# CURSO DE DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Entornos de desarrollo

**INICIATIVA Y COORDINACIÓN**
**IFP** Innovación en Formación Profesional

*Supervisión editorial y metodológica:*
Departamento de Producto de Planeta Formación

*Supervisión técnica y pedagógica:*
Departamento de Enseñanza de **IFP** Innovación en Formación Profesional

Módulo: Entornos de desarrollo / Desarrollo de aplicaciones web


Barcelona (España), 2017

# INTRODUCCIÓN AL MÓDULO

Un **entorno de desarrollo** es el conjunto de procedimientos y herramientas utilizadas para la creación o mantenimiento de programas informáticos (software). Si bien un entorno de desarrollo debe abarcar la entrada de código de programa a través de un editor de código, las pruebas a través de un compilador y un depurador y su empaquetado final para la entrega al usuario utilizando un constructor de interfaz gráfica (GUI), hay otros aspectos que debe tener en cuenta para ser un nexo útil entre la fase de análisis y la fase de instalación que un desarrollo de software debe contemplar. En ocasiones, el concepto puede también aplicarse a un entorno y elementos físicos que permiten alcanzar los objetivos planteados por un requerimiento de cliente.

Se etiqueta como IDE (*Integrated Development Environment* o Entorno de Desarrollo Integrado) al entorno de programación que contiene un conjunto de procesos, instrumentos y normas que trabajan de forma coordinada para facilitar al programador el control de las diferentes etapas del desarrollo en las que está directamente implicado: entrada de código de programa y pruebas y preparación (empaquetado) del producto final para su inmediata utilización. Puede dedicarse en exclusiva a un sólo lenguaje de programación o bien pueden utilizarse para varios.

En algunos lenguajes de programación, un IDE puede funcionar como un sistema en tiempo de ejecución donde se permite utilizar el lenguaje de programación de forma interactiva sin necesidad de trabajo orientado a archivos de texto.

# UNIDAD FORMATIVA 2

· Diseño y realización de pruebas

· Optimización dedocumentos

# 3. Diseño y realización de pruebas

La realización de pruebas es un hito muy importante en el marco del desarrollo de software. Debemos partir de la base de que no es posible tener en cuenta todas las circunstancias de datos posibles que un programa afrontará cuando sea promocionado a un entorno real de explotación. Teniendo en cuenta dicha imposibilidad, en este apartado estudiaremos cómo diseñar una batería de pruebas, que debe contemplar dos aspectos fundamentales: las pruebas que simulen las casuísticas más habituales que el programa procesará, y aquellas situaciones críticas en las que un fallo del producto podría generar un grave problema. "Lo cotidiano" (cubrir un alto porcentaje de situaciones) y "lo crítico" (evitar circunstancias indeseables y que producirían un significativo quebranto económico de orden empresarial) son los dos puntos guía de los que nos serviremos para realizar las pruebas y poder progresar en un entorno real de explotación con el máximo de garantías posibles.

## 3.1 Planificación de pruebas

Planificar correctamente un período de pruebas comporta tener en cuenta los siguientes aspectos (Figura 3.1):

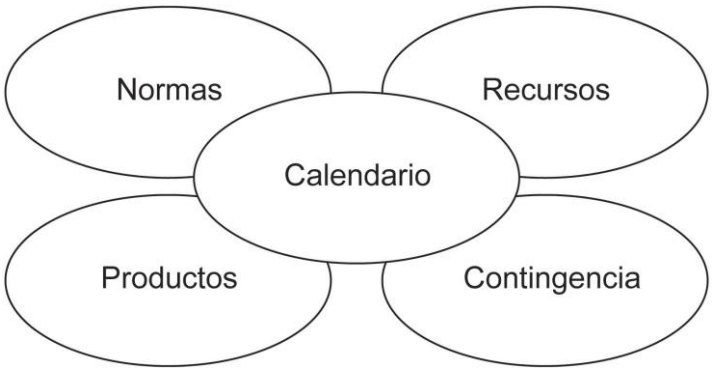


**Figura 3.1**
Principales aspectos implicados en una planificación de pruebas.

- **Calendario.** Cuándo y durante cuánto tiempo podemos hacer pruebas.
- **Normas.** Objetivos de la compañía, ámbito de las pruebas, metodología a utilizar, etcétera.
- **Recursos.** Hardware y software necesarios para los test, herramientas que van a utilizarse y perfiles de los técnicos y usuarios a los que se les requerirá su colaboración.

- **Productos.** Incluye todos los elementos que conforman la futura instalación en entorno real.

- **Contingencia.** A pesar de que no siempre es posible tener en cuenta todas las circunstancias por las que puede desarrollarse una prueba de producto, no conviene olvidar que las cosas pueden no salir como esperamos y que deberemos tener alternativas para ello.

## 3.2 Tipos de pruebas

Una vez finalizado el desarrollo del producto, es necesario comprobar su funcionamiento. Para ello, realizaremos una serie de test que lo pongan a prueba y verifique que los resultados no son aleatorios o motivados por otras causas que no sean las propias de las funcionalidades a instalar (Figura 3.2).

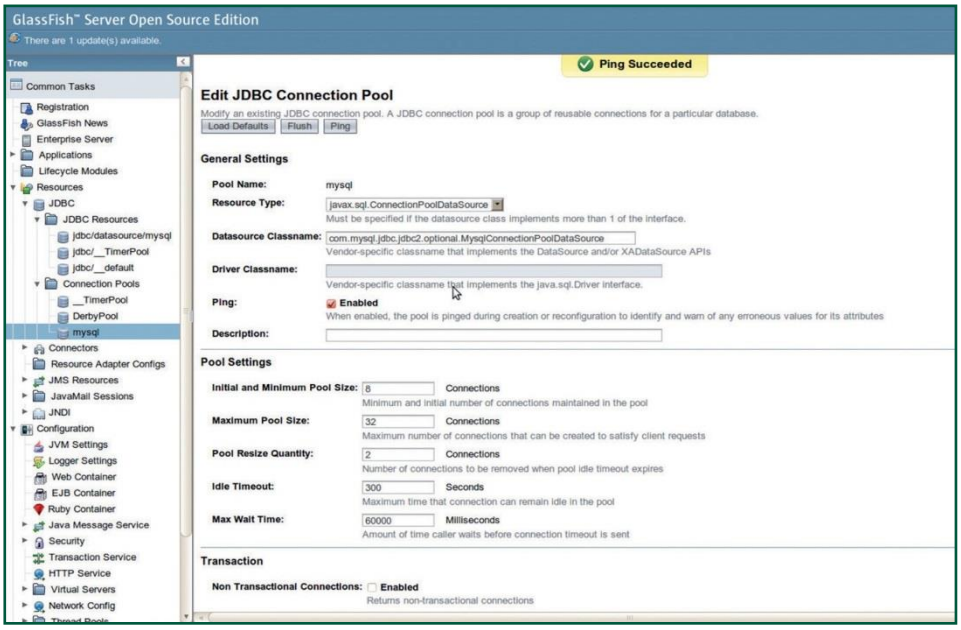


**Figura 3.2**
Una prueba de test debe detectar datos no concordantes con la realidad.

Veamos a continuación algunos tipos de pruebas:

- **Prueba funcional.** Es aquella que comprueba si el producto hace realmente lo que se espera que haga. De lo que se trata es de comparar las exigencias del cliente y la solución ofrecida por el analista a través del diseño técnico con los resultados del software. Para ello, se utilizan unos datos de test preparados para tal efecto y un caso de prueba para que, antes de que el software nos proporcione el resultado, éste se conozca a priori para poderlo valorar (Figura 3.3).


**Figura 3.3**
Un caso de prueba es aquel que debe dar la respuesta antes de que la dé el software probado.

- **Prueba estructural.** Es la que se adentra en el propio programa y, a partir del estudio del código, de su estructura interna y su confrontación con el diseño técnico, llega a las conclusiones necesarias para tomar acciones correctoras si así fuera preciso. Este tipo de prueba utiliza casos de prueba relacionados con la lógica del código fuente, estructuras de datos, tablas, tiempos de ejecución y control correcto de bucles de sentencias. En el caso de que sea un software configurable por el usuario deberemos tener en cuenta el control que se haga de selecciones imposibles o que puedan llevar a errores de computación. Dicha prueba estructural no debería, en teoría, realizarla el propio programador, aunque con frecuencia así se haga, rellenando un documento de test técnico (Figura 3.4).

