como parámetro y se lo pasa al constructor de **super**. Como todas las clases en Java, van definidas en un fichero independiente. De esta manera, la excepción propia quedaría así:



```
1 public class ExcepcionNombreFicher extends Exception {
2     public ExcepcionNombreFichero(String msg) {
3         super(msg);
4     }
5 }
```

A continuación se pone el código del *servlet* **ServletFichero**, que hará uso de la excepción propia anterior:



```
1
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStream;
6 import java.io.InputStreamReader;
7 import java.io.PrintWriter;
8
9 import javax.servlet.ServletException;
10 import javax.servlet.annotation.WebServlet;
11 import javax.servlet.http.HttpServlet;
12 import javax.servlet.http.HttpServletRequest;
13 import javax.servlet.http.HttpServletResponse;
14
15 import Exceptions.ExcepcionNombreFichero;
16
17 /**
18  * Servlet implementation class ServletFichero
19 */
20 @WebServlet("/ServletFichero")
21 public class ServletFichero extends HttpServlet {
22     private static final long serialVersionUID = 1L;
23     private String respuesta = "";
24
25     /**
26      * @see HttpServlet#HttpServlet()
27      */
28     public ServletFichero() {
29         super();
30         // TODO Auto-generated constructor stub
31     }
32
33     /**
34      * @see HttpServlet#doGet(HttpServletRequest request,
35      HttpServletResponse response)
36      */
37     protected void doGet(HttpServletRequest request, HttpServletResponse response)
38             throws ServletException, IOException {
39         String fichero = request.getParameter("fichero");
40         response.setContentType("text/html");
41         PrintWriter out = response.getWriter();
42
43         try {
44             if (!fichero.equals("prueba.txt"))
45                 throw new ExcepcionNombreFichero("Fichero incorrecto");
46
47             InputStream input =
48                 getServletContext().getResourceAsStream(fichero);
49             BufferedReader br = new BufferedReader(new
50                 InputStreamReader(input));
51
52             String s;
53
54             while ((s = br.readLine()) != null)
55                 out.println("<p>" + s + "</p>");
56             br.close();
57         }
58     }
59 }
```

```

53         } catch (ExceptionNombreFichero e) {
54             respuesta = e.getMessage();
55
56             out.println("<script>alert('" + respuesta + "')</script>");
57         }
58         finally {
59             out.close();
60         }
61     }
62
63     /**
64      * @see HttpServlet#doPost(HttpServletRequest request,
65      * HttpServletResponse response)
66
67     protected void doPost(HttpServletRequest request, HttpServletResponse
68     response) throws ServletException, IOException {
69         // TODO Auto-generated method stub
70         doGet(request, response);
71     }
72

```

Sobre este código, se deben hacer varias apreciaciones:

- El nombre del fichero se recibirá pasándolo mediante la URL al invocar al *servlet*.
- En el **try** se compara que el nombre del fichero coincida con “prueba.txt”. En caso de ser negativa esta comparación, se arrojará la excepción propia mediante la palabra reservada **throw**. Obsérvese que la excepción es creada pasando el mensaje a mostrar al constructor.
- Para crear los distintos flujos que van a permitir leer el contenido del fichero, hay que abrir este último con **getServletContext().getResourceAsStream(fichero)**. Este fichero debe estar dentro de la carpeta raíz que contiene el servicio web sobre el que se apoya el *servlet*.
- Con el While se va mostrando cada línea del fichero hasta que se alcance el final. Esto se consigue con el método **readLine()** del **BufferedReader** creado sobre el fichero, que leerá línea a línea hasta que se devuelva **null**.
- El bloque **finally** se ejecuta siempre (se dispare la excepción o no), así que se aprovechará para cerrar en él el flujo de escritura **out**, que es el que crea la página HTML que el *servlet* manda como respuesta a la petición.

Ahora que el *servlet* está definido, se procederá a la creación del HTML que llamará al servicio web. Básicamente, constará de un formulario que pedirá el nombre de un fichero para, acto seguido, invocar al *servlet* pasando este nombre como parámetro a través de la URL (este es el motivo por el cual se ha implementado el método *doGet* en vez de *doPost*). Se muestra el código:

```
1 <html>
2 <head>
3     <title>Nombre de fichero</title>
4 </head>
5
6 <body>
7     <form action="ServletFichero" method="get">
8         <label>Nombre del fichero</label>
9         <input type="text" name="fichero" />
10        <input type="submit" />
11    </form>
12 </body>
13 </html>
14
```

Un par de apuntes sobre el HTML:

- En el **method** del formulario se especifica **get** para que los datos del formulario sean pasados a través de la URL.
- En el **action** se indica el nombre del *servlet* (en este caso **ServletFichero**). Allí es donde se quiere que se envíen los datos.
- Únicamente hay dos *input* en el formulario. Uno es una caja de texto (destinada a recibir el nombre del fichero) y el otro es el botón de envío. El **label** simplemente acompaña al primer *input*, indicando lo que se quiere que ponga en la caja de texto.

