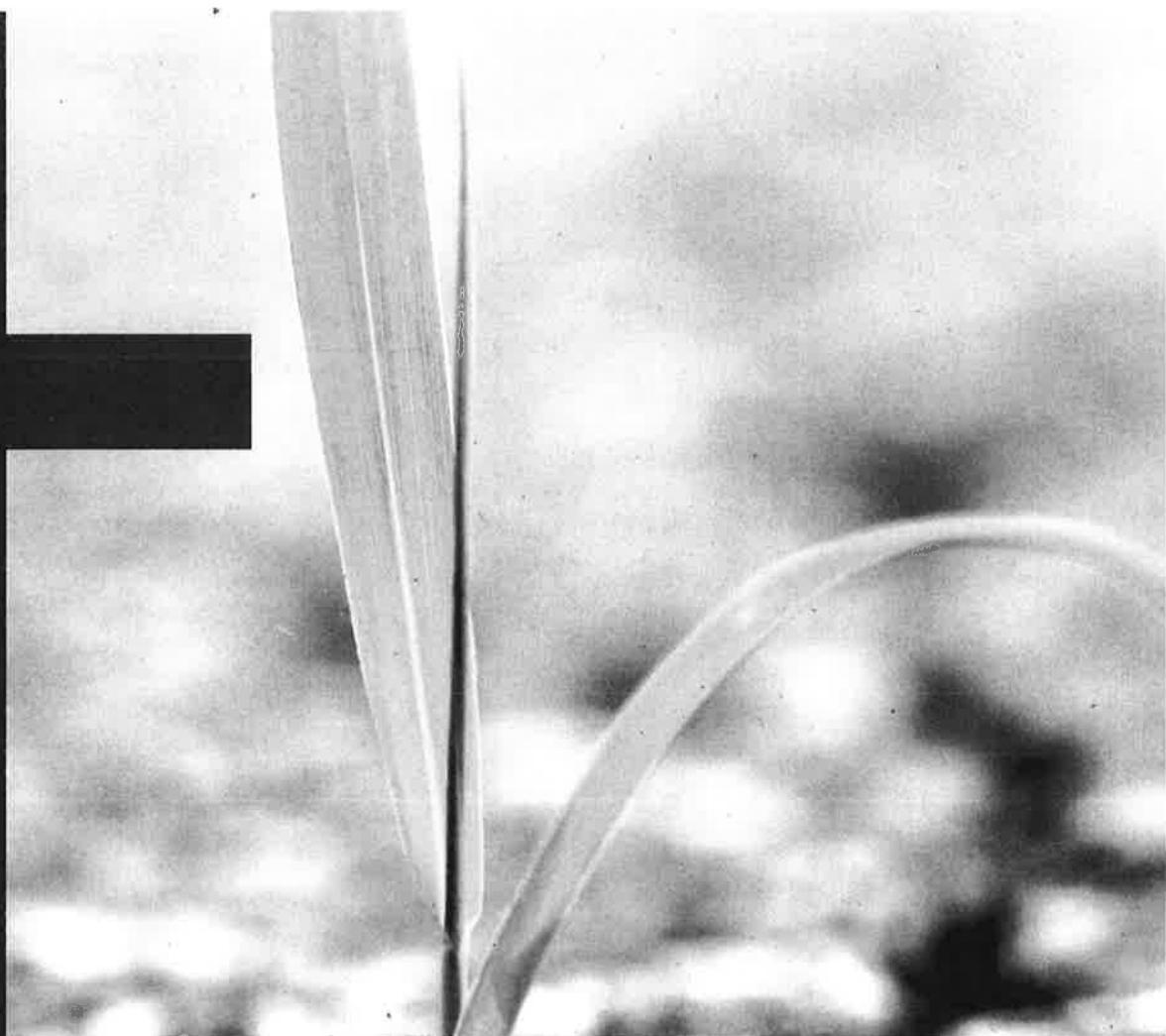


Técnico Superior en Desarrollo de Aplicaciones Multiplataforma y Web
DAM y DAW



Entornos de Desarrollo

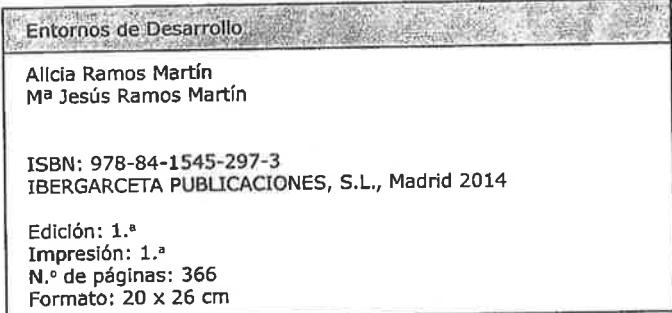
Alicia Ramos Martínez
M.º Jesús Ramos Martínez



Alicia Ramos Martín
Mª Jesús Ramos Martín

Entornos de Desarrollo

Técnico Superior en Desarrollo de Aplicaciones Multiplataforma y Web
DAM y DAW



Reservados los derechos para todos los países de lengua española. De conformidad con lo dispuesto en el artículo 270 y siguientes del código penal vigente, podrán ser castigados con penas de multa y privación de libertad quienes reprodujeren o plagiaren, en todo o parte, una obra literaria, artística o científica fijada en cualquier tipo de soporte sin la preceptiva autorización. Ninguna parte de la publicación, incluido el diseño de la cubierta, puede ser reproducida, almacenada o trasmisida de ninguna forma, ni por ningún medio sea éste electrónico, químico, mecánico, electro-óptico, grabación, fotocopia o cualquier otro, sin la previa autorización escrita por parte de la editorial.

Dírjase a CEDRO (Centro Español de Derechos Reprográficos), www.cedro.org, si necesita fotocopiar o escanear algún fragmento de esta obra.

COPYRIGHT © 2014 IBERGARCETA PUBLICACIONES, S.L.
info@garceta.es

Entornos de Desarrollo
© Alicia Ramos Martín, Mª Jesús Ramos Martín

1.^a edición, 1.^a impresión

OI: 211/2018

ISBN: 978-84-1545-297-3

Depósito Legal: M-20981-2014

Imagen de cubierta: © Elenathewise_fotolia.com

Impresión: Print House, marca registrada de Coplar S.A.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Nota sobre enlaces a páginas web ajena: Este libro puede incluir referencias a sitios web gestionados por terceros y ajenos a IBERGARCETA PUBLICACIONES, S.L., que se incluyen sólo con finalidad informativa. IBERGARCETA PUBLICACIONES, S.L., no asume ninguna responsabilidad por los daños y perjuicios derivados del uso de los datos personales que pueda hacer un tercero encargado del mantenimiento de las páginas web ajenas a IBERGARCETA PUBLICACIONES, S.L., y del funcionamiento, accesibilidad y mantenimiento de los sitios web no gestionados por IBERGARCETA PUBLICACIONES, S.L., directamente. Las referencias se proporcionan en el estado que se encuentran en el momento de publicación sin garantías, expresas o implícitas, sobre la información que se proporcione en ellas.

ÍNDICE

CAPÍTULO 1. DESARROLLO DE SOFTWARE.....	1
1.1. Introducción	2
1.2. El software de ordenador.....	2
1.2.1. Software basado en el tipo de tarea que realiza	3
1.2.2. Software basado en el método de distribución.....	3
1.2.3. Licencias de software. Software libre y propietario.....	4
1.3. Ciclo de vida del software	6
1.3.1. Definición.....	6
1.3.2. Modelos de ciclo de vida	6
1.4. Fases del desarrollo de una aplicación	12
1.4.1. Análisis	12
1.4.2. Diseño	15
1.4.3. Codificación.....	22
1.4.4. Pruebas	26
1.4.5. Documentación	28
1.4.6. Explotación	32
1.4.7. Mantenimiento	33
1.5. Concepto de programa.....	35
1.5.1. Programa y componentes del sistema informático.....	35
1.6. Lenguajes de programación.....	41
1.6.1. Clasificación y características	41
1.7. Obtención de código ejecutable	47
1.7.1. Tipos de código.....	47
1.7.2. Compilación.....	48
1.8. Máquinas virtuales	49
1.8.1. La máquina virtual de Java	49
1.9. Herramientas utilizadas en programación	51
1.10. Ejemplos.....	52
1.10.1. Diagramas de flujo de datos. DFD	52
1.10.2. Pseudocódigo y diagramas de flujo.....	57
COMPRUEBA TU APRENDIZAJE	63
ACTIVIDADES DE AMPLIACIÓN	64

CAPÍTULO 2. INSTALACIÓN Y USO DE ENTORNOS DE DESARROLLO	67
2.1. Introducción	68
2.2. Componentes de un entorno de desarrollo	68
2.3. Instalación de un entorno de desarrollo.....	69
2.3.1. Instalación de Eclipse.....	69
2.3.2. Instalación de plugins.....	72
2.3.3. Instalación de SQL Developer	75
2.4. Herramientas para el modelado de datos.....	78
2.4.1. Data Modeler de Oracle	78
1.4.2. Plugin ER Master de Eclipse.....	92
2.5. Comparación de entornos de desarrollo	100
COMPRUEBA TU APRENDIZAJE	102
ACTIVIDADES DE AMPLIACIÓN.....	102
CAPÍTULO 3. DISEÑO Y REALIZACIÓN DE PRUEBAS	105
3.1. Introducción	106
3.2. Técnicas de diseño de casos de prueba.....	106
3.2.1. Pruebas de caja blanca	107
3.2.2. Pruebas de caja negra	107
3.3. Estrategias de prueba del software	107
3.3.1. Prueba de unidad.....	108
3.3.2. Prueba de integración.....	109
3.3.3. Prueba de validación	109
3.3.4. Prueba del sistema.....	110
3.4. Documentación para las pruebas	110
3.5. Pruebas de código.....	110
3.5.1. Prueba del camino básico.....	111
3.5.2. Partición o clases de equivalencia.....	117
3.5.3. Análisis de valores límite.....	120
3.6. Herramientas de depuración	121
3.6.1. Puntos de ruptura y seguimiento	125
3.6.2. Examen y modificación de variables	128
3.7. Pruebas unitarias JUnit.....	129
3.7.1. Creación de una clase de prueba	130
3.7.2. Preparación y ejecución de las pruebas.....	132
3.7.3. Tipos de anotaciones	137
3.7.4. Pruebas parametrizadas	139
3.7.5. Suite de pruebas	140
COMPRUEBA TU APRENDIZAJE	143
ACTIVIDADES DE AMPLIACIÓN.....	146

CAPÍTULO 4. OPTIMIZACIÓN Y DOCUMENTACIÓN	149
4.1. Introducción	150
4.2. Control de versiones	150
4.2.1. Subversión. Ciclo de vida de subversión	151
4.2.2. Cliente TortoiseSVN	153
4.2.3. Servidor subversión. VisualSVN Server	164
4.2.4. Subversión en SQL Developer	166
4.2.5. Subversión en Eclipse	172
4.3. Documentación.....	183
4.3.1. Uso de JavaDoc en Eclipse	185
4.4. Refactorización	189
4.4.1. Cuándo refactorizar. Malos olores (bad smells).....	190
4.4.2. Refactorización en Eclipse	191
COMPRUEBA TU APRENDIZAJE	199
ACTIVIDADES DE AMPLIACIÓN.....	199
CAPÍTULO 5. ELABORACIÓN DE DIAGRAMAS DE CLASES	203
5.1. Introducción	204
5.2. Conceptos orientados a objetos	204
5.3. Qué es UML	205
5.3.1. Tipos de diagramas	207
5.4. Diagramas de clases	210
5.4.1. Clases	211
5.4.2. Relaciones	213
5.4.3. Estereotipos.....	221
5.5. Herramientas para el diseño de diagramas	223
5.5.1. ArgoUML.....	223
5.5.2. UML con Eclipse	233
5.5.3. UML con WhiteStarUML.....	245
5.6. Generación de código a partir de diagramas de clases	249
5.7. Ingeniería inversa	252
COMPRUEBA TU APRENDIZAJE	258
ACTIVIDADES DE AMPLIACIÓN.....	258
CAPÍTULO 6. ELABORACIÓN DE DIAGRAMAS DE COMPORTAMIENTO	261
6.1. Introducción	262
6.2. Diagrama de casos de uso	262
6.2.1. Elementos de diagrama de casos de uso	263
6.2.2. Identificar actores.....	264
6.2.3. Identificar casos de uso	265
6.2.4. Relaciones en un diagrama de casos de uso.....	270

6.3. Herramientas para la elaboración de diagramas	280
6.3.1. Eclipse.....	280
6.3.2. ArgoUML.....	284
6.3.3. WhiteStarUML.....	287
6.4. Diagramas de interacción	290
6.4.1. Diagrama de secuencia.....	290
6.4.2. Diagrama de comunicación o colaboración	300
6.4.3. Elaboración de diagramas de interacción.....	303
6.5. Diagrama de estado	317
6.5.1. Elaboración de diagramas de estado	323
6.6. Diagrama de actividad.....	330
6.6.1. Elaboración de diagramas de actividad.....	332
6.7. Ingeniería inversa en Eclipse.....	336
COMPRUEBA TU APRENDIZAJE	345
ACTIVIDADES DE AMPLIACIÓN.....	350

CAPÍTULO 1

DESARROLLO DE SOFTWARE

CONTENIDOS

- El software del ordenador
- Concepto de programa. Programa y componentes del sistema informático
- Ciclo de vida del software.
- Lenguajes de programación. Características y clasificación
- Fases del desarrollo de una aplicación
- Código fuente, código objeto y código ejecutable; máquinas virtuales

OBJETIVOS

- Reconocer la relación de los programas con los componentes del sistema informático
- Identificar las fases de desarrollo de una aplicación informática
- Diferenciar los conceptos de código fuente, código objeto y código ejecutable
- Clasificar los lenguajes de programación

RESUMEN DEL CAPÍTULO

En este capítulo aprenderemos a reconocer los elementos y herramientas que intervienen en el desarrollo de un programa informático, analizando sus características y las fases en las que actúan hasta llegar a su puesta en funcionamiento.

1.1. INTRODUCCIÓN

El ordenador se compone de dos partes: el hardware y el software. El hardware lo forman los componentes físicos que se pueden ver y tocar: el monitor, el teclado, el ratón, la placa base, la memoria RAM, el microprocesador, las tarjetas de expansión, el disco duro, etc. En cambio el software forma la parte lógica del ordenador que no se puede tocar. Comprende el conjunto de programas y aplicaciones que actúan sobre el hardware del ordenador y facilitan al usuario la realización de diferentes tareas.

En este capítulo se estudiarán conceptos básicos que tienen que ver con el desarrollo de software.

1.2. EL SOFTWARE DEL ORDENADOR

Podemos dar varias definiciones sobre la palabra software, por ejemplo:

- El diccionario de la Real Academia Española (RAE) define software como «*el conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora*».
- El estándar 729 del IEEE (*Institute of Electrical and Electronics Engineers*), define software como «*el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de computación*».

En cualquier caso, el software es todo aquello que se refiere a los programas y datos almacenados en un ordenador, programas encargados de dar instrucciones para realizar tareas con el hardware o para comunicarnos con otro software, y datos necesarios para la ejecución de los programas.

Podemos decir que el software se divide en dos categorías: según el **tipo de tarea que realiza** y según el **método de distribución**. Teniendo en cuenta la licencia bajo la cual se distribuye un determinado software aparecen diferentes subconjuntos dentro del conjunto global del software, entre ellas destacamos: **software libre**, **software propietario** y **software de dominio público**. Véase Figura 1.1.

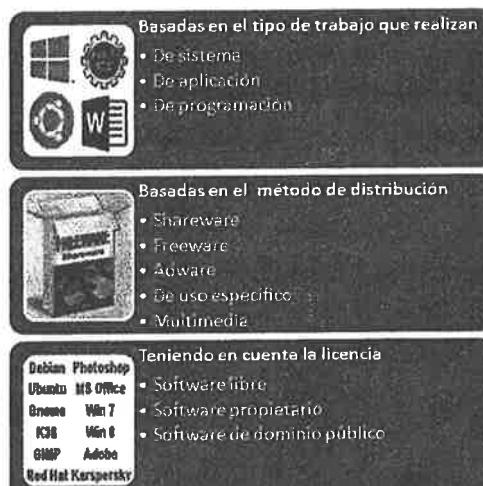


Figura 1.1. Clasificación del software.

1.2.1. Software basado en el tipo de tarea que realiza

Se distinguen tres tipos de software:

- ✓ • **Software de sistema;** es aquel que permite que el hardware funcione. Lo forman los programas que permiten la administración de la parte física o los recursos del ordenador, y es el que interactúa entre el usuario y los componentes hardware del ordenador. Ejemplo de esto son los sistemas operativos, los controladores de dispositivo, las herramientas de diagnóstico, las de corrección y optimización, etcétera.
- ✓ • **Software de aplicación;** lo forman los programas que nos ayudan a realizar tareas específicas en cualquier campo susceptible de ser automatizado o asistido. Este software hace que el ordenador sea una herramienta útil para el usuario. Por ejemplo: las aplicaciones de control y automatización industrial, las aplicaciones ofimáticas, el software educativo, el software médico, las aplicaciones de contabilidad, de diseño asistido (CAD), etcétera.
- ✗ • **Software de programación o desarrollo;** es el que proporciona al programador herramientas para ayudarle a escribir programas informáticos y a usar diferentes lenguajes de programación de forma práctica. Entre ellos se encuentran los entornos de desarrollo integrados (IDE), que agrupan las anteriores herramientas, normalmente en un entorno visual, de forma que el programador no necesite introducir múltiples comandos para compilar, interpretar, depurar, etc. Habitualmente, cuentan con una avanzada interfaz gráfica de usuario (GUI).

1.2.2. Software basado en el método de distribución

Entre estos se encuentran los así llamados programas enlatados, el software desarrollado por compañías y vendido principalmente por distribuidores; el freeware y software de dominio público, que se ofrece sin costo alguno; el shareware, que es similar al freeware, pero suele conllevar una pequeña tasa para los usuarios que lo utilicen profesionalmente:

- ✗ • **Shareware.** Es una modalidad de distribución de software, tanto juegos como programas utilitarios, para que el usuario pueda evaluar de forma gratuita el producto por un tiempo especificado. Para adquirir una licencia de software que permita el uso del software de manera completa se requiere de un pago (muchas veces modesto), aunque también existe el llamado «shareware de precio cero»; sin embargo, esta modalidad es poco común. Por ejemplo: los comprresores de archivos Winzip, WinRAR; herramientas de sistema como PC File, ZoneAlarm; edición de imágenes como Paint Shop Pro, The Logo Creator; antivirus como F-Prot, PC-Tools o Virus Scan.
- ✗ • **Freeware.** Es un software que se distribuye sin cargo. A veces se incluye el código fuente, pero no es lo usual. El freeware suele incluir una licencia de uso, que permite su redistribución pero con algunas restricciones, como no modificar la aplicación en sí, ni venderla y dar cuenta de su autor. Contrariamente a lo que se cree, los programas de software libre no necesariamente son freeware. Esto suele provenir de una confusión acerca del significado de la palabra free en inglés, que puede ser tanto «gratis» como «libre»; es decir, un tipo de software cuya licencia autoriza su uso, modificación y redistribución con y sin cambios.

- ↳ ■ **Adware.** Suelen ser programas Shareware que de forma automática descargan publicidad en nuestro ordenador cuando lo ejecutamos o instalamos, hemos de estar atentos a la hora de instalarlos porque a veces se puede evitar su descarga. Al comprar la licencia del programa se elimina la publicidad.
- ↳ ■ **Software multimedia.** El software multimedia se refiere a los programas utilizados para presentar de una forma integrada textos, gráficos, sonidos y animaciones; este tipo de software es considerado como una nueva tecnología. Sobre todo se usa en el ámbito educativo. Por ejemplo, las encyclopedias multimedia.
- ↳ ■ **Software de uso específico.** Este tipo de software es el que se desarrolla especialmente para resolver un problema determinado de alguna organización o persona; utilizar este software requiere de un experto en informática para su creación o adaptación. Ejemplos pueden ser los programas para llevar la gestión de un videoclub, los que se usan en las escuelas para registrar las calificaciones de los alumnos y los horarios de los profesores, o los que se usan en los bancos para el control de las cuentas y clientes, etc.

1.2.3. Licencias de software. Software libre y propietario¹

Una licencia de software es un contrato que se establece entre el desarrollador de un software, sometido a propiedad intelectual y a derechos de autor, y el usuario, en el cual se definen con precisión los derechos y deberes de ambas partes. Es el desarrollador, o aquel a quien este haya cedido los derechos de explotación, quien elige la licencia según la cual distribuye el software. La Figura 1.2 muestra la información de una licencia comercial y el contrato de licencia de usuario final.

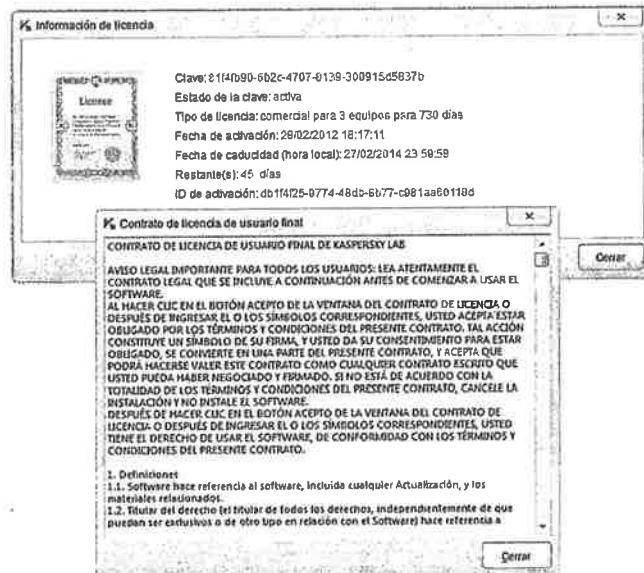


Figura 1.2. Información y contrato de licencia de usuario final.

El **software libre** es aquel en el cual el autor cede una serie de libertades básicas al usuario, en el marco de una licencia, que establece las siguientes libertades:

¹ Las definiciones de este epígrafe se han obtenido del documento: *Propuesta de recomendaciones a la Administración General del Estado sobre utilización del software libre y de fuentes abiertas*. NIPO: 326-05-044-3.

1. Libertad de utilizar el programa con cualquier fin en cuantos ordenadores se deseé.
2. Libertad de estudiar cómo funciona el programa y de adaptar su código a necesidades específicas; para ello, como condición previa, es necesario poder acceder al código fuente.
3. Libertad de distribuir copias a otros usuarios (con o sin modificaciones).
4. Libertad de mejorar el programa (ampliarlo, añadir funciones) y de hacer públicas y distribuir al público las modificaciones; para ello, como condición previa, es necesario poder acceder al código fuente.

A diferencia del software libre y de fuentes abiertas, el **software propietario** es aquel que, habitualmente, se distribuye en formato binario, sin posibilidad de acceso al código fuente según una licencia en la cual el propietario, por regla general, prohíbe alguna o todas las siguientes posibilidades: la redistribución, modificación, copia, uso en varias máquinas simultáneamente, transferencia de titularidad, difusión de fallos y errores que se pudiesen descubrir en el programa, entre otras. Se dan diversas variantes: Freeware y Shareware.

Finalmente, un **software de dominio público** es aquel que carece de licencia o no hay forma de determinarla pues se desconoce al autor. Esta situación se produce bien cuando su propietario abandona los derechos que le acreditan como titular o bien cuando se produce la extinción de la propiedad por expiración del plazo de la misma, es decir, el fin del plazo de protección de los derechos de autor. El software de dominio público no pertenece a una persona concreta, sino que todo el mundo lo puede utilizar; e incluso cabe desarrollar una oferta propietaria sobre la base de un código que se encuentra en el dominio público.

La licencia más utilizada en los productos y desarrollos de software libre y de fuentes abiertas es la **licencia GPL** (*GNU General Public License - Licencia Pública General*) que da derecho al usuario a usar y modificar el programa con la obligación de hacer públicas las versiones modificadas de este. La Figura 1.3 muestra las condiciones de la licencia de Moodle que es un paquete de software libre que se distribuye bajo licencia GPL.

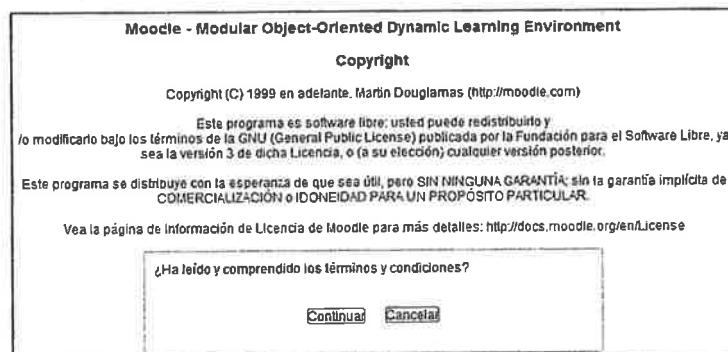


Figura 1.3. Condiciones de licencia de Moodle.

¡INTERESANTE!

En esta dirección encontrarás más información sobre licencias de software libre compatibles con la licencia GPL <http://www.gnu.org/licenses/license-list.es.html#SoftwareLicenses>.

1.3. CICLO DE VIDA DEL SOFTWARE

El proceso de desarrollo del software implica un conjunto de actividades que se tienen que planificar y gestionar de tal manera que aseguren un producto final que dé solución a las necesidades de todas aquellas personas que lo van a utilizar.

1.3.1. Definición

El estándar ISO/IEC 12207-1 define ciclo de vida del software como: *Un marco de referencia que contiene los procesos, las actividades y las tareas involucradas en el desarrollo, la explotación y el mantenimiento de un producto de software, abarcando la vida del sistema desde la definición de los requisitos hasta la finalización de su uso.*

El ciclo de vida de un producto software comprende el periodo que transcurre desde que el producto es concebido hasta que deja de estar disponible o es retirado. Normalmente, se divide en etapas y en cada etapa se realizarán una serie de tareas. Usualmente se consideran las siguientes etapas: especificación y análisis de requisitos, diseño del sistema, implementación del software, aplicación y pruebas, entrega y mantenimiento:

- ✓ 1. **Análisis.** Construye un modelo de los requisitos. En esta etapa se debe entender y comprender de forma detallada el problema que se va a resolver. Es muy importante producir en esta etapa una documentación entendible, completa y fácil de verificar y modificar.
- ✓ 2. **Diseño.** En esta etapa ya sabemos qué es lo que hay que hacer, ahora hay que definir cómo se va a resolver el problema. Se deducen las estructuras de datos, la arquitectura de software, la interfaz de usuario y los procedimientos. Por ejemplo, en esta etapa hay que seleccionar el lenguaje de programación, el Sistema Gestor de Bases de Datos, etc.
- ✓ 3. **Codificación.** En esta etapa se traduce lo descrito en el diseño a una forma legible por la máquina. La salida de esta fase es código ejecutable.
- ✓ 4. **Pruebas.** Se comprueba que se cumplen criterios de corrección y calidad. Las pruebas deben garantizar el correcto funcionamiento del sistema.
- ✓ 5. **Mantenimiento.** Esta fase tiene lugar después de la entrega del software al cliente. En ella hay que asegurar que el sistema pueda adaptarse a los cambios. Se producen cambios porque se han encontrado errores, es necesario adaptarse al entorno (por ejemplo se ha cambiado de sistema operativo) o porque el cliente requiera mejoras funcionales.

Cada etapa tiene como entrada uno o varios documentos procedentes de las etapas anteriores y produce otros documentos de salida, por ello una tarea importante a realizar en cada etapa es la documentación.

1.3.2. Modelos de ciclo de vida

Existen varios modelos de ciclo de vida, es importante tener en cuenta las características del proyecto software para elegir un modelo u otro. Los modelos más importantes son: *Cascada, Incremental y Evolutivo*.

2/

1.3.2.1. Ciclo de vida en cascada

En este modelo las etapas para el desarrollo del software tienen un orden, de tal forma que para empezar una etapa es necesario finalizar la etapa anterior; después de cada etapa se realiza una revisión para comprobar si se puede pasar a la siguiente, véase Figura 1.4 (1). Este modelo permite hacer iteraciones, por ejemplo, durante la etapa de mantenimiento del producto el cliente requiere una mejora, esto implica que hay que modificar algo en el diseño, lo cual significa que habrá que hacer cambios en la codificación y se tendrán que realizar de nuevo las pruebas, es decir, si se tiene que volver a una de las etapas anteriores hay que recorrer de nuevo el resto de las etapas, véase Figura 1.4 (2).

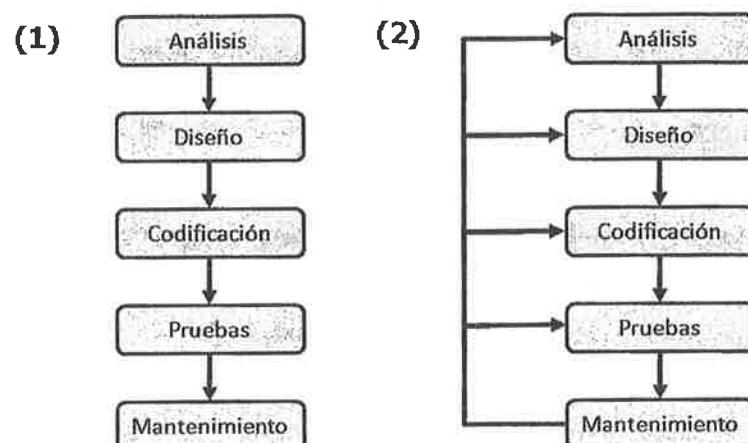


Figura 1.4. Modelo en Cascada.

Tiene varias variantes, una de la más utilizada es la que produce una realimentación entre etapas, se la conoce como *Modelo en Cascada con Realimentación*. Por ejemplo, supongamos que la etapa de *Análisis* (captura de requisitos) ha finalizado y se puede pasar a la de *Diseño*. Durante el desarrollo de esta etapa se detectan fallos (los requisitos han cambiado, han evolucionado, ambigüedades en la definición de los mismos, etc.), entonces será necesario retornar a la etapa anterior, realizar los ajustes pertinentes y continuar de nuevo con el *Diseño*. A esto se le conoce como realimentación, pudiendo volver de una etapa a la anterior o incluso de varias etapas a la anterior, véase Figura 1.5.

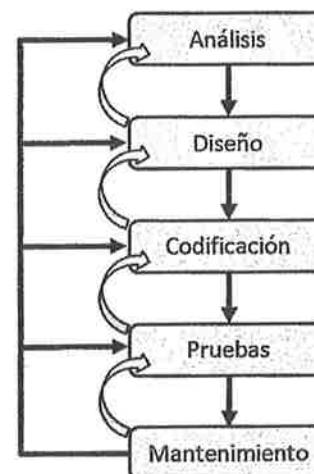


Figura 1.5. Modelo en Cascada con Realimentación.

Ventajas:

- Fácil de comprender, planificar y seguir.
- La calidad del producto resultante es alta.
- Permite trabajar con personal poco cualificado.