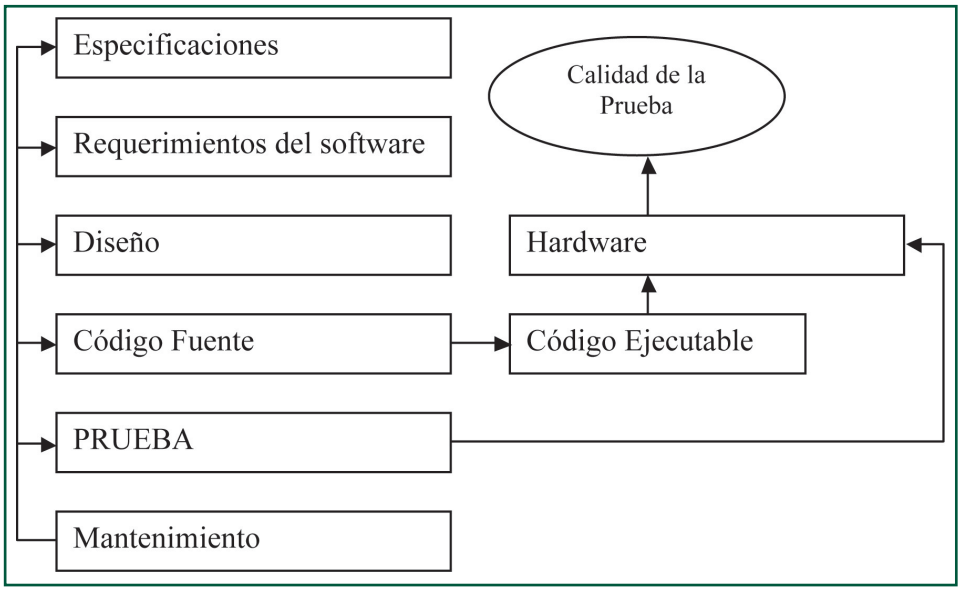


**Figura 3.4**
Componentes principales de una prueba estructural.

**Recuerda**

**Las pruebas funcionales son externas al código del programa, es decir, solo miden resultados. Las pruebas estructurales, sin embargo, entran en el propio código de programa.**

- **Prueba de regresión.** Esta prueba se aplica en el mantenimiento de un software ya existente al que se le quiere realizar una modificación, ya sea porque contenía un error o por una simple mejora del producto. En caso de que sea un nuevo producto, la prueba de regresión se utiliza después de haber realizado el test principal, e implica la realización de cambios. En ese caso, este tipo de prueba intenta probar que nuestra modificación no ha generado nuevos errores o influye negativamente en la ejecución normal del programa. La forma más sencilla de esta prueba consistiría en crear un "caso de prueba" y probar el programa dos veces: una sin la modificación, y la otra con la modificación, para chequear el impacto que ha tenido y corregirlo si es necesario.

## 3.3 Procedimientos y casos de prueba

Los **procedimientos** y **casos de prueba** están relacionados entre sí, pues los primeros, al ser especificaciones formales de los casos de prueba, contienen a éstos últimos e indican las circunstancias en las que deben aplicarse. Veámoslo a continuación:

- **Procedimientos.** Deben contener los requerimientos necesarios para realizar la prueba los recursos (humanos y materiales) que se utilizarán, así como los casos de prueba.

- **Casos de prueba.** Definen una ejecución específica del programa desarrollado examinando aspectos específicos de entrada y salida de datos. Define qué datos de entrada serán permitidos y cuáles serán los esperados en el resultado. Un caso de prueba puede dividirse en varias fases, teniendo, cada grupo de ellos, un cuadro de datos asumido como correcto en el resultado (Figura 3.5).

| **Caso de Prueba:** | | | | |
|---|---|---|---|---|
| **Descripción:** | | | | |
| **Requerimiento de Datos:** | | | | |
| **Número Paso** | **Descripción** | **Resultado esperado** | **Nombre Transacción** | **Tiempo esperado** |
| 01 | Invocar aplicación desde escritorio | Se muestra la pantalla de menú principal | Usuario_login | 5 |
| 02 | Seleccionar 'Búsqueda' en el menú | Se muestra la pantalla de 'Búsqueda' | Seleccionar_búsqueda | 3 |

**Figura 3.5**
Inicio de un documento descriptivo de un caso de prueba.

## 3.4 Herramientas de depuración

Las herramientas de depuración son unos programas cuyo objetivo principal es el de probar el funcionamiento de otros programas informáticos. Para ello se utiliza una serie de acciones entre las que destacamos las siguientes:

- **Puntos de ruptura (*Breakpoint*).** El programador puede poner en el código fuente unas señales que la herramienta de depuración interpretará como un punto de parada de ejecución. Cuando el control del programa llegue a ese punto, el programa se detendrá temporalmente, pero mantendrá los valores de variables, funciones y objetos en la memoria. Esto permitirá al programador conocer el valor o estado de cada elemento que deberá comprobar y modificar si fuese necesario (Figura 3.6). Una vez analizados los datos, es posible continuar la ejecución a partir de dicho punto.

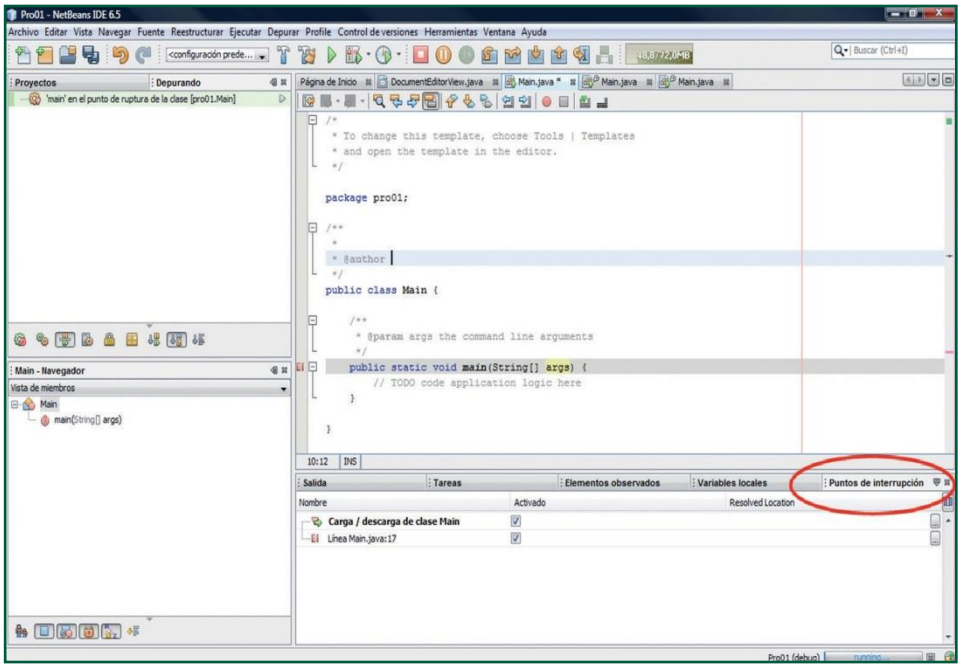


**Figura 3.6**
Puntos de ruptura (*Breakpoint*).

- **Tipos de ejecución.** Según la naturaleza de lo que pretendamos comprobar, podremos utilizar uno de los siguientes métodos:
  - **Ejecución línea a línea.** Se emplea cuando el problema lo tenemos muy delimitado o deseamos una ejecución detallada. Por cada línea de código el programa se detiene.
  - **Ejecución por grupos.** Mientras no termina una declaración completa de sentencias, el programa no se detiene.

- **Examinadores de variables.** Las ventanas de salida en modo "depuración" presentan los valores que tienen las variables en un momento concreto. A partir de un punto de interrupción también podemos interrogar al entorno de desarrollo acerca de un valor determinado. Existen, además, otras formas dinámicas de examinar variables, que provocan puntos de interrupción cuando una variable tiene un valor marcado por el programador.

- **Utilización de herramientas de depuración.** Veamos un ejemplo de utilización de las herramientas de depuración. Para ello marcaremos una línea como ruptura. En primer lugar crearemos una impresión básica "Hola mundo!" mediante lenguaje Java sobre NetBeans.

  1. Abrimos un nuevo proyecto Java mediante Archivo / Proyecto Nuevo y seleccionar categoría Java.
  2. Introducimos en el editor después de la declaración:

     ```
     public static void main(String[] args) {
     ```

     la sentencia:

     ```
     System.out.println("Hola  mundo!");
     ```

  Comprobamos mediante la opción de menú Ejecutar que no ha habido error de sintaxis. Una vez realizado este paso, veamos cómo utilizamos la herramienta de depuración al haber marcado un punto de interrupción. Procedemos siguiendo los pasos descritos a continuación:

  1. Abrimos NetBeans como proyecto nuevo de Java.
  2. Sustituimos en el editor el texto "//TODO code application logic here" por la instrucción de presentación:

     ```
     System.out.println("Hola mundo!");
     ```

  3. Marcamos la línea en vídeo inverso.
  4. Con la opción "Depurar / Nuevo punto de interrupción" marcamos la línea entrada.
  5. Usamos "Depurar / Ocultar Mostrar línea de punto de interrupción" para ver la línea marcada como ruptura.
  6. Usamos "Depurar Debug main Project".

  Una vez realizados todos los pasos, podremos observar en la pantalla de salida cómo nos avisa de que ha llegado al punto de interrupción. Esto nos permitirá conocer los valores hasta este punto y poder realizar modificaciones si fuese necesario.

## 3.5 Validaciones

El entorno de desarrollo nos ofrece una serie de ayudas que nos evita posibles errores. El editor puede obligar al programador a especificar todos los tipos de variables declaradas. El control de los límites de las tablas y las pruebas aisladas de las rutinas complementa una buena técnica de **validación**. Si fuese necesario, podríamos también configurar la carga de datos de salida en ficheros temporales para comprobar que los cálculos son los correctos. Activar todos los filtros que el editor nos proporciona para evitar errores de asignación y de coherencia interna es una buena práctica; pero, antes que nada, una buena validación exige la previa identificación de lo que vamos a comprobar (Figura 3.7).