Y, para terminar, solo quedaría el fichero “prueba.txt”, con el texto que se quiera. Una simple cadena, por ejemplo: “Este texto se debe ver si el fichero se ha abierto correctamente”. La distribución de los ficheros en la nueva aplicación web quedará así:

- Se crea la carpeta **ServletFichero** dentro del directorio **webapps** en el directorio de instalación de *Tomcat*.
- Los ficheros “index.html” y “prueba.txt” van dentro del directorio anterior, añadiendo también una nueva carpeta llamada **WEB-INF**.
- Dentro de **WEB-INF** se copia “web.xml” (el descriptor) y se crea una carpeta **clases** que contendrá “ExpcionNombreFichero.class” y “ServletFichero.class”.

Transacciones y persistencia

Una aplicación web normalmente tiene que estar conectada a una base de datos, con fines varios: acceder a datos, registrar accesos, verificar usuarios, etc. En este apartado, se explicará cómo realizar la conexión con una base de datos, además de leer datos, modificar y escribir en la misma.

Acceso a bases de datos. Conectores

El acceso a una base de datos se realiza a través de una API que permite realizar operaciones básicas sobre dicha base de datos.

Por otra parte, el conector es un programa que permite conectar con una base de datos.

Normalmente, se presenta en forma de librería o paquete, siendo necesaria su importación.

Recuerda

La comunicación con la base de datos se realiza a través de la API, requiriéndose el uso de un conector que proporciona el apoyo para establecer dicha conexión.

Estándares para el acceso a bases de datos

Hay varios estándares que definen una API para el acceso a una base de datos, siendo los dos más importantes los siguientes:

- **Open DataBase Connectivity (ODBC)**: es una API desarrollada originalmente por Microsoft. Como toda API consta de una especificación que cualquier programa que la implemente entenderá. Está escrita en lenguaje C e históricamente siempre ha estado asociada a Windows.
- **Java DataBase Connectivity (JDBC)**: lo mismo que la anterior, pero surgió para que fuese compatible con programas escritos en Java, de tal manera que se mantuviese la portabilidad de la que hace gala este lenguaje.

Gestión de la configuración de acceso a bases de datos

En el caso de Java, hay que decir que mediante JDBC se manejan cuatro clases básicas, cada una con una funcionalidad muy definida:

- **DriverManager**: permite cargar el driver (conector) que facilita la conectividad con la base de datos.
- **Connection**: establece la conexión con la base de datos.
- **Statement**: permite ejecutar sentencias.
- **ResultSet**: permite trabajar con el resultado devuelto por una sentencia.

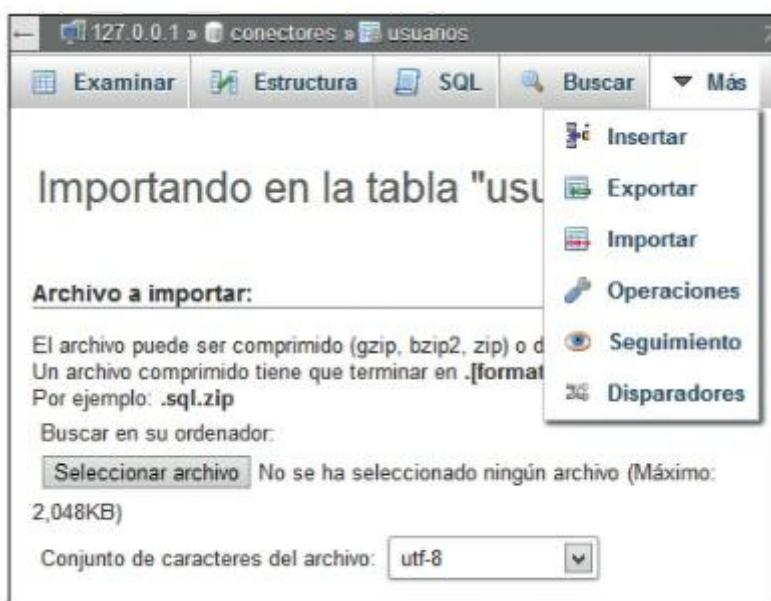
Para ilustrar un ejemplo, se creará una base de datos MySQL con el siguiente script:

```

1 CREATE DATABASE IF NOT EXISTS 'conectores';
2 USE 'conectores';
3
4 CREATE TABLE IF NOT EXISTS 'usuarios' (
5     'usuario' varchar(30) COLLATE utf8_unicode_ci NOT NULL,
6     'password' varchar(30) COLLATE utf8_unicode_ci NOT NULL,
7     PRIMARY KEY ('usuario')
8 );
9
10 INSERT INTO 'conectores'. 'usuarios' VALUES('rafa', 'rafa');
11 INSERT INTO 'conectores'. 'usuarios' VALUES('luis', 'luis');
12