Inconvenientes:

- La necesidad de tener todos los requisitos definidos desde el principio (algo que no siempre ocurre ya que pueden surgir necesidades imprevistas).
- Es difícil volver atrás si se cometen errores en una etapa.
- El producto no está disponible para su uso hasta que no está completamente terminado.

Se recomienda cuando:

- El proyecto es similar a alguno que ya se haya realizado con éxito anteriormente.
- Los requisitos son estables y están bien comprendidos.
- Los clientes no necesitan versiones intermedias.

1.3.2.2. Modelos evolutivos

El software evoluciona con el tiempo, es normal que los requisitos del usuario y del producto cambien conforme se desarrolla el mismo. La competencia en el mercado del software es tan grande que las empresas no pueden esperar a tener un producto totalmente completo para lanzarlo al mercado, en su lugar se van introduciendo versiones cada vez más completas que de alguna manera alivian las presiones competitivas.

El modelo en cascada asume que se va a entregar un producto completo, en cambio los modelos evolutivos permiten desarrollar versiones cada vez más completas hasta llegar al producto final deseado. En estos modelos se asume que las necesidades del usuario no están completas y se requiere una vuelta a planificar y diseñar después de cada implantación de los entregables.

Los modelos evolutivos más conocidos son: el *Iterativo incremental* y el *Espiral*.

MODELO ITERATIVO INCREMENTAL

Está basado en varios ciclos cascada realimentados aplicados repetidamente. El modelo incremental entrega el software en partes pequeñas, pero utilizables, llamadas «incrementos». En general, cada incremento se construye sobre aquél que ya ha sido entregado². En la Figura 1.6 se muestra un diagrama del modelo bajo un esquema temporal, se observa de forma iterativa el modelo en cascada para la obtención de un nuevo incremento mientras progresan el tiempo en el calendario (por simplificar el diagrama se ha puesto el modelo en cascada más básico, el lineal secuencial).

² Ingeniería del software. Un enfoque práctico. Roger S. Pressman.

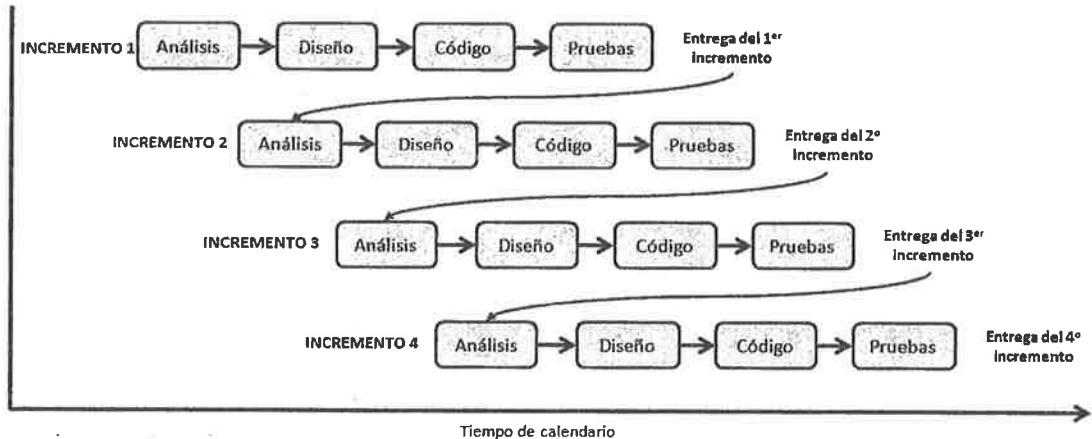


Figura 1.6. Modelo Iterativo Incremental.

Como ejemplo de software desarrollado bajo este modelo se puede considerar un procesador de textos, en el primer incremento se desarrollan funciones básicas de gestión de archivos y de producción de documentos; en el segundo incremento se desarrollan funciones gramaticales y de corrección ortográfica, en el tercer incremento se desarrollan funciones avanzadas de paginación, y así sucesivamente.

Ventajas:

- No se necesitan conocer todos los requisitos al comienzo.
- Permite la entrega temprana al cliente de partes operativas del software.
- Las entregas facilitan la realimentación de los próximos entregables.

Inconvenientes:

- Es difícil estimar el esfuerzo y el coste final necesario.
- Se tiene el riesgo de no acabar nunca.
- No recomendable para desarrollo de sistemas de tiempo real, de alto nivel de seguridad, de procesamiento distribuido, y/o de alto índice de riesgos.

Se recomienda cuando:

- Los requisitos o el diseño no están completamente definidos y es posible que haya grandes cambios.
- Se están probando o introduciendo nuevas tecnologías.



MODELO EN ESPIRAL

Este modelo combina el *Modelo en Cascada* con el modelo iterativo de construcción de prototipos. El proceso de desarrollo del software se representa como una espiral, donde en cada ciclo se desarrolla una parte del mismo. Cada ciclo está formado por cuatro fases, véase Figura 1.7, y cuando termina produce una versión incremental del software con respecto al ciclo anterior. En este aspecto se parece al *Modelo Iterativo Incremental* con la diferencia de que en cada ciclo se tiene en cuenta el análisis de riesgos.

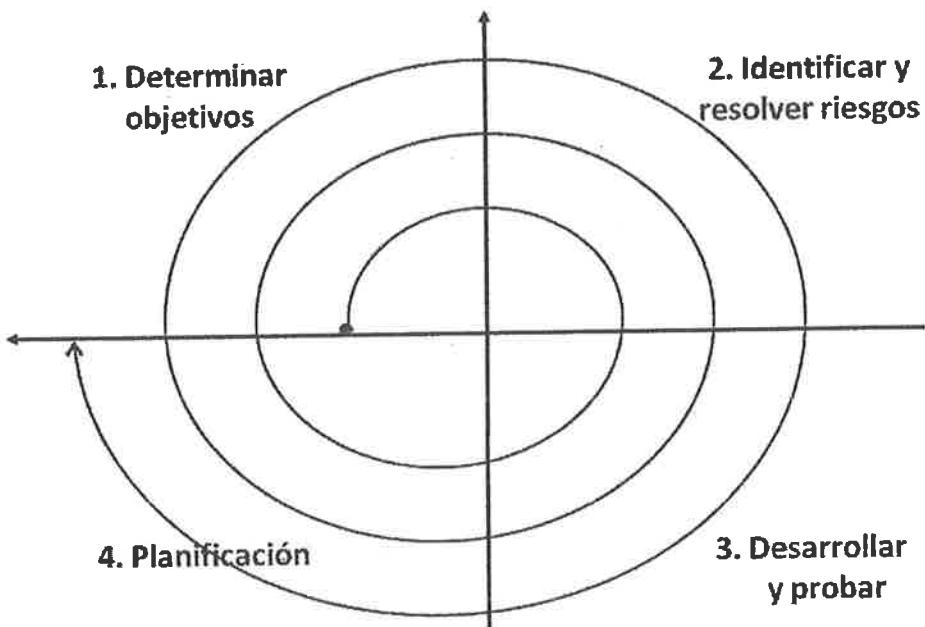


Figura 1.7. Modelo en espiral.

Durante los primeros ciclos la versión incremental podría ser maquetas en papel o modelos de pantallas (prototipos de interfaz); en el último ciclo se tendría un prototipo operacional que implementa algunas funciones del sistema. Para cada ciclo, los desarrolladores siguen estas fases:

1. **Determinar objetivos.** Cada ciclo de la espiral comienza con la identificación de los objetivos, las alternativas para alcanzar los objetivos (diseño A, diseño B, reutilización, compra, etc.), y las restricciones impuestas a la aplicación de las alternativas (costos, plazos, interfaz, etc.)
2. **Análisis del riesgo.** A continuación hay que evaluar las alternativas en relación con los objetivos y limitaciones. Con frecuencia, en este proceso se identifican los riesgos involucrados y (si es posible) la manera de resolverlos. Un riesgo puede ser cualquier cosa: requisitos no comprendidos, mal diseño, errores en la implementación, etc. Utiliza la construcción de prototipos (representación limitada de un producto) como mecanismo de reducción de riesgos.
3. **Desarrollar y probar.** Desarrollar la solución al problema en este ciclo, y verificar que es aceptable.
4. **Planificación.** Revisar y evaluar todo lo que se ha hecho, y con ello decidir si se continua, entonces hay que planificar las fases del ciclo siguiente.

La Figura 1.8 muestra un ejemplo de este modelo con cuatro ciclos. En el ciclo más interno se comienza con los requisitos y un plan inicial de desarrollo; se evalúan los riesgos y se construyen prototipos de las alternativas. Para terminar, se construye un documento con «el concepto de las operaciones» que describen la funcionalidad del sistema. Al final del ciclo se genera el plan de requisitos del sistema y el plan para todo el ciclo de vida útil.

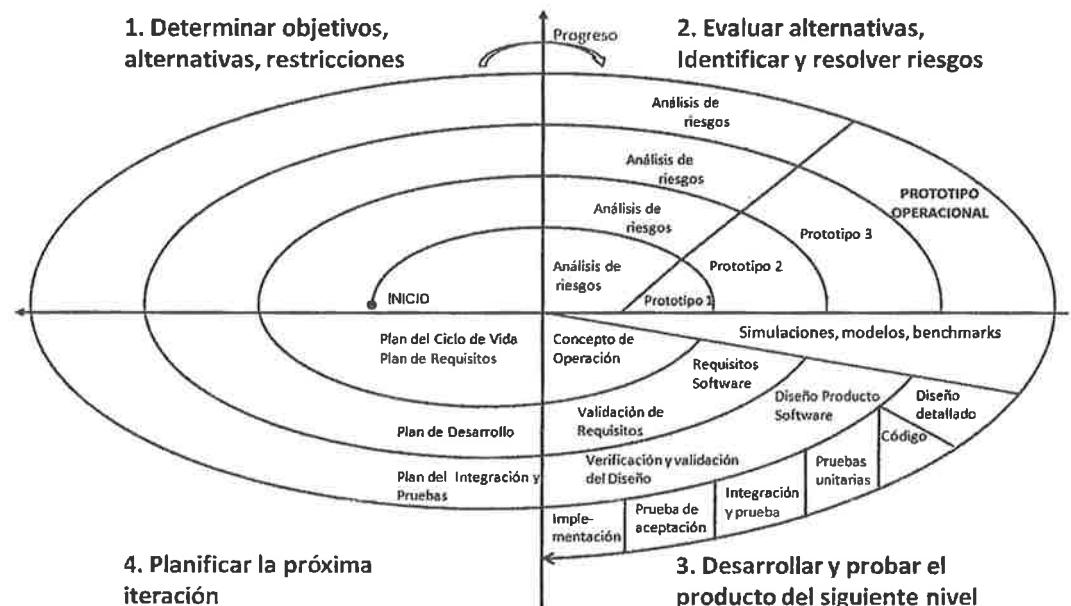


Figura 1.8. Ejemplo de Modelo en espiral.

A partir de aquí comienza el segundo ciclo, el resultado será la especificación y validación de los requisitos del software y la elaboración del plan de desarrollo. En el tercer ciclo se hace el diseño del producto software, la verificación y validación del diseño y se produce el plan de integración y pruebas. En el cuarto ciclo los desarrolladores producen un diseño detallado; implementan los módulos, realizan pruebas unitarias, de integración y aceptación. En este punto del desarrollo se puede haber alcanzado el éxito o no, en cuyo caso será necesario otro ciclo.

En todos los ciclos se hace un análisis de riesgo, donde se evalúan las alternativas según los requisitos y restricciones, y se construyen prototipos para analizarlas y seleccionar una. Estos prototipos pueden ser modelos en papel, modelos de pantallas, simulaciones del sistema, prototipos operacionales, y dependerán del riesgo a evaluar, del ciclo en el que se esté y del tipo de aplicación.

Ventajas:

- No requiere una definición completa de los requisitos para empezar a funcionar.
- Análisis del riesgo en todas las etapas.
- Reduce riesgos del proyecto
- Incorpora objetivos de calidad

Inconvenientes:

- Es difícil evaluar los riesgos.
- El costo del proyecto aumenta a medida que la espiral pasa por sucesivas iteraciones.
- El éxito del proyecto depende en gran medida de la fase de análisis de riesgos.

Se recomienda para:

- Proyectos de gran tamaño y que necesitan constantes cambios.
- Proyectos donde sea importante el factor riesgo.

Este sistema es muy utilizado para el desarrollo de sistemas orientados a objetos.

¡INTERESANTE!

El modelo de construcción de prototipos comienza con la recogida de requisitos, se definen los objetivos globales para el software y se identifican los requisitos y las áreas donde es obligatoria más definición. Entonces aparece un diseño rápido que se centra en una representación de esos aspectos software que serán visibles al usuario o cliente. Esto lleva a la construcción del prototipo que puede servir como primer sistema construido.

El objetivo es crear un producto intermedio antes de realizar el producto final y así con el prototipo se puede ver cómo responde la funcionalidad prevista para el producto final.

1.4. FASES DEL DESARROLLO DE UNA APLICACIÓN

Antes de desarrollar un proyecto software hay que elegir un modelo de ciclo de vida. Será necesario examinar las características del proyecto para elegir un modelo u otro. Independientemente del modelo elegido hay una serie de etapas que se deben seguir para construir un proyecto de calidad, algunas se nombraron en el epígrafe anterior. Estas etapas se tratan a continuación.

1.4.1. Análisis

Lo más importante del éxito de un proyecto software es entender y comprender el problema que se necesita resolver, y una vez comprendido darle solución. En esta fase se analizan y especifican los **requisitos** o capacidades que el sistema debe tener porque el cliente así lo ha pedido.

La obtención de requisitos no es tarea fácil ya que el cliente puede no tenerlos claros, pueden surgir nuevos requisitos, puede cambiar lo especificado, pueden existir malos entendidos por falta de conocimiento del equipo de desarrollo sobre el problema a resolver, el cliente puede no expresarse de forma clara debido a la falta de conocimientos informáticos, etc. Para realizar un proyecto satisfactorio es necesario obtener unos buenos requisitos, y para ello es esencial una buena comunicación entre el cliente y los desarrolladores. Para facilitar esta comunicación se utilizan varias técnicas, algunas son las siguientes:

- **Entrevistas.** Es la técnica más tradicional que consiste en hablar con el cliente. Hay que tener sobre todo conocimientos de psicología.
- **Desarrollo conjunto de aplicaciones (JAD, Joint Application Development).** Se apoya en la dinámica de grupos, es un tipo de entrevista muy estructurada aplicable a grupos de personas (usuarios, administradores, analistas, desarrolladores, etc.) Cada persona juega un rol concreto y todo lo que se hace está reglamentado.
- **Planificación conjunta de requisitos (JRP Joint Requirements Planning).** Es un subconjunto de JAD, se caracterizan por estar dirigidas a la alta dirección y en consecuencia los productos resultantes son los requisitos de alto nivel o estratégicos.
- **Brainstorming.** Es un tipo de reuniones en grupo cuyo objetivo es generar ideas desde diferentes puntos de vista para la resolución de un problema. Su utilización es adecuada al principio del proyecto, pues puede explorar un problema desde muchos puntos de vista.

- **Prototipos.** Es una versión inicial del sistema, se utiliza para clarificar algunos puntos, demostrar los conceptos, en definitiva, para enterarse más acerca del problema y sus posibles soluciones. Después se tira o bien se usa como base para añadir más cosas.
- **Casos de uso.** Es la técnica definida en UML (*Unified Modeling Language*), se basa en la representación de escenarios que describen el comportamiento deseado del sistema, es decir lo que queremos que haga el sistema. Representan **requisitos funcionales** del mismo. Es importante resaltar que describen qué hace el sistema, no cómo lo hace.

Se especifican dos tipos de requisitos:

- **Requisitos funcionales.** Describen con detalle la función que realiza el sistema, cómo reacciona ante determinadas entradas, cómo se comporta en situaciones particulares, etc.
- **Requisitos no funcionales.** Tratan sobre las características del sistema, como puede ser la fiabilidad, mantenibilidad, sistema operativo, plataforma hardware, restricciones, limitaciones, etc.

La siguiente tabla muestra un ejemplo de requisitos funcionales y no funcionales para una aplicación de gestión de una agenda de contactos:

Requisitos funcionales	Requisitos no funcionales
El usuario puede agregar un nuevo contacto	La aplicación debe funcionar en sistemas operativos Linux y Windows
El usuario puede ver una lista con todos los contactos	El tiempo de respuesta a consultas, altas, bajas y modificaciones ha de ser inferior a 5 segundos
A partir de la lista de contactos el usuario puede acceder a un contacto	Utilizar un sistema gestor de base de datos para almacenar los datos
El usuario puede eliminar un contacto o varios de la lista	Utilizar un lenguaje multiplataforma para el desarrollo de la aplicación
El usuario puede modificar los datos de un contacto seleccionado de la lista	La interfaz de usuario es a través de ventanas, debe ser intuitiva y fácil de manejar
El usuario puede seleccionar determinados contactos	El manejo de la aplicación se realizará con el teclado y el ratón
El usuario puede imprimir la lista de contactos	Espacio libre en disco, mínimo: 1GB. Mínima cantidad de memoria 2GB

Para representar los requisitos se utilizan diferentes técnicas:

- **Diagramas de flujo de datos, DFD.** Es un diagrama que representa el flujo de datos entre los distintos procesos, entidades externas y almacenes que forman el sistema. Los procesos identifican funciones dentro del sistema, se representan mediante burbujas ovaladas o circulares. Las entidades externas representan componentes que no forman parte del sistema (por ejemplo una persona, un departamento, etc.) pero proporcionan datos al sistema o los reciben de él, se representan mediante rectángulos. Los almacenes representan los datos desde el punto de vista estático, es decir, representan el lugar donde se almacenan los datos procesados o desde donde se recuperan para apoyar un proceso; se representan gráficamente mediante dos líneas horizontales y paralelas. Por último, el flujo de datos representa el movimiento de datos dentro del sistema, se representa mediante flechas. En la Figura 1.9 se muestran varios procesos que aceptan datos de entrada y los trasforman en datos de salida, las entidades externas proporcionan datos al sistema o los reciben de él, el

almacenamiento recibe datos de un proceso y el mismo proceso consulta datos sobre él, por último las flechas representan el flujo de los datos. Al final del capítulo se muestran ejemplos de DFDs.

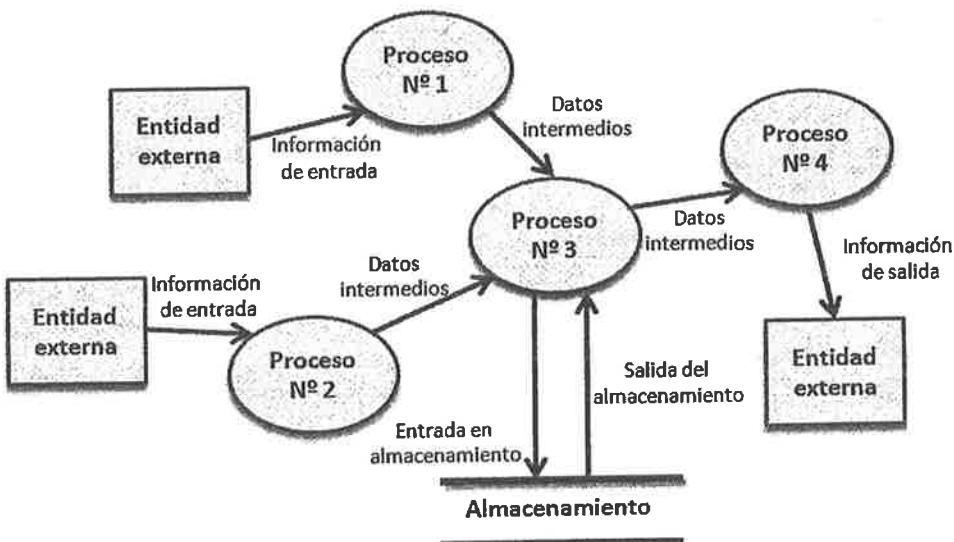


Figura 1.9. Diagrama de flujo de datos o DFD.

- ✓ • **Diagramas de flujo de control, DFC.** Similar a los DFD con la diferencia de que muestra el flujo de control, en lugar de datos.
- ✓ • **Diagramas de transición de estados, DTE.** Representa cómo se comporta el sistema como consecuencia de sucesos externos.
- ✓ • **Diagrama Entidad / Relación, DER.** Usado para representar los datos y la forma en la que se relacionan entre ellos. La Figura 1.10 representa un DER con dos entidades: EMPLEADOS y DEPARTAMENTOS y sus relaciones. La relación TIENE significa que un departamento tiene como mínimo 1 empleado y como máximo M empleados (1:M); y un empleado pertenece a uno y solo un departamento (1:1). La relación DIRIGE significa que un empleado dirige a 0 o M empleados (0:M), y un empleado es dirigido por 0 o un empleado (0:1).



Figura 1.10. Diagrama Entidad/Relación.

- ✓ • **Diccionario de datos, DD.** Es una descripción detallada de los datos utilizados por el sistema que gráficamente se encuentran representados por los flujos de datos y almacenes presentes sobre el conjunto de DFDs.

Todo lo realizado en esta fase debe quedar reflejado en el documento de *Especificación de Requisitos del Software* (ERS), este documento no debe tener ambigüedades, debe ser completo, consistente, fácil de verificar y modificar, fácil de utilizar en la fase de explotación y mantenimiento, y fácil de identificar el origen y las consecuencias de los requisitos. Sirve como entrada para la siguiente fase en el desarrollo de la aplicación. La estructura del documento ERS propuesta por el IEEE en la última versión del estándar 830 [IEEE, 1998]³ es la siguiente:

- 1. Introducción.
 - 1.1 Propósito.
 - 1.2 Ámbito del Sistema.
 - 1.3 Definiciones, Acrónimos y Abreviaturas.
 - 1.4 Referencias.
 - 1.5 Visión general del documento.
- 2. Descripción General.
 - 2.1 Perspectiva del Producto.
 - 2.2 Funciones del Producto.
 - 2.3 Características de los usuarios.
 - 2.4 Restricciones.
 - 2.5 Suposiciones y Dependencias.
 - 2.6 Requisitos Futuros.
- 3. Requisitos Específicos.
 - 3.1 Interfaces Externas.
 - 3.2 Funciones.
 - 3.3 Requisitos de Rendimiento.
 - 3.4 Restricciones de Diseño.
 - 3.5 Atributos del Sistema.
 - 3.6 Otros Requisitos.
- 4 Apéndices.

¡INTERESANTE!

Consulta el documento <http://www.fdi.ucm.es/profesor/gmendez/docs/is0809/ieee830.pdf> para ver qué es lo que se desarrolla dentro de cada apartado.

1.4.2. Diseño

Una vez identificados los requisitos es necesario componer la forma en que se solucionará el problema. En esta etapa se traducen los requisitos funcionales y no funcionales en una representación de software.

Principalmente hay dos tipos de diseño, el **diseño estructurado** que está basado en el flujo de los datos a través del sistema; y el **diseño orientado a objetos** donde el sistema se entiende como un conjunto de objetos que tienen propiedades y comportamientos, además de eventos que activan operaciones que modifican el estado de los objetos; los objetos interactúan de manera formal con otros objetos.

El **diseño estructurado** (diseño clásico) produce un modelo de diseño con 4 elementos (véase Figura 1.11):

³ <http://www.fdi.ucm.es/profesor/gmendez/docs/is0809/ieee830.pdf>

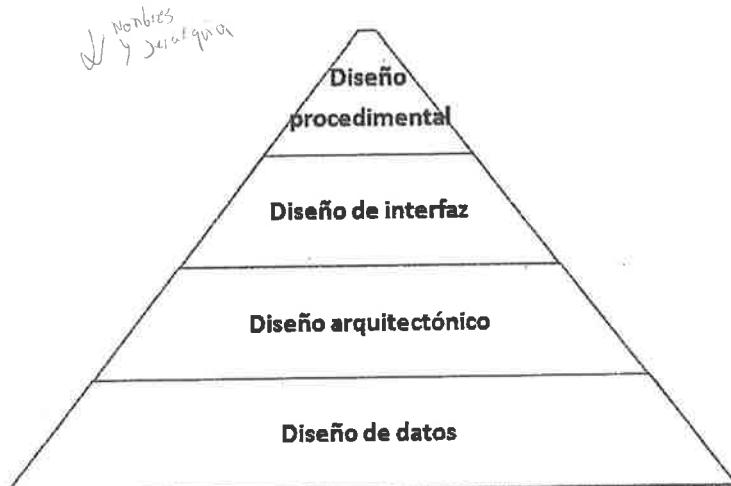


Figura 1.11. Modelo de diseño.

- **Diseño de datos.** Se encarga de transformar el modelo de dominio de la información creado durante el análisis, en las estructuras de datos que se utilizarán para implementar el software. El diseño de datos está basado en los datos y las relaciones definidos en el diagrama entidad relación y en la descripción detallada de los datos (definidos en el diccionario de datos).
- **Diseño arquitectónico.** Se centra en la representación de la estructura de los componentes del software, sus propiedades e interacciones⁴. Un componente del software puede ser tan simple como un módulo de programa, pero también puede ser algo tan complicado como una base de datos o los conectores que permitan la comunicación, coordinación y cooperación entre los componentes. Partiendo de los DFD se establece la estructura modular del software que se desarrolla.
- **Diseño de la interfaz.** Describe cómo se comunica el software consigo mismo, con los sistemas que operan con él, y con las personas que lo utilizan. El resultado de esta tarea es la creación de formatos de pantalla.
- **Diseño a nivel de componentes (diseño procedimental).** Transforma los elementos estructurales de la arquitectura del software en una descripción procedural de los componentes del software. El resultado de esta tarea es el diseño de cada componente software con el suficiente nivel de detalle para que pueda servir de guía en la generación de código fuente en un lenguaje de programación. Para llevar a cabo este diseño se utilizan representaciones gráficas mediante diagramas de flujo, diagramas de cajas, tablas de decisión, pseudocódigo, etc.

Los fundamentos del diseño a nivel de componentes se establecieron cuando se propuso el uso de un conjunto de construcciones lógicas con las que podía formarse cualquier programa. Las construcciones son: secuencial, condicional y repetitiva⁴ (Figuras 1.12, 1.13 y 1.14); y son fundamentales para la **programación estructurada**:

- La construcción **secuencial**: implementa los pasos del proceso esenciales para la especificación de cualquier algoritmo.

⁴ Ingeniería del software. Un enfoque práctico. Roger S. Pressman.

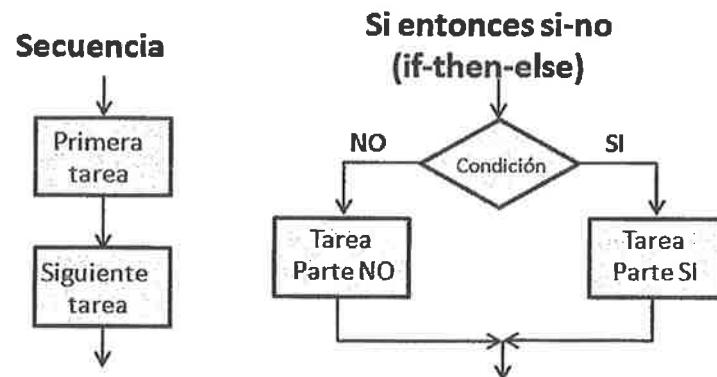


Figura 1.12. Construcción secuencial y condicional.

- La **condicional**: permite seleccionar un proceso u otro a partir de la evaluación de una condición lógica (que se representa mediante un rombo). Si se cumple la condición se realiza la tarea de la *Parte SI*, si no se cumple se realiza la tarea de la *Parte NO*. La selección múltiple es una extensión de la estructura *Si entonces si-no*; en este caso un parámetro se prueba por decisiones sucesivas hasta que ocurre una condición verdadera y se ejecuta el camino de procesamiento asociado. Por ejemplo, en la Figura 1.13 si al evaluar el parámetro ocurre la *Condición 1* se ejecuta la tarea *Parte Caso 1*, si ocurre la *Condición 2* se ejecuta la tarea *Parte Caso 2* y si ocurre la *Condición 3* se ejecuta la tarea *Parte Caso 3*.

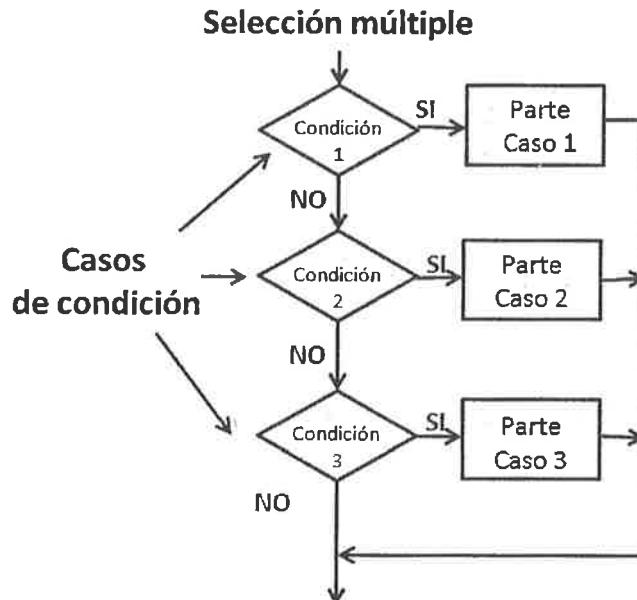


Figura 1.13. Construcción selección múltiple.

- La **repetitiva**: proporciona los bucles. *Repetir-hasta*: primero ejecuta la tarea del bucle y después comprueba la condición, si no se cumple se vuelve a realizar la tarea; si la condición se cumple finaliza el bucle, la tarea se realiza al menos una vez. *Hacer-mientras*: primero se comprueba la condición y después se realiza la tarea del bucle repetidamente siempre y cuando la condición se cumpla; el bucle finaliza cuando la condición no se cumple.

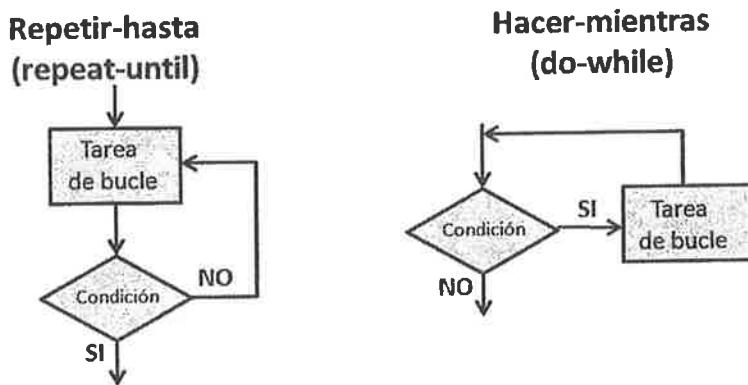


Figura 1.14. Construcciones repetitivas.

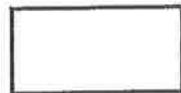
1.4.2.1. Notaciones gráficas para el diseño

Para representar el diseño se emplean algunas herramientas gráficas como son: diagramas de flujo, diagramas de cajas, tablas de decisión o pseudocódigo.