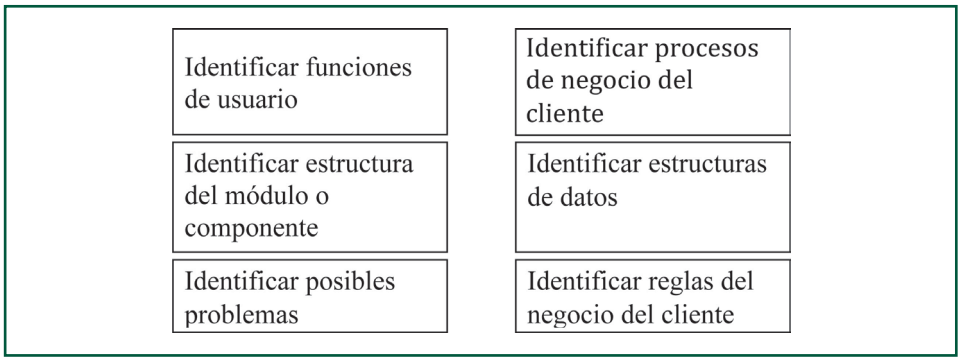


**Figura 3.7**
La identificación del entorno es crucial para poder validar con garantías.

Supongamos que estamos programando una rutina de cálculo de fecha de entrega de mercancías que estaría dentro de un aplicativo web de una empresa de mensajería. Nuestra rutina, en ese caso, deberá tener en cuenta los siguientes ítems:

- Fecha de entrega del producto a la mensajería.
- Fecha de entrega del producto al camión.
- Calendario laboral del sector del transporte.
- Duración del recorrido.
- Cálculo de la fecha de entrega posible.

Se trata de una buena práctica utilizar el editor para que nos audite la entrada de código. En Visual Basic podemos obligar a declarar las variables antes de usarlas, de esta manera el compilador nos avisará de que nos hemos equivocado al emplear una variable que no existe. Esto se consigue con la instrucción "Option Explicit" al inicio del programa, si bien podemos ir más allá e indicar al editor que nos obligue a entrar, además de la declaración de la variable, el tipo de datos que almacenará y tenerlo en cuenta en cada conversión que hagamos. Podemos indicarlo con la instrucción "Option Strict".

A continuación, utilizaremos un fichero de entrada con múltiples fechas al azar y procesaremos las fechas con nuestra rutina, grabando en un segundo fichero nuestro cálculo estimado de fecha de entrega. Tendremos, entonces, dos ficheros, y cada elemento del primer fichero tendrá su pareja en el segundo fichero. Escogeremos luego fechas límite (año bisiesto, último día de mes) y valores de fecha medios y observaremos qué ha calculado nuestra rutina. Lo compararemos con un cálculo manual y comprobaremos que nuestra rutina tiene en cuenta el calendario laboral y la particularidad de los años bisiestos, así como la duración de los recorridos.

## 3.6 Pruebas de código

Probar todo un programa, es decir, comprobar todas las posibilidades de ejecución que pueden darse, es técnicamente imposible. Por ello, es preciso determinar hasta qué nivel de pruebas queremos llegar. La naturaleza del negocio, en donde el software se sitúa, y el presupuesto económico son dos elementos que exigen la aplicación de un tipo de pruebas u otro. Veamos algunas de ellas:

- **Prueba de cubrimiento.** La forma más sencilla es la de contar manualmente las veces que una línea de código se ejecuta. Los entornos de desarrollo ofrecen herramientas para contabilizar de forma automática las líneas y las veces que se ejecuta (Figura 3.8). Podemos omitir el detalle hasta la línea y controlar ejecuciones de áreas funcionales o rutinas.

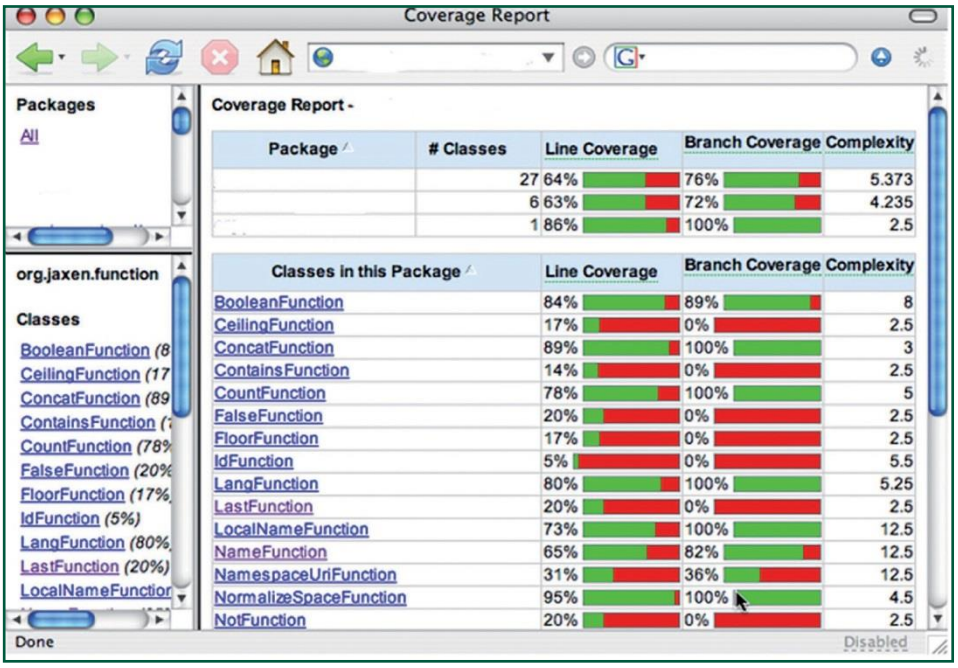


**Figura 3.8**
Preparación de una prueba de programa para comprobar el tanto por ciento de código utilizado.

- **Clase de equivalencia.** Se define como un modelo en el que unos datos son equivalentes a otros. De esta forma, una ingente cantidad de datos a probar en un test puede reducirse a unos pocos grupos de datos. Cada grupo se obtendrá por datos equivalentes. Se determina que es suficiente probar un elemento de cada grupo, siendo un valor medio de cada clase aquel considerado más representativo. No hay un método único para detectar las clases. Cada vez que obtenemos un dato, o grupo de datos de entrada necesarios, se generan tres clases: el grupo de valores por debajo de los necesarios, el grupo de los necesarios y aquellos que están por encima.

Los valores frontera o límite son aquellos que están en el borde de cambio de una clase de equivalencia a otra. Es conveniente probarlos, ya que suelen tener un porcentaje de riesgo más elevado de provocar problemas al software. Es conveniente utilizar dos valores límite por clase: el valor mínimo y el valor máximo.

## 3.7 Normas de calidad

Las normas de calidad son documentos publicados por organismos competentes en los que se especifican los requisitos, las reglas y las formas de trabajo que una empresa debe seguir para poder ser considerada "certificada" en dicha norma. Las empresas de software también pueden necesitar certificarse en calidad (Figura 3.9).

**Figura 3.9**
La certificación en calidad abarca importantes flujos de datos en la empresa. La documentación utilizada puede ser objeto de una mejora constante.

En primer lugar, porque el auditor que compruebe si seguimos la normativa nos corregirá mediante "no conformidades" que deberemos eliminar con el tiempo; en segundo lugar porque, en un sistema económico globalizado, se exige poseer unas conductas de trabajo acordes con los requisitos del cliente; y, por último, porque los futuros accionistas desearán conocer en qué tipo de compañía están poniendo su dinero. Veamos a continuación cuál es la normativa más común:

- **ISO/IEC 15504 SPICE - Software Process Improvement Capability Determination.** Este grupo de normas facilita la mejora continua de los procesos de las organizaciones, especialmente aquellos relacionados con el desarrollo y mantenimiento del software. Se basa en la evaluación, a través de niveles de madurez de la empresa, entendiendo por éstos el conjunto de procesos predefinidos que ayudan a una organización a mejorar en el desarrollo del software (Figura 3.10).

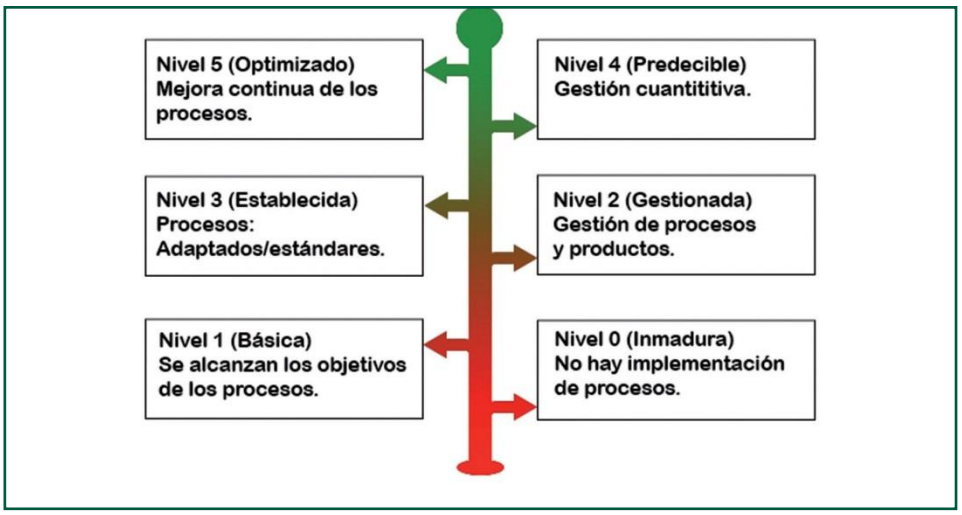


**Figura 3.10**
Niveles de madurez de una empresa.

- **ISO 9001.** Elaborada por la Organización Internacional para la Estandarización. Va más allá del desarrollo del software y pretende abarcar otros ámbitos de la empresa, como el servicio al cliente, sistemas de mantenimiento y flujos administrativos, entre otros.

- **ISO IEC 90003 2004.** Esta normativa sirve de guía para una constante mejora. Se aborda desde diferentes ámbitos:
  - Requerimientos del sistema empresarial.
  - Requerimientos de gestión.
  - Requerimientos de recursos.
  - Requerimientos de control y realización.
  - Requerimientos de contingencia y reparación.