```

La creación de la base de datos no se incluye como objetivo práctico en este manual, aunque cabe señalar que *PhpMyAdmin* (accesible desde <http://localhost/phpmyadmin>) permite la importación directa de un *script*.



A continuación, se definirá un servlet que extraerá la lista de usuarios disponibles. Para ello, se añade a *Eclipse* un nuevo elemento de tipo *Servlet*.

```
 1
 2
 3 import java.io.IOException;
 4 import java.io.PrintWriter;
 5 import java.sql.Connection;
 6 import java.sql.DriverManager;
 7 import java.sql.ResultSet;
 8 import java.sql.SQLException;
 9 import java.sql.Statement;
10
11 import javax.servlet.ServletException;
12 import javax.servlet.annotation.WebServlet;
13 import javax.servlet.http.HttpServlet;
14 import javax.servlet.http.HttpServletRequest;
15 import javax.servlet.http.HttpServletResponse;
16
17 /**
18 * Servlet implementation class ServletBBDD
19 */
20 @WebServlet("/ServletBBDD")
21 public class ServletBBDD extends HttpServlet {
22     private static final long serialVersionUID = 1L;
23
24     PrintWriter out;
25
26     /**
27      * @see HttpServlet#HttpServlet()
28     */
29     public ServletBBDD() {
30         super();
31         // TODO Auto-generated constructor stub
32     }
33
34     /**
35      * @see HttpServlet#doGet(HttpServletRequest request,
36      HttpServletResponse response)
37     */
38     protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
39         response.setContentType("text/html");
40         out = response.getWriter();
41
42         Connection cid;
43         Statement cs;
44         ResultSet rs;
45
46         try {
47             cid = conecta("localhost", "conectores", "root", "");
48             cs = cid.createStatement();
49             rs = cs.executeQuery("SELECT * FROM usuarios");
50
51             while (rs.next())
52                 out.println("Usuario : " + rs.getString(1));
53
54             desconecta(cid);
55
56         } catch (SQLException e) {
57             out.println(e.getMessage());
58         }
59     }
60 }
61
```

```

62     /**
63      * @see HttpServlet#doPost(HttpServletRequest request,
64      * HttpServletResponse response)
65     */
66     protected void doPost(HttpServletRequest request, HttpServletResponse
67                           response) throws ServletException, IOException {
68         // TODO Auto-generated method stub
69         doGet(request, response);
70     }
71
72     public Connection conecta(String host, String bd, String user, String
73                               pwd) {
74         Connection cid = null;
75
76         try {
77             System.out.println("Conectando a BD...");
78             Class.forName("com.mysql.jdbc.Driver");
79             String url = "jdbc:mysql://" + host + "/" + bd;
80
81             cid = DriverManager.getConnection(url, user, pwd);
82         } catch (Exception e) {
83             out.println(e.getMessage());
84         } finally {
85             return cid;
86         }
87     }
88
89     public void desconecta(Connection cid) {
90         try {
91             cid.close();
92         } catch (SQLException e) {
93             out.println(e.getMessage());
94         }
95     }
96 }
97 }
98

```

En este código, se han definido dos funciones: **conecta** y **desconecta**. La primera recibe cuatro parámetros (el servidor sobre el que conectar, la base de datos, el usuario y el *password*). Por defecto, la base de datos no tiene *password*, y el usuario es **root**. Si se desea modificar, puede hacerse desde la página <<http://localhost/security/index.php>>.

La función **conecta** inicialmente carga el conector mediante **Class.forName ("com.mysql.jdbc.Driver")**. Después se define la URL de conexión y se crea la conexión con **DriverManager.getConnection** para que sea devuelta al final de la función.

En el método **doGet** del servlet se utiliza la función **conecta**. Sobre el resultado que devuelve se crea un objeto de tipo **Statement**, con el fin de ejecutar la sentencia “**SELECT * FROM usuarios**” que obtendrá todos los usuarios disponibles para almacenarlos en un **ResultSet**.