DIAGRAMA DE FLUJO

Es una herramienta muy usada para el diseño procedimental. Se utilizan los símbolos vistos anteriormente:

Una caja indica un paso del proceso.



Un rombo representa una condición lógica.



Las flechas indican el flujo de control.

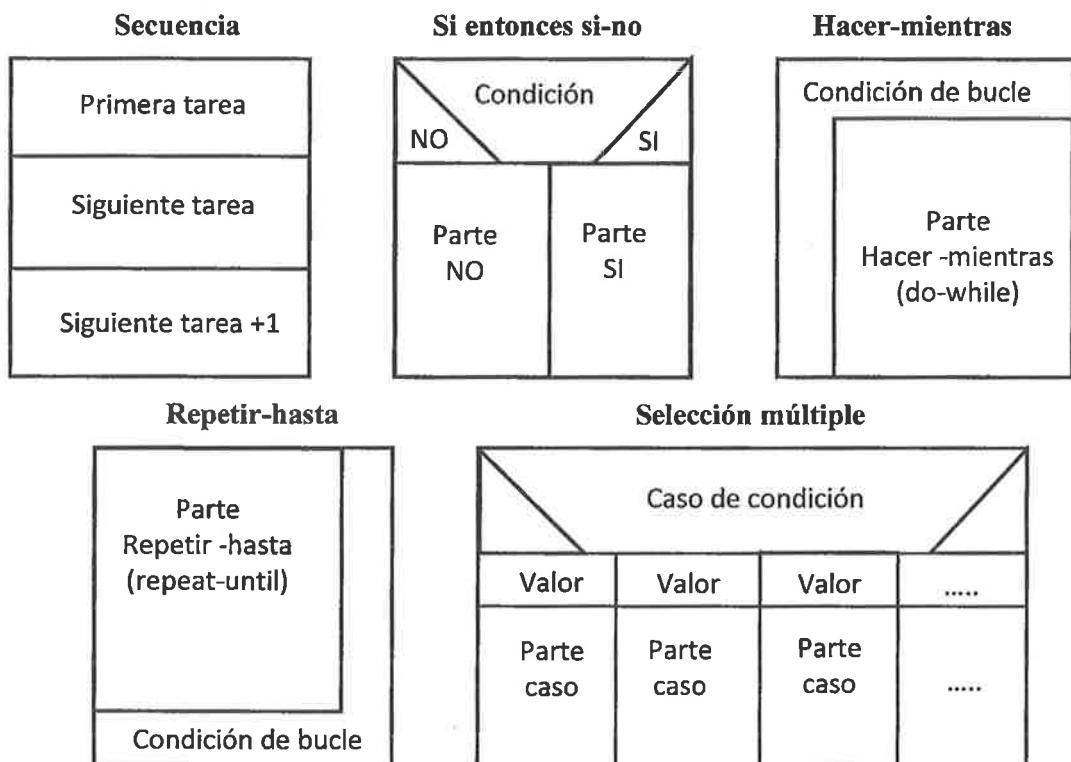


DIAGRAMA DE CAJAS

Surgió con el deseo de desarrollar una representación de diseño procedimental que no permitiera la violación de las construcciones estructuradas. El elemento fundamental es la caja:

- Para representar una secuencia se conectan varias cajas seguidas.
- Para una condicional se representa una caja para la *Parte SI* y otra para la *Parte NO*, encima se indica la condición.
- En las repetitivas, el proceso a repetir se encierra en una caja que está dentro de otra caja donde en la parte superior (*do-while*) o inferior (*repeat-until*) se indica la condición del bucle.
- Por último, en la selección múltiple en la parte superior se indica el caso de condición, se definen tantas columnas como valores se vayan a comprobar en la condición, debajo de cada valor se indica la parte a realizar.

Son las siguientes:



TABLAS DE DECISIÓN

A veces en una aplicación es necesario que un módulo de programa evalúe una combinación compleja de condiciones y seleccione las acciones según esas condiciones. Las tablas de decisión permiten representar en forma de tabla las condiciones y las acciones que se llevan a cabo combinando esas condiciones. Se dividen en cuatro cuadrantes:

- Cuadrante superior izquierdo: contiene la lista de todas las condiciones posibles.
- Cuadrante inferior izquierdo: lista de todas las acciones posibles basándose en la combinación de condiciones.
- Cuadrante superior derecho: entrada de las condiciones.
- Cuadrante inferior derecho: entrada de las acciones.

Reglas					
Condiciones	1	2	3	4	n
Condición nº 1	✓		✓		
Condición nº 2		✓	✓		
Condición nº 3	✓			✓	
Acciones					
Acción nº1	✓			✓	
Acción nº 2	✓	✓			
Acción nº 3		✓	✓	✓	
Acción nº 4			✓	✓	

Cada columna de los cuadrantes de la derecha forma una regla de procesamiento, las reglas establecen las acciones a realizar con cada combinación de condiciones. Para construir una tabla se realizan los siguientes pasos:

1. Hacer una lista de todas las acciones y todas las condiciones.
2. Asociar conjuntos específicos de condiciones con acciones específicas, eliminando combinaciones imposibles de condiciones.
3. Determinar las reglas indicando qué acción o acciones ocurren para un conjunto de condiciones.

Por ejemplo, en la siguiente tabla de decisión se definen las siguientes reglas:

	Reglas			
Condiciones	1	2	3	4
Cliente registrado	SI	SI	NO	NO
Importe compra > 800 €	SI	NO	SI	NO
Acciones				
Aplicar 1% Bonificación sobre el importe compra	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
Aplicar 3% Descuento sobre el importe compra	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	
Calcular Factura	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Regla 1: Si el cliente SI está registrado y el importe de la compra SI supera 800 €, entonces se aplica un 1% de bonificación y un 3% de descuento sobre el importe de la compra y se calcula la factura.

Regla 2: Si el cliente SI está registrado y el importe de la compra NO supera 800 €, entonces se aplica un 1% de bonificación sobre el importe de la compra y se calcula la factura.

Regla 3: Si el cliente NO está registrado y el importe de la compra SI supera 800 €, entonces se aplica un 3% de descuento sobre el importe de la compra y se calcula la factura.

Regla 4: Si el cliente NO está registrado y el importe de la compra NO supera 800 €, entonces se calcula la factura.

✓ PSEUDOCÓDIGO

El pseudocódigo utiliza texto descriptivo para realizar el diseño de un algoritmo. A primera vista se parece a un lenguaje de programación, ya que mezcla frases en lenguaje natural con estructuras sintácticas que incluyen palabras clave que permiten construir las estructuras básicas de la programación estructurada, declarar datos, definir subprogramas y establecer características de modularidad.

El pseudocódigo dependerá mucho del que lo escriba ya que no hay un estándar definido. Al no ser un lenguaje de programación no puede ser compilado. A continuación, se muestra una representación en pseudocódigo de las estructuras básicas de la programación estructurada.

Secuencial	Instrucción 1 Instrucción 2 Instrucción n
Condicional	Si <condición> Entonces <Instrucciones> Si no <Instrucciones> Fin si
Condicional múltiple.	Según sea <variable> Hacer Caso valor 1: <Instrucciones> Caso valor 2: <Instrucciones> Caso valor 3: <Instrucciones> Otro caso: <Instrucciones> Fin según
Repetir-hasta	Repetir <instrucciones> Hasta que <condición>
Hacer-mientras	Mientras <condición> Hacer <instrucciones> Fin mientras

El siguiente ejemplo muestra el pseudocódigo de un proceso repetitivo de lectura y tratamiento de los registros de un fichero secuencial:

```

Inicio
    Abrir Fichero
    Leer Registro del Fichero
    Mientras no sea Fin de Fichero Hacer
        Procesar Registro leído
        Leer Registro del Fichero
    Fin mientras
    Cerrar Fichero
Fin.

```

1.4.2.2. Diseño orientado a objetos

El diseño de software orientado a objetos (DOO) es difícil. Para llevarlo a cabo hay que partir de un análisis orientado a objetos (AOO). En dicho análisis se definen todas las clases que son importantes para el problema que se trata de resolver, las operaciones y los atributos asociados, las relaciones y comportamientos y las comunicaciones entre clases.

El diseño orientado a objetos define 4 capas de diseño⁵:

- **Subsistema.** Se centra en el diseño de los subsistemas que implementan las funciones principales del sistema.

⁵ Ingeniería del software. Un enfoque práctico. Roger S. Pressman.

- **Clases y Objetos.** Especifica la arquitectura de objetos global y la jerarquía de clases requerida para implementar un sistema.
- **Mensajes.** Indica cómo se realiza la colaboración entre los objetos.
- **Responsabilidades.** Identifica las operaciones y atributos que caracterizan cada clase.

Para el análisis y diseño orientado a objetos se utiliza **UML** (*Unified Modeling Language - Lenguaje de Modelado Unificado*). Es un lenguaje de modelado basado en diagramas que sirve para expresar modelos (un modelo es una representación de la realidad donde se ignoran detalles de menor importancia). Se ha convertido en el estándar de facto de la mayor parte de las metodologías de desarrollo orientado a objetos que existen hoy en día. En los capítulos 5 y 6 se tratará más ampliamente UML.

1.4.3. Codificación

Una vez realizado el diseño se realiza el proceso de codificación. En esta etapa, el programador recibe las especificaciones del diseño y las transforma en un conjunto de instrucciones escritas en un lenguaje de programación, almacenadas dentro de un programa. A este conjunto de instrucciones se le llama código fuente. El programador debe conocer la sintaxis del lenguaje de programación utilizado.

En cualquier proyecto en el que trabaja un grupo de personas debe haber unas normas de codificación y estilo, claras y homogéneas. Estas normas facilitan las tareas de corrección y mantenimiento de los programas, sobre todo cuando se realizan por personas que no los han desarrollado. Por ejemplo, estos dos programas son el mismo pero uno de ellos resulta más fácil de leer:

<pre>public class Ejemplo { public static void main(String[] args) { int suma = 0; int contador = 0; while(contador < 10) { contador++; suma = suma + contador; } System.out.println("Suma => " + suma); } }</pre>	<pre>public class Ejemplo { public static void main(String[] args) { int suma=0; int contador=0; while(contador <10) { contador++; suma=suma+contador; } System.out.println("Suma => + suma); } }</pre>
--	---

A continuación se describen una serie de normas de escritura de código fuente en Java que facilita la lectura de los programas haciendo que sea más sencillo mantenerlos y encontrar errores, incluso por personas que no han desarrollado el código⁶.

Nombres de ficheros

La extensión para los ficheros de código fuente es *.java* y para los ficheros compilados es *.class*.

⁶ Fuente: <http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html>

Organización de ficheros

Cada fichero debe contener una sola clase pública y debe ser la primera. Las clases privadas e interfaces asociados con esa clase pública se pueden poner en el mismo fichero después de la clase pública. Las secciones en las que se divide el fichero son:

- Comentarios. Todos los ficheros fuente deben comenzar con un comentario que muestra el nombre de la clase, información de la versión, la fecha, y el aviso de derechos de autor.
- Sentencias del tipo *package* e *import*. Van después de los comentarios, la sentencia *package* va delante de *import*. Ejemplo:

```
/*
 * Nombre de clase
 *
 * Información de la versión
 *
 * Fecha
 *
 * Aviso de Copyright
 */
package paquete.ejemplo;
import java.io.*;
```

- Declaraciones de clases e interfaces. Consta de las siguientes partes:
 1. Comentario de documentación (*/** ... */*) acerca de la clase o interface.
 2. Sentencia *class* o *interface*.
 3. Comentario de la implementación (*/* ... */*) de la clase o interface.
 4. Variables estáticas, en este orden: públicas, protegidas y luego privadas.
 5. Variables de instancia, en este orden: públicas, protegidas y luego privadas.
 6. Constructores.
 7. Métodos. Se agrupan por su funcionalidad, no por su alcance.

Identación

- Como norma general se usarán cuatro espacios.
- La longitud de las líneas de código no debe superar 80 caracteres.
- La longitud de las líneas de comentarios no debe superar 70 caracteres.
- Cuando una expresión no cabe en una sola línea: romper después de una coma, romper antes de un operador, alinear la nueva línea al principio de la anterior.

Comentarios

Los comentarios deben contener solo la información que es relevante para la lectura y la comprensión del programa. Existen dos tipos de comentarios: de documentación y de implementación.

Los comentarios de documentación están destinados a describir la especificación del código. Se utilizan para describir las clases Java, los interfaces, los constructores, los métodos y los

campos. Debe aparecer justo antes de la declaración. Existe una herramienta llamada *Javadoc* que genera páginas HTML partiendo de este tipo de comentarios. Tienen el siguiente formato:

```
/***
 * La clase Ejemplo proporciona ...
 */
public class Ejemplo {...
```

Los comentarios de implementación son para comentar algo acerca de la aplicación particular. Puede ser de 3 tipos:

- Comentarios de bloque:

```
/*
 * Esto es un comentario de bloque
 */
```

- Comentarios de línea: /* Esto es un comentario de línea */
- Comentario corto: // Esto es un comentario corto

Declaraciones

- Se recomienda declarar una variable por línea.
- Inicializar las variables locales donde están declaradas y colocarlas al comienzo del bloque. Ejemplo:

```
void miMetodo() {
    int var1 = 0;           //comienza el bloque de miMetodo
    int var2 = 10;
    if (var1 == var2) {
        int suma = 0;      // comienza el bloque if
        ....
    } else {
        var2 = var1;
        ....
    }
    ....
```

- En las clases e interfaces:

1. No se ponen espacios en blanco entre el nombre del método y el paréntesis “(”.
2. La llave de apertura “{” se coloca en la misma línea que el nombre del método o clase.
3. La llave de cierre “}” aparece en una línea aparte y en la misma columna que el inicio del bloque, excepto cuando el bloque está vacío.
4. Los métodos se separan por una línea en blanco.

```
class Ejemplo extends Object {
    int var1;
    int var2;

    Ejemplo(int i, int j) {
        var1 = i;
        var2 = j;
```

```

    }

    int metodoVacio() {}

    ...
}

```

Sentencias

- Cada línea debe contener una sentencia.
- Si hay un bloque de sentencias, este debe ser sangrado con respecto a la sentencia que lo genera y debe estar entre llaves aunque solo tenga una sentencia.
- Sentencias *if-else*, *if else-if else*. Definen bloques a los que se aplican las normas anteriores. Todos los bloques tienen el mismo nivel de sangrado.
- Bucles. Definen un bloque, que sigue las normas anteriores. Si el bucle está vacío no se abren ni se cierran llaves.
- Las sentencias *return* no deben usar paréntesis.

Separaciones

Mejoran la legibilidad del código. Se utilizan:

- Dos líneas en blanco: entre las definiciones de clases e interfaces.
- Una linea en blanco: entre los métodos, la definición de las variables locales de un método y la primera instrucción, antes de un comentario, entre secciones lógicas dentro de un método para mejorar la legibilidad.
- Un carácter en blanco: entre una palabra y un paréntesis, después de una coma, los operadores binarios menos el punto, las expresiones del *for*, y entre un cast y la variable.

Nombres

Los nombres de las variables, métodos, clases, etc., hacen que los programas sean más fáciles de leer ya que pueden darnos información acerca de su función. Las normas para asignar nombres son las siguientes:

- Paquetes: el nombre se escribe en minúscula, se pueden utilizar puntos para reflejar algún tipo de organización jerárquica. Ejemplo: *java.io*.
- Clases e interfaces: los nombres deben ser sustantivos. Se deben utilizar nombres descriptivos. Si el nombre está formado por varias palabras la primera letra de cada palabra debe estar en mayúscula: Ejemplo: *HiloServidor*.
- Métodos: se deben usar verbos en infinitivo. Si está formado por varias palabras el verbo debe estar en minúscula y la siguiente palabra debe empezar en mayúscula. Ejemplo: *ejecutar()*, *asignarDestino()*.
- Variables: deben ser cortas (pueden ser de una letra) y significativas. Si son varias palabras la primera debe estar en minúscula. Ejemplos: *i*, *j*, *sumaTotal*.
- Constantes: el nombre debe ser descriptivo. Se escriben en mayúsculas y si son varias palabras van unidas por un carácter de subrayado. Ejemplo: *MAX_VALOR*.

Las herramientas utilizadas para el desarrollo de los programas suelen ayudar a formatear correctamente el código. Por ejemplo, desde el entorno Eclipse con el fichero Java abierto se

puede pulsar en la opción de menú *Source->Format* para formatear el código, véase Figura 1.15 (las reglas de formato incorporadas por defecto en Eclipse cumplen en su mayoría las reglas definidas por Oracle).

Una vez generado el código fuente es necesario traducirlo a un lenguaje que sea entendido por la máquina, para ello se utilizan otros programas llamados **compiladores** o **intérpretes**. El resultado de la traducción es el **código objeto**. Este último aún no es ejecutable por la máquina. Se necesita un programa enlazador o montador que tome el código objeto y lo conecte con una serie de librerías para construir el **código ejecutable**. Una vez obtenido el código ejecutable es necesario probar el programa con el fin de comprobar y verificar que cumple con las especificaciones del diseño.

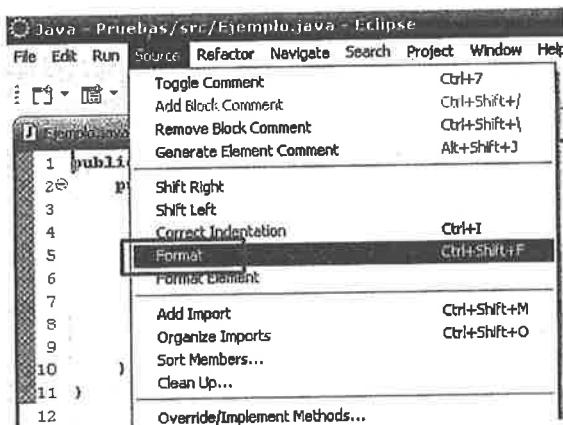


Figura 1.15. Formateo de texto desde Eclipse.

A la vez que se va desarrollando código se deben ir escribiendo los manuales técnicos y de referencia necesarios, así como la parte inicial correspondiente del manual de usuario. Esta documentación es esencial para la etapa de pruebas y mantenimiento, así como para la entrega final del producto.

1.4.4. Pruebas

En esta etapa ya se dispone del software y se trata de encontrar errores, no solo de codificación sino también relativos a la especificación o el diseño. Durante la prueba del software se realizarán tareas de verificación y validación del software (V&V):

- **La verificación:** se refiere al conjunto de actividades que tratan de comprobar si se está construyendo el producto correctamente, es decir, si el software implementa correctamente una función específica.
- **La validación:** se refiere al conjunto de actividades que tratan de comprobar si el producto es correcto, es decir, si el software construido se ajusta a los requisitos del cliente.

El objetivo en esta etapa es planificar y diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y de esfuerzo. Una prueba tiene éxito si descubre un error no detectado hasta entonces. Un caso de prueba es un documento que especifica los valores de entrada, salida esperada y las condiciones previas para la ejecución de la prueba.

Las recomendaciones para las pruebas son las siguientes:

- Cada prueba debe definir los resultados de salida esperados.
- Un programador o una organización debe evitar probar sus propios programas.
- Es necesario revisar los resultados de cada prueba en profundidad.
- Las pruebas deben incluir datos de entrada válidos y esperados, así como no válidos e inesperados.
- Centrar las pruebas en dos objetivos: 1) comprobar si el software no hace lo que debe hacer y 2) comprobar si el software hace lo que no debe hacer.
- Evitar hacer pruebas que no estén documentadas ni diseñadas con cuidado.
- No planear pruebas asumiendo que no se encontrarán errores.
- La probabilidad de encontrar errores en una parte del software es proporcional al número de errores ya encontrados.
- Las pruebas son una tarea creativa.

El flujo de proceso para probar el software se muestra en la Figura 1.16, es el siguiente:

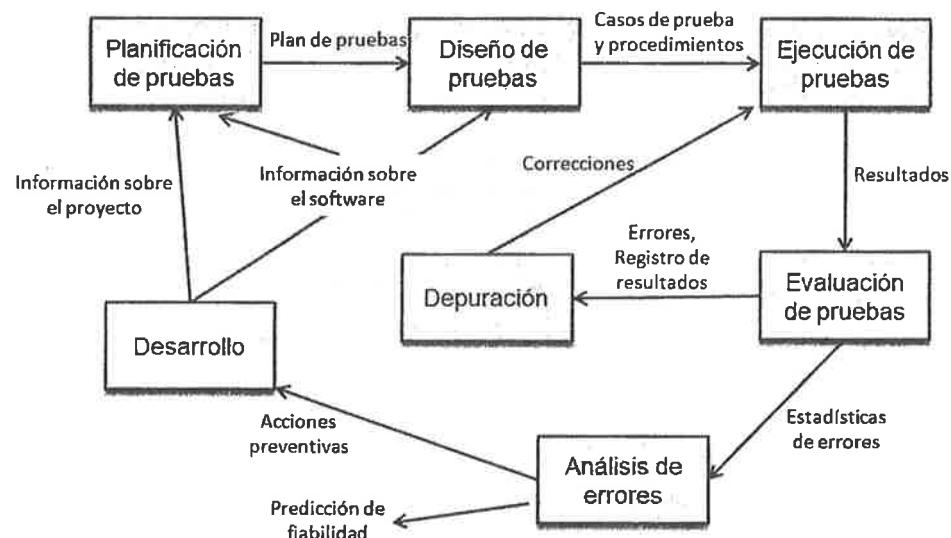


Figura 1.16. Flujo de proceso para probar el software.

1. En primer lugar, hay que **generar un plan de pruebas** partiendo de la documentación del proyecto y de la documentación sobre el software a probar.
2. A partir del plan se **diseñan las pruebas**. Se identifican las técnicas a utilizar para probar el software.
3. **Generación de los casos de prueba.** Se han de confeccionar los distintos casos de prueba según la técnica o técnicas identificadas previamente.
4. **Definición de los procedimientos de la prueba.** Hay que especificar cómo se va a llevar a cabo el proceso, quién lo va a realizar, cuándo, etc.
5. **Ejecución de las pruebas** aplicando los casos de prueba generados previamente.
6. **Evaluación.** Se identifican los posibles errores producidos al comparar los resultados obtenidos en la ejecución con los esperados. Es necesario realizar un informe con el

- resultado de ejecución de las pruebas, qué casos de prueba pasaron satisfactoriamente, cuáles no, y qué fallos se detectaron.
7. **Depuración.** Trata de localizar y corregir los errores. Si se corrige un error se debe volver a probar el software para ver que se ha resuelto el problema. Si no se consigue localizar un error puede ser necesario realizar más pruebas para obtener más información.
 8. **El análisis de errores** puede servir para predecir la fiabilidad del software y mejorar los procesos de desarrollo.

Para llevar a cabo el diseño de casos de prueba se utilizan dos técnicas: **prueba de caja blanca** y **prueba de caja negra** (véase Figura 1.17). Las primeras se centran en validar la estructura interna del programa (necesitan conocer los detalles procedimentales del código) y las segundas se centran en validar los requisitos funcionales sin fijarse en el funcionamiento interno del programa (necesitan saber la funcionalidad que el código ha de proporcionar). Estas pruebas no son excluyentes y se pueden combinar para descubrir diferentes tipos de errores.

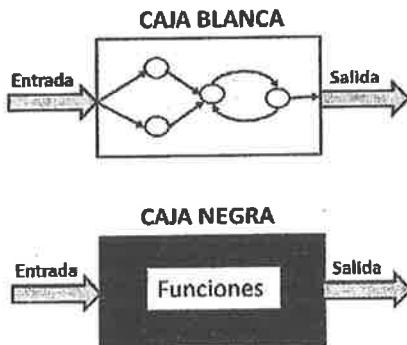


Figura 1.17. Pruebas de caja blanca y negra.

En el Capítulo 3 se tratarán más ampliamente el diseño y la realización de las pruebas.

1.4.5. Documentación

Todas las etapas del desarrollo deben quedar perfectamente documentadas. En esta etapa será necesario reunir todos los documentos generados y clasificarlos según el nivel técnico de sus descripciones.

Los documentos relacionados con un proyecto de software:

- Deben actuar como un medio de comunicación entre los miembros del equipo de desarrollo.
- Deben ser un repositorio de información del sistema para ser utilizado por el personal de mantenimiento.
- Deben proporcionar información para ayudar a planificar la gestión del presupuesto y programar el proceso del desarrollo del software.
- Algunos de los documentos deben indicar a los usuarios cómo utilizar y administrar el sistema.

Por lo general se puede decir que la documentación presentada se divide en dos clases:

- **La documentación del proceso.** Estos documentos registran el proceso de desarrollo y mantenimiento. Son documentos en los que se indican planes, estimaciones y horarios que se utilizan para predecir y controlar el proceso de software, que informan sobre cómo usar los recursos durante el proceso de desarrollo, sobre normas de cómo se ha de implementar el proceso.
- **La documentación del producto.** Esta documentación describe el producto que está siendo desarrollado. Define dos tipos de documentación: la documentación del sistema que describe el producto desde un punto de vista técnico, orientado al desarrollo y mantenimiento del mismo; y la documentación del usuario que ofrece una descripción del producto orientada a los usuarios que utilizarán el sistema.

La documentación del proceso se utiliza para gestionar todo el proceso de desarrollo del software. La documentación del producto se utiliza después de que el sistema está en funcionamiento, aunque también es esencial para la gestión del desarrollo del sistema.

DOCUMENTACIÓN DEL USUARIO

Los usuarios de un sistema no son todos iguales. Hay que distinguir entre los usuarios finales y los usuarios administradores del sistema.

 **Los usuarios finales** utilizan el software para realizar alguna tarea, quieren saber cómo el software les ayuda a realizar su tarea, no están interesados en el equipo informático o en los detalles del programa. **Los administradores del sistema** son responsables de la gestión de los programas informáticos utilizados por los usuarios finales, pueden actuar como un gestor de red si el sistema implica una red de estaciones de trabajo o como un técnico que arregla problemas de software de los usuarios finales y que sirve de enlace entre los usuarios y el proveedor de software.

Para atender a los distintos tipos de usuarios con diferentes niveles de experiencia, se definen una serie de documentos que deben ser entregados con el sistema de software (aunque se pueden entregar en el mismo documento), véase Figura 1.18:



Figura 1.18. Diferentes tipos de documentación de usuario.

Descripción funcional del sistema

Proporciona una descripción general de las funciones del sistema. Los usuarios deben ser capaces de leer este documento como un manual introductorio y decidir si el sistema es lo que necesitan

Documento de instalación del sistema	Está destinado a los administradores del sistema. Describe cómo instalar el sistema en un entorno particular. Debe contener una descripción de los ficheros que componen el sistema, su configuración, el hardware mínimo requerido, cómo iniciar el sistema, etc.
Manual introductorio.	Proporciona explicaciones sencillas de cómo empezar a utilizar el sistema. Para facilitar las explicaciones se deben usar ejemplos. Los usuarios principiantes cometan errores, se debe describir la manera de recuperarse ante ellos
Manual de referencia del sistema	Proporciona la descripción detallada de la instalación del sistema y su uso. Debe proporcionar una lista completa de mensajes de error y debe describir cómo recuperarse ante los errores detectados. Debe ser completo, formal y descriptivo
Guía del administrador del sistema	Para algunos sistemas donde hay que introducir comandos de control se deben indicar las tareas del operador, se deben describir los mensajes que se producen y las respuestas que el operador tiene que dar al sistema. Por ejemplo, se deben describir los mensajes generados cuando el sistema interactúa con otros sistemas
Tarjeta de referencia rápida	Sirven para hacer búsquedas rápidas. Por ejemplo, un sistema de ayuda en línea que contiene información breve sobre alguna parte del sistema, permite al usuario ahorrar el tiempo invertido en consultar manuales (aunque no debe ser visto como sustituto para la documentación más completa)

✓ DOCUMENTACIÓN DEL SISTEMA

La documentación del sistema incluye todos los documentos que describen el sistema, desde la especificación de requisitos hasta las pruebas de aceptación. Los documentos que describen el diseño, la implementación y las pruebas del sistema son esenciales para entender y mantener el software.

En los grandes sistemas software la documentación debe incluir:

Fundamentos del sistema	Describe los objetivos de todo el sistema
El análisis y especificación de requisitos.	Proporciona información exacta sobre la especificación de los requisitos, según lo acordado entre las partes interesadas (usuarios, clientes, desarrollador)
Diseño	Describe la arquitectura del sistema: cómo se aplican los requisitos del sistema, la forma en que se descompone el sistema en un conjunto de unidades de programas que interactúan y la función de cada unidad de programa

Implementación	Proporciona la descripción de: la forma en que se expresa el sistema detallado en algún lenguaje de programación formal y las acciones del programa en forma de comentarios dentro de los programas
Plan de pruebas del sistema	Proporciona una descripción de cómo las unidades de programa son evaluadas individualmente y como todo el sistema se prueba después de la integración
Plan de pruebas de aceptación	Se describen las pruebas que el sistema debe pasar antes de que los usuarios las acepten
Los diccionarios de datos	Contiene las descripciones de todos los términos que se relacionan con el sistema de software en cuestión

Para los sistemas más pequeños, la documentación suele ser menos amplia, como mínimo se debe tratar de mantener la especificación del sistema, el documento de diseño arquitectónico y el código fuente del programa.

Por desgracia, el mantenimiento de la documentación a menudo se descuida y puede no estar en sintonía con el software asociado, causando problemas para los usuarios y mantenedores del sistema. La mejor solución a este problema consiste en apoyar el mantenimiento de documentos con herramientas software que registran los documentos, las relaciones entre ellos, avisan a los desarrolladores cuando hay cambios en un documento, registran posibles inconsistencias en la documentación.

ESTRUCTURA DEL DOCUMENTO

La documentación debe estar correctamente organizada para permitir a los lectores localizar la información más fácilmente, organizada en capítulos, y dentro de estos capítulos, en secciones y subsecciones. La estructura del documento tiene un impacto importante en la legibilidad y usabilidad. Algunas pautas mínimas de estructuración son las siguientes:

- Todos los documentos, por muy breves que sean, deben tener una portada que identifica el proyecto, el tipo de documento, el autor, la fecha de la creación, la versión, los revisores, los destinatarios del documento y la clase de la confidencialidad del documento.
- Deben contener un índice con capítulos, secciones y subsecciones. Se debe definir un esquema de numeración coherente para capítulos, secciones y subsecciones y para numerar las páginas (el número de página debe ser capítulo – página).
- Se debe incluir al final un glosario de términos ya que el documento está dirigido a un amplio espectro de lectores que pueden tener diferentes vocabularios.