## 3.8 Pruebas unitarias. Herramientas

Una **prueba unitaria** es aquella que se realiza en un sólo componente del programa de forma aislada. Dicho componente puede ser un módulo o una clase, si bien el concepto de unidad será manipulado por el programador para aplicarlo allí donde considere oportuno. El aislamiento del componente del resto de la aplicación es, en ocasiones, complicado, ya que éste puede depender de las interacciones que tenga el analizado con el resto. Algunas herramientas que facilitan dicha tarea son:

- **JUnit.** Conjunto de clases que asiste a los desarrolladores de Java para comprobar si un programa funciona correctamente. En caso de que el valor devuelto por un proceso no se corresponda con el esperado, es posible integrar JUnit en entornos de desarrollo, como NetBeans o Eclipse, a través de los complementos o *plugins* (Figura 3.11).

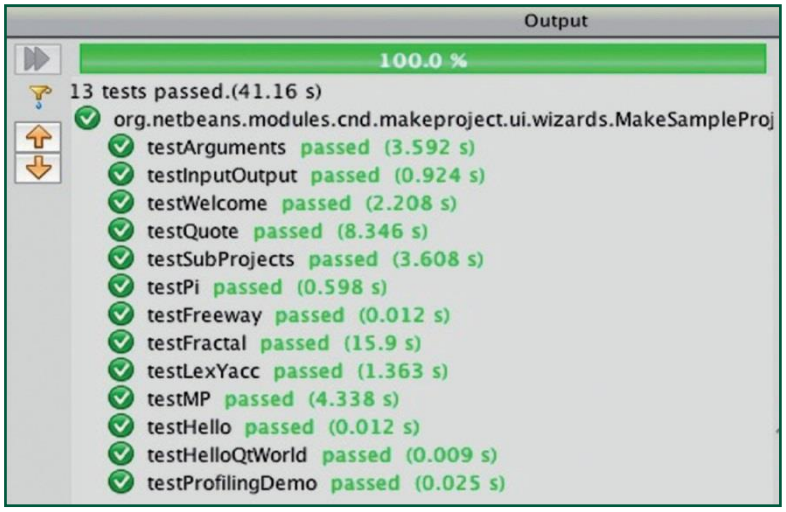


**Figura 3.11**
Las herramientas para pruebas unitarias se integran en los entornos de desarrollo.

- **PHPUnit.** Orientado a la creación de juegos de pruebas unitarias en lenguaje PHP.

- **NUNit.** Un gestor de pruebas unitarias para los lenguajes .NET. La propia herramienta está escrita en C#.

## 3.9 Automatización de pruebas

**Los asistentes informáticos de pruebas de software** son muy adecuados para los entornos de desarrollo. La posibilidad de automatizar una prueba en un software mediante otro software, confiere una gran habilidad para detectar errores. En general, esta automatización puede orientarse hacia dos áreas concretas:

- **Prueba de código.** Es la más frecuente. Módulos, librerías y clases se prueban mediante la acción de entrar datos y comprobar que los valores retornados son los correctos.

- **Interfaz gráfica de usuario.** Un formulario gráfico que interacciona con un usuario es puesto a prueba mediante la simulación de entradas por teclado y clics de ratón, lo que permitirá comprobar el comportamiento que tiene nuestra interfaz gráfica.

Algunas herramientas que pueden asistirnos en la automatización de pruebas:

- **HP Unified Functional Testing (UFT).** Este software permite la automatización de pruebas funcionales y de regresión. El producto utiliza el lenguaje Visual Basic Script para gestionar las pruebas: http://www8.hp.com/us/en/software-solutions/unified-functional-automated-testing/index.html

- **Microsoft Test Manager.** Esta herramienta permite planificar, crear y ejecutar casos de test, además de su automatización. Mediante la pantalla 'Test Plan Manager' es posible configurar y construir casos de test: https://msdn.microsoft.com/es-es/library/jj635157.aspx

- **Selenium.** Automatización de pruebas para aplicaciones web. Permite pruebas de regresión, buena comunicación con el desarrollador, ilimitadas iteraciones en las ejecuciones de código, documentación de casos de test e informes a medida: http://seleniumhq.org/

## 3.10 Documentación de pruebas

Existen infinidad de documentos utilizados en la etapa de pruebas. La IEEE 829 estándar es un intento por "normalizar" una determinada documentación. Consta de tres grupos de documentos, cada uno de ellos con sus propios formularios estándar, que deberán rellenarse con la información adecuada:

- Preparación de las pruebas:
  - Declaración del Plan.
  - Especificaciones del diseño.
  - Especificaciones de los casos de test.
  - Procedimiento.
  - Detalle de los elementos a probar.

- Ejecución de las pruebas:
  - Registro detallado de cada test.
  - Incidentes acaecidos durante las pruebas.

- Finalización de las pruebas:
  - Informe resumen.

Por último, es preciso utilizar una gestión documental de las copias digitales de los documentos o bien tener un sistema adecuado de archivo en papel; ya que, si no podemos encontrar un documento, es, de hecho, como si no existiera.

## Resumen

Es importante diseñar previamente el plan de pruebas, pues nos marcará el camino a seguir e impedirá desvíos no deseables. Los principales agentes implicados en un plan de pruebas estándar son el calendario, las normas, los recursos, los productos y las posibles contingencias.

Los principales tipos de pruebas son los siguientes: funcional (externa al código, se observan los resultados), estructural (controlando el código de ejecución se observa internamente el programa) y regresión (para comprobar que nuestro cambio no influye en el resto del programa).

Los procedimientos de una prueba son las normas o requisitos necesarios para realizarla. Su materialización se consigue mediante el caso de prueba, es decir, plasmar dicho procedimiento mediante una prueba de datos concreta en la que se deciden unos datos de entrada y se espera una respuesta ya conocida.

Si así fuera necesario, podremos utilizar las herramientas de depuración para detectar posibles fallos de software y 'ralentización' de la ejecución del programa como forma de evidenciar fallos ocultos.

Como programadores, podemos probar nuestro código mediante 'Cubrimiento' (veces que una sentencia de programa se ejecuta), 'Clase de equivalencia' (por la que utilizaremos valores representativos de diversas casuísticas) o 'valores límite' (fuente de posibles errores, ya que son datos que están en el límite entre clases de equivalencia).

Las pruebas unitarias (de una parte concreta del código) y la automatización de pruebas (mediante módulos específicos que simulan situaciones reales) son los complementos ideales de una buena batería de pruebas.

Por último, y no por ello menos importante, el estar amparados por una metodología que obliga a unas normas de actuación de aseguramiento de la calidad puede sernos muy útil porque, en última instancia, nos generará una documentación correcta y susceptible de ser auditada bajo la normativa de calidad a la que nos hayamos adscrito.

# Ejercicios de autocomprobación

**Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F):**

1. Con la prueba funcional se comparan las exigencias del cliente y la solución ofrecida por el analista a través del diseño técnico con los resultados del hardware.

2. La prueba estructural utiliza casos de prueba relacionados con la lógica del código fuente, estructuras de datos, tablas, tiempos de ejecución y control correcto de bucles de sentencias.

3. Los procedimientos y casos de prueba están relacionados entre sí, pues los primeros, al ser especificaciones formales de los casos de prueba, contienen a éstos últimos e indican las circunstancias en las que deben aplicarse.

4. Las herramientas de depuración son unos programas cuyo objetivo principal es el de cambiar el funcionamiento de otros programas informáticos.

5. Los casos de prueba definen una ejecución específica del programa desarrollado examinando aspectos específicos de entrada y salida de datos.

6. Cuando el control del programa llegue al punto de ruptura, el programa se detendrá temporalmente, pero mantendrá los valores de variables, funciones y objetos en la memoria.

7. Los valores frontera o límite son aquellos que están en el borde de cambio de una clase de equivalencia a otra.

**Completa las siguientes afirmaciones:**

8. Planificar correctamente un período de pruebas comporta tener en cuenta los siguientes aspectos: normas,_____, calendario, _____ y_____.

9. Las_____de_____son documentos publicados por organismos competentes en los que se especifican los requisitos, las reglas y las formas de trabajo que una empresa debe seguir para poder ser considerada "_____" en dicha norma.

10. Selenium es una_____de pruebas para aplicaciones web, que permite pruebas de_____, buena comunicación con el _____, ilimitadas_____en las ejecuciones de código, documentación de casos de test e informes a medida.

Las soluciones a los ejercicios de autocomprobación se encuentran al final de esta Unidad Formativa. En caso de que no los hayas contestado correctamente, repasa la parte de la lección correspondiente.

# 4. Optimización de documentos

Según hemos visto en la unidad 1, los procesos informáticos suelen simular aspectos concretos del funcionamiento de la realidad para hacerlo más controlable e inteligible. El **modelo de realidad**, al que imita, es totalmente cambiante, por lo que no es extraño pensar que un proyecto informático vaya evolucionando a lo largo del tiempo de forma dinámica por lo que, postulados que al principio eran de indiscutible vigencia, van pasando a un segundo plano o simplemente desapareciendo a medida que el proyecto avanza. Este dinamismo de los proyectos informáticos, en los que también incluimos las fases de mantenimiento, hacen de la **optimización** y la **mejora continua** dos aspectos indispensables para que el material informático evolucione paralelamente a esa realidad, muchas veces de apariencia caprichosa, y que sigue unas reglas no siempre claras, pero que el software debe imitar.