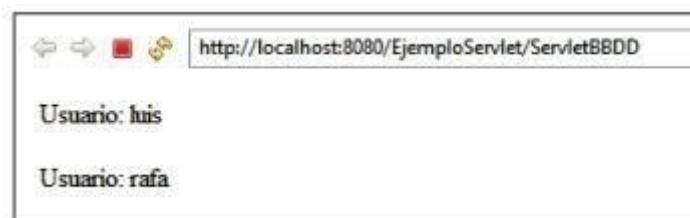
El código del *servlet* concluye recorriendo el **ResultSet** devuelto con un **while**. Como solo interesa el primer campo que almacena cada entrada de la base de datos (el segundo campo es la contraseña),

se utilizará el método **getString(1)** para obtenerlo. También se finaliza la conexión con la base de datos mediante la función **desconecta**.

Hay que indicar que el *servlet* no funcionará a no ser que se incluya en el proyecto el conector correspondiente. El conector se descargará en la siguiente URL, seleccionando la opción **TAR comprimido para plataforma independiente**:

<http://dev.mysql.com/downloads/connector/j/>

El archivo descargado se descomprimirá, copiando el **mysql-connector-java-x.x.xx-bin.jar** a la carpeta lib dentro del **WEB-INF** del proyecto. La ejecución mostrará a los dos usuarios contenido en las tablas de la base de datos.



Acceso a directorios y otras fuentes de datos

En ejemplos anteriores se ha accedido a un archivo externo, siempre que esté dentro del contenedor del servlet (al mismo nivel que **WEB-INF**).

```
1 protected void doGet(HttpServletRequest request, HttpServletResponse
2 response) throws ServletException, IOException {
3     response.setContentType("text/html");
4     PrintWriter out = response.getWriter();
5
6     InputStream input =
7         getServletContext().getResourceAsStream(fichero);
8     BufferedReader br = new BufferedReader(new
9         InputStreamReader(input));
10
11    String s;
12
13    while ((s = br.readLine()) != null)
14        out.println("<p>" + s + "</p>");
15    br.close();
16    out.close();
17 }
```

Se crea el flujo de entrada sobre el fichero, usando la siguiente línea:

```
1 ...
2
3 InputStream input = getServletContext().getResourceAsStream(fichero);
4
5 ...
```

Posteriormente, se crea un *buffer* de lectura sobre el flujo de entrada:

```
1 ...
2
3 BufferedReader br = new BufferedReader(new InputStreamReader(input));
4
5 ...
```

Y se recorre el fichero con un **while** hasta que se alcance el final:

```
1 ...
2
3 while ((s = br.readLine()) != null)
4   out.println("<p>" + s + "</p>");
5
6 ...
```



El fichero tiene que estar dentro del contenedor del *servlet* (al mismo nivel que **WEB-INF**)

Programación de transacciones

Las transacciones sirven para que una base de datos se mantenga consistente cuando haya una modificación o inserción de datos, de tal forma que cualquier error devuelva a la base de datos a un estado consistente. Antes de continuar con la explicación, hay que presentar dos conceptos:

- **Commit:** es la orden con la cual debe terminar una transacción. De esta manera los cambios serán permanentes.
- **Rollback:** vuelve al estado resultado del último commit.

Sabiendo estos dos conceptos, el funcionamiento genérico de una transacción debería ser el siguiente:

1. Desactivar el *commit* Automático. De esta manera, se podrá controlar que no se haga una modificación errónea.
2. Realizar las actualizaciones o inserciones problemáticas.
3. Si no ha habido error, se realiza el *commit*. En caso contrario, se vuelve al estado anterior con *rollback*.
4. Volver a activar el *commit* automático.

Los comandos que aplican estas operaciones en Java se realizan sobre la variable que contiene la conexión a la base de datos (supóngase que se trata de *con*). Estos comandos se exponen a continuación:

- ***con.setAutoCommit(false)*:** desactiva el commit automático.
- ***con.commit()*:** realiza un commit a voluntad.
- ***con.rollback()*:** vuelve al estado anterior.
- ***con.setAutoCommit(true)*:** activa el commit automático.

Actividad 7

Teniendo en cuenta la base de datos utilizada al principio del capítulo, que almacena un usuario y su correspondiente *password*, realice un servlet que efectúe la inserción de una nueva entrada. Las condiciones son:

- Recepción de datos por URL, usando ***response.getParameter()***.
- Controlar con transacciones que el usuario no está repetido.
- Para realizar la inserción se usará ***executeUpdate(insert)***, siendo *insert* la cadena que contiene “**INSERT INTO usuarios (usuario, password) VALUES ('usuario','password')**”. Recordar sustituir *usuario* y *password* por los valores recogidos de la URL.
- Usar los métodos conecta y desconecta, definidos en el ejemplo para crear la conexión con la base de datos (y para desconectar).
- Se comprobará el funcionamiento con usuario y *password* “juan”, para posteriormente repetir el proceso y ver el *rollback*.