El estándar IEEE para la documentación del usuario (IEEE , 2001) propone que la estructura de un documento debe incluir los componentes que se muestran en la tabla:

COMPONENTE	DESCRIPCIÓN
Datos de identificación	Datos como el título y el identificador que únicamente identifican el documento
Tabla de contenidos	Capítulo, nombres de las secciones y los números de página
Lista de ilustraciones	Números de figura y títulos
Introducción	Define el propósito del documento y un breve resumen de los contenidos
Información para el uso de la documentación	Sugerencias para diferentes lectores sobre cómo utilizar la documentación con eficacia
Conceptos de las operaciones	Una explicación de la base conceptual para el uso del software
Procedimientos	Las instrucciones sobre cómo utilizar el software
Información sobre los comandos software	Una descripción de cada uno de los comandos soportados por el software
Mensajes de error y resolución de problemas	Una descripción de los mensajes de error que pueden aparecer y cómo recuperarse de ellos
Glosario	Las definiciones de términos especiales utilizados
Fuentes de información relacionadas	Las referencias o enlaces a otros documentos que proporcionan información adicional
Características de navegación	Las características que permiten a los lectores encontrar su actual ubicación y moverse por el documento
Índice	Una lista de términos clave y las páginas en las que estos términos se mencionan
Capacidad de búsqueda	En la documentación electrónica, indica la manera de encontrar términos específicos en la documentación



1.4.6. Explotación

Una vez que se han realizado todas las pruebas y documentado todas las etapas, se pasa a la explotación del sistema. En esta etapa se lleva a cabo la instalación y puesta en marcha del producto software en el entorno de trabajo del cliente.

En esta etapa se llevan a cabo las siguientes tareas:

- Se define la estrategia para la implementación del proceso. Se desarrolla un plan donde se establecen las normas para la realización de las actividades y tareas de este proceso. Se definen los procedimientos para recibir, registrar, solucionar, hacer un seguimiento de los problemas y para probar el producto software en el entorno de trabajo.

- ✓ ▪ **Pruebas de operación.** Para cada release del producto software, se llevarán a cabo pruebas de funcionamiento y tras satisfacerse los criterios especificados, se libera el software para uso operativo.
- ✓ ▪ **Uso operacional del sistema.** El sistema entrará en acción en el entorno previsto de acuerdo con la documentación de usuario.
- ✓ ▪ **Soporte al usuario.** Se deberá proporcionar asistencia y consultoría a los usuarios cuando la soliciten. Estas peticiones y las acciones subsecuentes se deberán registrar y supervisar.

1.4.7. Mantenimiento

El mantenimiento del software se define (*Norma IEEE 1219 para Mantenimiento de Software*) como la modificación de un producto de software después de la entrega para corregir los fallos, para mejorar el rendimiento u otros atributos, o para adaptar el producto a un entorno modificado.

Existen cuatro tipos de mantenimiento del software que dependen de las demandas de los usuarios del producto software a mantener (véase Figura 1.19):

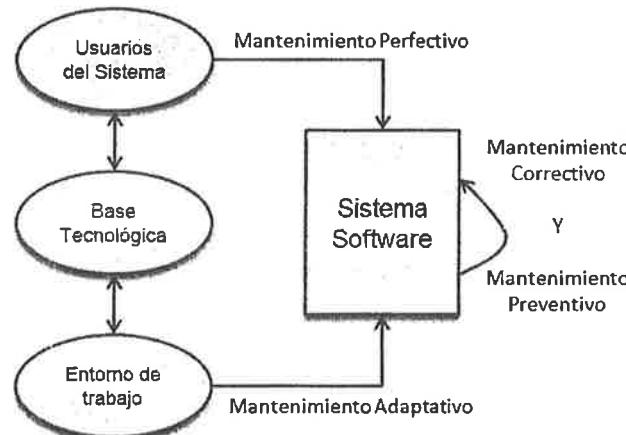


Figura 1.19. Tipos de mantenimiento del software.

- ✓ ▪ **Mantenimiento adaptativo.** Con el paso del tiempo es posible que se produzcan cambios en el entorno original (CPU, sistema operativo, reglas de la empresa, etc.) para el que se desarrolló el software. El mantenimiento adaptativo tiene como objetivo la modificación del producto por los cambios que se produzcan, tanto en el hardware como en el software del entorno en el que se ejecuta. Este tipo de mantenimiento es el más usual debido a los rápidos cambios que se producen en la tecnología informática, que en la mayoría de ocasiones dejan obsoletos los productos software desarrollados.
- ✓ ▪ **Mantenimiento correctivo.** Es muy probable que después de la entrega del producto, el cliente encuentre errores o defectos, a pesar de las pruebas y verificaciones realizadas en las etapas anteriores. Este tipo de mantenimiento tiene como objetivo corregir los fallos descubiertos.

- **Mantenimiento perfectivo.** Conforme el cliente utiliza el software puede descubrir funciones adicionales que le pueden aportar beneficios. El mantenimiento perfectivo es la modificación del producto de software orientado a incorporar nuevas funcionalidades (más allá de los requisitos funcionales originales) y nuevas mejoras en el rendimiento o la mantenibilidad del producto.
- **Mantenimiento preventivo.** Consiste en la modificación del producto de software sin alterar las especificaciones del mismo, con el fin de mejorar y facilitar las tareas de mantenimiento. Este tipo de mantenimiento hace cambios en programas con el fin de que se puedan corregir, adaptar y mejorar más fácilmente, por ejemplo, se pueden reestructurar los programas para mejorar su legibilidad o añadir nuevos comentarios que faciliten la comprensión del mismo. A este mantenimiento también se le llama *reingeniería del software*.

Las tareas que se llevan a cabo en esta etapa son (véase Figura 1.20):

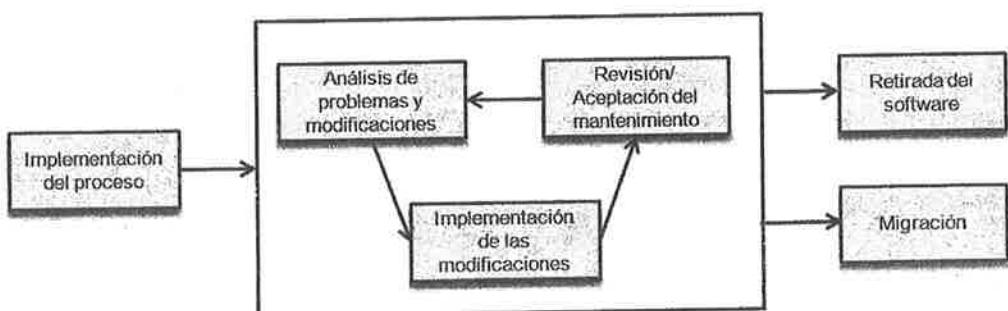


Figura 1.20. Tareas en el mantenimiento del software.

- **Implementación del proceso.** El responsable de mantenimiento deberá preparar, documentar y ejecutar planes y procedimientos para llevar a cabo las actividades y tareas del proceso de mantenimiento. Deberá establecer procedimientos para recibir, registrar y hacer seguimiento a los informes de problemas y a las peticiones de modificaciones de los usuarios y proporcionar información a los usuarios sobre su situación.
- **Análisis de problemas y modificaciones.** El responsable de mantenimiento deberá analizar el informe del problema o la petición de modificación, de acuerdo con su impacto en la organización, el sistema existente y los sistemas con los que interacciona. Deberá verificar el problema, desarrollar opciones para la implementación de la modificación y documentar la solicitud de modificación, los resultados del análisis y las opciones de implementación.
- **Implementación de las modificaciones.** El responsable de mantenimiento tiene que llevar a cabo el análisis y determinar qué documentación, unidades de software y versiones se deben ser modificar, es preciso documentar este análisis. Deberá ejecutar el proceso de desarrollo para implementar las modificaciones.

- **Revisión/aceptación del mantenimiento.** El responsable de mantenimiento deberá llevar a cabo revisiones, con la organización que autoriza las modificaciones, para determinar la integridad del sistema modificado. Deberá obtener aprobación para la finalización satisfactoria de la modificación.
- **Migración.** Se deberá preparar, documentar y ejecutar un plan de migración. Las actividades de planificación deberán incluir a los usuarios a los que se les dará notificación de los planes y actividades de migración.
- **Retirada del software.** El producto software se retirará por petición del propietario. Se deberá preparar y documentar un plan de retirada para el cese del soporte activo por parte de las organizaciones de operación y mantenimiento. Se deberá notificar a los usuarios los planes y actividades de la retirada. Para facilitar la transición al nuevo sistema, conviene que se lleve a cabo la operación en paralelo del sistema a retirar y del nuevo producto software. Durante este período, se deberá proporcionar formación a los usuarios.

1.5. CONCEPTO DE PROGRAMA

Un **programa informático** es un conjunto de instrucciones escritas en un lenguaje de programación que aplicadas sobre un conjunto de datos resuelven un problema o parte del mismo. Para que el programa pueda ser ejecutado es necesario traducirlo a un lenguaje entendible por el ordenador, el lenguaje máquina; esta tarea se lleva a cabo por otro programa llamado compilador. Una vez que tenemos el programa en código entendible por la máquina hay que cargarlo en la memoria principal para que el procesador ejecute una a una todas las instrucciones.

1.5.1. Programa y componentes del sistema informático

Para ejecutar un programa se necesitan los recursos hardware del ordenador: el procesador (también conocido como UCP *Unidad Central de Proceso* o CPU), la memoria RAM (o memoria principal), los dispositivos de E/S, etc. Las instrucciones de un programa se cargan en la memoria principal y la CPU será la encargada de ejecutarlas.

En una arquitectura Von Neumann, véase Figura 1.21, son varios los componentes que forman la CPU:

- **La Unidad de Control (UC)** que interpreta y ejecuta las instrucciones máquina almacenadas en la memoria principal y genera las señales de control necesarias para ejecutarlas.
- **La Unidad Aritmético-Lógica (UAL)**, recibe los datos sobre los que efectúa operaciones de cálculo y comparaciones, y toma decisiones lógicas (determina si una afirmación es cierta o falsa mediante las reglas del álgebra de Boole), y devuelve el resultado, todo ello bajo la supervisión de la Unidad de Control.
- **Los registros** de trabajo o de propósito general donde se almacena información temporal, es el almacenamiento interno de la CPU.

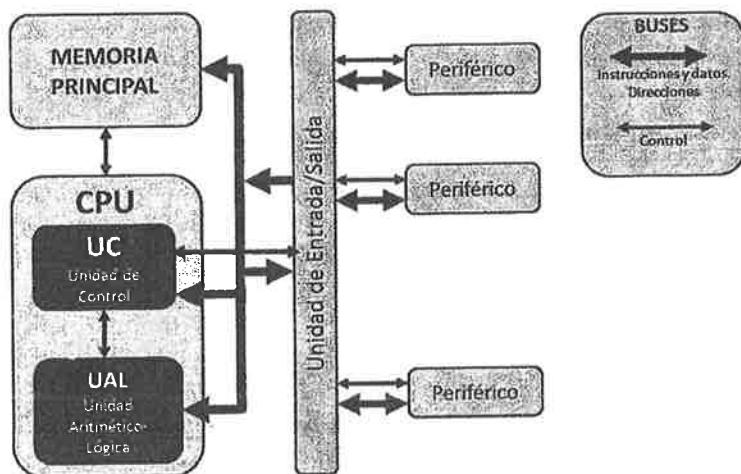


Figura 1.21. Arquitectura Von Neumann.

La UC consta de una serie de registros que intervienen en la ejecución de las instrucciones, un decodificador y un reloj:

- **Contador de Programa** o de instrucciones (CP, *Program Counter*). Contiene la dirección de la siguiente instrucción a ejecutar, su valor es actualizado por la CPU después de capturar una instrucción.
- **Registro de Instrucción** (RI, *Instruction Register*). Contiene el código de la instrucción a ejecutar, aquí se analiza el código de operación. Está dividido en dos zonas: el código de operación y la dirección de memoria donde se encuentra el operando.
- **Registro de dirección de memoria** (RDM, *Memory Address Register*). Contiene la dirección de una posición de memoria, donde se encuentra o se almacenará la información, este intercambio se realiza a través del bus de direcciones.
- **Registro de intercambio de memoria** (RIM, *Memory Buffer Register*). Recibe o envía, dependiendo de si es una operación de lectura o escritura, la información o el dato contenido en la posición apuntada por el RDM, el intercambio de datos con la memoria se realiza a través al bus de datos.
- **Decodificador de instrucción** (DI). Se encarga de extraer y analizar el código de operación de la instrucción en curso contenida en el RI y genera las señales de control necesarias para ejecutar correctamente la instrucción.
- **El Reloj**. Proporciona una sucesión de impulsos eléctricos a intervalos constantes, va marcando los tiempos de ejecución de los pasos a realizar para cada instrucción, marca el ritmo de funcionamiento del decodificador de instrucción.
- **El secuenciador**. Este dispositivo genera órdenes o microórdenes elementales, que sincronizadas con los impulsos de reloj hace que se ejecute paso a paso y de manera ordenada la instrucción cargada en él.

A la hora de ejecutar una instrucción se distinguen dos fases:

- **Fase de búsqueda.** Consiste en localizar la instrucción a ejecutar dentro de la memoria principal y llevarla a la Unidad de Control o UC para procesarla.

2. Fase de ejecución. Es la realización de las acciones que llevan asociadas las instrucciones. Por ejemplo una suma, una resta o una carga de datos.

Para poder realizar operaciones de lectura de una celda de memoria o de escritura en una celda de memoria, se utilizan: el **registro de dirección (RDM)** y el **registro de intercambio o de datos (RIM)**, y el **selector de memoria o decodificador de direcciones** que es el dispositivo que conecta la celda de memoria cuya dirección figura en el RDM con el registro de intercambio RIM, posibilitando la transferencia de los datos en un sentido o en otro dependiendo de la operación de lectura o de escritura.

Vamos a ver con un ejemplo cómo se llevarían a cabo las fases de búsqueda y ejecución para un programa almacenado en memoria que lee dos números, los suma y visualiza la suma; podemos imaginarnos la memoria como una serie de casillas con su dirección asociada. El mapa de memoria se muestra a continuación:

MEMORIA			
PROGRAMA		DATOS	
1	9	17	25
2	10	18	26
3	11	19	27 Dato A
4 Leer A	12	20	28 Dato B
5 Leer B	13	21	29 Dato C
6 Calcular C=A+B	14	22	30
7 Visualizar C	15	23	31
8	16	24	32

Las instrucciones del programa comienzan en la dirección de memoria 4. Los datos se van a almacenar en las posiciones 27, 28 y 29, tal y como se muestra en el dibujo. Cada instrucción tiene su fase de búsqueda y su fase de ejecución. Se ejecuta la instrucción *Leer A*. En la Figura 1.22 podemos ver la fase de búsqueda de esta instrucción

Fase de búsqueda

1. En el CP se almacena la dirección de memoria de comienzo del programa, la 4.
2. La UC envía una orden para que el contenido del CP se transfiera al RDM (registro de dirección de memoria).
3. El selector de memoria localiza la posición 4, y transfiere su contenido al RIM (registro de intercambio). El RIM contiene: *Leer A*.
4. La UC da la orden de transferir el contenido del RIM al registro de instrucción RI en el que deposita el código de la instrucción a ejecutar.
5. Seguidamente, el decodificador de instrucción (DI) analiza el código contenido en el RI (en el ejemplo la operación es *Leer A*) y genera las señales de control para ejecutar correctamente la instrucción.
6. El CP se incrementa en uno y apuntará a la siguiente instrucción, en este caso es la 5 *Leer B*.

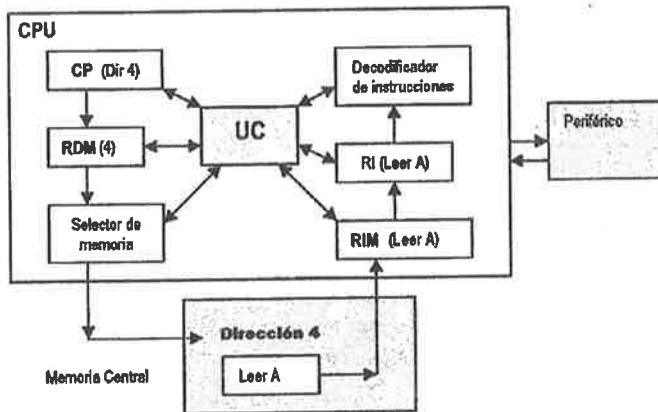


Figura 1.22. Fase de búsqueda de la instrucción Leer A.

Fase de ejecución

7. Una vez conocido el código de operación, la U.C. establece las conexiones con el dispositivo de entrada para aceptar el dato A.
8. La UC da la orden de que el dato leído se cargue en el RIM y que en el RDM se cargue la dirección de memoria donde se va a almacenar el dato leído, en el ejemplo esta dirección es la 27.
9. El selector de memoria se encarga de guardar en la dirección indicada por el RDM (la dirección 27) el contenido del RIM.

Aquí termina la ejecución de la primera instrucción, véase Figura 1.23.

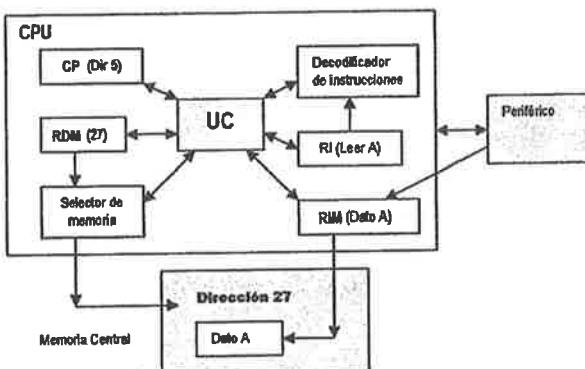


Figura 1.23. Fase de ejecución de la instrucción Leer A.

Las fases de la instrucción Leer B son similares a Leer A. Veamos las fases en la instrucción Calcular $C=A+B$.

Fase de búsqueda

1. El CP contiene la dirección 6.
2. La UC ordena que el contenido del CP pase al RDM.

3. El selector de memoria localiza la posición 6, y transfiere su contenido al RIM. El RIM contiene: *Calcular C=A+B*.
4. La UC da la orden de transferir el contenido del RIM al registro de instrucción RI.
5. Seguidamente, el decodificador de instrucción (DI) analiza el código contenido en el RI y genera las señales de control para ejecutar correctamente la instrucción.
6. El CP se incrementa en uno y apunta la siguiente instrucción, en este caso es la 7, *Visualizar C*.

Fase de ejecución:

7. Se transfiere la dirección del primer operando A (27) desde el RI hasta el RDM.
8. El selector extrae el contenido de la posición 27 y lo deposita en el RIM.
9. El contenido del RIM se carga en el registro acumulador de la UAL, véase Figura 1.24.

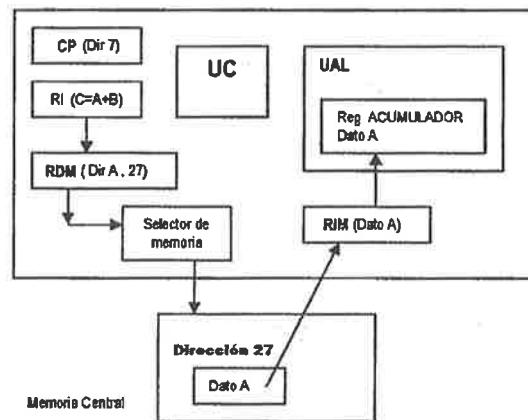


Figura 1.24. Carga en el acumulador el dato A.

10. Se transfiere la dirección del segundo operando B (28) desde el RI hasta el RDM.
11. El selector extrae el contenido de la posición 28 y lo deposita en el RIM.
12. Se envía una orden para que el contenido del RIM se sume al contenido del registro acumulador de la UAL, y el resultado se almacene en el acumulador. Véase Figura 1.25.

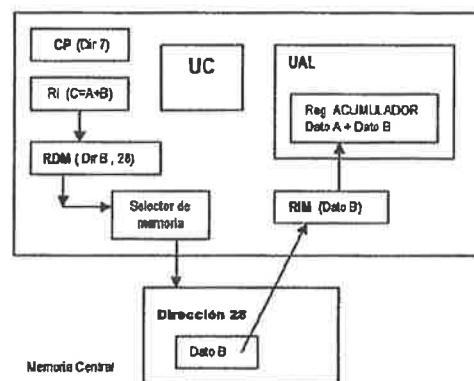


Figura 1.25. Suma al acumulador el dato B.

13. Se envía este resultado desde el registro acumulador al RIM para almacenarlo en memoria, pero antes hay que localizar en qué dirección se va a almacenar.
14. Se transfiere desde el RI (recuerda que en el RI está la instrucción del cálculo de la suma, es decir, el código de operación y la dirección de los operandos), al RDM la dirección donde se va a almacenar el resultado, esto es, la dirección de C (29).
15. Finalmente se transfiere el resultado desde el RIM a la dirección indicada por el RDM, el resultado C queda almacenado en la posición 29. Véase Figura 1.26.

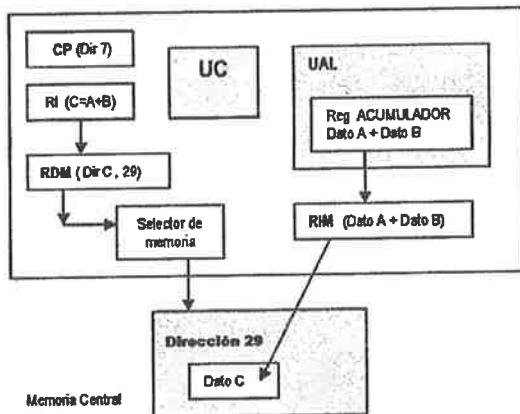


Figura 1.26. Carga en Dato C el acumulador que contiene la suma.

Por último se ejecuta la instrucción *Visualizar C*.

Fase de búsqueda

1. El CP contiene la dirección 7.
2. El contenido del CP se transfiera al RDM.
3. El selector de memoria localiza la posición 7, y transfiere su contenido al RIM.
4. La UC da la orden de transferir el contenido del RIM al registro de instrucción RI.
5. El decodificador de instrucción (DI) analiza el código contenido en el RI y genera las señales de control para ejecutar la instrucción.
6. El CP se incrementa en uno y su valor es 8, fin de programa.

Fase de ejecución

7. Una vez conocido el código de operación, la U.C. establece las conexiones con el dispositivo de salida para visualizar el dato C.
8. La dirección del dato a visualizar se carga en el RDM, la 29.
9. El selector de memoria selecciona esa posición y guarda en el RIM el contenido del dato C.
10. Se generan las órdenes para que el contenido del RIM salga por la unidad de salida.

1.6. LENGUAJES DE PROGRAMACIÓN

Anteriormente se definió lo que es un programa informático, un conjunto de instrucciones escritas en un lenguaje de programación. Podemos definir un lenguaje de programación como un conjunto de caracteres, las reglas para la combinación de esos caracteres y las reglas que definen sus efectos cuando los ejecuta un ordenador. En definitiva, un lenguaje de programación consta de los siguientes elementos:

- **Un alfabeto o vocabulario (léxico):** formado por el conjunto de símbolos permitidos.
- **Una sintaxis:** son las reglas que indican cómo realizar las construcciones con los símbolos del lenguaje.
- **Una semántica:** son las reglas que determinan el significado de cualquier construcción del lenguaje.

1.6.1. Clasificación y características

Los lenguajes de programación se pueden clasificar atendiendo a varios criterios:

		Lenguajes de bajo nivel
↳	Según su nivel de abstracción	Lenguajes de nivel medio
		Lenguajes de alto nivel
↳	Según la forma de ejecución	Lenguajes compilados
		Lenguajes interpretados
		Lenguajes imperativos
		Lenguajes funcionales
↳	Según el paradigma de programación	Lenguajes lógicos
		Lenguajes estructurados
		Lenguajes orientados a objetos

SEGÚN SU NIVEL DE ABSTRACCIÓN

Lenguajes de bajo nivel

Son lenguajes de programación que se acercan al funcionamiento de un ordenador. El lenguaje de más bajo nivel por excelencia es el lenguaje máquina, que es entendible directamente por la máquina. Las instrucciones están formadas por cadenas de ceros y unos, es decir, utilizan el alfabeto binario (0 y 1). Los programas en este lenguaje son específicos para cada procesador.

A este le sigue el lenguaje ensamblador. Este lenguaje es difícil de aprender y es específico para cada procesador. Un programa escrito en este lenguaje necesita ser traducido a lenguaje máquina para poder ejecutarse. Se programa utilizando nombres nemotécnicos y las instrucciones trabajan directamente con registros de memoria física de la máquina. La Figura 1.27 muestra un programa en ensamblador.

-u 100 1a	OCFD :0100 BA0B01 OCFD :0103 B409 OCFD :0105 CD21 OCFD :0107 B400 OCFD :0109 CD21	MOV DX,010B MOV AH,09 INT 21 MOV AH,00 INT 21
-d 10b 13f	OCFD :0100 20 65 73 74 65 20 65 73-20 75 6E 20 70 72 6F 67 OCFD :0110 72 61 6D 61 20 68 65 63-68 6F 20 65 6E 20 61 73 OCFD :0120 73 65 6D 62 6C 65 72 20-70 61 72 61 20 6C 61 20 OCFD :0130 57 69 6B 69 70 65 64 69-61 24 OCFD :0140	48 6F 6C 61 2C

Hola,
este es un progra
ma hecho en as
sembler para la
Wikipedia\$

Figura 1.27. Programa en ensamblador⁷.

Lenguajes de nivel medio

Este tipo de lenguajes tienen ciertas características que los acercan a los lenguajes de bajo nivel, pero a la vez también tienen características de los lenguajes de alto nivel. Un lenguaje de programación de este tipo es el lenguaje C. Se suelen utilizar para aplicaciones como la creación de sistemas operativos.

Lenguajes de alto nivel

Son normalmente más fáciles de aprender porque están formados por palabras del lenguaje natural, como el inglés. Para poder ejecutarlos en el ordenador se necesita un programa intérprete o compilador que traduzca las instrucciones escritas en este lenguaje, en instrucciones en lenguaje máquina que el ordenador pueda entender.

Los lenguajes de programación de alto nivel son independientes de la máquina, es decir, no dependen del hardware del ordenador y no requieren ningún conocimiento de código máquina por parte del usuario que lo utiliza. El alfabeto utilizado se acerca más a la del problema que se trata de resolver que al código máquina.

Algunos lenguajes de alto nivel son: ALGOL, Basic, C++, C#, Clipper, COBOL, Fortran, Java, Logo, Pascal, Object Pascal, Perl, PHP, PL/SQL, Python, Modula-2, etc.

SEGÚN LA FORMA DE EJECUCIÓN

» Lenguajes compilados

Un programa que se escribe en un lenguaje de alto nivel tiene que traducirse a un código que pueda utilizar la máquina. Los programas traductores que pueden realizar esta operación se llaman compiladores o intérpretes.

Un compilador (véase Figura 1.28) es un programa que puede leer un programa escrito en un determinado lenguaje (un lenguaje fuente) y traducirlo en un programa equivalente en otro lenguaje (lenguaje destino). Hay que tener en cuenta que el compilador devolverá errores si el programa en el lenguaje fuente no está bien escrito. El programa destino se podrá ejecutar si el lenguaje destino es directamente ejecutable por la máquina.

⁷ http://commons.wikimedia.org/wiki/File:Codigo_de_maquina.png.

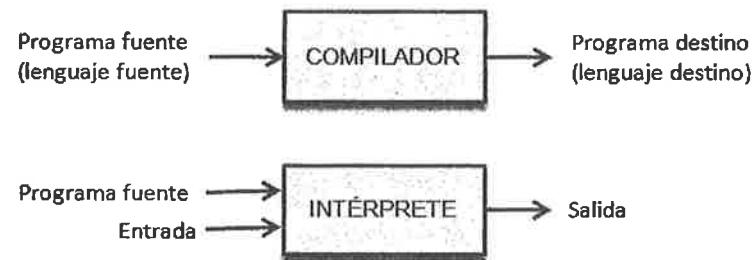


Figura 1.28. Compilador e Intérprete.

✓ Lenguajes interpretados

Los intérpretes son otra alternativa para traducir los programas escritos en lenguaje de alto nivel. En este caso, en vez de producir un programa destino como resultado del proceso de traducción, el intérprete nos da la apariencia de ejecutar directamente las operaciones especificadas en el programa fuente con las entradas proporcionadas por el usuario (véase Figura 1.28). Cada vez que se ejecuta una instrucción, se debe interpretar y traducir a lenguaje máquina.

El programa destino en el lenguaje máquina que produce un compilador es por lo general más rápido que un intérprete al momento de asignar las entradas a las salidas. El intérprete elimina la necesidad de realizar una compilación después de cada modificación del programa. Algunos ejemplos de lenguajes interpretados son: PHP, Java Script, Python, Perl, Logo, Ruby, ASP, Basic, etc.

híbrido Los procesadores del lenguaje Java combinan la compilación y la interpretación, como se muestra en la Figura 1.29. Un programa fuente en Java puede compilarse primero en un formato intermedio, llamado *bytecodes*, después una máquina virtual los interpreta.

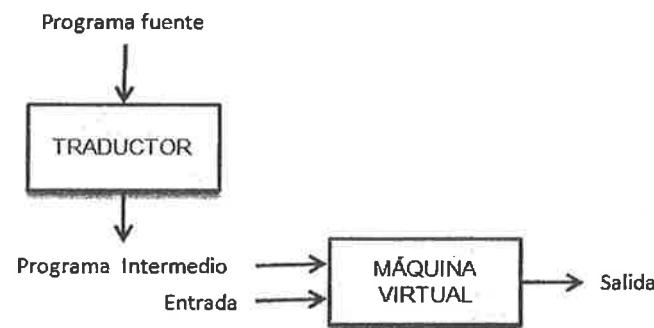


Figura 1.29. Un compilador híbrido.

SEGÚN EL PARADIGMA DE PROGRAMACIÓN

Un paradigma de programación es un enfoque particular para la construcción del software. Define un conjunto de reglas, patrones y estilos de programación que susan los lenguajes de programación. Un lenguaje de programación puede usar más de un paradigma. Dependiendo del problema a resolver un paradigma resultará más apropiado que otro.

Constituyen varias categorías de los lenguajes de programación: imperativos, funcionales (o aplicativos) y lógicos (o declarativos).

Lenguajes imperativos

Los primeros lenguajes imperativos fueron los lenguajes máquina de los ordenadores, en los que las instrucciones eran muy simples, después se utilizaron los lenguajes ensambladores. En los lenguajes imperativos un cálculo consiste en una serie de sentencias que establecen explícitamente cómo se debe manipular la información digital presente en memoria, y/o cómo se debe recoger o enviar información desde/hacia los dispositivos.

La sentencia principal es la asignación. Las estructuras de control permiten establecer el orden de ejecución y cambiar el flujo del programa dependiendo de los resultados de las acciones primitivas.

Ejemplos de lenguajes imperativos son: Basic, Fortran, Algol, Pascal, C, Ada, C++, Java, C#. La mayoría de los lenguajes usados para desarrollo de software comercial son imperativos. Ejemplo de un programa en lenguaje C que calcula el factorial de un número leído por teclado y lo muestra en pantalla:

```
int main() {
    int x;
    int fact = 1;
    scanf("%i", &x);

    for(int i = 2; i <= x; i++) {
        fact = fact * i;
    }
    printf ("Factorial = %i", fact);
}
```

Dentro de esta categoría se engloba la **programación estructurada**, la **programación modular** (paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable), y la **programación orientada a objetos** (paradigma de programación que usa objetos y sus interacciones para diseñar programas).