En esta unidad abordaremos los procesos controlados de la evolución del material de software y de su documentación. Para ello, utilizaremos herramientas informáticas, pero también aquellas de desarrollos metodológicos en los que analizaremos un modo de hacer que, en el pasado, sirvió para resolver problemas similares a los que tenemos actualmente.

## 4.1 Concepto de refactorización

**Refactorizar** (*Refactoring*) es una técnica que consiste en modificar la estructura interna de un programa para mejorar su comprensión, y, de este modo, garantizar que sea más sencillo su futuro mantenimiento (Figura 4.1). Hablaremos de refactorización cuando los cambios realizados en el código no afecten al funcionamiento del programa, es decir, el programa debe hacer exactamente lo mismo que hacía antes de los cambios; en el caso de que funcione de forma diferente no podremos hablar de refactorización sino de cambios o adición de funcionalidades.

**Recuerda**

**Un usuario no debe darse cuenta nunca de que un código ha pasado por un proceso de refactorización, ya que ningún cambio funcional debe realizarse. Los cambios deben ser pequeños para poder probarlos fácilmente.**

### 4.1.1 Limitaciones

La refactorización, como técnica, se manifiesta en el marco de la realidad de las empresas, es decir, con una serie de limitaciones, como plazos, presupuestos y recursos finitos, por lo que probablemente nunca podremos refactorizar tanto como desearíamos. El equilibrio entre lo necesario, por un lado, y el tiempo, los costes y los recursos por el otro, hará que nuestra refactorización sea la mejor que podía hacerse en ese momento concreto. Veamos cuáles son algunas de estas limitaciones:

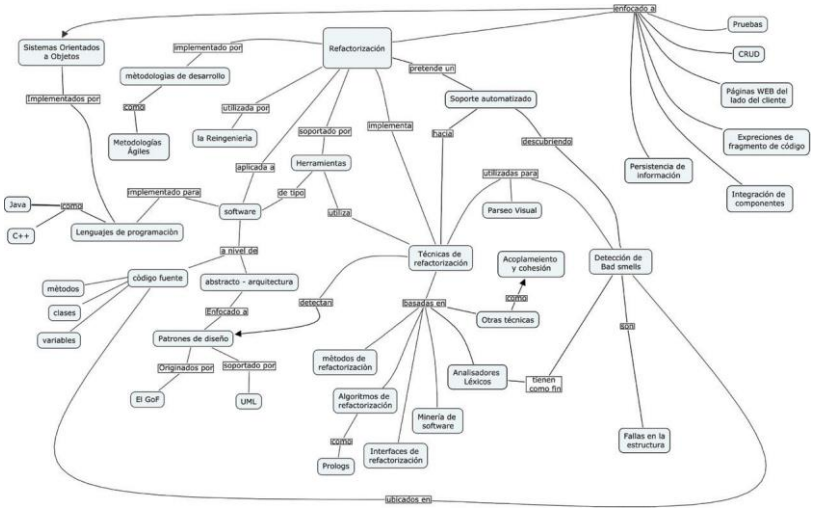


**Figura 4.1**
Mapa de la refactorización de una empresa.

- Los proyectos suelen estar condicionados por el aspecto económico.
- La refactorización no puede facturarse a un cliente inmediatamente.
- El programador que "limpia" el código puede no ser el mismo que el creador, con lo que se corre el riesgo de que entren nuevos errores.
- Las modificaciones o cambios han de poder probarse sin esfuerzo, por lo que hay que tener preparadas unas pruebas funcionales o unitarias.
- Si se trata de un código antiguo en el que han trabajado muchos programadores, algunos de ellos noveles, podemos encontrarnos con un auténtico lío cuya mejora puede costar cara (Figura 4.2).
- A veces no podemos cambiar la interfaz de usuario y complica la refactorización.

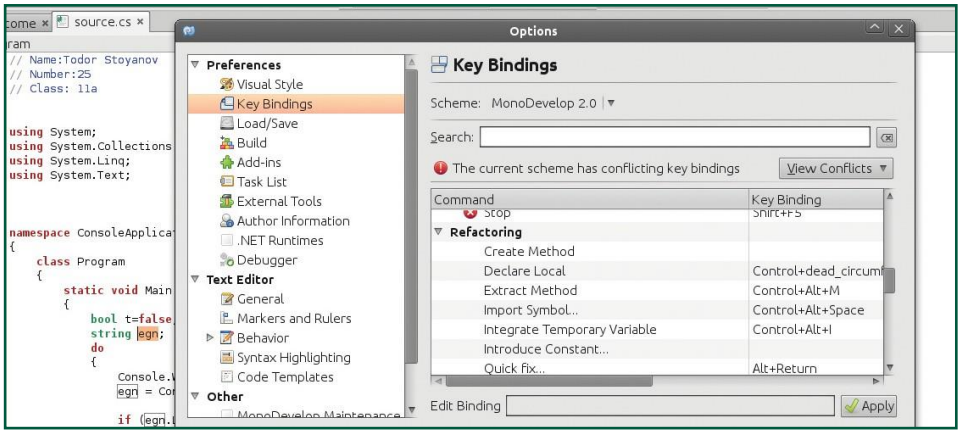


**Figura 4.2**
La refactorización vuelve a poner en orden las clases y el resto de componentes del software, armoniza sus relaciones y evita el caos.

### 4.1.2 Patrones de refactorización más usuales

Desde hace tiempo, los **patrones de diseño** han sido utilizados en la programación orientada a objetos. Suele considerarse un patrón de diseño la solución a un problema que se repite en el tiempo infinidad de veces. Se trata de ideas, sugerencias y plantillas que pueden ser transformadas en código, seguidas de forma total o parcialmente, y cuyo funcionamiento ha estado demostrado innumerables veces. Los patrones también pueden ser útiles a la hora de refactorizar y, por lo tanto, de mejorar la claridad y estructura del código. Los patrones pueden clasificarse en tres grandes grupos:

- **Patrones de creación.** Ayudan a la creación y configuración de objetos; existe un mayor amoldamiento de creación de objetos ante un caso determinado:
  - ***Simple Factory*** **(Fábrica simple).** Clase utilizada para la creación de instancias de objetos.
  - ***Factory Method*** **(Método de Fábrica).** Define una interfaz para crear objetos y permite a las subclases decidir la clase a instanciar.
  - ***Abstract Factory*** **(Fábrica Abstracta).** Interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases.
  - ***Builder*** **(Constructor).** La construcción de un objeto se separa de su representación, por lo que pueden crearse diferentes representaciones con el mismo proceso de construcción (Figura 4.3).

| | |
|---|---|
| Builder | Clase que especifica un interfaz abstracto para crear partes de un producto objeto. |
| ConcreteBuilder | Construye y junta partes del producto que implementa el 'Builder' interfaz. |
| Director | Clase que construye un objeto complejo usando el interfaz 'Builder' |
| Producto | Representa el objeto complejo que está siendo construído. |

**Figura 4.3**
Elementos que componen el patrón *Builder*.

- **Patrones de estructura.** Asisten en la generación de grupos de objetos en estructuras mayores. Se adecuan a la forma de combinar clases y objetos en el montaje de dichas estructuras:
  - ***Adapter*** **(Adaptador)**. Hace de intermediario entre dos clases que posean interfaces incompatibles y, por lo tanto, de difícil comunicación (Figura 4.4).
  - ***Bridge*** **(Puente).** Descompone un componente en dos jerarquías de clase: una abstracción funcional (algo que hace) y una implementación (la cosa en sí).
  - ***Decorator*** **(Decorador).** Agrega o limita competencias a un objeto.

- **Patrones de comportamiento.** Conforman el flujo y tipo de comunicación entre los objetos de un programa.
  - ***Chain of Responsibility*** **(Cadena de responsabilidad).** La cadena de mensajes puede ser respondida por más de un receptor.
  - ***Command*** **(Orden).** Representa una petición con un objeto.
  - ***Mediator*** **(Mediador).** Se introduce un nuevo objeto que administra los mensajes entre objetos. Una Comisaría de Policía que transmite información a los coches patrulla sería un ejemplo en la vida cotidiana.

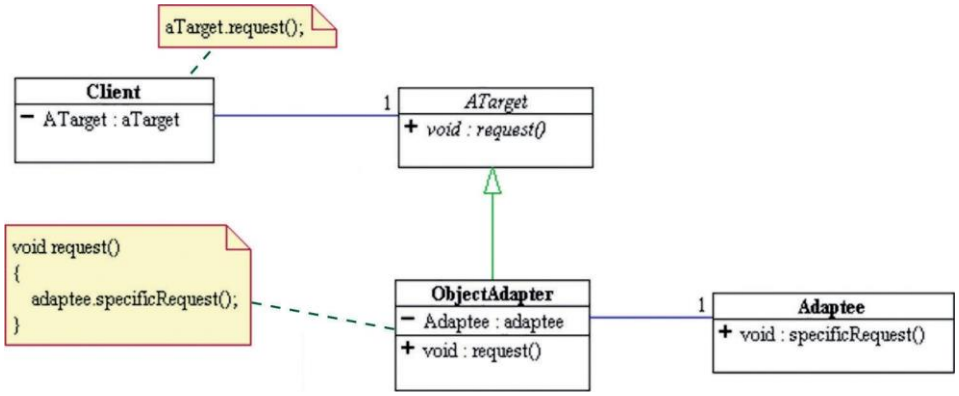


**Figura 4.4**
En la vida real, un adaptador universal de enchufes, para poder utilizar nuestros electrodomésticos en cualquier país, sería un ejemplo de adaptador.