Componentes en servidor. Ventajas e inconvenientes en el uso de contenedores de componentes

A estas alturas, está claro que un servidor es el lugar donde se almacenan y ejecutan ciertas aplicaciones web. Con ciertos lenguajes de servidor no hace falta nada más. Una aplicación web escrita en PHP necesitará únicamente de un servidor instalado como Apache. Hablando estrictamente, un servidor web sirve páginas estáticas en HTML y nada más (después, en la práctica, estas páginas son dinámicas, puesto que se generan de manera dinámica).

Un contenedor, por otra parte, es lo que se ha venido usando hasta ahora en la forma de *Apache Tomcat*. Un contenedor crea una especie de entorno que permite manejar el ciclo de vida de los *servlets*.

En otras palabras: los *servlets* son los componentes y el contenedor (*Apache Tomcat*) proporciona la interfaz para que estos componentes puedan funcionar. El contenedor de aplicaciones es absolutamente necesario desde el momento en que un servidor web tiene problemas ejecutando el contenido dinámico de un *servlet*.

Las ventajas de los contenedores vienen determinadas por las utilidades que proporcionan. Potencian aspectos tales como la seguridad y el acceso concurrente, facilitan el control de sesiones, etc. Desventajas en sí no presentan, pero sí es cierto que requieren una mayor atención y exigencia respecto al nivel de configuración.



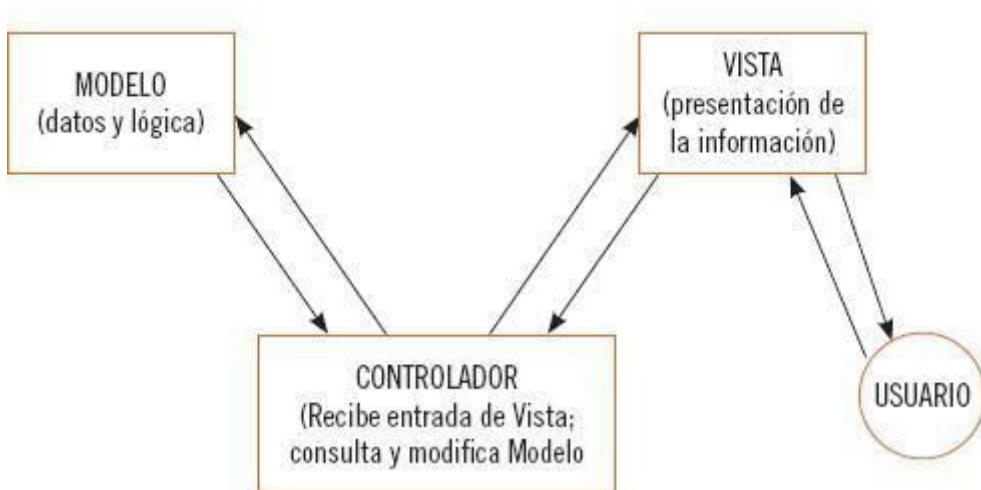
El *servlet* actúa como componente, mientras que *Apache Tomcat* (el contenedor) proporciona el entorno para que el primero se pueda ejecutar.

Modelos de desarrollo. El modelo vista controlador (MVC)

El modelo vista controlador es un patrón de diseño, basado en tres componentes claramente diferenciados:

- **Modelo:** contiene los datos de la aplicación, implementando la lógica y funciones.
- **Vista:** representa la información contenida en el modelo.
- **Controlador:** actúa de intermediario, recibiendo la interacción del usuario a través de la vista y manipulando el modelo.

Modelo vista controlador



Modelo: programación de acceso a datos

En el modelo se representan los datos que manejará el programa. Estos pueden estar representados de muy diversas formas (en forma de base de datos, por ejemplo), pero resulta de suma importancia que se incluya todo el código necesario que permita manipular estos datos. Evidentemente, los métodos dependen del objetivo buscado, pero, por regla general, se implementarán funciones que permiten los ya clásicos modificación, inserción y borrado. En el código propuesto a continuación, se creará un modelo que trabaja con un valor numérico. Los métodos de los que se dispondrá serán:

- Dos constructores: uno vacío y otro que recibe el valor del atributo x para su inicialización.
- Tres métodos modificadores: para incremento, decremento y reinicio.
- Un método de acceso: devuelve el valor del atributo x.

```
 1 public class Model {  
 2     private int x;  
 3  
 4     public Model() {  
 5         x = 0;  
 6     }  
 7  
 8     public Model(int x) {  
 9         this.x = x;  
10    }  
11  
12    public void incrementarX() {  
13        x++;  
14    }  
15  
16    public void decrementarX() {  
17        x--;  
18    }  
19  
20    public void reiniciarX() {  
21        x = 0;  
22    }  
23  
24    public int getX() {  
25        return x;  
26    }  
27 }
```