Lenguajes funcionales

El paradigma funcional está basado en el concepto matemático de función. Los programas escritos en lenguajes funcionales estarán constituidos por un conjunto de definiciones de funciones (entendiendo estas no como subprogramas clásicos de un lenguaje imperativo) junto con los argumentos sobre los que se aplican.

En los lenguajes funcionales:

- No existe la operación de asignación.
- Las variables almacenan definiciones o referencias a expresiones.
- La operación fundamental es la aplicación de una función a una serie de argumentos.
- La computación se realiza mediante la evaluación de expresiones.

Ejemplos de lenguajes funcionales son: Lisp, Scheme, ML, Miranda o Haskell. Se utilizan menos que los anteriores para el desarrollo de software comercial. A continuación, se muestra una función en lenguaje Lisp que calcula el factorial de N de forma recursiva:

```
(defun factorial (N)
  "Calcula el factorial de N."
  (if (= N 1)
    1
    (* N (factorial (- N 1)))))
```

✓ Lenguajes lógicos

En este tipo de lenguajes un cálculo es el proceso de encontrar qué elementos de un dominio cumplen determinada relación definida sobre dicho dominio o el proceso de determinar si un determinado elemento cumple o no dicha relación.

Los programas escritos en estos lenguajes se pueden ver como una base de datos formada por listas de declaraciones lógicas (reglas) que se pueden consultar. La ejecución consistirá en realizar preguntas de forma interactiva.

El lenguaje lógico por excelencia es Prolog, está especialmente indicado para aplicaciones muy específicas como: sistemas expertos, demostración de teoremas, consulta de bases de datos relacionales, procesamiento del lenguaje natural. El siguiente ejemplo muestra varias relaciones *padrede* en Prolog y a continuación muestra varias consultas sobre esas relaciones y el valor devuelto por el sistema:

```
padrede(pedro,maria). % pedro es padre de maria
padrede(juan,luis). % juan es padre de luis
padrede(juan,marta).
padrede(manuel,ana).
```

Las siguientes consultas (símbolos `?`) producen las respuestas especificadas (yes o no):

```
?- padrede(maria,juan).
no.
?- padrede(manuel,ana).
yes.
?- padrede(pedro,marta).
no.
```

Lenguajes de programación estructurados

En el apartado de diseño se nombraron las tres construcciones lógicas que son el fundamento de la programación estructurada: la estructura secuencial, la condicional y la repetitiva. Los lenguajes de programación basados en la programación estructurada son los llamados lenguajes de programación estructurados.

Un programa estructurado, que utiliza las tres construcciones lógicas nombradas anteriormente, resulta fácil de leer, es decir, puede ser leído secuencialmente desde el comienzo hasta el final sin perder la continuidad de lo que hace. El problema de estos programas es que todo el código se concentra en un único bloque, si el programa es demasiado grande o el problema a resolver es complejo, resulta difícil su lectura y manejo.

Actualmente, cuando se habla de programación estructurada, nos solemos referir a la división de un programa en partes más manejables conocidas como módulos. Así un programa estructurado puede estar compuesto por un conjunto de estos módulos, cada uno tendrá una entrada y una salida. La comunicación entre ellos debe estar perfectamente controlada y se debe poder trabajar de forma independiente con cada uno de ellos. La Figura 1.30 muestra un programa que se divide en varios módulos, el módulo raíz (o programa principal) controla el

resto de los módulos; el primero muestra una pantalla de entrada de datos, en el segundo módulo se hace la entrada y comprobación de los datos y en el tercer módulo se insertan los datos en la base de datos.

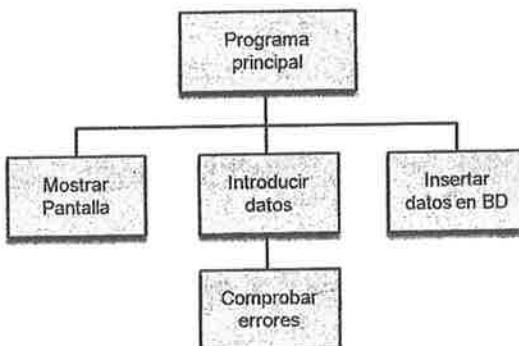


Figura 1.30. Programa dividido en módulos.

A esta evolución de la programación estructurada se le llama **programación modular**. Esta división en módulos aporta una serie de ventajas:

- Al dividir el programa en varios módulos, varios programadores pueden trabajar simultáneamente.
- Los módulos se pueden reutilizar en otras aplicaciones.
- Es menos costoso resolver pequeños problemas de forma aislada que abordar el problema a resolver de forma global.

Ejemplos de lenguajes estructurados son: Pascal, C, Fortran, Modula-2, etc.

Lenguajes de programación orientados a objetos

En la programación orientada a objetos un programa está compuesto por un conjunto de objetos no por un conjunto de instrucciones o un conjunto de módulos, como en la programación estructurada.

Un **objeto** consta de una estructura de datos y de una colección de métodos u operaciones que manipulan esos datos. Los datos definidos dentro de un objeto son sus atributos. Las operaciones definen el comportamiento del objeto y cambian el valor de uno o más atributos. Estas operaciones pueden ser vistas como módulos en un sentido convencional.

Los objetos se comunican unos con otros a través del paso de mensajes. Por ejemplo, un objeto A como parte de la ejecución del *Método2* requiere el proceso asociado con la operación Alta de otro objeto B, entonces el objeto A envía un mensaje de la forma: *B.Alta(datos)*; la operación de Alta completa su ejecución y envía un valor de retorno al objeto A (Figura 1.31).

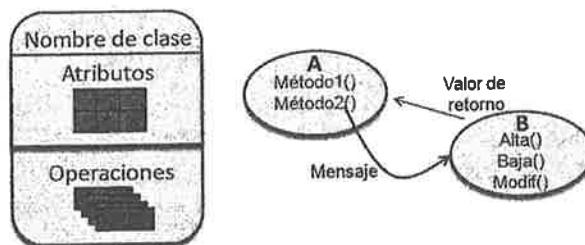


Figura 1.31. Clase y paso de mensajes entre objetos.

Una clase no es más que una plantilla para la creación de objetos. Cuando se crea un objeto (instanciación) se ha de especificar de qué clase es el objeto instanciado, para que el compilador comprenda las características del objeto. Una clase Java de nombre *Persona* con dos atributos, un constructor y cuatro métodos es la siguiente:

```

public class Persona {
    //Atributos
    private String nombre;
    private int edad;

    //Constructor
    public Persona(String nombre, int edad) {
        this.nombre=nombre;
        this.edad=edad;
    }

    //Métodos
    public void setNombre(String nom) {nombre=nom; }
    public void setEdad(int ed) {edad=ed; }
    public String getNombre() {return nombre;} //devuelve nombre
    public int getEdad() {return edad;}         //devuelve edad
}

```

La creación de un objeto de esa clase sería: *Persona persona=new Persona("Manuel", 20);*

Desde el punto de vista de la programación estructurada, una clase se asemejaría a un módulo, los atributos a las variables globales de dicho módulo, y los métodos a las funciones del módulo. Entre las ventajas de la programación orientada a objetos destacamos que facilita la reutilización de código, el trabajo en equipo o el mantenimiento del software. Sin embargo, la principal desventaja es la complejidad para adaptarse a esta programación ya que es menos intuitiva que la estructurada.

Ejemplos de lenguajes orientados a objetos son: C++, Java, Ada, Smalltalk, etc.,

1.7. OBTENCIÓN DE CÓDIGO EJECUTABLE

Durante la etapa de diseño se construyen los componentes software con el suficiente nivel de detalle de tal forma que sirvan como guía en la generación de código fuente en un lenguaje de programación; algunas herramientas utilizadas para ello son los diagramas de flujo o el pseudocódigo.

La generación del código fuente se lleva a cabo en la etapa de codificación. En esta etapa el código pasa por diferentes estados.

1.7.1. Tipos de código

El código de un programa pasa por diferentes estados desde que se escribe hasta que se ejecuta en el ordenador:

- **Código fuente.** Es el código escrito por los programadores utilizando algún editor de texto o alguna herramienta de programación. Se utiliza un lenguaje de programación de alto nivel apropiado para el problema que se trata de resolver. Para escribir el código se parte de los diagramas de flujo o pseudocódigos diseñados en la etapa de diseño. Este código no es directamente ejecutable por el ordenador.

- ✓ □ • **Código objeto.** Es el código resultante de compilar el código fuente. No es directamente ejecutable por el ordenador ni entendido por el ser humano. Es un código o representación intermedia de bajo nivel.
- ✓ □ • **Código ejecutable.** Es el resultado de enlazar el código objeto con una serie de rutinas y bibliotecas, obteniendo así el código que es directamente ejecutable por la máquina.

1.7.2. Compilación

El proceso de compilación de un programa se lleva a cabo mediante dos programas, el compilador y el enlazador, véase Figura 1.32. Si el compilador en el proceso de traducción devuelve algún error (falta algún signo de puntuación, sentencias mal escritas, tipos de datos no compatibles, variables no definidas, etc.) no se generará el programa objeto (o código objeto), será necesario modificar el programa fuente y pasarlo de nuevo por el compilador.

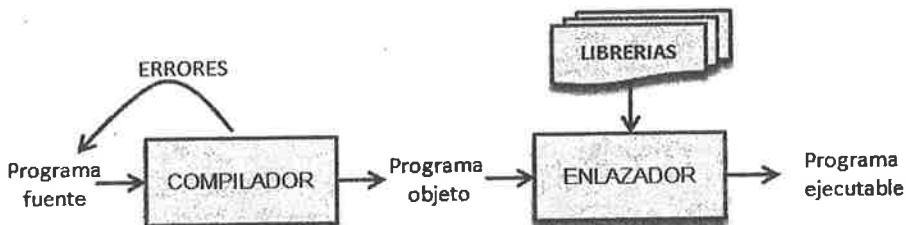


Figura 1.32. Proceso de compilación.

El compilador se compone internamente de varias etapas o fases que realizan distintas operaciones (Figura 1.33):

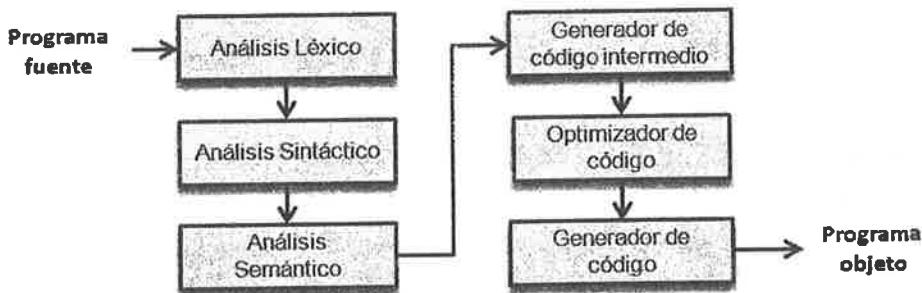


Figura 1.33. Fases de un compilador.

- ✓ □ • **Análisis léxico:** se lee secuencialmente todo el código fuente obteniendo unidades significativas de caracteres denominadas tokens. Por ejemplo la instrucción: suma = x + 2, generaría 5 tokens: suma, =, x, +, 2.
- ✓ □ • **Análisis sintáctico:** recibe el código fuente en forma de tokens y realiza el análisis sintáctico que determina la estructura del programa; es decir, se comprueba si las construcciones de tokens cumplen las reglas de sintaxis definidas en el lenguaje de programación correspondiente. El proceso es semejante al análisis gramatical sobre alguna frase en lenguaje natural.
- ✓ □ • **Análisis semántico:** se comprueban que las declaraciones son correctas, se verifican los tipos de todas las expresiones, si las operaciones se pueden realizar sobre esos tipos, si los arrays tienen el tamaño y tipo adecuados, etc.

- U ▪ **Generación de código intermedio:** después del análisis se genera una representación intermedia similar al código máquina con el fin de facilitar la tarea de traducir al código objeto.
- Q ▪ **Optimización de código:** trata de mejorar el código intermedio generado en la fase anterior, de tal forma que el código resultante sea más fácil y rápido de interpretar por la máquina.
- U ▪ **Generación de código:** genera el código objeto de nuestro programa.

El programa enlazador inserta en el código objeto las funciones de librería necesarias para producir el programa ejecutable. Por ejemplo, en un programa escrito en C si el fichero fuente hace referencia a funciones de una biblioteca o a funciones que están definidas en otros ficheros fuentes, entonces el ligador combina estas funciones con el programa principal para crear un fichero ejecutable.

1.8. MÁQUINAS VIRTUALES

Una máquina virtual es una aplicación software de una máquina (por ejemplo, un ordenador) que ejecuta los programas como si fuese una máquina real. Se pueden clasificar en dos categorías:

- U ▪ **Máquinas virtuales de sistema** (en inglés *System Virtual Machine*). Permiten ejecutar en la misma máquina física varias máquinas virtuales cada una con un sistema operativo, de esta manera pueden coexistir diferentes sistemas operativos sobre una misma máquina. Un ejemplo de este software es *VMware Workstation* y *Virtual Box*; que se pueden utilizar para evaluar y probar nuevos sistemas operativos, para ejecutar aplicaciones sobre distintos sistemas operativos, etc.

Las máquinas virtuales agregan gran complejidad al sistema en tiempo de ejecución y pueden hacer que al ejecutar algún proceso el ordenador vaya lento; por lo tanto hay que tener en cuenta que la máquina donde se ejecute la máquina virtual debe tener una capacidad notable.

- U ▪ **Máquinas virtuales de proceso** (en inglés *Process Virtual Machine*). Una máquina virtual de proceso, a veces llamada "máquina virtual de aplicación", se ejecuta como un proceso normal dentro de un sistema operativo y soporta un solo proceso. La máquina se inicia automáticamente cuando se lanza el proceso que se desea ejecutar y se detiene cuando finaliza. Su objetivo es el de proporcionar un entorno de ejecución independiente de la plataforma de hardware y del sistema operativo, que oculte los detalles de la plataforma subyacente y permita que un programa se ejecute siempre de la misma forma sobre cualquier plataforma. El ejemplo más conocido actualmente de este tipo de máquina virtual es la máquina virtual de Java⁸.

1.8.1. La máquina virtual de Java

Los programas compilados en lenguaje Java se pueden ejecutar en cualquier plataforma: es decir, pueden ejecutarse en entornos UNIX, Mac, Windows, Solaris, etc. Esto es debido a que el código generado por el compilador no lo ejecuta el procesador del ordenador, sino que lo ejecuta la *Máquina Virtual de Java (JVM, Java Virtual Machine)*. El proceso de compilación y ejecución de un programa Java se muestra en la Figura 1.34.

⁸ Fuente: http://es.wikipedia.org/wiki/Máquina_virtual.



Figura 1.34. Compilación y ejecución de un programa Java.

El código fuente del programa está escrito en ficheros de texto plano que tienen la extensión *.java*. La compilación del fichero, mediante el compilador Java *javac*, genera un fichero (o varios) con la extensión *.class* (siempre y cuando no haya errores). Un fichero *.class* contiene código en un lenguaje intermedio cercano al lenguaje máquina pero independiente del ordenador y el sistema operativo en que se ejecuta, este código se llama *bytecode*. La *Máquina Virtual de Java* toma y traduce el *bytecode* en código binario para el procesador que se utiliza para ejecutar el programa. Como está disponible en muchos sistemas operativos diferentes, los ficheros *.class* pueden ser ejecutados en distintas plataformas: Microsoft Windows, Solaris, Linux o Mac OS, Figura 1.35.

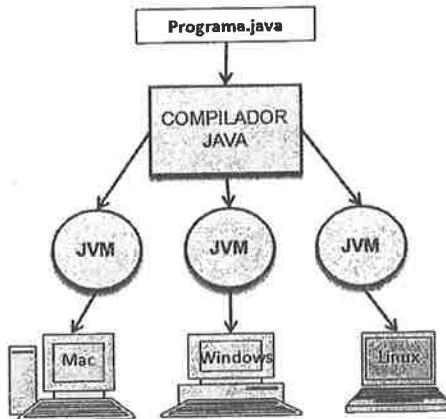


Figura 1.35. Ejecución de un programa Java en distintas plataformas.

Las tareas principales de la *Máquina Virtual Java* son las siguientes:

- Reservar espacio en memoria para los objetos creados y liberar la memoria no utilizada.
- Comunicarse con el sistema huésped (sistema donde se ejecuta la aplicación) para ciertas funciones, como controlar el acceso a dispositivos hardware.
- Vigilar el cumplimiento de las normas de seguridad de las aplicaciones Java.

La principal desventaja de los lenguajes basados en máquina virtual es que son más lentos que los lenguajes completamente compilados, debido a la sobrecarga que genera tener una capa de software intermedia entre la aplicación y el hardware del ordenador, sin embargo, esta desventaja no es demasiado crítica.

Para instalar el entorno de ejecución de la *Máquina Virtual Java* hemos de acceder a la web de Oracle <http://www.oracle.com/technetwork/java/javase/downloads/index.html>, pulsar en *Download* donde aparece **JRE** (*Java Runtime Environment*) y descargar el fichero que nos interese según el sistema operativo del ordenador donde vayamos a instalarlo. Por ejemplo para instalarlo en una máquina con sistema operativo Windows 7 de 64 bits, descargamos el fichero:

jre-7u51-windows-x64.exe (Figura 1.36). El proceso de instalación es sencillo, tan solo hay que ir siguiendo los pasos. Puede que debamos reiniciar el equipo para completar la instalación.

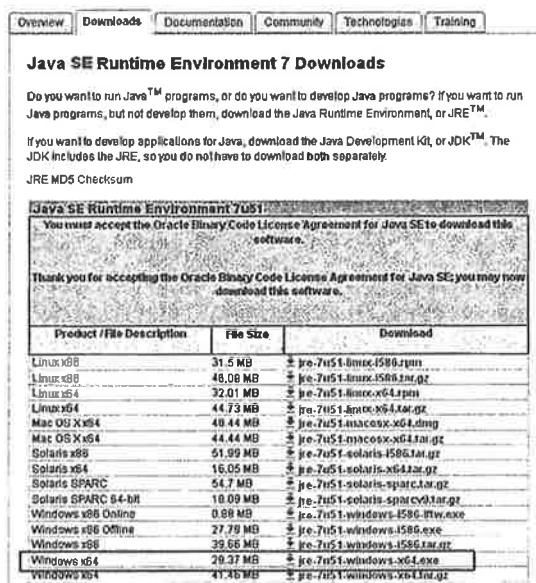


Figura 1.36. Descarga de la máquina virtual de Java.

1.9. HERRAMIENTAS UTILIZADAS EN PROGRAMACIÓN

Para llevar a cabo la codificación y prueba de los programas se suelen utilizar entornos de programación. Estos entornos nos permiten realizar diferentes tareas:

- Crear, editar y modificar el código fuente del programa.
- Compilar, montar y ejecutar el programa.
- Examinar el código fuente.
- Ejecutar el programa en modo depuración.
- Realizar pruebas del programa de forma automática.
- Generar documentación.
- Gestionar los cambios que se van haciendo en el programa (control de versiones).
- Etc.

A estos entornos de programación se les suele llamar entornos de desarrollo integrado o **IDE** (*Integrated Development Environment*). Los IDEs están diseñados para maximizar la productividad del programador. Un IDE es un programa informático formado por un conjunto de herramientas de programación que facilitan las tareas de creación, modificación, compilación, implementación y depuración de software.

La mayoría de los IDEs actuales proporcionan un entorno de trabajo visual formado por ventanas, barras de menús, barras de herramientas, paneles laterales para presentar la estructura en árbol de los proyectos o del código del programa que estamos editando, pestañas adicionales con el resultado de aplicar determinadas herramientas o bien que nos permiten tener abiertos varios ficheros del proyecto, asistentes que ayudan al programador a desarrollar parte del proyecto o del código, etc.

Los editores suelen ofrecer facilidades como el resaltado de la sintaxis utilizando diferentes colores y tipos de letra, el emparejamiento de llaves o paréntesis poniendo el cursor sobre uno de ellos, el plegado y desplegado de código, por ejemplo una función se puede reducir a una simple línea de cabecera.

Un mismo IDE pueda funcionar con varios lenguajes de programación, este es el caso de *Eclipse* o *Netbeans* que mediante la instalación de plugins se le puede añadir soporte de lenguajes adicionales. En el siguiente capítulo se estudiarán los entornos de desarrollo.

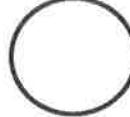
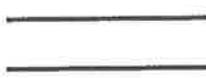
1.10. EJEMPLOS

En este apartado se presenta un ejemplo de una de las técnicas gráficas más utilizadas en el análisis estructurado y que se nombró en el apartado de análisis dentro de las fases del desarrollo de una aplicación (Epígrafe 1.4.1). Se trata de los diagramas de flujo de datos o DFD. En primer lugar, se exponen las reglas básicas para construir este tipo de diagramas y a continuación se muestra un ejemplo elemental sobre un sistema de gestión de almacén.

Después de los DFDs se muestran ejemplos sencillos de elaboración de programas utilizando dos notaciones gráficas: el pseudocódigo y los diagramas de flujo.

1.10.1. Diagramas de flujo de datos. DFD

A la hora de construir un DFD utilizamos los siguientes símbolos (notación *Yourdon*):

Flujos de datos	 
Procesos	
Almacenes de datos	
Entidades externas	

Conviene tener en cuenta una serie de reglas:

- Los elementos del DFD tienen que tener un nombre con significado. En el caso de los flujos de datos que entran y salen de un almacén, no hace falta indicar un nombre, en este caso se asume que entra y sale la totalidad de los datos de un determinado registro.
- Los flujos de datos deben mostrar en qué sentido se mueven los datos.
- Se permiten flujos de datos entre: dos procesos, un proceso y un almacén, un proceso y una entidad externa.

- No se permiten flujos de datos entre: dos almacenes, dos entidades externas, un almacén y una entidad externa.
- Realizar un DFD para cada parte del sistema que haya sido identificada.
- Los almacenes de datos y entidades externas se pueden representar varias veces en el DFD si con ello mejoran la legibilidad.
- En un DFD no puede haber elementos aislados, significaría que por él no pasa ningún flujo de datos, por tanto su presencia es inútil.

Normalmente para representar un sistema grande se utilizan varios DFDs siguiendo una estructura jerárquica. En el nivel más alto de la jerarquía se suele representar un único proceso, identificado con un 0, que representa el sistema completo. Se representa el proceso, los flujos de entrada y salida de datos y las entidades externas, nunca los almacenes. A este diagrama se le llama **diagrama de contexto** o **DFD de nivel 0**. Véase Figura 1.37.

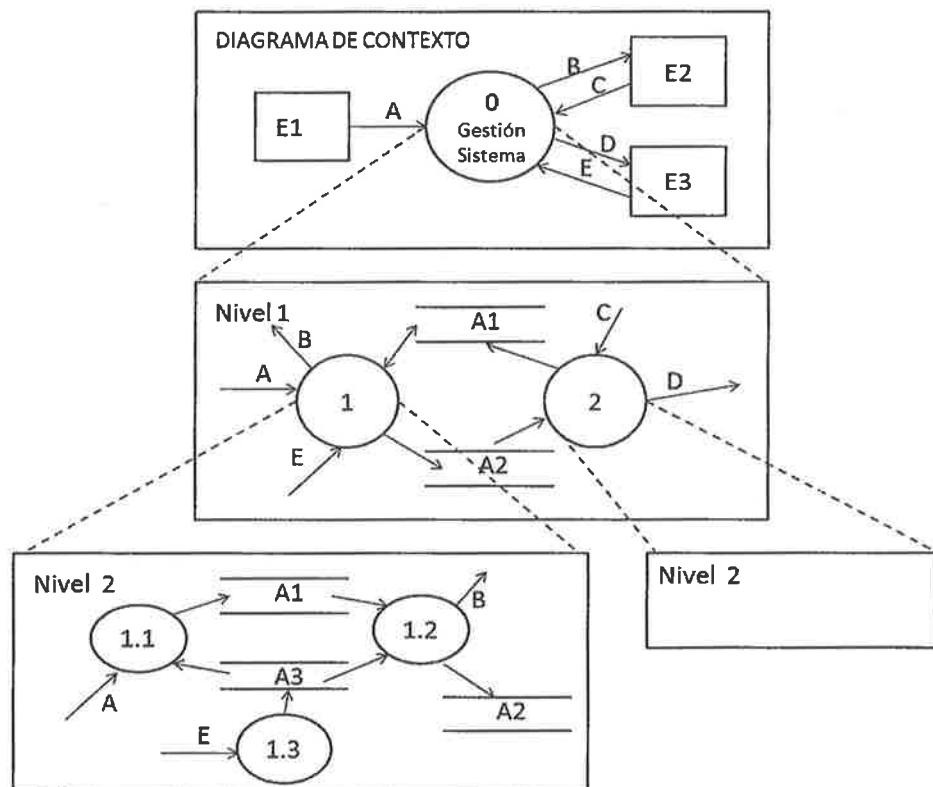


Figura 1.37. Descomposición en niveles de un DFD.

A continuación se descompone el proceso identificado con un 0 en otro DFD en el que se representan las funciones principales del sistema, el diagrama generado se llama **DFD de nivel 1**; los procesos que aparecen en este DFD se enumeran de 1 en adelante. En este DFD se pueden observar los flujos de entrada y salida procedentes de las entidades externas y nombrados como A, B, C, D y E. En este diagrama y los siguientes no se representan las entidades externas, aunque a veces es conveniente que se incluyan para ayudar a la comprensión de los mismos.

Seguidamente se descompone cada uno de los procesos en nuevos procesos que representan funciones más simples (explosión de cada proceso en otro DFD). La descomposición por niveles

permite analizar el sistema desde el ámbito general al detalle, pasando por sucesivos niveles intermedios (se sigue una filosofía *top-down*). En la Figura 1.37 se observa la descomposición del proceso numerado como 1 en tres subprocessos: 1.1, 1.2 y 1.3. Esta descomposición obtiene el **DFD de Nivel 2**. Al pasar de un nivel superior a otro inferior (por ejemplo del nivel 1 al nivel 2) hay que verificar que la información que entra y sale de un proceso de nivel superior sea consistente con la información que entra y sale del DFD en que este proceso se descompone; es decir en el DFD de nivel 2 de un proceso, se deben mostrar los flujos de entrada y salida del proceso en el nivel 1.

En el nivel 2 de la Figura 1.37, se muestran los flujos de entrada y salida del proceso numerado como 1 en el nivel 1: A y E de entrada, B de salida, flujo de entrada y salida sobre el almacén A1 y flujo de salida sobre el almacén A2. En este DFD se define el almacén A3.

Se recomienda utilizar un máximo de cuatro niveles de descomposición de diagramas, aunque, dependiendo del problema a resolver, puede que se necesiten menos. Los niveles de descomposición recomendados son:

- Nivel 0: Diagrama de contexto
- Nivel 1: Subsistemas
- Nivel 2: Funciones de cada subsistema
- Nivel 3: Subfunciones asociadas
- Nivel 4: Procesos necesarios para el tratamiento de cada subfunción

Supongamos un ejemplo muy sencillo, por ejemplo, nos han pedido realizar un sistema de gestión de un almacén que almacena un único tipo de productos. Este almacén vende productos a clientes y a su vez el almacén compra productos al proveedor con el fin de no quedarse sin existencias. Este sistema debe controlar la gestión de clientes y la de proveedores. Las operaciones a realizar en cada una son:

- **Gestión de clientes:** el sistema se encarga de recibir y servir los pedidos de compra de productos que solicita el cliente. Cuando un cliente solicita un producto rellena un pedido de compra con la siguiente información: número de pedido, producto, código de cliente, fecha de pedido y el número de unidades pedidas. El sistema registra el pedido, comprueba si hay unidades disponibles y sirve el pedido; si no hay suficientes unidades se sirven las que haya en el almacén. La respuesta que recibe el cliente a esta acción es una factura con los datos: número de factura, fecha, código de cliente, producto, unidades del producto, precio e importe total del pedido. El sistema deberá actualizar las existencias del producto restando las que se han servido al cliente.
- **Gestión de proveedores:** el sistema se encarga de enviar pedidos al proveedor cuando las existencias del producto están por debajo de una cantidad; y de recibir los pedidos cuando llegan al almacén actualizando las existencias del producto con la cantidad que entra. Si en el almacén no se dispone de una mínima cantidad del producto, el sistema realiza un pedido de abastecimiento al proveedor. Los datos para realizar el pedido son: número de pedido, producto, fecha de pedido, código de proveedor y número de unidades que se piden. La operación de recepción de los pedidos se cumple mediante un albarán, la información del mismo es la siguiente: número de albarán, fecha, código de proveedor, producto, número de unidades suministradas y coste total del pedido. Entonces se debe actualizar el stock del producto sumándole las unidades recibidas.

El DFD de nivel 0 está formado por un proceso, el proceso de *Sistema de Gestión de Almacén*, que representa el funcionamiento general del sistema; dos entidades externas CLIENTE y PROVEEDOR, y los flujos de entrada (pedidos de compra de los clientes y albaranes procedentes de los proveedores) y salida (facturas a clientes y pedidos de abastecimiento a los proveedores) al sistema, véase Figura 1.38. El proceso principal se numera con un cero.



Figura 1.38. DFD de nivel 0.

En el DFD de nivel 1 se representan los dos procesos fundamentales del sistema: la gestión de clientes y la de proveedores. Se numeran con un uno y un dos. En ambos procesos se representan los flujos de datos de entrada y de salida. La información de los productos se representa en el almacén PRODUCTOS, se muestran flujos que entran y salen del almacén, para consultar y actualizar información del producto, véase Figura 1.39.

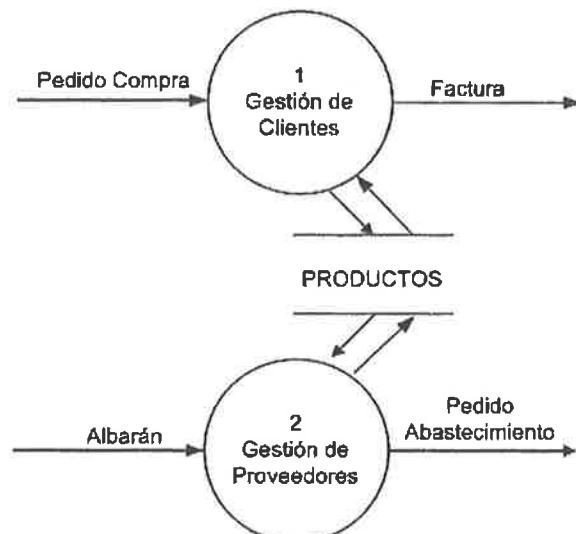


Figura 1.39. DFD de nivel 1.

La gestión de clientes se puede dividir en dos subprocessos: registrar pedido del cliente (numerado como 1.1) y servir pedido al cliente (numerado como 1.2). Estos forman el DFD de nivel 2, véase Figura 1.40.



Figura 1.40. DFD de nivel 2, Gestión de Clientes.

La entrada del primer proceso será el pedido de compra del cliente, y la salida, el registro de los pedidos en el almacén PEDIDOSCLIENTE. El proceso *Servir Pedido* toma la entrada de los pedidos registrados y la salida será la factura al cliente, este proceso se divide en otros dos procesos que son: comprobar disponibilidad de las existencias del producto (1.2.1) y generar la factura (1.2.2), estos forman el DFD de nivel 3, véase Figura 1.41.