Realicemos pruebas utilizando el entorno de desarrollo NetBeans.

- Crearemos un nuevo proyecto de Java.

- Asignamos las variables enteras "suma1", "suma2" y "suma3" con los valores 1, 2 y 3 respectivamente.

- Refactorizamos el código cambiando la asignación de las variables por un array de tres elementos: el primer elemento tiene valor 1; el segundo, 2, y el tercero, 3.

- Diseñamos un caso de prueba para comprobar la refactorización.

- Ejecutamos el programa antes de la modificación, medimos del tiempo de ejecución y comprobamos los valores. Realizamos el cambio en el programa.

- Ejecutamos el programa con el cambio, medimos del tiempo de ejecución y comprobamos los valores.

- Ejecutamos el programa como el usuario y constatamos que no ha habido ningún cambio funcional.

### 4.1.3 Analizadores de código

Un **análisis de código** consiste en chequear el código fuente con la intención de realizar posteriormente trabajos de mantenimiento (refactorización). Algunas de las habilidades que los analizadores de código externos tienen están implementadas en los propios depuradores de los entornos de desarrollo. Se justifica el analizador de código externo para aspectos peculiares de depuración. Podemos clasificar los análisis de código en dos tipos:

- **Análisis dinámico.** Se ejecuta el programa para detectar defectos de ejecución:
  - Valgrind Tool Suite (C / C++).
  - IBM Rational AppScan. Chequea las vulnerabilidades de seguridad.

- **Análisis estático.** Se examina el código sin ejecutar el programa:
  - FxCop .NET Framework 2.0. Ofrece información acerca del diseño, localización, rendimiento y mejoras de la seguridad.
  - PMD Rule Designer. Chequea código fuente Java encontrando errores, variables no usadas y código duplicado (Figura 4.5).

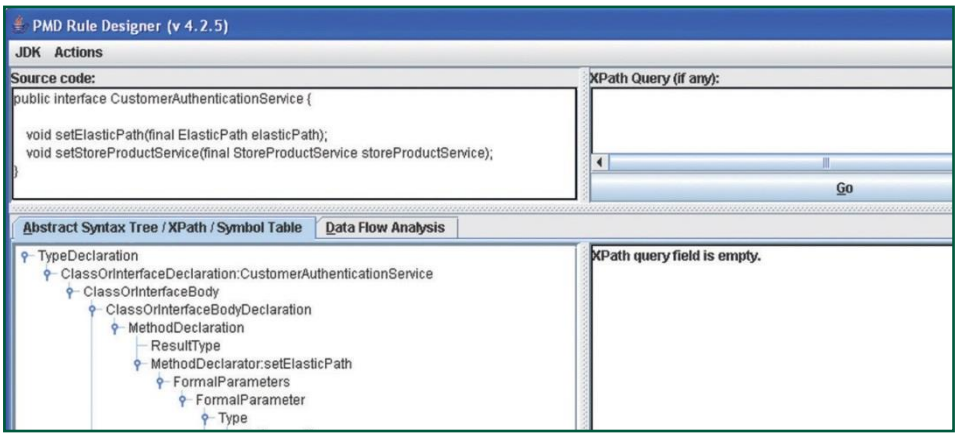


**Figura 4.5**
PMD Rule Designer chequea código fuente Java buscando errores.

**Ejemplo de uso de un analizador de código bajo NetBeans**

Veamos cómo sería la revisión del código fuente usando un analizador de código en este caso bajo NetBeans.

- Comprobamos que tenemos instalado el plugin PMD bajo NetBeans. Si no es el caso, lo descargamos de sourceforge.net. El plugin tiene extensión ".nbm". Usaremos "Herramientas / complementos / Descargados / Añadir complemento".

- Creamos un proyecto nuevo de Java.

- Insertamos este código después de la sentencia

```
"public static void main (string[] args) ("
                    Int suma1;
                    Int suma2;
                    Int suma3;
                    suma1 = 1;
                    suma2 = 2;
                    suma3 = 3;
```

- Ejecutamos el analizador de código y detectamos el resultado.

- Buscamos en la ventana de proyectos del IDE la clase creada (si no le hemos puesto nombre, se llamará "javaApplicationX.java", donde X es un número que depende del número de proyectos que hayamos creado previamente).

- Hacemos clic con el botón derecho sobre la clase.

- Buscamos "Herramientas" (Tools para la versión inglesa).

- Ejecutamos "Run PMD".

- Finalmente comprobamos el resultado. PMD muestra en ventana de salida "Avoid unused local variables such as suma1, suma2, suma3". PMD ha detectado que hemos hecho una asignación de variables, pero después no las usamos en el resto del código.

Bajo el entorno de desarrollo NetBean podemos personalizar el funcionamiento de PMD. Para ello cambiamos las reglas de funcionamiento y deshabilitamos la opción de escaneo.

Utilizamos la opción "Herramientas / Opciones / Miscellaneous / PMD". Mediante el tick Enable scan podemos activarlo y desactivarlo. Los botones Manage rules y Manage rulesets permiten crear y modificar las reglas y el set de reglas.

### 4.1.4 Refactorización y pruebas

Si tenemos un software que funciona, pero que debe ser refactorizado como una medida de mejora de su estructura, deberemos procurar no modificar algo que ya funcionaba e introducir nuevos errores. No debemos olvidar que **la funcionalidad del software no cambia**. La refactorización es la parte del mantenimiento del código que no arregla errores ni añade funcionalidad. El objetivo es mejorar la facilidad de comprensión del código para el manteni-

miento en el futuro. La forma de garantizar que nuestros cambios no tendrán un impacto negativo es mediante la realización de pequeños cambios, y siempre con combinaciones de pruebas unitarias y funcionales preparadas; dichas pruebas deberán ejecutarse antes y después de la refactorización (Figura 4.6).

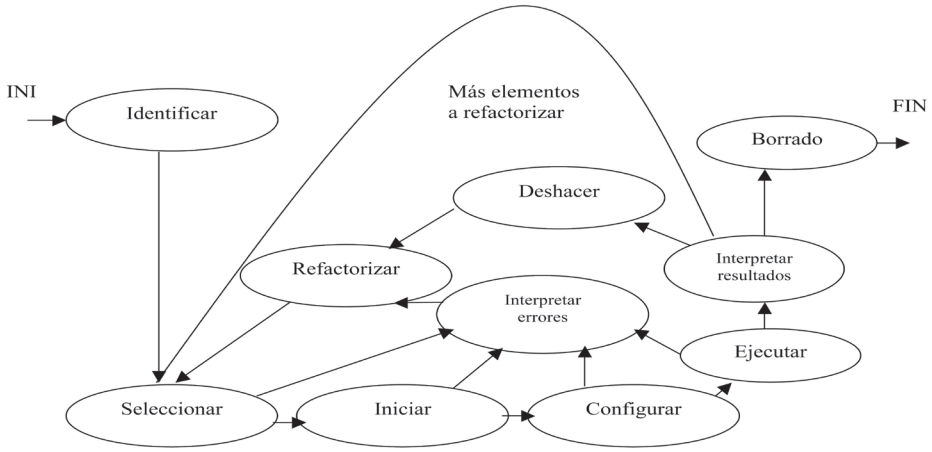


**Figura 4.6**
Modelo de conducta de los programadores con herramientas de refactorización.

### 4.1.5 Herramientas de ayuda a la refactorización

A la hora de hablar de **herramientas**, vamos a centrarnos en dos de las plataformas más utilizadas:

- **Java.** Los dos entornos más extendidos entre los programadores de Java son NetBeans, que ya hemos mencionado anteriormente, y el entorno Eclipse:
  **-Eclipse** provee de un set completo de *refactorings* automáticos, renombre de elementos, mover clases, crear interfaces, etcétera.
  -**RefactorIt** contiene un reestructurador de diseño. Puede usarse sólo o como *plug-in* de entornos de desarrollo, como Eclipse o NetBeans entre otros.

- **NET.** La plataforma de Microsoft multilenguaje también posee potentes herramientas de optimización de código como:
  **-ReSharper.** Inspección de código y *refactoring*.
  **-Visual Estudio** para los lenguajes soportados.

En cuanto a la aplicación de patrones de refactorización, veamos una aplicación de patrones de refactorización en el entorno NetBeans.

- Creamos un proyecto nuevo de Java.

- Insertamos este código después de la sentencia

```
"public static void main(string[] args) ("":
                Int suma1;
                Int suma2;
                Int suma3;
                suma1 = 1;
                suma2 = 2;
                suma3 = 3;
```

- Creamos un método nuevo que contenga estas instrucciones como medida de limpiar el código principal mediante el menú Refactor.
- Marcamos en vídeo inverso la declaración de enteros y sus asignaciones.
- En el menú "Reestructurar" (Refactor), introducimos Method.
- Ponemos nombre al método nuevo.
- Comprobamos que se ha creado el nuevo método. Aparecerá en la ventana de navegación de NetBeans.
- Hacemos clic sobre el nuevo método y comprobamos que contiene el código seleccionado.

## 4.2 Control de versiones

Una correcta gestión del código fuente, de los documentos, de los gráficos y del resto de ficheros que conforman un proyecto informático necesita de un software de **control de versiones** que provea de una base de datos que incluya la trazabilidad de las revisiones hechas por todos los programadores. La nomenclatura más utilizada para las diferentes versiones es la unión de números ( y, en ocasiones, también letras), que indican el nivel que definen (1.1, 1.2…).

### 4.2.1 Estructura de las herramientas de control de versiones

El sistema gestiona una base de datos en forma de árbol en el que se van apilando los diferentes cambios que se realizan en el software (Figura 4.7).