Vista: Desarrollo de aplicaciones en cliente. Eventos e interfaz de usuario

La vista se encarga de representar la información del modelo, debiendo responder a los cambios que sucedan sobre este. Una vista consta de varios componentes, a los cuales se asociarán detectores de eventos con el fin de capturar interactividad con el usuario. Esto se comentará en el apartado relacionado con el controlador. Ahora, simplemente, se creará la vista usando tres botones y una etiqueta (tipos **JButton** y **JLabel** respectivamente).

```
 1 import javax.swing.*;
 2 import java.awt.BorderLayout;
 3
 4 public class View {
 5     private JFrame frame;
 6     private JLabel texto;
 7     private JButton btIncrementar;
 8     private JButton btDecrementar;
 9     private JButton btReiniciar;
10
11     public View(String text) {
12         frame = new JFrame("View");
13         frame.getContentPane().setLayout(new BotderLayout());
14         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15         frame.setSize(300, 200);
16         frame.setVisible(true);
17
18         texto = new JLabel(text);
19         frame.getContentPane().add(texto, BorderLayout.CENTER);
20
21         btIncrementar = new JButton("Incrementar");
22         btDecrementar = new JButton("Decrementar");
23         btReiniciar = new JButton("Reiniciar");
24
25         frame.getContentPane().add(btDecrementar, BorderLayout.WEST);
26         frame.getContentPane().add(btIncrementar, BorderLayout.SOUTH);
27         frame.getContentPane().add(btReiniciar, BorderLayout.EAST);
28     }
29
30     public JButton getBtIncrementar() {
31         return btIncrementar;
32     }
33
34     public JButton getBtDecrementar() {
35         return btDecrementar;
36     }
37
38     public JButton getBtReiniciar() {
39         return btReiniciar;
40     }
41
42     public void setText(String text) {
43         texto.setText(text);
44     }
45 }
```

Del código se deduce que los componentes van dentro de un contenedor, en este caso un **JFrame**. Ambos, contenedor y componentes, se crean llamando a los constructores de sus respectivas clases (la cadena que reciben es el nombre que muestran).

Los componentes son añadidos al **JFrame** a través de `getContentPane().add()`. Lo más destacable que se puede decir de los componentes es que existen tres métodos que devuelven cada uno de los botones (la utilidad se verá en el controlador). También hay un cuarto que modifica el valor de la etiqueta, que será necesario para actualizar el valor mostrado por la vista.

Respecto al **JFrame** en sí, se define un tamaño (300 x 200), indicando que sea visible. También se especifica la operación por defecto asociada al ícono de cerrar ventana, siendo en este caso la finalización y cierre del programa. Lo que más puede llamar la atención es el uso de un **Layout** (en este caso **BorderLayout**). Un **Layout** rige la distribución que van a tomar los componentes dentro del contenedor. Con **BorderLayout** los elementos se pueden añadir sobre los cuatro puntos cardinales, aparte del centro del **JFrame**. Obsérvese cómo se tienen tres botones ocupando las posiciones *west*, *south* y *east*, mientras que la etiqueta con el valor numérico se distribuirá sobre el centro del **JFrame**.

Programación del controlador

El controlador es el punto de unión entre la vista y el modelo, actuando de comunicación entre ambos. Se encarga de recibir las entradas de información procedentes de la vista y, si el modelo es modificado, se ocupa de que la vista se refresque (así lo representado por la vista siempre es coherente con el modelo). Ahora se pondrá en práctica el concepto de evento introducido en el punto anterior.

Los eventos ya se han usado en algún ejemplo durante este capítulo. Un evento es fruto de una interacción sobre un componente, produciendo la ejecución de un código asociado. Para que esto se produzca, además del código asociado al evento, el componente tiene que estar “a la escucha” de eventos. Todo quedará más claro con el código del controlador:

```
1 import java.awt.event.*;
2
3 public class Controller implements ActionListener {
4     private Model model;
5     private View view;
6
7     public Controller(Model model, View view) {
8         this.model = model;
9         this.view = view;
10    }
11
12    public void iniciar() {
13        view.getBtIncrementar().addActionListener(this);
14        view.getBtDecrementar().addActionListener(this);
15        view.getBtReiniciar().addActionListener(this);
16    }
17
18    public void actionPerformed(ActionEvent e) {
19        if (e.getSource() == view.getBtIncrementar())
20            model.incrementarX();
21        else if (e.getSource() == view.getBtDecrementar())
22            model.decrementarX();
23        else if (e.getSource() == view.getBtReiniciar())
24            model.reiniciarX();
25
26        view.setText(Integer.toString(model.getX()));
27    }
28 }
```