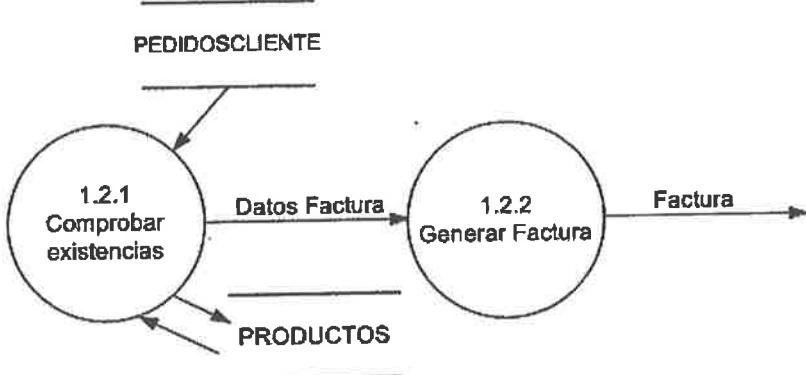


Figura 1.41. DFD de nivel 3, Gestión de Clientes.

Al comprobar la disponibilidad de existencias del producto se accede al almacén de PRODUCTOS, una vez comprobada se deben actualizar las existencias restando las unidades pedidas por el cliente. La salida de este proceso serán los datos para generar la factura y servirán de entrada al proceso de generar factura.

La gestión de proveedores se puede dividir en dos subprocessos: generar pedido a proveedor (numerado como 2.1) y recibir pedido del proveedor (numerado como 2.2). Estos forman el DFD de nivel 2, véase Figura 1.42.

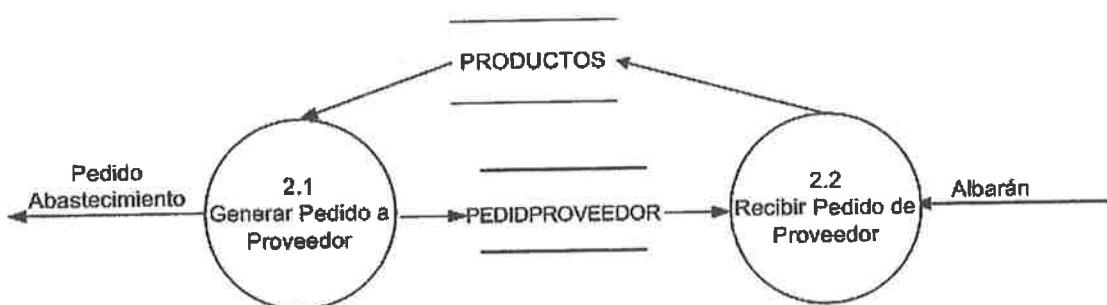


Figura 1.42. DFD de nivel 2, Gestión de Proveedores.

El proceso de generar pedido toma como entrada el almacén de PRODUCTOS, la salida será el registro de los pedidos en el almacén de PEDIDPROVEEDOR y la generación del pedido de abastecimiento. El proceso de recibir pedido toma como entrada los albaranes recibidos y los comprueba con los pedidos realizados, la salida del proceso será la actualización del stock de los productos cuyas unidades han sido abastecidas.

El proceso de generar pedido a proveedor lo dividimos en dos subprocessos: *Calcular unidades a pedir* (2.1.1) y *Crear Pedido* (2.1.2), Figura 1.43. El primer proceso parte de los productos del almacén, se comprueba si las existencias no superan un mínimo y se calcula las unidades a pedir. A partir de esas unidades se creará el pedido de abastecimiento.

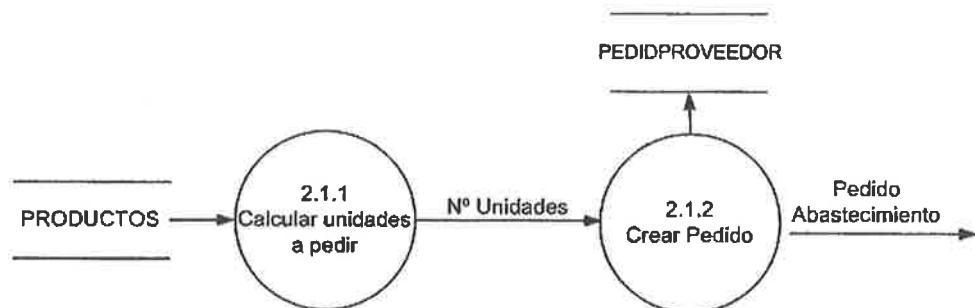


Figura 1.43. DFD de nivel 3, Generar Pedido a Proveedor.

El proceso de recibir pedido de proveedor lo dividimos en dos subprocessos: *Comprobar Albarán* (2.2.1) y *Actualizar stock* (2.2.2). El primero recibe los datos del albarán que se comprobarán con el pedido registrado en el almacén. Después de la comprobación se realizará el proceso de actualización del stock de los productos cuyas unidades se han recibido, Figura 1.44.

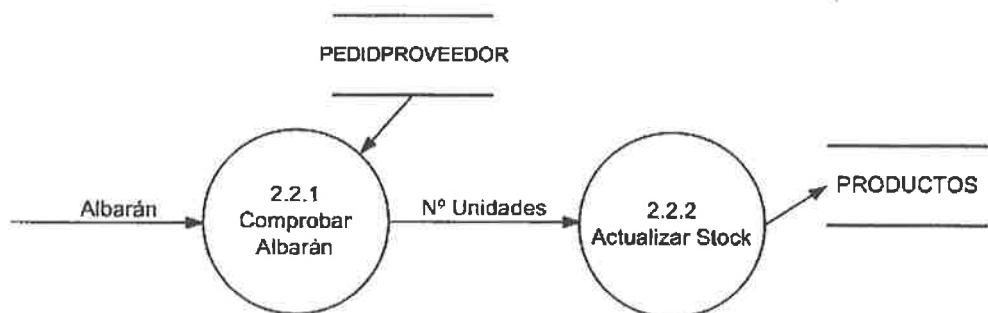


Figura 1.44. DFD de nivel 3, Recibir Pedido de Proveedor.

1.10.2. Pseudocódigo y diagramas de flujo

En el siguiente apartado se muestran algunos ejemplos de pseudocódigos y el correspondiente diagrama de flujo. Además de los símbolos vistos anteriormente utilizaremos los siguientes símbolos:

Símbolo <i>Terminador</i> , representa el inicio y final de un programa.	
Símbolo de <i>Entrada/Salida</i> , representa entrada y salida de datos.	
<i>Pantalla</i> , se utiliza para representar salida por pantalla.	

<i>Impresora</i> , se utiliza como símbolo de entrada (documento de entrada) y salida (impresora).	
<i>Teclado</i> , se utiliza como símbolo de entrada por teclado.	
<i>Conektor</i> , se utiliza para unir una parte del diagrama con otra.	
Llamada a subrutina o procedimiento.	
Disco magnético, representa una función de entrada/salida para soporte en un disco magnético.	

Ejemplo 1. Programa que lee dos números y muestra la suma en pantalla.

Se utilizan estructuras secuenciales básicas.

Pseudocódigo	Diagrama de flujo
<p>Inicio</p> <p>Leer Nume1, Nume2</p> <p>Suma=Nume1+Nume2</p> <p>Visualizar "La suma es:" Suma</p> <p>Fin</p>	<pre> graph TD A([INICIO]) --> B[/Nume1, Nume2/] B --> C[Suma = Nume1 + Nume2] C --> D(["'La suma es:' Suma"]) D --> E([FIN]) </pre>

Ejemplo 2. Programa que lee dos números y muestra el mayor en pantalla, si son iguales deberá mostrar un mensaje indicándolo. Se utiliza la estructura condicional para comprobar los valores: Si <condición> Entonces <Instrucciones> Si no <Instrucciones> Fin si.

Además se muestra una estructura condicional dentro de otra.

Pseudocódigo	Diagrama de flujo
<p>Inicio</p> <p>Leer A, B</p> <p>Si A > B Entonces</p> <p> Visualizar "El mayor es:" A</p> <p>Si no</p> <p> Si A = B Entonces</p> <p> Visualizar "Son iguales"</p> <p> Si no</p> <p> Visualizar "El mayor es:" B</p> <p> Fin si</p> <p>Fin si</p> <p>Fin</p>	<pre> graph TD INICIO([INICIO]) --> PARALELOGRAMO[/A, B/] PARALELOGRAMO --> DECISION{A > B} DECISION -- NO --> DECISION_A_B{A = B} DECISION_A_B -- NO --> OVAL_B([El mayor es: B]) DECISION_A_B -- SI --> OVAL_IIGUALES([Son Iguales]) OVAL_B --> FIN([FIN]) OVAL_IIGUALES --> FIN DECISION -- SI --> OVAL_A([El mayor es: A]) OVAL_A --> FIN </pre>

Ejemplo 3. Programa que lee dos números en un proceso repetitivo. Este proceso terminará cuando los números leídos sean iguales.

Se utiliza la estructura repetitiva Repetir <instrucciones> Hasta que <condición>.

Pseudocódigo	Diagrama de flujo
<p>Inicio</p> <p>Repetir</p> <p> Visualizar "Escribe dos números"</p> <p> Leer A, B</p> <p> Hasta A = B</p> <p>Fin</p>	<pre> graph TD INICIO([INICIO]) --> PARALELOGRAMO[/Escribe dos números/] PARALELOGRAMO --> DECISION{A = B} DECISION -- NO --> PARALELOGRAMO DECISION -- SI --> FIN([FIN]) </pre>

Ejemplo 4. Programa que lee 10 números en un proceso repetitivo y muestra la suma.

Es necesario declarar variables para contar los números que se van leyendo y para ir guardando la suma. Se deben inicializar a 0.

Se utiliza la estructura repetitiva Repetir <instrucciones> Hasta que <condición>.

Pseudocódigo	Diagrama de flujo
<p>Inicio</p> <p>Declarar Cuenta=0</p> <p>Declarar Suma =0</p> <p>Repetir</p> <p> Visualizar "Escribe un número"</p> <p> Leer A</p> <p> Cuenta = Cuenta +1</p> <p> Suma = Suma + A</p> <p>Hasta Cuenta = 10</p> <p> Visualizar "La suma es:" Suma</p> <p>Fin</p>	<pre> graph TD INICIO([INICIO]) --> ASIGNACION[Cuenta = 0 Suma = 0] ASIGNACION --> PREGUNTA("Escribe un número") PREGUNTA --> LEER[/A/] LEER --> ACTUALIZACION[Cuenta = Cuenta + 1 Suma = Suma + A] ACTUALIZACION --> DECISION{Cuenta = 10} DECISION -- NO --> PREGUNTA DECISION -- SI --> SALIDA("La suma es: Suma") SALIDA --> FIN([FIN]) </pre>

Ejemplo 5. Programa que lee registros de un fichero secuencial. Cada registro contiene información de un alumno: Nombre, Curso, Nota.

El programa debe mostrar por cada registro leído el Nombre, el Curso y la Nota.

Al final del proceso de lectura debe mostrar la nota media. Esta se calcula sumando las notas de todos los alumnos y dividiéndola por el número de alumnos que hay.

Se utiliza la estructura repetitiva Mientras <condición> Hacer <instrucciones> Fin mientras.

Se declaran tres variables, una para contar alumnos, otra para sumar notas y una tercera para guardar la nota media.

Pseudocódigo	Diagrama de flujo
<p>Inicio</p> <p>Declarar Cuenta=0</p> <p>Declarar Suma =0</p> <p>Declarar Media</p> <p>Abrir Fichero Notas</p> <p>Leer Registro (Nombre, Curso, Nota)</p> <p>Mientras NO sea Final de Fichero Hacer</p> <ul style="list-style-type: none"> Visualizar Nombre, Curso, Nota Cuenta = Cuenta +1 Suma = Suma + Nota Leer Registro (Nombre, Curso, Nota) <p>Fin mientras</p> <p>Media= Suma/Cuenta</p> <p>Visualizar "Nota media:" Media</p> <p>Cerrar Fichero Notas</p> <p>Fin</p>	<pre> graph TD INICIO([INICIO]) --> Cuenta[Cuenta = 0 Suma = 0] Cuenta --> Abrir[Abrir fichero] Abrir --> Leer{Nombre, Curso, Nota} Leer --> Decision{¿NO es Fin de Fichero?} Decision -- SI --> Nombre([Nombre, Curso, Nota]) Nombre --> CuentaP[Cuenta = Cuenta + 1 Suma = Suma + Nota] CuentaP --> Leer Decision -- NO --> Media[Media = Suma / Cuenta] Media --> NotaMedia("Nota media: Media") NotaMedia --> Cerrar[Cerrar fichero] Cerrar --> FIN([FIN]) </pre>

Ejemplo 6. Programa que lee un número de teclado y muestra a qué día de la semana se corresponde.

Si el número leído es 1 visualizará Lunes, si es 2 visualizará Martes, si es 3 visualizará Miércoles, si es 4 Jueves, si es 5 Viernes, si es 6 Sábado y si es 7 Domingo.

Para cualquier otro valor visualizará *No válido*.

Se utilizará la estructura **Según sea**:

Según sea <variable> Hacer

Caso valor 1: Instrucciones

Caso valor 2: Instrucciones

Otro caso: Instrucciones

Fin según

Pseudocódigo	Diagrama de flujo
<p>Inicio</p> <p>Leer N</p> <p>Según sea N Hacer</p> <p>Caso 1: Visualizar "Lunes"</p> <p>Caso 2: Visualizar "Martes"</p> <p>Caso 3: Visualizar "Miércoles"</p> <p>Caso 4: Visualizar "Jueves"</p> <p>Caso 5: Visualizar "Viernes"</p> <p>Caso 6: Visualizar "Sábado"</p> <p>Caso 7: Visualizar "Domingo"</p> <p>Otro caso: Visualizar "No válido"</p> <p>Fin según</p> <p>Fin</p>	<pre> graph TD INICIO([INICIO]) --> N[/N/] N --> D1{N=1} D1 -- SI --> Lunes([Lunes]) D1 -- NO --> D2{N=2} D2 -- SI --> Martes([Martes]) D2 -- NO --> D3{N=3} D3 -- SI --> Miércoles([Miércoles]) D3 -- NO --> D4{N=4} D4 -- SI --> Jueves([Jueves]) D4 -- NO --> D5{N=5} D5 -- SI --> Viernes([Viernes]) D5 -- NO --> D6{N=6} D6 -- SI --> Sábado([Sábado]) D6 -- NO --> D7{N=7} D7 -- SI --> Domingo([Domingo]) D7 -- NO --> NoVálido([No válido]) NoVálido --> FIN([FIN]) </pre>

COMPRUEBA TU APRENDIZAJE

1. ¿Cuál de las siguientes afirmaciones es correcta?
 - a) En los modelos evolutivos no se necesita conocer todos los requisitos al comienzo.
 - b) Es muy común en el modelo en cascada el uso de prototipos.
 - c) El análisis de riesgos se lleva a cabo en cada incremento del modelo iterativo incremental.
 - d) El modelo en cascada es apropiado cuando se necesita una versión inicial del software a desarrollar.

2. Relaciona:

Modelo en cascada Modelo iterativo incremental Modelo en espiral	Es fácil de comprender. Los clientes necesitan versiones intermedias. No se necesita conocer todos los requisitos al comienzo. Reduce riesgos del proyecto. Los requisitos son estables. Genera mucho trabajo adicional. No se sabe cuándo va a terminar. Los requisitos son estables. El proyecto es similar a uno ya realizado Se acomoda bien a los cambios de requisitos.
--	--

3. ¿Qué se hace en la etapa de análisis del desarrollo de una aplicación? ¿Qué tipos de requisitos se especifican en esta fase? Nombra algunas herramientas para obtener y representar los requisitos. ¿Dónde debe quedar reflejado todo lo realizado en esta fase?

4. ¿Cuál de estas afirmaciones sobre la fase de diseño en el desarrollo de una aplicación es correcta?
 - a) En esta fase se especifica qué hay que hacer.
 - b) En esta fase se especifica cómo hacerlo.
 - c) En esta fase se realiza el proceso de programación.
 - d) En esta fase se realizan las pruebas.

5. ¿Cuáles son los 4 elementos fundamentales que intervienen en el diseño estructurado? Nombra algunas notaciones gráficas utilizadas para representar la fase de diseño.

6. ¿Por qué crees que debe haber normas de codificación y estilo a la hora de llevar a cabo la fase de codificación en el desarrollo de una aplicación?

7. ¿En qué consisten las tareas de verificación y validación del software?

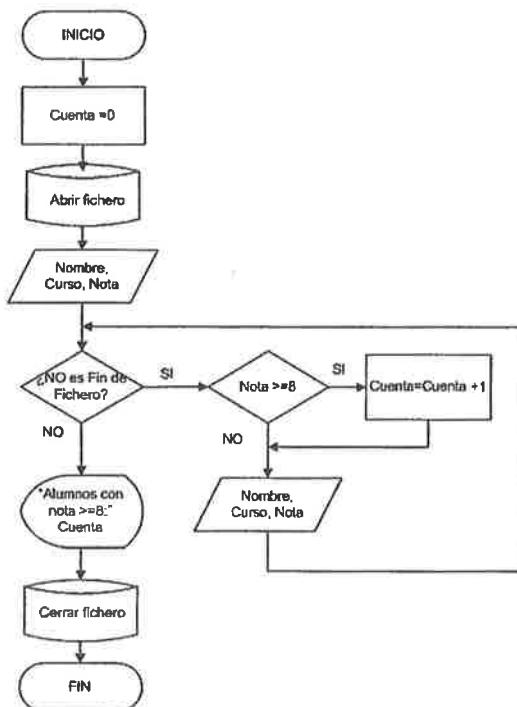
8. ¿Qué técnicas se utilizan para llevar a cabo el diseño de casos de prueba? ¿En qué se consisten?
9. La documentación de un proyecto software se divide en dos clases: la documentación del proceso y la del producto. ¿Qué contiene cada una?
10. ¿Qué información debe contener la documentación del usuario y la del sistema?
11. ¿Qué tareas se llevan a cabo durante la explotación del sistema?
12. ¿Cuántos tipos de mantenimiento del software existen? ¿En qué consisten?
13. ¿Qué tareas se llevan a cabo en la etapa de mantenimiento del software?
14. ¿Qué relación tiene un programa con el hardware del ordenador donde se ejecuta?
15. ¿De qué tres elementos consta un lenguaje de programación?
16. ¿Qué diferencia hay entre un lenguaje de alto nivel y otro de bajo nivel?
17. Diferencias entre un compilador y un intérprete.
18. ¿Qué ventajas aporta la programación modular?
19. ¿Cuáles son los diferentes estados por los que pasa el código de un programa desde que se escribe hasta que se ejecuta por el ordenador?
20. ¿Cuáles son las etapas o fases de un compilador?
21. Cuáles de las siguientes afirmaciones sobre la máquina virtual de Java es cierta:
 - a) Un fichero *.class* contiene código en un lenguaje máquina.
 - b) La máquina virtual de Java toma y traduce el *bytecode* en código binario.
 - c) Los ficheros *.class* sólo pueden ser ejecutados en Microsoft Windows y Linux.
 - d) A la hora de instalar el entorno de ejecución de la máquina virtual Java necesitamos saber en qué sistema operativo se va a instalar.

ACTIVIDADES DE AMPLIACIÓN

1. Modifica el pseudocódigo y el diagrama de flujo de los Ejemplos 3 y 4, de forma que se utilice la estructura *Mientras <condición> Hacer <instrucciones> Fin mientras*.
2. Modifica el pseudocódigo y el diagrama de flujo del Ejemplo 5, de forma que se utilice la estructura *Repetir <instrucciones> Hasta <condición>*.
3. Utilizando como modelo el Ejemplo 6, realiza un pseudocódigo y un diagrama de flujo que lea un número y muestre la nota a la que corresponde. Si el valor es 5 debe mostrar *Suficiente*, si es 6 *Bien*, si es 7 *Notable bajo*, si es 8 *Notable alto*, si es 9 *Sobresaliente*, si es 10 *Matrícula de honor*; para cualquier otro valor debe mostrar *Suspens*.
4. Modifica el pseudocódigo del Ejemplo 6 de manera que se utilice la estructura *Si <condición> Entonces <Instrucciones> Si no <Instrucciones> Fin si*, en lugar de *Según sea*.

5. Utilizando como modelo el Ejemplo 2, realiza un pseudocódigo y un diagrama de flujo que lea un número y muestre el menor de ellos. Si los números son iguales debe mostrar un mensaje indicándolo.

1. Escribe el pseudocódigo para el siguiente diagrama de flujo:



2. Escribe el diagrama de flujo para el siguiente pseudocódigo:

Inicio

Declarar Cuenta=0, Suma =0,

Declarar Positivos = 0, Negativos = 0

Repetir

Visualizar “Escribe un número”

Leer A

Cuenta = Cuenta +1

Si A >= 0 Entonces

Positivos=Positivos +1

Si no

Negativos = Negativos +1

Fin si

Suma = Suma + A

Hasta Cuenta = 10

Visualizar Suma, Positivos, Negativos

Fin.

6. Se desea realizar un sencillo sistema de gestión de notas en el que van a interactuar profesores y alumnos. El sistema permitirá a un alumno consultar la nota de las asignaturas en que está matriculado y a un profesor introducir las notas de los cursos a los que imparte clase. Tanto el profesor como el alumno deben identificarse para entrar al sistema mediante un nombre de usuario y una clave, por lo tanto debe existir un almacén con los usuarios que pueden entrar en el sistema. Si la identificación no es correcta el sistema rechaza al usuario. Los procesos para alumno y profesor son los siguientes:

- Para alumno, 2 procesos: debes comprobar si el usuario está registrado y consultar la nota de la asignatura solicitada, estas notas estarán en un almacén. Cuando el usuario se identifica en el sistema puede introducir el nombre de la asignatura para que este le devuelva la nota. Para nombrar el flujo de datos entre el proceso de comprobación de usuario y de consulta de notas puedes usar: *pantalla de usuario*.

- Para profesor, 3 procesos: debes comprobar si el usuario está registrado, solicitar el curso al que introducirá las notas e introducir las notas del curso solicitado. Para nombrar el flujo de datos entre el proceso de comprobación y de solicitud de curso puedes usar: *pantalla de profesor*, y para el de solicitud de curso e introducción de notas: *pantalla de curso*. Cuando el profesor se identifica envía al sistema el curso del que desea introducir las notas y las notas, estas se irán registrando en un almacén.

Se pide realizar los DFDs de nivel 0, 1 y 2. Define dos almacenes, uno para los usuarios que se identifican en el sistema y otro para las notas.

CAPÍTULO 2

INSTALACIÓN Y USO DE ENTORNOS DE DESARROLLO

CONTENIDOS

Funciones de un entorno de desarrollo

Instalación de un entorno de desarrollo

Configuración y actualización de un entorno de desarrollo

Uso básico de un entorno de desarrollo

Análisis y comparación de entornos de desarrollo libres y comerciales más usuales

OBJETIVOS

Conocer las características de los entornos de desarrollo

Evaluar entornos integrados de desarrollo, analizando, sus características para editar código fuente y generar ejecutables

Instalar y configurar entornos de desarrollo

Utilizar entornos de desarrollo para crear modelos de datos y desarrollo de programas

RESUMEN DEL CAPÍTULO

En este capítulo estudiaremos para qué se utilizan los entornos de desarrollo. Instalaremos dos tipos de entornos de desarrollo: SQLDEVELOPER, orientado al uso de bases de datos y creación de modelos de datos E/R y relacionales, y ECLIPSE más orientado al desarrollo de programas, en este se instalarán plugins para añadir funcionalidades a este entorno y trabajar con otras herramientas.

2.1. INTRODUCCIÓN

IDE son las siglas de Integrated Development Environment, en español *Entorno Integrado de Desarrollo*, es una aplicación informática que está compuesta por un conjunto de herramientas de programación que van a facilitar la tarea al programador y obtener mayor rapidez en el desarrollo de las aplicaciones. Esta herramienta puede estar pensada para utilizarse con un único lenguaje de programación o bien puede dar cabida a varios lenguajes.

En la fase de desarrollo de software intervienen varias herramientas en cada una de las distintas etapas hasta llegar a obtener el resultado final, podremos encontrar IDEs para todas las etapas del desarrollo.

2.2. COMPONENTES DE UN ENTORNO DE DESARROLLO

Un IDE normalmente consiste en un editor de texto donde escribir el código con resaltado de sintaxis y corrector sintáctico, un compilador y/o intérprete, un depurador, un control de versiones, un constructor de interfaz gráfica (GUI), entre otras funcionalidades.

- **Editor de texto:** Es la parte que nos permite escribir el código fuente del programa, ofrece funciones propias de la edición como como copiar, cortar, pegar, o buscar. Además es capaz de reconocer, resaltar y cambiar los colores de las variables, las cadenas de caracteres, las palabras reservadas, las instrucciones, el inicio y fin de los corchetes, de esta manera el código fuente será mucho más visual, cómodo y se podrán reconocer los errores a simple vista.
- **Compilador:** Es el encargado de traducir el código fuente tecleado por el programador y escrito en un lenguaje de alto nivel, a un programa escrito en lenguaje de bajo nivel llamado lenguaje máquina, capaz de ser interpretado y ejecutado por la máquina. El proceso de traducción se conoce como compilación.
- **Intérprete o interpretador:** Los intérpretes se diferencian de los compiladores en que solo realizan la traducción a medida que se va ejecutando la instrucción. Normalmente no guardan el resultado de dicha traducción. Son más lentos que los compilados debido a la necesidad de traducir el programa mientras se ejecuta, pero a cambio son más flexibles como entornos de programación y depuración y permiten ofrecer al programa interpretado un entorno no dependiente de la máquina donde se ejecuta el intérprete, sino del propio intérprete (lo que se conoce comúnmente como máquina virtual).
- **Depurador (Debugger):** Es el encargado de depurar y limpiar los errores en el código fuente de un programa informático. El depurador permite examinar paso a paso, instrucción a instrucción, la ejecución de un programa y examinar las distintas situaciones y cambios que se produzcan en las variables del programa o en los registros del procesador. El depurador va a permitir detener el programa en cualquier punto de ruptura para examinar la ejecución.
- **Constructor de interfaz gráfica:** Esta herramienta de programación simplifica la creación de interfaces gráficas de usuario permitiendo al diseñador colocar los controles (botones, listas, menús, y demás elementos) utilizando un editor WYSIWYG de arrastrar y soltar. Algunos IDE incorporan estas herramientas con el plugin correspondiente, es el caso del Eclipse.

- **Control de versiones:** Estas herramientas permiten controlar los cambios que se realizan sobre las aplicaciones, de esta manera se obtendrán revisiones y versiones de las aplicaciones en un momento dado de su desarrollo.

2.3. INSTALACIÓN DE UN ENTORNO DE DESARROLLO

Existen en el mercado multitud de entornos de desarrollo, en esta unidad se ha optado por instalar dos entornos de desarrollo, un entorno orientado al trabajo con bases de datos para crear modelos de datos (SQLDEVELOPER con Data Modeler, herramienta de Oracle para trabajar con SQL, PL/SQL y crear modelos de datos), y el otro orientado al desarrollo de programas en este caso ECLIPSE. Son dos aplicaciones que se pueden descargar gratis desde los sitios webs de Oracle <http://www.oracle.com/us/downloads/index.html>, en este caso hay que registrarse y aceptar las condiciones; y Eclipse <http://www.eclipse.org/>.

Para el uso de estas herramientas se necesita instalar también Java, aconsejable instalar el JDK 7 o superior para el software que se va a utilizar.

Se ha considerado el uso de estas herramientas pues son de las que tienen más éxito en el mercado, no se necesitan licencias y además dado el perfil del ciclo son herramientas que las utilizarán en el estudio de los módulos del ciclo y las seguirán utilizando una vez acaben sus estudios.

2.3.1. Instalación de Eclipse

Para instalar Eclipse basta con descargarse el archivo comprimido y descomprimirlo en el disco duro. Hay multitud de versiones de Eclipse, a nosotros nos interesa la versión estándar que se llama **Standard**, suficiente para lo que vamos a estudiar. En esta versión se irán añadiendo plugins para ir aumentando las funcionalidades.

La versión a utilizar en esta unidad es *Eclipse Kepler Standard*, versiones para 32 y 64 bits, antes de instalar comprueba desde las propiedades de tu equipo o desde el *Panel de control/Sistema*, la versión del sistema operativo de tu ordenador para ver si trabaja a 32 o 64 bits.

Una vez descomprimido se lanza el ejecutable *eclipse.exe*, aparecerá un cuadro de diálogo en el que pedirá seleccionar el *workspace* de trabajo (véase la Figura 2.1), Eclipse almacena los proyectos en espacios de trabajo llamados *Workspaces*. Se puede crear en ese momento la carpeta para el workspace o seleccionar una ya creada en el equipo.

Antes de entrar aparecerá una ventana de bienvenida y una serie de opciones para ver tutoriales, o para ver lo nuevo, o para ver los ejemplos. En nuestro caso seleccionamos *Workbench* para entrar en la ventana principal.

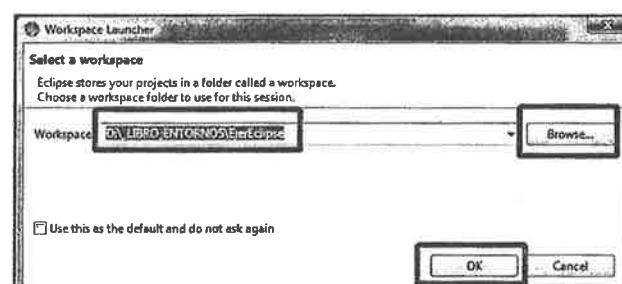


Figura 2.1. Selección del workspace de trabajo.

A continuación aparece la ventana principal de Eclipse. Para movernos por la herramienta lo primero que vamos a hacer es cargar un proyecto ya realizado. Para ello hay que copiar el proyecto *FicheroAleatorioVentana* que se encuentra en la carpeta de recursos del capítulo, al *Workspace* de trabajo. Y dentro de Eclipse abrimos el menú *File/Import*. En el siguiente diálogo se despliega la carpeta *General* y se selecciona *Existing Projects into Workspace* (véase la Figura 2.2). En la ventana que aparece a continuación se selecciona la carpeta de trabajo, se marca el proyecto, y se pulsa el botón *Finish*, véase la Figura 2.3.

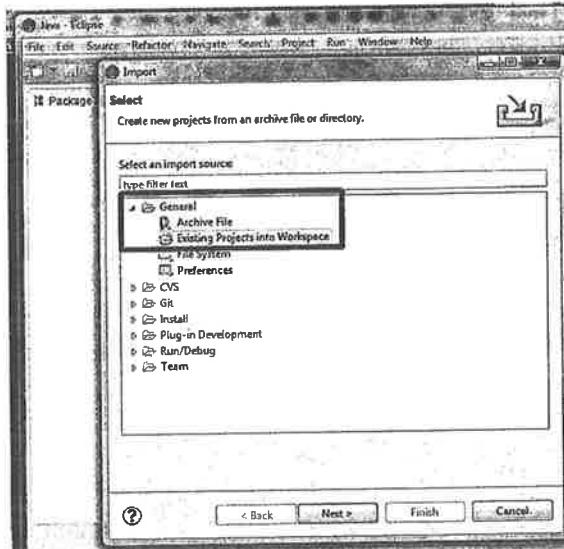


Figura 2.2. Importar un proyecto existente.