Algunos elementos de esta configuración son:

- **Tronco.** Es el camino de cambio principal (1.1, 1.2, 1.3).

- **Cabeza.** Es la última versión del tronco (1.4).
- **Ramas.** Son los caminos secundarios (2.1, 3.1).
- **Delta.** Son los cambios de una versión respecto a otra.

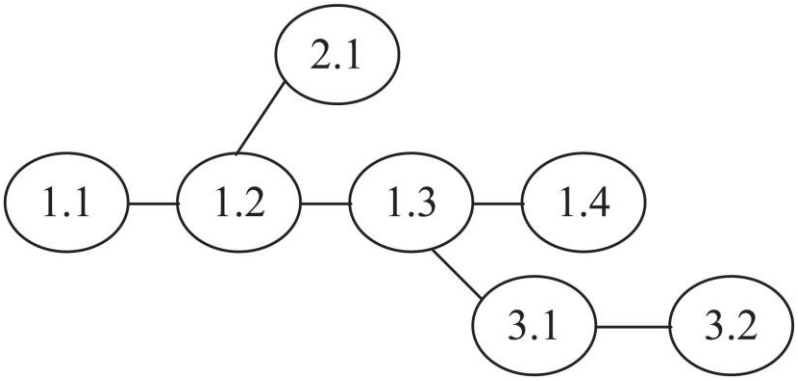


**Figura 4.7**
Estructura en ramales (2.1 y 3.1).

### 4.2.2 Repositorio

El **repositorio** es el lugar en donde se almacena una copia completa de todos los ficheros y directorios que están bajo la gestión del software de control de versiones. Los proyectos diferentes suelen estar ubicados en distintos repositorios. En cualquier caso, éstos no deberán atenderse entre ellos. Un mismo repositorio controla la concurrencia de varios programadores e intenta acceder al mismo elemento y a la misma versión de código mediante avisos y posibilidad de lectura sin grabación de código.

### 4.2.3 Herramientas de control de versiones

Existen muchas herramientas que gestionan el control de versiones. Veamos a continuación dos de ellas:

- **CVS (*Concurrent Versions System*).** Control exhaustivo de la trazabilidad de nuevas funcionalidades, así como de la introducción involuntaria de errores en el código.
- **Mercurial.** Es un gestor de código fuente que gestiona el desarrollo colaborativo entre varios programadores y mantiene una traza de los cambios que se realizan.

A modo de ejemplo con Mercurial podríamos actuar del siguiente modo:

- Comprobamos que el plugin Mercurial está instalado en NetBeans. En caso contrario, lo instalamos.
- Creamos un proyecto nuevo en Java con NetBeans.
- Añadimos código Java. Guardamos el proyecto.
- Creamos el repositorio de control de versiones en Mercurial.
- Añadimos el cambio realizado a nuestro repositorio.
- Añadimos más código Java. Guardamos el proyecto.
- Añadimos el cambio realizado a nuestro repositorio.
- Comprobamos el historial de cambios.

En un ejemplo práctico sería:

- Creamos un proyecto nuevo de Java en NetBeans.
- Insertamos este código después de la sentencia

```
"public static void main(string[] args) ("
                Int suma1;
                Int suma2;
                Int suma3;
                suma1 = 1;
                suma2 = 2;
                suma3 = 3;
```

- Guardamos el proyecto.
- En la opción "Control de Versiones" (Team), seleccionamos "Mercurial / Initialize Project".
- Introducimos el directorio de Windows en el que deseamos colocar el repositorio de control de versiones para este proyecto.
- Seleccionamos la opción "Control de versiones /Mercurial / Update".
- Seleccionamos la opción "Control de versiones /Mercurial / Commit".

- Añadimos la línea de código "suma4 = 1;".
- Guardamos el proyecto.
- Seleccionamos la opción "Control de versiones /Mercurial / Update".
- Seleccionamos la opción "Control de versiones /Mercurial / Commit".
- Seleccionamos la opción "Control de versiones /Mercurial / Show History".

### 4.2.4 Clientes de control de versiones integrados en el entorno de desarrollo

NetBeans accede a los repositorios de los siguientes controladores de versiones (Figura 4.8):

- **Git.** Su característica principal es la velocidad de respuesta.
- **Mercurial.** Posee un buen rendimiento y predisposición para trabajos colaborativos y distribuidos.
- **Subversion.** Buen control de los cambios en el tiempo. Facilidad de acceso a versiones anteriores.
- **CVS** (*Concurrent Versions System*).

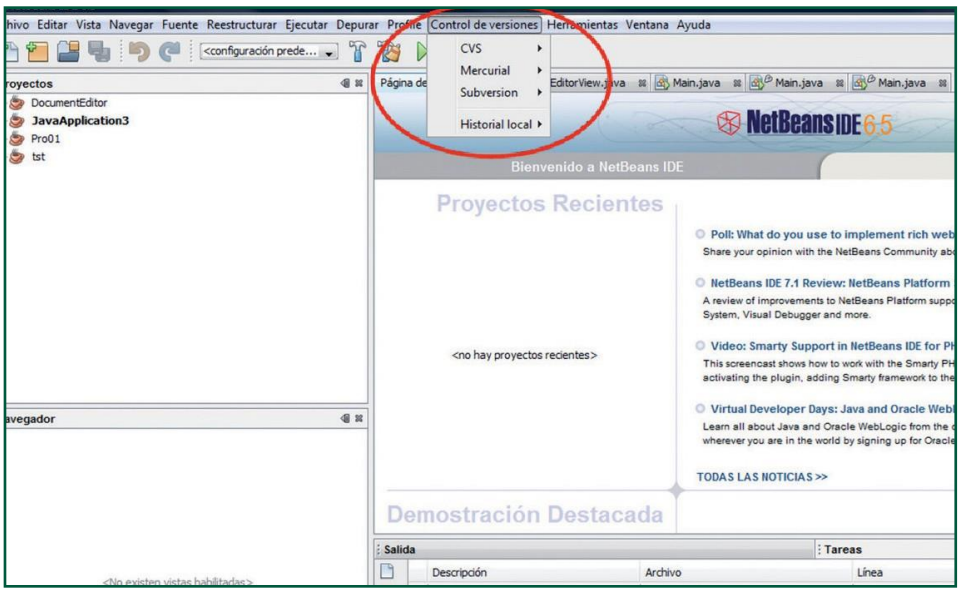


**Figura 4.8**
Control de versiones en el entorno de desarrollo NetBeans.

## 4.3 Documentación

La **documentación** es una serie de textos en diferentes soportes que acompañan al desarrollo de software y aportan información adicional a todo el flujo de datos que comporta dicho desarrollo. Desde un punto de vista de la gestión del negocio y de la gestión tecnológica, la podemos agrupar del siguiente modo:

- **Documentación de gestión.** Incluye los documentos destinados a informar a los responsables del proyecto (presupuestos, desviaciones, roles a adoptar por el personal, recursos y plazos de entrega).

- **Documentación de requerimientos.** Información relativa a las necesidades planteadas de cambio. Pueden mostrarse descripciones del actual sistema y los motivos del cambio.

- **Diseño de función y tecnológico.** A partir del sistema que hay que cambiar se describen las nuevas funcionalidades, es decir, "lo que se debe hacer". Incluye los diseños tecnológicos, es decir, "cómo lo vamos a hacer".

- **Documentación de usuario.** Manuales, diagramas explicativos, nueva normativa de trabajo, etc. En definitiva, todo el material relacionado con el usuario y que es preciso que conozca.

### 4.3.1 Uso de comentarios

Los **comentarios** son textos que se introducen dentro del código fuente para ofrecer información adicional cuando sea necesario. No se trata de sentencias ejecutables, por lo que son omitidos cuando el programa está funcionando.

Es conveniente utilizar una metodología rigurosa acerca de cómo documentar los programas para que cualquier desarrollador utilice una forma uniforme y sistemática. Podemos insertar código de comentario en cabecera de fuente, y especificar la información relativa a la versión, el programador, fechas, la compilación y todo aquello que resulte necesario para el desarrollo del proyecto.

Existe, además, la posibilidad de comentar secciones de código, aunque no debería ser necesario hacerlo si estamos programando como es debido. También podemos comentar aspectos relacionados con alguna solución funcional que el software aporta; quizás pueda ser reutilizado por otros programadores.

## Para saber más

**Para saber más sobre refactorización y acerca de sus peligros y soluciones, consultar el siguiente enlace: http://www.willydev.net/descargas/WillyDev_Refactorizacion.pdf**

En cuanto a los textos destinados a cada área funcional de la empresa, éstos podrán crearse con mayor facilidad mediante el uso de palabras clave dentro de un código fuente que pueda ser interpretado por un generador de documentación (Figura 4.9).