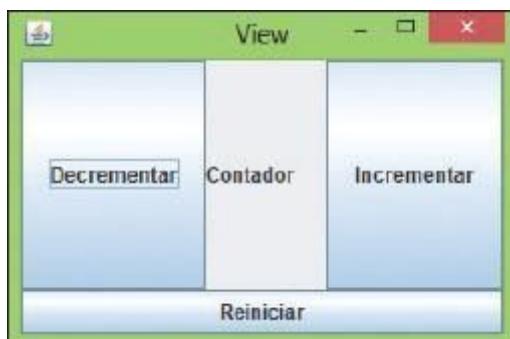
El controlador implementa el interfaz **ActionListener** (recuérdese que una interfaz es una clase cuyos métodos son abstractos, requiriéndose su implementación), lo cual obliga a la presencia del método `actionPerformed`. Se puede decir que el controlador está a la escucha de eventos, pero esto debe particularizarse en los componentes que interesen.

Obsérvese que la vista y el modelo son atributos del controlador, permitiéndose acceder a sus métodos públicos. Gracias a ello, se pueden obtener los tres botones de la vista y añadirles el `actionListener` que hará que se ejecute el código del `actionPerformed` cada vez que se haga clic sobre alguno de ellos.

El método `actionPerformed` es compartido por los tres botones (los tres están a la escucha) así que, para distinguir el objeto que dispara el evento, se usa `e.getSource()`. Este irá comparando los botones a través de tres condicionales. En el momento en que hay un emparejamiento, se invoca a un método del modelo, ya sea para incrementar, decrementar o reiniciar.

Sin embargo, este ejemplo de modelo vista controlador está incompleto, se necesita una clase Lanzadera que sirva de punto de entrada al programa.

```
● ○ ●  
1 public class Lanzadera {  
2     public void static main(String[] args) {  
3         Model model = new Model(0);  
4         View view = new View("Contador");  
5         Controller controller = new Controller(model, view);  
6         controller.iniciar();  
7     }  
8 }
```



Recuerda

El modelo vista controlador es un patrón de diseño que consta de tres componentes. El elemento central es el controlador, que recibe la entrada de información procedente de la vista y se la manda al modelo, haciendo que este se modifique y propagando los cambios a la vista. En Java, el modelo vista controlador se apoya en eventos y componentes de interfaz de usuario.

Documentación del software. Inclusión en código fuente. Generadores de documentación

Uno de los aspectos más importantes durante el desarrollo de un software es el proceso de documentación. Este proceso se puede referir a la creación de manuales técnicos y de usuario, pero hay otra vertiente de la documentación: la documentación dentro del código fuente.

Esta es una de las documentaciones más descuidadas, cuando es vital en el proceso de mantenimiento del *software*. No hay que olvidar que es muy raro que un *software* se quede cerrado, requiriendo a lo largo de su vida alguna actualización para incluir mejoras o solucionar errores. La forma de poner comentarios en lenguajes como Java es la siguiente:

- **Comentarios multilínea:** se pone /* al principio del comentario, y */ para cerrarlo. Algunas veces el IDE con el que se esté desarrollando introduce un * en cada una de las líneas. No influye para nada. Lo importante es que se respeten los caracteres de principio y fin del comentario.
- **Comentarios de línea:** es una forma abreviada de introducir un comentario. Simplemente se antepone // al principio del comentario y automáticamente quedará comentada esa línea.

Estos comentarios son útiles para comentar ciertos aspectos del código que pueden ser de utilidad para una posterior revisión o simplemente para explicar algo que no está suficientemente claro a primera vista. Hay que tener presente que muchas de las cosas que hoy están claras (por la frescura de la programación), seguramente no lo estarán tanto pasado un tiempo. De ahí que haya que incidir en la importancia de este tipo de comentarios.

Quedaría otro aspecto, también íntimamente relacionado con la idea anterior. Es de buena costumbre comentar la cabecera de las funciones y clases, de tal forma que se especifiquen aspectos tales como parámetros, valores de retorno, atributos de la clase, etc.

Esta idea ya se mencionó, pero ahora se avanzará un poco sobre ella. Se parte de la siguiente clase usada en la definición del modelo en el ejemplo de modelo vista controlador:

Resumen

Una aplicación web, por regla general, es el resultado de la combinación de varios lenguajes de programación, tanto del lado del cliente como del lado del servidor. Los primeros proporcionan interactividad, mientras que los segundos ofrecen funcionalidad gracias a la potencia de los scripts ejecutados en el servidor. Existen también tecnologías como AJAX, que permiten invocar scripts del servidor desde el lado del cliente, sin necesidad de realizar una petición que suponga un refresco completo de página.