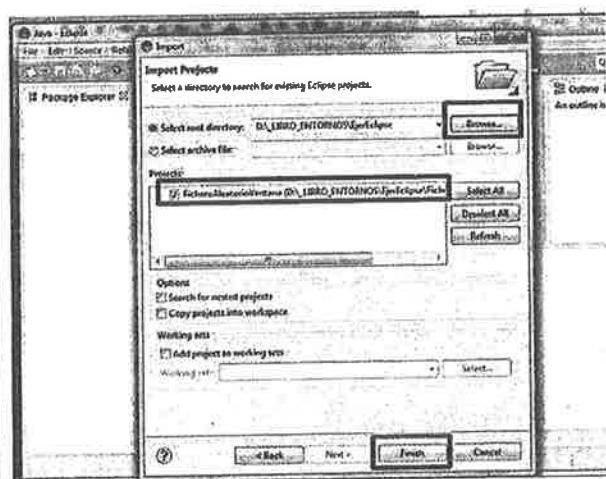


Figura 2.3. Selección del proyecto a importar.

En la ventana principal de trabajo de Eclipse (Figura 2.4) podremos distinguir los siguientes bloques:

- *Pakage Explorer* o la zona de proyectos, desde aquí se podrá navegar por todo el workspace, por los proyectos y los elementos que los componen, carpetas, archivos, librerías, etc.
- *Zona de edición*, desde aquí se escribirá el código de los programas. Observa que el texto aparece resaltado para identificar la sintaxis y palabras del lenguaje. Si se escribe una instrucción errónea el editor marcará la línea y ofrecerá posibles soluciones. Además al escribir código muestra propuestas de plantilla, muy útil para simplificar la escritura, por ejemplo de los métodos de las clases.
- *Outline*, en esta zona se muestra el esquema de la clase que se edita en ese momento, permite acceder de forma más rápida a sus métodos y atributos.
- *Consola Java*, en esta zona se muestra el resultado de la ejecución de los programas, los mensajes de salida por consola, y los errores de ejecución.

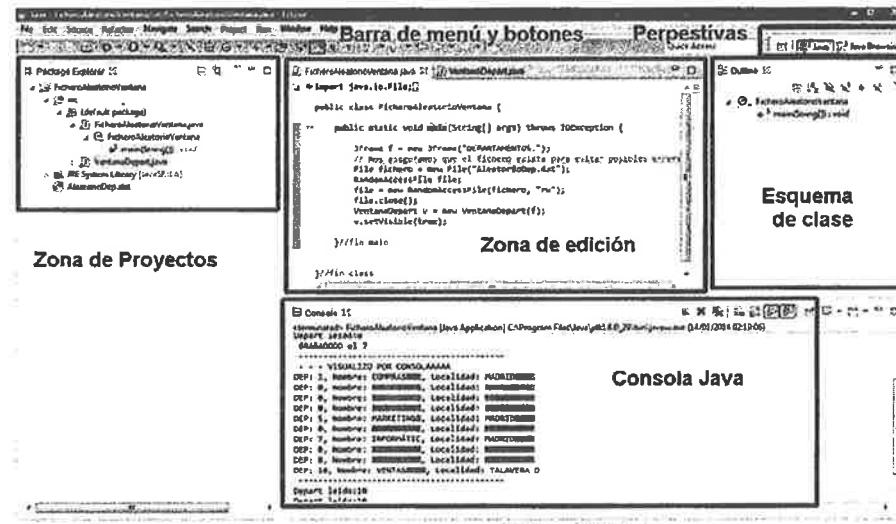


Figura 2.4. Ventana de trabajo de Eclipse.

La visualización de la ventana principal puede cambiar eligiendo una perspectiva diferente y dependiendo del tipo de desarrollo que se vaya a realizar y de los plugins que se tengan instalados (véase la Figura 2.5). También se podrán mostrar otras vistas dentro del Eclipse, desde el menú *Windows>Show View* (véase la Figura 2.6). Para ejecutar el proyecto se pulsa el botón *Run* de la barra de botones.

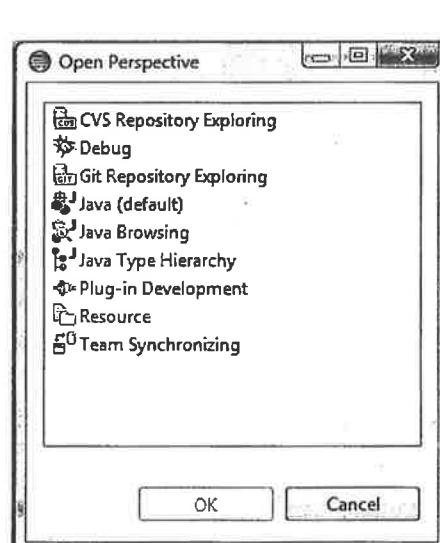


Figura 2.5. Perspectivas de desarrollo de Eclipse.

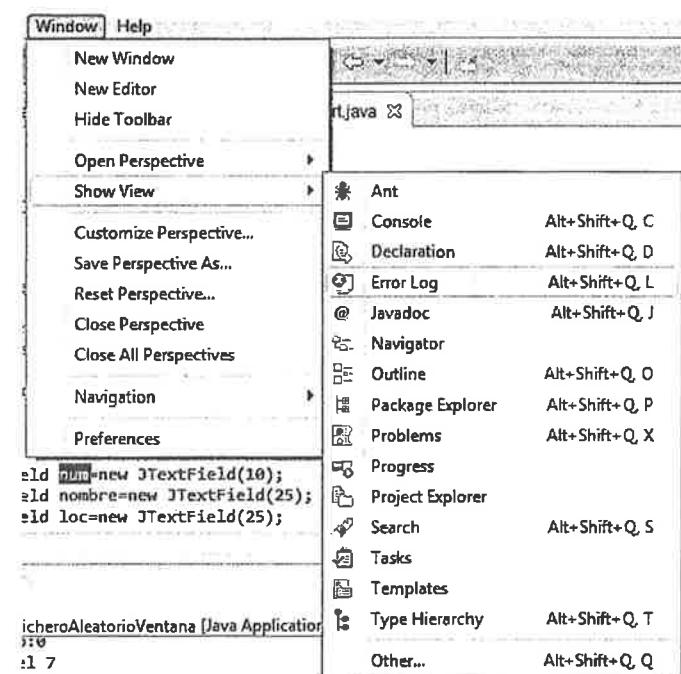


Figura 2.6. Vistas de Eclipse.

2.3.2. Instalación de plugins

Un *plugin* es un programa o una aplicación informática que añade funcionalidades específicas a otro programa que hace de programa principal. Su nombre procede del inglés (plug-in significa “enchufable”) y su presencia es muy habitual en los navegadores web, en reproductores de música, en sistemas de gestión de contenidos y en herramientas de desarrollo como Eclipse.

Los plugins no son parches ni actualizaciones, sino propiedades añadidas a programas originales, aparecidas por primera vez a mediados de los años 70 y conocidas también como complementos, extensiones y addons (del inglés add-on, “agregado”). Más información en <http://www.saberia.com/2010/01/que-es-un-plugin/>.

2.3.2.1. Plugin WindowBuilder

Eclipse no dispone de una interfaz gráfica de usuario para crear aplicaciones con ventanas, para ello es necesario instalar plugins. Los plugins aumentan las funcionalidades del Eclipse. Uno de los plugins más populares para esto es *WindowBuilder*, desarrollado por Google, que permite desarrollar de forma rápida y cómoda la GUI (interfaz gráfica de usuario) de las aplicaciones Java.

Para instalar *WindowBuilder* es necesario saber la versión de Eclipse que se tiene instalada y así saber qué versión del plugin se necesita. Para saber la versión de Eclipse basta con abrir el menú **Help/About Eclipse**. Una vez que se sabe la versión de Eclipse, el plugin se descarga de: <http://www.eclipse.org/windowbuilder/download.php>.

Desde esa página se puede acceder al Link de descarga del plugin y descargarlo en un archivo comprimido. Para instalarlo se abre el menú **Help/Install New Software**. Y en la ventana que aparece pulsamos el botón *Add* para añadir el fichero, se marcan los paquetes a instalar y se pulsa el botón *Next* (Véase la Figura 2.7)

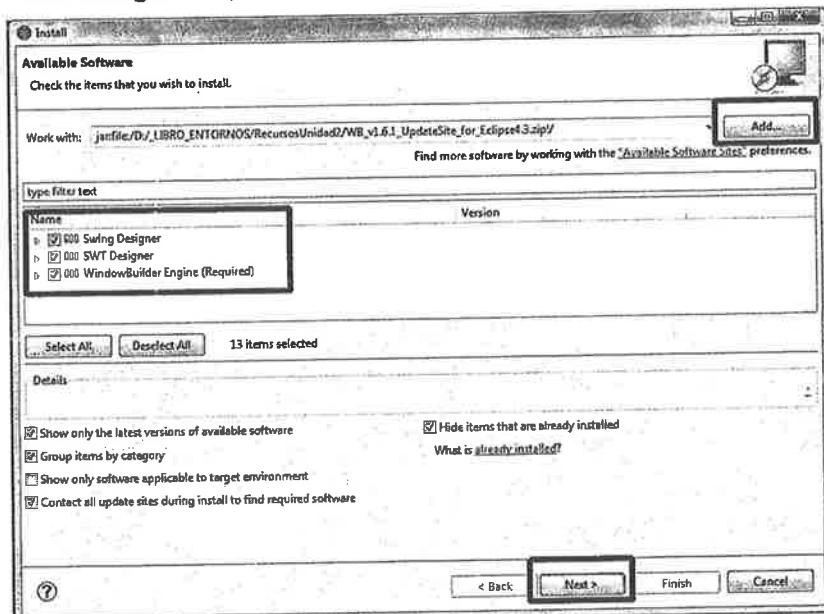


Figura 2.7. Instalación del plugin *WindowBuilder*.

Se visualizará una ventana indicando todo lo que se va a instalar, y pedirá aceptar los términos de la licencia. Se acepta y comenzará la descarga e instalación. Una vez instalado pedirá reiniciarse.

También se puede instalar escribiendo la siguiente URL:

<http://download.eclipse.org/windowbuilder/WB/release/R201406251200/4.3/>

en el campo *Work With* desde la opción de menú *Help/Install New Software*.

Más información de cómo se instala el WindowBuilder:

<http://download.eclipse.org/windowbuilder/WB/release/R201406251200/4.3/>

CREACIÓN DE UN PROYECTO CON UNA INTERFAZ GRÁFICA

Para crear un proyecto abrimos el menú *File/New/Java Project*, y en la ventana que aparece (véase la Figura 2.8) se escribe el nombre del proyecto, se dejan las opciones por defecto y se pulsa el botón *Finish*. A continuación, se añade una ventana al proyecto, para ello nos posicionamos sobre el nombre del proyecto, pulsamos el botón derecho del ratón y en el menú contextual se elige *New/Other*. A continuación se abre la carpeta *WindowBuilder/Swing Designer* y se elige *Application Window*, véase la Figura 2.9. Seguidamente se escribe el nombre de la ventana y se pulsa *Finish*. Así se crea una ventana de aplicación. Si se ejecuta el proyecto se visualizará la ventana inicialmente vacía.

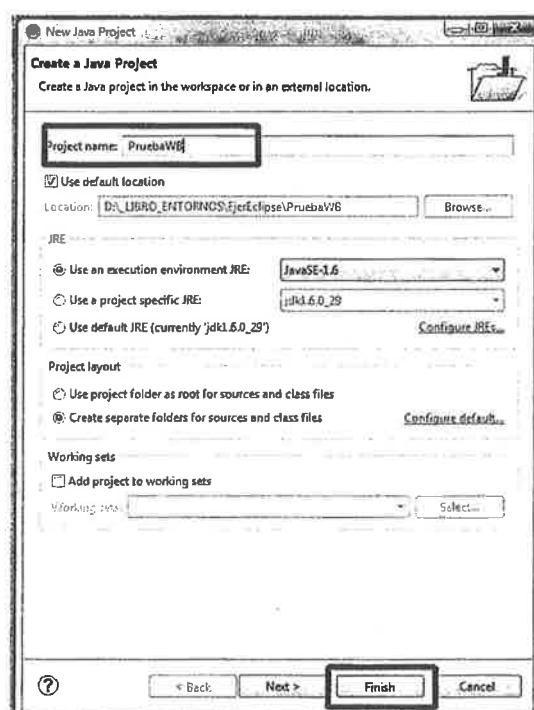


Figura 2.8. Creación de un proyecto Java

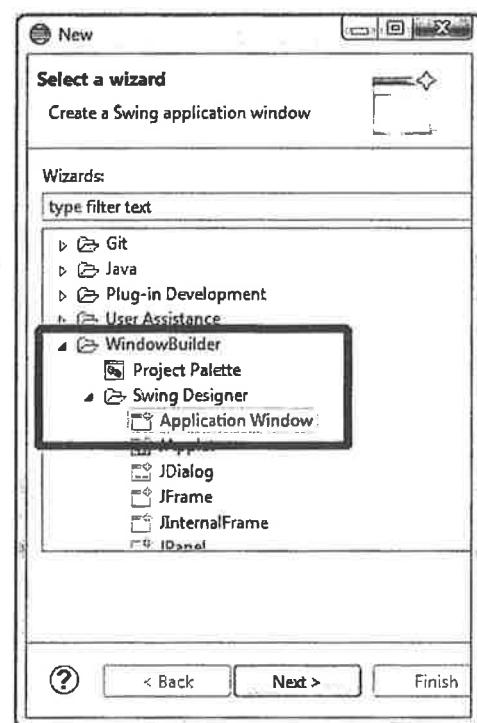


Figura 2.9. Creación de una ventana de aplicación de Windows.

Observa que se ha creado una clase con el nombre puesto a la ventana, esa clase se puede editar en modo *Source* (fuente) o en modo *Design* (diseño). En la Figura 2.10 se muestra la ventana en modo diseño, se distinguen varios bloques:

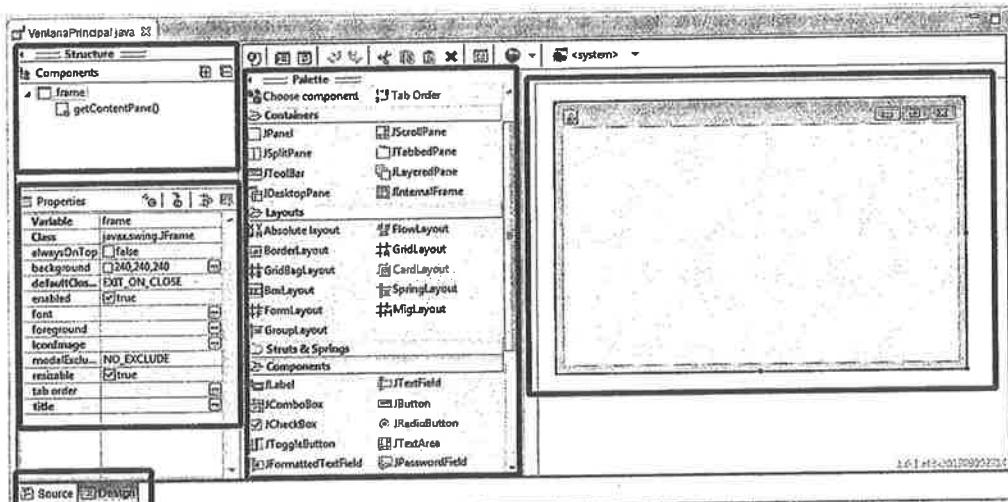


Figura 2.10. Vista diseño de una interfaz gráfica.

- **Structure**, donde se ven de manera jerárquica los componentes que se han agregado a la ventana.
- **Properties**, se muestran las propiedades del elemento o componente seleccionado, como el título, tipo de letra, tamaño, color, etc, estas propiedades se pueden cambiar.
- **Palette**, se muestran los elementos de tipo contenedor que se pueden añadir a la ventana; los layouts, es decir, la distribución de los elementos en la ventana; o los componentes, tipo etiquetas, cajas de texto, listas desplegables, etc, que se pueden añadir.
- **La ventana o formulario**, que es donde se van añadiendo los elementos.

Para añadir componentes a la ventana primero se pulsa **Absolute layout** (en **Palette/Layouts**) y se arrastra al marco interno de la ventana esto, va a permitir colocar los componentes en cualquier parte de la ventana.

Otra forma de abrir la ventana en modo diseño es pulsar sobre la clase con el botón derecho del ratón y seleccionar **Open With/WindowBuilder Editor**

ACTIVIDAD 2.1

Utiliza los controles JLabel, JTextField, JButton y JPanel (caja que encierra a los controles) para crear la ventana de la Figura 2.11:



Figura 2.11. Ventana de la actividad.

Cambia las propiedades: *Variable* para poner el nombre del control, y para cambiar el aspecto utiliza: *background, font, foreground, text, horizontalAlignment, verticalAlignment* etc.

Observa el código fuente que se va generando en la clase al ir añadiendo los controles.

Para añadir acción a los botones de las ventanas se hace doble clic sobre el botón y se visualiza el código ***actionPerformed*** asociado al botón, las acciones se añadirían en ese método. Por ejemplo, si queremos visualizar un mensaje en la consola al pulsar el botón limpiar y al pulsar el botón insertar escribimos dentro del ***actionPerformed*** de cada botón:

```
 JButton btnLimpiarDatos = new JButton("Limpiar Datos");
btnLimpiarDatos.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e)
    {
        System.out.println(" SE HA PULSADO EL BOTÓN LIMPIAR");
    }
});

JButton btnInsertarDatos = new JButton("Insertar Datos");
btnInsertarDatos.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e)
    {
        System.out.println(" SE HA PULSADO EL BOTÓN INSERTAR");
    }
});
```

2.3.3. Instalación de SQL Developer

SQL Developer es una herramienta gráfica de Oracle que simplifica las tareas de desarrollo para bases de datos, principalmente ORACLE. Con la herramienta se podrá navegar, editar y crear objetos de base de datos Oracle, ejecutar sentencias SQL, editar y depurar PL SQL, ejecutar informes y colocar archivos bajo control de versiones. Además se podrán crear modelos de datos, generar los DDL correspondientes, realizar modelos relacionales e ingeniería entre el modelo lógico y el relacional. En esta unidad se instalará SQL Developer versión ***sqldeveloper-3.2.10.09.57*** (se instala esta versión para la creación de modelos de datos ya que las jerarquías no se crean correctamente en las versiones 4.0.x de Sql Developer).

La instalación es similar a Eclipse, basta con descomprimir el archivo descargado. Para ejecutarlo se accede a la carpeta y se hace doble clic en el ejecutable ***sqldeveloper.exe***. Puede que al ejecutar pida la ruta donde se encuentra instalado el JDK, en ese caso se añade. Hay que instalar la versión que corresponde con el sistema operativo, es decir versión de 32 o 64 bits. Si no es así se visualizarán errores indicando que no se puede lanzar la máquina virtual.

Aunque esta herramienta está pensada para trabajar con Oracle, también podemos crear conexiones y trabajar con otras bases de datos como MySQL, bastaría con instalar el conector correspondiente. Para utilizar la herramienta Data Modeler o el Control de versiones que incorpora SQL Developer no es necesario tener instalada la BD Oracle, sin embargo conviene tener instalada la BD Oracle, y también MySQL, pues los ejercicios de modelos de datos luego se crearán en una base de datos.

¡IMPORTANTE!

Consulta el documento que aparece en la carpeta de recursos de esta unidad para instalar la BD ORACLE, y para instalar MySQL. En ORACLE se desbloqueará al usuario HR para luego utilizarlo, en MySQL se creará la BD *Prueba* con las tablas empleados y departamentos. El usuario de MySQL será *root* sin clave. A lo largo de esta unidad se crearán y utilizarán conexiones a estas bases de datos.

Al arrancar SQL Developer se muestra la ventana de la Figura 2.12, en la que distinguimos el bloque de la izquierda donde se configurarán las conexiones a la BD y el bloque central que es la página de bienvenida y que permite entrar en las ayudas y tutoriales de la herramienta.

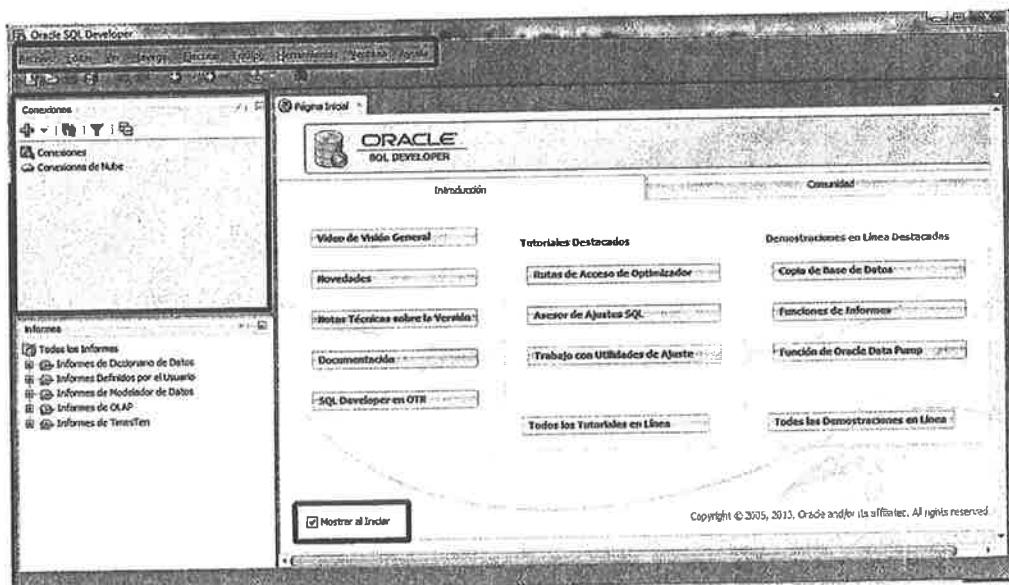


Figura 2.12. Ventana inicial de SQL Developer.

CREACIÓN DE CONEXIONES A BD DESDE LA HERRAMIENTA

Para crear una conexión pulsamos al botón más(+) del panel de conexiones (O también menú *Archivo/Nuevo/Conexión a Base de Datos*). Se visualiza el cuadro de diálogo para añadir los datos de la conexión, en esta ventana hay que rellenar:

El *nombre de la conexión*, que es el nombre que luego se verá el panel, debe ser un nombre que nos informe del usuario y de la BD a la que se conecta. Luego se teclea el nombre del usuario y la contraseña del usuario (en esta unidad se utilizará el usuario HR que se crea con la instalación de Oracle).

En el *tipo de conexión* se dejan las opciones por defecto. Si se va a crear una conexión con el administrador SYS, en *Rol* hay que seleccionar SYSDBA. También es importante saber el puerto y el SID (nombre de la instancia). Las versiones exprés de Oracle crean una única instancia llamada XE. Esos datos se indican al final de la instalación de la BD. Pulsaremos los botones *Probar* para ver si hay conexión, *Guardar* para que luego aparezca en el panel de conexiones, y *Conectar* para empezar a trabajar, véase la Figura 2.13.

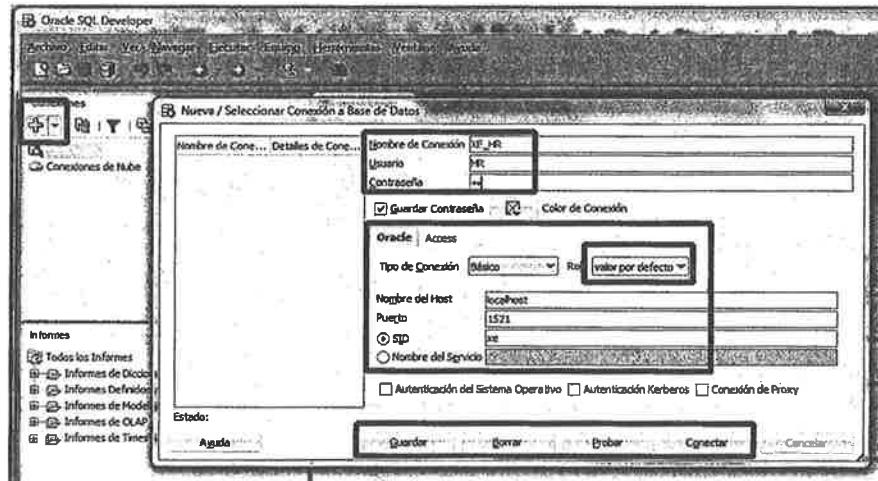


Figura 2.13. Crear una conexión a Oracle con SQL Developer.

Una vez creada la conexión al hacer clic sobre ella se abre la ventana de trabajo de SQL, desde aquí se podrán escribir y ejecutar sentencias SQL (véase la Figura 2.14). Además se puede navegar por todos los objetos que tiene el usuario asociado a la conexión, se pueden ver las tablas que tiene el usuario HR si se pulsa al nodo *Tablas*, si se hace doble clic en una tabla se podrá ver el detalle de la misma.

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	RESELLER SUPPORT	40
20	SALES	10
30	ACCOUNTING	20
40	SHIPPING	30
50	IT	40
60	MARKETING	50
70	OPERATIONS	60
80	FINANCIAL	70
90	GENERAL	80

Figura 2.14. Vista de la hoja de trabajo, y el navegador de objetos

Si al conectarnos con el usuario HR aparece una ventana indicando que la contraseña va a caducar, escribe en la hoja de trabajo ***alter user hr identified by hr2;*** donde *hr2* será la nueva clave. Ejecuta la sentencia pulsando al botón ▶ (Sentencia de Ejecución). No olvides luego poner la nueva contraseña en las propiedades de la conexión (botón derecho sobre la conexión opción *Propiedades*).

CREACIÓN DE UNA CONEXIÓN A UNA BD MySQL

Para crear una conexión a una BD distinta de Oracle, es necesario instalar el driver JDBC correspondiente. Para MySQL se instalará ***mysql-connector-java-5.1.18-bin.jar*** que se encuentra en la carpeta de recursos. Se copia en el disco en una carpeta y se abre el menú *Herramientas/Preferencias*, en la ventana que se muestra se accede a la sección *Bases de datos* y se selecciona la opción *Controladores JDBC de Terceros*. Una vez aquí se selecciona la ruta donde se encuentra el fichero *.jar*. Una vez instalado aparecerá una pestaña nueva en la ventana de creación de conexiones. Se reinicia el SQL Developer.

Para crear una conexión a la BD *pruebas* de Mysql, se pulsa al botón de nueva conexión, se teclea el nombre y el usuario, el usuario es *root* sin contraseña. El servidor es *localhost*, la BD MySQL escucha por el puerto *3306*, y para elegir la BD *pruebas* se pulsa al botón *Seleccionar Base de Datos* y se selecciona (véase la Figura 2.15). Se guarda la conexión.

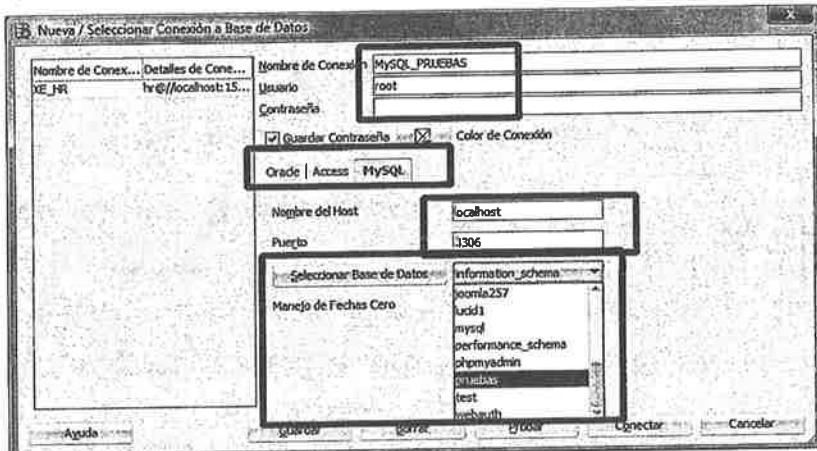


Figura 2.15. Creación de conexión a una BD MySQL

Al conectarnos se podrán realizar consultas y otros tipos de sentencias SQL sobre las tablas de la BD asociada. Y además por ser el usuario *root*, también se podrá trabajar con las tablas de las otras bases de datos.

2.4. HERRAMIENTAS PARA EL MODELADO DE DATOS

En este apartado se va a estudiar cómo crear modelos de datos utilizando las dos herramientas instaladas. Este apartado es un complemento a los primeros temas del módulo de Bases de Datos en el que se estudia el modelo E/R y el relacional. En este apartado se crearán modelos partiendo de diagramas E/R ya elaborados, no se trata de que aprendan a hacer los diagramas, eso se estudia en bases de datos, se trata de que esos diagramas los realicen con este tipo de herramientas.

2.4.1. Data Modeler de Oracle

Oracle Data Modeler es una herramienta de modelado de bases de datos gratis desarrollada por Oracle, se puede descargar por separado o incorporada al *SQL Developer*. Las características más importantes son las siguientes:

- Los modelos se almacenan en el sistema de ficheros, bajo una estructura de directorios (por lo que son fáciles de transportar, archivar, etc.)
- Puede trabajar con cualquier base de datos, no está restringido a Oracle.
- Permite realizar ingeniería inversa.
- Dispone de los siguientes niveles de diseño: lógico, relacional y físico, con herramientas de generación automática en ambos sentidos.
- Permite definir dominios de tipos de datos.
- Herramienta de diseño visual y rica en herramientas (colores personalizables, deshacer, búsqueda de objetos, etc.)
- Múltiples opciones en la generación del DDL.

En este apartado se estudiará la versión de *Data Modeler* que incorpora SQL Developer versión *sqldeveloper-3.2.10.09.57*, que se encuentra en la carpeta de recursos del capítulo. Para abrir el explorador de Data Modeler se pulsa al menú *Ver/Data Modeler/Explorador*, se visualiza el módulo del *Data Modeler*, véase la Figura 2.16. Para abrir la ventana del modelo lógico, desde el menú contextual de *Modelo Lógico* se elige *Mostrar*. Igualmente para crear un modelo relacional, se elige *Nuevo Modelo Relacional* del menú contextual de *Modelos Relacionales*.

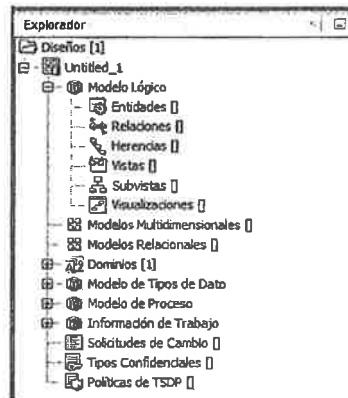


Figura 2.16. Explorador del Data Modeler.