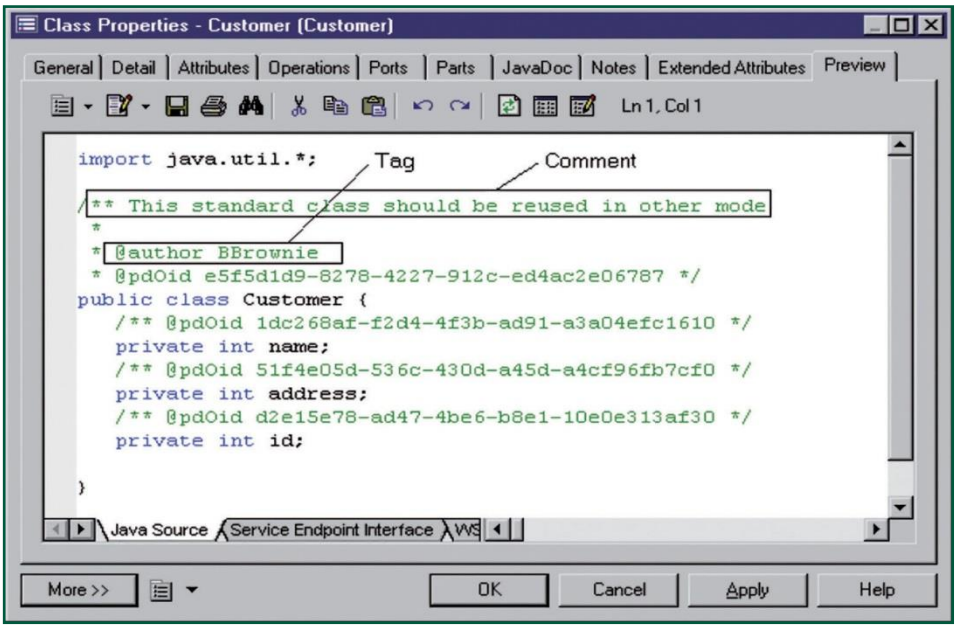


**Figura 4.9**
Control de versiones en el entorno de desarrollo NetBeans.

Veamos un ejemplo de cómo sería una documentación de clases. Para ello, seleccionamos la clase que deseamos documentar y generamos un documento HTML en el que figure la información técnica a través de la herramienta Javadoc. De forma práctica sería:

- Creamos un proyecto nuevo de Java en NetBeans.
- Insertamos este código después de la sentencia

```
"public static void main(string[] args) ("
```

```
Int suma1;
```

```
suma1 = 1;
```

- Comprobamos que Javadoc está instalado en nuestro NetBeans. En caso contrario, lo instalamos.
- En la opción Ejecutar (*Run*), seleccionamos "Generate Javadoc".

- Una página HTML deberá abrirse en nuestro explorador por defecto con toda la información técnica documentada de forma ordenada.

### 4.3.2 Documentación de clases

Las **clases** pueden documentarse a través de un diagrama o a través de un texto específico que detalla la información principal de la clase. Suele generarse en la fase de diseño y su uso principal está en la implementación.

La documentación de clases debe contener la siguiente información:

- Nombre.
- Atributos.
- Métodos.
- Desarrollo de herencias.
- Condiciones previas y posteriores de los métodos y atributos.

Veamos un ejemplo de cómo sería una documentación de clases. Para ello, seleccionamos la clase que deseamos documentar y generamos un documento HTML en el que figure la información técnica a través de la herramienta Javadoc.

De forma práctica sería:

- Creamos un proyecto nuevo de Java en NetBeans.
- Insertamos este código después de la sentencia

```
"public static void main(string[] args) (":
                Int suma1;
                suma1 = 1;
```

- Comprobamos que Javadoc está instalado en nuestro NetBeans. En caso contrario, lo instalamos.
- En la opción Ejecutar (*Run*), seleccionamos "Generate Javadoc".
- Una página HTML deberá abrirse en nuestro explorador por defecto con toda la información técnica documentada de forma ordenada.

### 4.3.3 Herramientas

Las **herramientas externas** generadoras de documentación suelen trabajar dentro del entorno de desarrollo, por lo que, en ocasiones, resulta difícil saber qué es lo que forma parte del entorno y qué ha sido añadido posteriormente.

Algunas de estas herramientas son:

- **Javadoc.** Genera documentación en formato HTML de programas en lenguaje Java.

- **Doxygen.** Es un sistema de documentación para C++, C, Java, Python, PHP y otros.

## Resumen

La documentación es un complemento necesario en el desarrollo de software.

El concepto de *refactorización* como mejora del software sin modificar la funcionalidad y su importancia como método de trabajo necesario para mantener el código lo más claro posible, son factores a tener en cuenta a la hora de diseñar un plan de trabajo de desarrollo. Los analizadores de código y los patrones estándar de refactorización son de gran ayuda y un buen punto de partida hacia la 'limpieza' del código. Los patrones de estructura y de comportamiento son asistentes que pueden mejorar nuestro rendimiento ofreciendo soluciones a los problemas más frecuentes.

Como entidad evolutiva que es, el desarrollo de software necesita un control de los cambios que se realizan; así como una organización de la edición cuando son varios los programadores que trabajan en el mismo proyecto. Esto se consigue mediante el control de versiones que también se integra en los entornos de desarrollo. CVS y Mercurial son dos herramientas que permiten el control de los cambios.

La documentación adquiere una gran importancia si se piensa en mantenimientos futuros, así como los comentarios en el código fuente y su integración con las herramientas de generación automática de documentos. Según su uso y a quién va dirigida, la documentación puede agruparse en gestión, requerimientos, funcional y orientada a usuario. Javadoc y Doxygen son dos herramientas de documentación que pueden integrarse en los entornos de desarrollo más comunes.

# Ejercicios de autocomprobación

**Indica si las siguientes afirmaciones son verdaderas (V) o falsas (F):**

1. Un usuario debe darse siempre cuenta de que un código ha pasado por un proceso de refactorización.

2. El *Factory Method* (Método de Fábrica) define una interfaz para crear objetos y permite a las subclases decidir la clase a instanciar.

3. Con el *Builder* (Constructor) la construcción de un objeto se separa de su representación, por lo que pueden crearse diferentes representaciones con el mismo proceso de construcción.

4. El repositorio es el lugar en donde se almacena el original de todos los ficheros y directorios que están bajo la gestión del software de control de versiones.

5. Un mismo repositorio controla la concurrencia de varios programadores e intenta acceder al mismo elemento y a la misma versión de código mediante avisos y posibilidad de lectura sin grabación de código.

6. Mercurial es un gestor de código fuente que gestiona el desarrollo colaborativo entre varios programadores y mantiene una traza de los cambios que se realizan.

7. La documentación es una serie de textos en soportes iguales, que acompañan al desarrollo de software y aportan información adicional a todo el flujo de datos que comporta dicho desarrollo.

**Completa las siguientes afirmaciones:**

8. Los_____son textos que se introducen dentro del código fuente para ofrecer información adicional cuando sea necesario. No se trata de sentencias_____, por lo que son omitidos cuando el programa está _____.

9. Las_____pueden documentarse a través de un _____ o a través de un texto específico que detalla la información principal de la _____.

10. Las dos plataformas de_____más utilizadas son Java y _____. Los dos entornos más extendidos entre los programadores de Java son NetBeans y el entorno_____, que provee de un set completo de_____automáticos, renombre de elementos, mover clases, crear interfaces, etcétera.

Las soluciones a los ejercicios de autocomprobación se encuentran al final de esta Unidad Formativa. En caso de que no los hayas contestado correctamente, repasa la parte de la lección correspondiente.

# Soluciones de los ejercicios de autocomprobación

## Unidad 3

1. F. Con la prueba funcional se comparan las exigencias del cliente y la solución ofrecida por el analista a través del diseño técnico con los resultados del software.
2. V
3. V
4. F. Las herramientas de depuración son unos programas cuyo objetivo principal es el de probar el funcionamiento de otros programas informáticos.
5. V
6. V
7. V
8. recursos, productor, contingencia.
9. normas, calidad, certificada.
10. automatización, regresión, desarrollador, iteraciones.

## Unidad 4

1. F. Un usuario no debe darse cuenta nunca de que un código ha pasado por un proceso de refactorización.
2. V
3. V
4. F. El repositorio es el lugar en donde se almacena una copia completa de todos los ficheros y directorios que están bajo la gestión del software de control de versiones.
5. V
6. V
7. F. La documentación es una serie de textos en diferentes soportes que acompañan al desarrollo de software y aportan información adicional a todo el flujo de datos que comporta dicho desarrollo.
8. comentarios, ejecutables, funcionando.
9. clases, diagrama, clase.
10. refactorización, NET, Eclipse, *refactorings*.

# Índice

**MÓDULO: ENTORNOS DE DESARROLLO**

**UNIDAD FORMATIVA 2**

**3. Diseño y realización de pruebas** ..... 41
3.1 Planificación de pruebas ..... 41
3.2 Tipos de pruebas ..... 42
3.3 Procedimientos y casos de prueba ..... 44
3.4 Herramientas de depuración ..... 44
3.5 Validaciones ..... 46
3.6 Pruebas de código ..... 48
3.7 Normas de calidad ..... 49
3.8 Pruebas unitarias. Herramientas ..... 50
3.9 Automatización de pruebas ..... 51
3.10 Documentación de pruebas ..... 52
Resumen ..... 53
Ejercicios de autocomprobación ..... 54

**4. Optimización de documentos** ..... 56
4.1 Concepto de refactorización ..... 56
4.1.1 Limitaciones ..... 56
4.1.2 Patrones de refactorización más usuales ..... 58
4.1.3 Analizadores de código ..... 60
4.1.4 Refactorización y pruebas ..... 61
4.1.5 Herramientas de ayuda a la refactorización ..... 62
4.2 Control de versiones ..... 63
4.2.1 Estructura de las herramientas de control de versiones ..... 63
4.2.2 Repositorio ..... 64

4.2.3 Herramientas de control de versiones ..... 64
4.2.4 Clientes de control de versiones integrados en el entorno de desarrollo ..... 66
4.3 Documentación ..... 67
4.3.1 Uso de comentarios ..... 67
4.3.2 Documentación de clases ..... 69 4.3.3
Herramientas ..... 70
Resumen ..... 71
Ejercicios de autocomprobación ..... 72
Soluciones a los ejercicios de autocomprobación ..... 74