Las posibilidades de una aplicación web son enormes, permitiendo acceder tanto a ficheros almacenados en el servidor como a bases de datos para realizar consulta y modificación.

Cuando se programa una aplicación web, hay que tener en cuenta aspectos como la seguridad y gestión de errores. Para la primera, es de utilidad la gestión de sesiones, aparte de obligar al usuario a realizar una autenticación ante el sistema (cotejando una base de datos, por ejemplo). Para el segundo, se recomienda el uso de excepciones, con el fin de tratar los errores de una manera elegante y amigable con el usuario.

Para la creación de aplicaciones web (y, por extensión, la creación de cualquier programa) existen varios patrones de diseño. El modelo vista controlador, por ejemplo, divide el programa en tres componentes claramente diferenciados: vista, modelo y controlador, siendo este último el eje central y coordinador entre los dos anteriores.

Resulta sumamente recomendable documentar el código con el fin de facilitar tareas de mantenimiento y extensibilidad. Java tiene una herramienta llamada Javadoc, que permite generación automática de documentación en formato HTML.

Ejercicios de repaso y autoevaluación

1. JavaScript es un lenguaje que se ejecuta en...

- a. el cliente.
- b. el servidor.
- c. Ambas son correctas.
- d. JavaScript no es un lenguaje. Es un complemento del navegador.

2. Complete la siguiente oración.

Un código programado en lenguaje de lado del _____ se ejecuta en el servidor, respondiendo a una _____ de un cliente y generando una _____ que será enviada a dicho cliente.

3. Enumere los lenguajes del lado del servidor.

4. ¿Cuáles de los siguientes son tipos de datos en Java?

- a. Tipo entero (int).
- b. Tipo cadena (String).
- c. Tipo real (float).
- d. Todas las respuestas anteriores son correctas.

5. Los paquetes en Java se importan con la palabra clave...

- a. package.
- b. import.
- c. include.
- d. Ninguna de las respuestas anteriores es correcta.

6. Para realizar una autorización...

- a. se comprueba la autenticación realizada por el usuario.
- b. el usuario introduce una contraseña.
- c. el usuario introduce un identificador.
- d. Ninguna de las respuestas anteriores es correcta.

7. Un descriptor de un servicio web en Java...

- a. es un fichero XML.
- b. establece los parámetros de conexión con una base de datos.
- c. no es editable con un editor de texto.
- d. Todas las respuestas anteriores son correctas.

8. El fichero de Java necesario en un servicio web llevará la extensión...

- a. ".java".
- b. ".exe".
- c. ".class".
- d. Todas las respuestas anteriores son correctas.

9. Una aplicación web puede acceder...

- a. a una base de datos.
- b. a un fichero.
- c. Las respuestas a. y b. son correctas.
- d. únicamente a datos especialmente habilitados para ello en un repositorio.

10. Los parámetros básicos para conectar a una base de datos son:

- a. Servidor, base de datos y contraseña.
- b. Servidor, base de datos, tabla y contraseña.
- c. Servidor, base de datos, usuario y contraseña.
- d. Servidor, base de datos, tabla, usuario y contraseña.

11. De las siguientes afirmaciones respecto a los servlets, diga cuál es verdadera o falsa.

- a. Pueden estar programados en un lenguaje diferente a Java.
 - Verdadero
 - Falso
- b. Requieren de un contenedor como Apache Tomcat.
 - Verdadero
 - Falso
- c. Solo son invocables a través de la URL, no estando permitido su acceso a través de un formulario.
 - Verdadero
 - Falso
- d. Se pueden ejecutar en el lado del cliente.
 - Verdadero
 - Falso

12. Las transacciones en una base de datos...

- a. requieren una activación mediante la palabra clave trans.
- b. permiten acceder a cualquier estado anterior de una base de datos.
- c. sirven para que esta se mantenga consistente.
- d. Todas las respuestas anteriores son correctas.

13. Relacione los siguientes elementos.

- a. PHP.
- b. AJAX.
- c. Apache Tomcat.
- d. JavaScript.
- Lenguaje de cliente.
- Lenguaje de servidor.
- Lenguaje que permite ejecutar scripts de servidor desde el cliente.
- Contenedor.

14. ¿Qué es el modelo vista controlador?

15. Complete la siguiente oración

Un evento es fruto de una _____ del usuario sobre un _____, _____ permitiendo la _____ de un código asociado.