Antes de crear ningún modelo vamos a cargar las tablas que tiene el usuario HR en un modelo relacional. Para ello desde el menú contextual de *Modelos Relacionales*, se elige *Nuevo Modelo Relacional*. Se muestra una hoja de trabajo con la barra de botones para crear el modelo. Se seleccionan todas las tablas del usuario HR y se arrastran a la hoja de trabajo (para que se vean bien las relaciones se recomienda crear las tablas dando nombre a las restricciones de claves primarias y ajena).

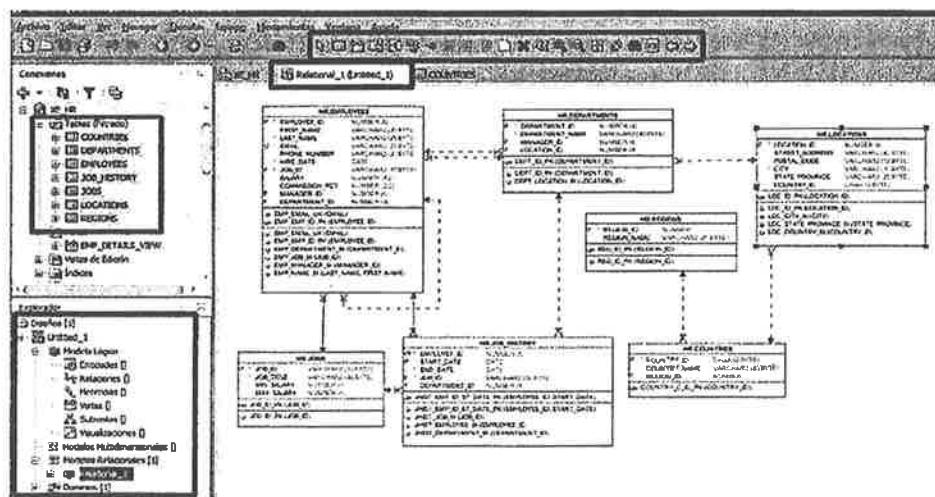


Figura 2.17. Modelo relacional asociado a las tablas del usuario HR.

Desde la barra de botones se podrán añadir tablas, vistas, claves ajenas, dividir o fusionar tablas, realizar la ingeniería al modelo lógico o generar las sentencias DDL (*Data Definition Language*, sentencias para la definición de datos). Si se pulsa el botón *Realizar ingeniería al modelo lógico* , se crearán las entidades y relaciones correspondientes a este modelo relacional. Se puede seleccionar qué elementos se quieren convertir al modelo lógico, véase Figuras 2.18 y 2.19.

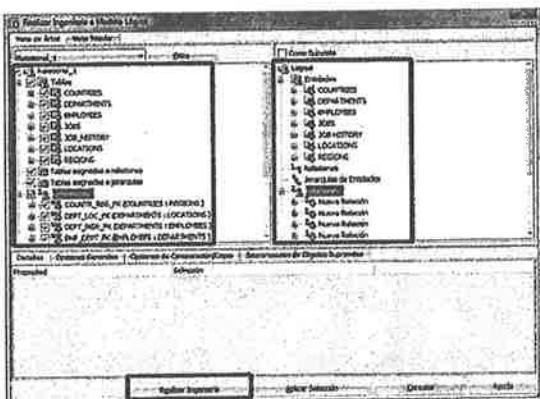


Figura 2.18. Generación del Modelo Lógico a partir del modelo relacional.

Para guardar este diseño de datos se pulsa al menú *Archivo/Data Modeler/Guardar o Guardar como*, se selecciona la carpeta y se guarda (elimina la contrabarra del nombre). Si se desea abrir un modelo, igualmente se hace desde el menú *Archivo/Data Modeler/Abrir*. Al guardar el proyecto se genera un archivo con la extensión *.dmd* (por ejemplo *ModeloHR.dmd*) y una carpeta con el mismo nombre.

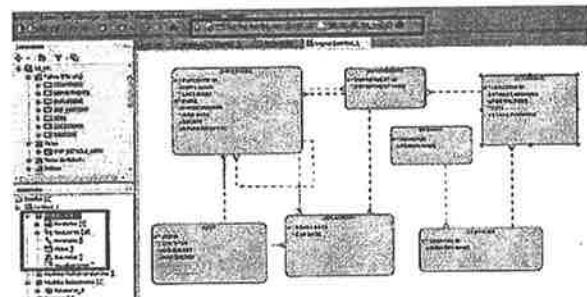


Figura 2.19. Modelo Lógico Entidad/Relación asociado a las tablas del usuario HR.

2.4.1.1. Creación de modelos de datos con Data Modeler

El siguiente paso es construir un modelo lógico de datos con la herramienta. Partimos de diseños ya realizados. Suponemos que se ha realizado un modelo de datos de un enunciado y se ha obtenido el esquema que se muestra en la Figura 2.20.

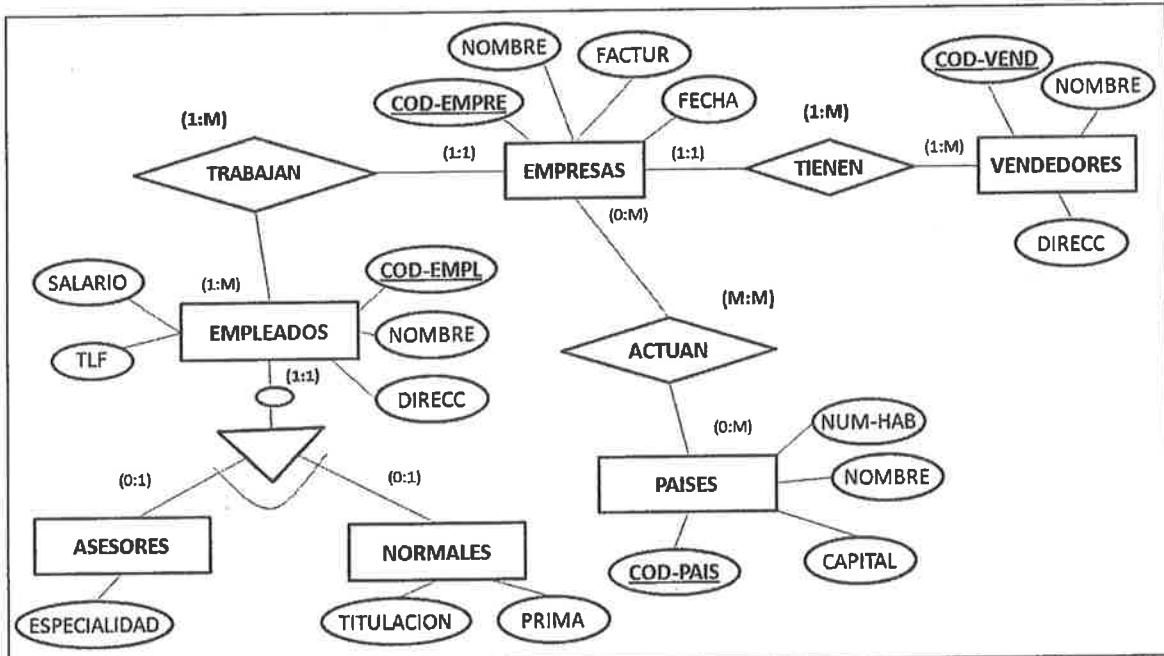


Figura 2.20. Esquema Lógico E/R.

Creación de un nuevo diseño, para ello dentro del explorador de *Data Modeler*, se abre el menú contextual del nodo *Diseños* y se selecciona *Nuevo Diseño*. Utilizaremos la barra de botones (véase la Figura 2.21) para ir añadiendo las entidades y las relaciones entre ellas. Una vez creado el modelo se realiza la ingeniería inversa para crear el modelo relacional.



Figura 2.21. Barra de botones modelo lógico.

Para crear una entidad se pulsa el botón *Nueva Entidad*, se muestra la ventana de propiedades de la entidad, véase la Figura 2.22. Desde esta ventana se irán añadiendo cada una de las propiedades de la entidad. Nos fijaremos en las propiedades de los apartados general y atributos.

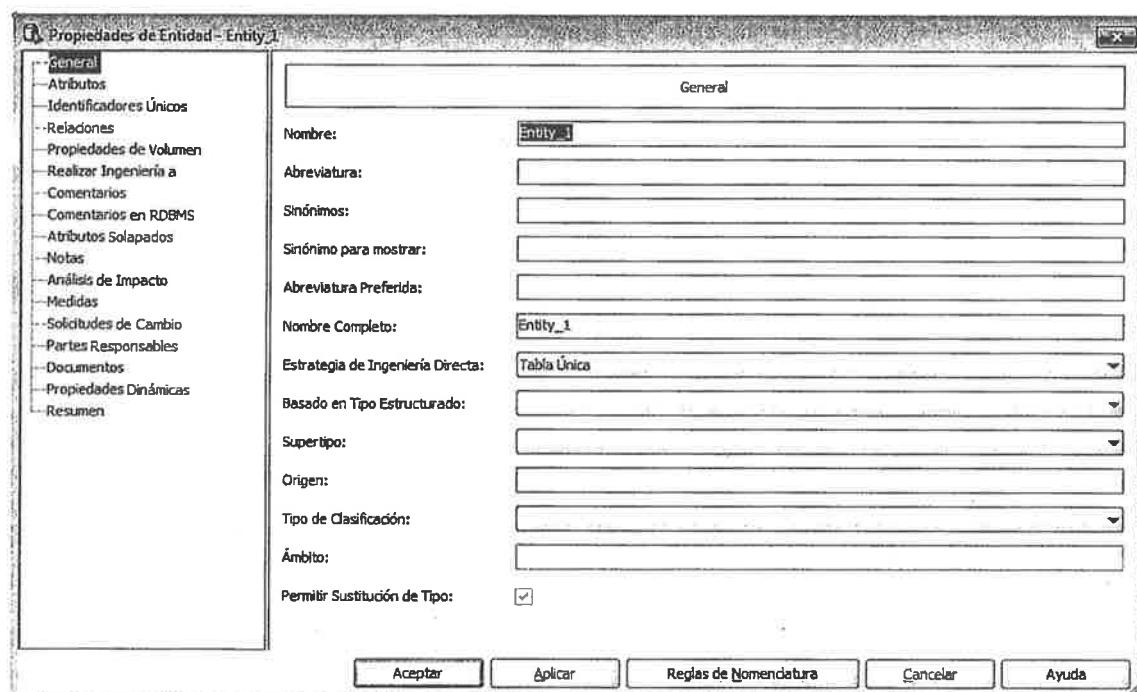


Figura 2.22. Propiedades de Entidad (versión *sqldDeveloper-3.2.10.09.57*).

PROPIEDADES DE GENERAL

En este apartado escribiremos los datos que tienen que ver con la entidad. Lo más importante es poner el nombre de la entidad, y si se trabaja con jerarquías indicar el supertipo, y como se desea la ingeniería, el resto se dejan las opciones por defecto. Las propiedades más utilizadas son:

- **Nombre:** Nombre que tiene la entidad.
- **Abreviatura:** Abreviatura o nombre corto para la entidad.

- **Sinónimos:** Sinónimos de la entidad.
- **Sinónimo para mostrar:** Sinónimo a mostrar para la entidad.
- **Abreviatura preferida:** Nombre que se usará para cualquier correspondencia de tabla durante el avance de ingeniería si la opción *Usar abreviaturas Preferidas* está activada en el cuadro de diálogo de *Ingeniería*.
- **Nombre completo:** Nombre largo que se utiliza en la visualización de nombre de entidades y nombre de atributos.
- **Estrategia Ingeniería directa:** Este campo es importante cuando se trabaja con entidades supertipo y subtipo. Se define la estrategia de las entidades para hacer la ingeniería a las tablas del modelo relacional:
 - **Tabla única:** Los datos de la entidad subtipo se incluyen en la tabla supertipo. No hay tablas subtipo.
 - **Tabla por secundario:** Los datos entidad supertipo se incluye en las tablas del subtipo. No hay una tabla supertipo.
 - **Tabla para cada entidad:** se genera una tabla separada para cada entidad supertipo y subtipo.
- **Basado en tipo estructurado:** Si la entidad se basa en un tipo estructurado (XML, SDO_GEOmetry) se selecciona de la lista. La entidad contendrá los atributos definidos en el tipo estructurado.
- **Supertipo:** Si la entidad es un subtipo, se selecciona el supertipo de la lista. La entidad hereda todos los atributos del supertipo especificado.

PROPIEDADES DE ATRIBUTOS

Desde aquí se añadirán los atributos a la entidad, véase la Figura 2.23. Las propiedades de cada atributo incluye su nombre y tipo de datos, y si es el identificador principal se marca (UID), si es un identificador de relación se marca (UID de Relación, comparable a la clave ajena), y si es un campo obligatorio se marca la casilla. En tipo de datos se puede seleccionar dominios (más adelante se verá cómo crear dominios), datos de tipo lógico (son los tipos de datos comunes), y datos de tipo estructurado (datos compuestos de otros datos). Inicialmente seleccionaremos datos de tipo lógico, y de ellos trabajaremos con los tipos básicos: **NUMERIC** para datos numéricos, **VARCHAR** para cadenas, y **DATE** para fechas.

Al seleccionar datos NUMERIC escribiremos una precisión, que indicará el número total de dígitos para la cantidad, y una escala para indicar las posiciones decimales. Al seleccionar VARCHAR se escribe el tamaño de la cadena, en unidades se deja la opción por defecto.

Para agregar un atributo, se hace clic en el botón *Agregar* (+), para eliminarlo se hace clic en *Eliminar* (X). Se utilizará las flechas para mover de posición los atributos. Para ver las propiedades de un atributo, se hace doble clic en el nombre.

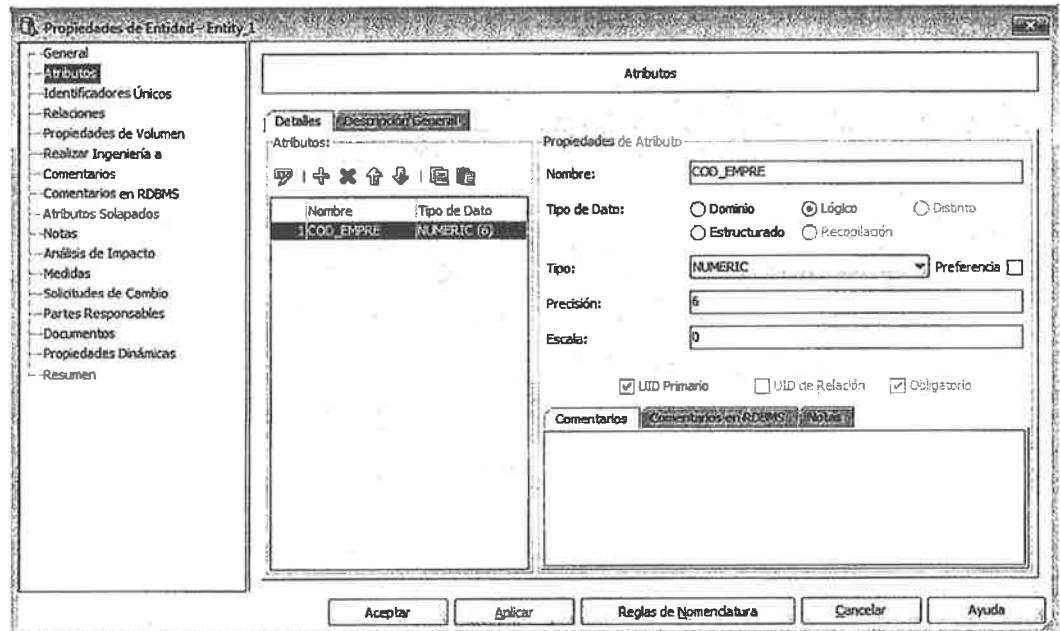


Figura 2.23. Propiedades de Atributos.

RESTO DE PROPIEDADES

Para el resto de propiedades se dejarán las opciones que aparecen, a no ser que se tenga que añadir alguna restricción. Las más importantes son:

- **Identificadores únicos:** se visualizan los identificadores únicos UID (restricción UNIQUE que son los atributos cuyo valor no se pueden repetir en la entidad) y las claves primarias de la entidad PUID (restricción PRIMARY KEY). Se podrán añadir y eliminar identificadores.
- **Relaciones:** muestra las relaciones asociadas con la entidad, para ver las propiedades de una relación se hace clic en el nombre.
- **Propiedades de volumen:** se indica el volumen de datos de la entidad, la tasa de crecimiento, en formulario normal se indica la forma requerida de normalización para la entidad (ninguna, primera, segunda, tercera, cuarta). Con normalización adecuada, se indica que el modelo está suficientemente normalizado, NO indica que el modelo no está normalizado, y que la normalización puede ser requerida en el modelo relacional.
- **Realizar Ingeniería:** se especifica el modelo relacional en el que esta entidad deberá ser propagada en las siguientes operaciones de ingeniería. Controla cuándo la entidad se propaga según el modelo Relacional especificado durante las operaciones de ingeniería.
- **Los comentarios,** se utilizan para añadir comentarios en la generación del modelo de datos.

NOTA

En versiones *sqldeveloper-4.0.x*, para crear jerarquías existe el apartado **Propiedades de subtipos** en donde se especificará las opciones que son importantes cuando se trabaja con las entidades supertipo y subtipo. Destacamos los siguientes apartados:

Generación de árbol secundario: define la estrategia de ingeniería al convertir las entidades a tablas en el modelo relacional:

- **Dont not present:** No cambian las propiedades.
- **Tabla única:** Los datos de la entidad subtipo se incluyen en la tabla supertipo. No hay tablas subtipo.
- **Tabla por secundario:** Los datos entidad supertipo se incluye en las tablas del subtipo. No hay una tabla supertipo.
- **Tabla para cada entidad:** se genera una tabla separada para cada entidad supertipo y subtipo.

Implementación de subtipos: se indica cómo se implementan las relaciones y los atributos de los supertipos y los subtipos:

- **Referencias:** define cómo se relacionarán las tablas al implementarlas en el modelo relacional: *Ninguno*, *Identifying* (identificación de las claves ajenas, se crean claves ajenas desde los subtipos al supertipo), o *Reverse arc* (solo un registro de la tabla subtipo puede existir para cada registro de la tabla supertipo, es la jerarquía completa de los subtipos).
- **Herencia de atributos:** Define cómo se heredan los atributos de un supertipo, tanto en el supertipo y subtipo (Atributos primarios solo, *Primary Attributes Only* o todos los atributos *All Attributes*).

Columna de discriminador: se indicará si hay algún atributo discriminador de los subtipos.

- **Generar discriminador:** Si se activa esta opción, se genera una columna discriminadora.
- **Usar Atributo:** Permite elegir un atributo existente como la columna discriminadora.
- **Nombre de la columna:** Para crear el atributo discriminador con el nombre que se escriba.
- **Valor de discriminador:** El valor relacionado con la entidad actual; puede existir en la columna de discriminador. Si no se especifica este campo, se utiliza el nombre corto entidad o el nombre.
- **Subtipos Completados:** Si esta opción se activa, los subtipos se marcan como completos, que afecta a las relaciones generadas y la lista de los posibles valores de la columna discriminadora. Si esta opción no está activada, se generan relacionesopcionales y el valor para la entidad actual se incluye en la lista de posibles valores para la columna discriminadora.

El problema de estas versiones (las *sqldeveloper-4.0.x*), es que las jerarquías no se crean correctamente, es por eso que para este capítulo se ha decidido utilizar la versión anterior, para crear modelos de datos con jerarquías.

CREAR LAS RELACIONES

Para crear una relación se utilizan los botones de la barra de botones del modelo lógico (véase la Figura 2.21) primero se crean las dos entidades y luego se relacionan. Las relaciones pueden ser 1:N, M:N y 1:1. Al crear una relación 1:M, se hace clic en el botón 1:N, y primero se selecciona la entidad de 1 y se arrastra la relación a la entidad del N.

Por ejemplo la empresa tiene muchos vendedores, es decir, 1 empresa N vendedores. La relación va de EMPRESAS a VENDEDORES. Y la cardinalidad (1,M) va en el sentido de EMPRESAS a VENDEDORES, porque una empresa tiene como mínimo 1 vendedor y como máximo N. La cardinalidad (1,1) va de vendedor a empresa porque un vendedor pertenece a 1 empresa como mínimo y a 1 empresa como máximo, es decir los vendedores pertenecen obligatoriamente a 1 empresa (Véase la Figura 2.24).

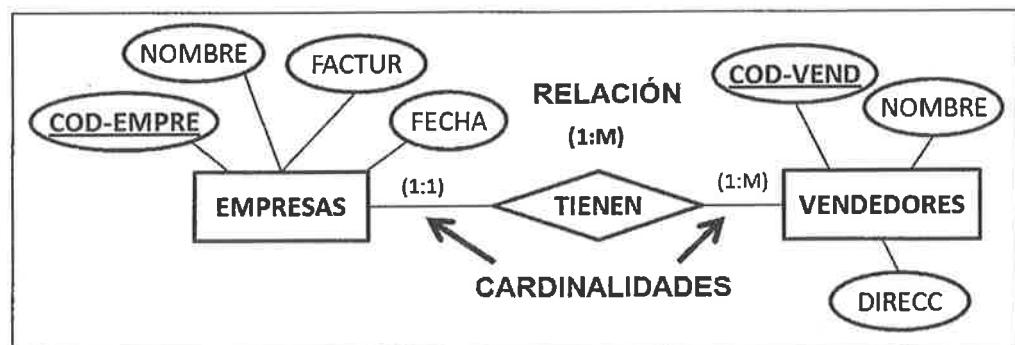


Figura 2.24. Representación de una relación.

Al crear la relación con la herramienta, se abre la ventana de propiedades de la relación (véase la Figura 2.25) en ella se indicará lo siguiente:

- **Nombre:** se escribirá el nombre de la relación, en el ejemplo TIENEN.
- **Nombre completo:** se mostrará cómo queda la relación, en el ejemplo: TIENEN (EMPRESAS - 1..*:1..1 - VENDEDORES), quiere decir una empresa tiene muchos vendedores (1..*) y un vendedor es de una empresa (1..1).
- **Cardinalidad del Origen:** *Origen*, *Nombre en origen* y *Sinónimo*, indican la entidad origen de la relación, en nuestro caso EMPRESAS. En *Cardinalidad de Origen a Destino*, aparece la cardinalidad máxima de EMPRESAS a VENDEDORES (*). En *Origen Opcional* se indica la cardinalidad mínima, es decir la obligatoriedad de que la empresa como mínimo tenga un vendedor, en el ejemplo la casilla se desmarca porque no es opcional sino obligatoria. Esta casilla se ha de marcar cuando nos encontramos cardinalidades mínimas a 0.
- **Cardinalidad del Destino:** igual que la anterior pero en este caso con la entidad destino, en el ejemplo VENDEDORES. La cardinalidad de *Destino a Origen* es 1, y se desmarca la casilla de *Destino Opcional*, pues la cardinalidad mínima es 1 porque se ha obligado (en el modelo lógico) que el vendedor sea de una empresa.

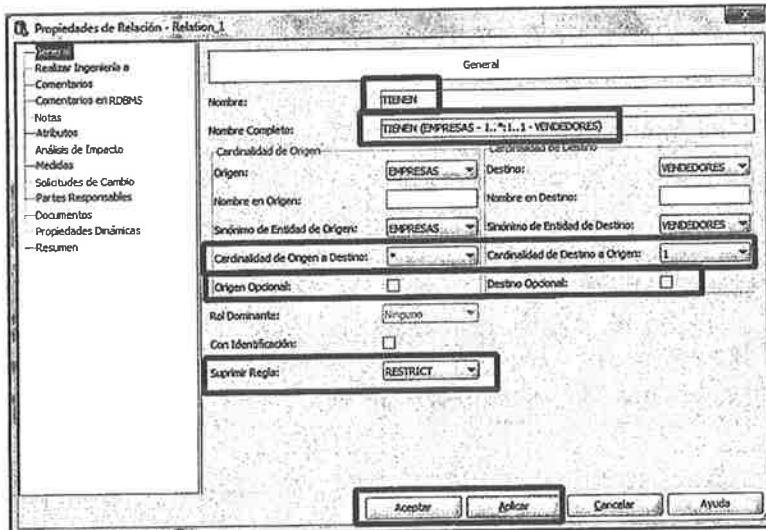


Figura 2.25. Propiedades de la Relación TIENEN.

- **Suprimir Regla:** esto es útil para la conversión al modelo relacional, pues se indica cómo se borrarán los registros relacionados al eliminar un registro de la tabla principal. Con RESTRICT, no se podrá eliminar si la tabla tiene registros relacionados (es decir no se podrá borrar una empresa si hay vendedores relacionados). Con NO ACTION, no permite las operaciones, similar al anterior, y con SET NULL permanecen los registros relacionados pero se asigna el valor NULL a las columnas relacionadas (claves ajenas).
- Se pulsa el botón *Aplicar* para ir viendo los cambios, y *Aceptar* para salir al modelo lógico.

Observa cómo quedan las líneas de las relaciones en el esquema, dependiendo de la cardinalidad aparecen punteadas o sólidas. Prueba a cambiar el *Origen* y *Destino Opcional* de las propiedades de la relación para ver cómo se muestran las cardinalidades en el modelo lógico, véase la Figura 2.26:

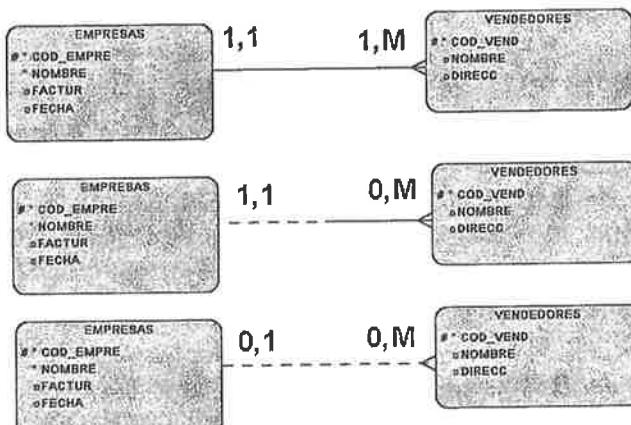


Figura 2.26 Representación de cardinalidades en el modelo lógico.

NOTA: Se puede cambiar la notación en la representación de los modelos lógicos desde el menú contextual de la hoja de diseño, por defecto los modelos se hacen con notación de *Barker*, pero se puede elegir la *Notación de Bachman*, o notación de *Ingeniería de la Información*.

CREAR JERARQUIAS

Para crear las jerarquías, primero se crean todas las entidades y en las propiedades de las entidades subtipo se indicará: en *Estrategia Ingeniería directa* se seleccionará el tipo de transformación que se realizará: **Tabla única**, para englobar a todos los atributos de los subtipos en el supertipo; **Tabla por secundario**, se elimina el supertipo y se quedan los subtipos con los atributos comunes y los suyos propios; o **Tabla para cada entidad**: se genera una tabla separada para cada entidad supertipo y subtipo. Y en la propiedad *Supertipo* se selecciona la entidad supertipo de los subtipos.

Qué transformación elegir:

- **Tabla única.** En general, se elige esta solución cuando los subtipos se diferencian en muy pocos atributos y las relaciones que tienen con otras entidades del esquema son las mismas para todos o casi todos los subtipos. Para diferenciar un tipo de otro se añade un atributo discriminante.
- **Tabla para cada entidad.** Esta es la solución adecuada cuando existen muchos atributos distintos entre los subtipos y se quieren mantener los atributos comunes a todos ellos en una tabla.
- **Tabla por secundario.** Se elegirá esta opción cuando muchos atributos distintos y los accesos realizados sobre los datos de los distintos subtipos siempre afectan a atributos comunes.

Para el ejemplo se elige la primera opción, las entidades tienen las mismas relaciones y los subtipos pocos atributos, será necesario añadir un atributo discriminante para saber el tipo de empleado, este se indicará que no pueda tomar valores nulos. En la Figura 2.17 se muestra el modelo lógico del esquema de la Figura 2.20.

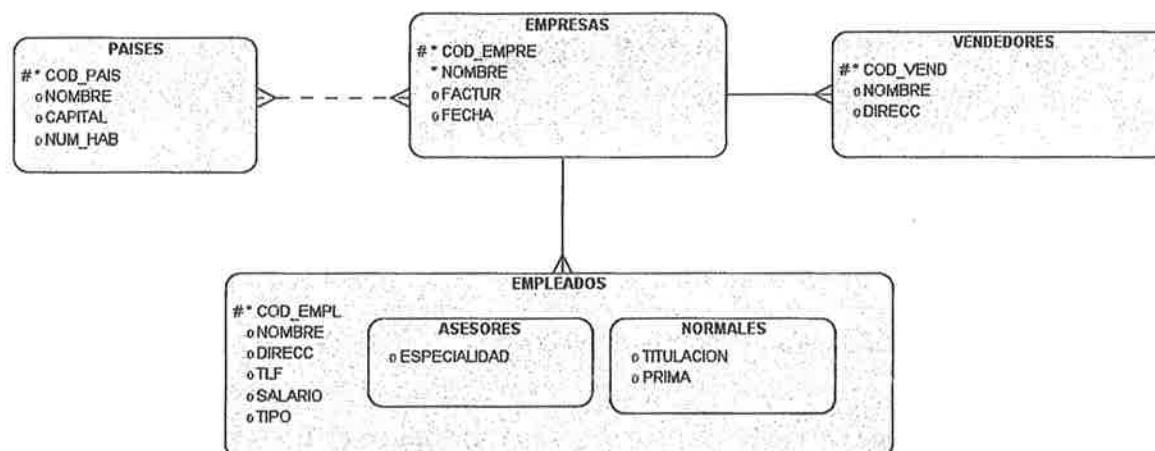


Figura 2.27. Representación del modelo lógico del ejemplo.

NOTA: En la conversión de jerarquías se hace necesario las restricciones semánticas de totalidad/parcialidad y de exclusividad/solapamiento. Para controlar estas restricciones se podrán añadir disparadores (TRIGGERS) y aserciones (ASSERTION). En las jerarquías si se elige la transformación *Tablas por secundario* o *Tablas por cada entidad* hay que cambiar los nombres de las restricciones, porque si no, se llamarán igual y ocurrirá un error (caso de las FK y PK por ejemplo).

NOTA: Para abrir, guardar, cerrar, importar, exportar o crear una imagen del modelo se hace desde el menú *Archivo/Data Modeler*.

INGENIERÍA AL MODELO RELACIONAL

Una vez creado el modelo lógico, lo siguiente es crear el modelo relacional, para ello las herramientas de modelado añaden la función automática de convertir el modelo lógico al modelo relacional. En *Data Modeler* se pulsa al botón *Realizar ingeniería al Modelo Relacional* de la barra de botones (véase la Figura 2.21). Se inicia el asistente para *Realizar Ingeniería al Modelo Relacional*, desde esa ventana se indicarán los elementos a realizar la ingeniería, también se pueden cambiar algunas opciones para realizar la ingeniería. Se pulsa el botón *Realizar ingeniería* y se genera automáticamente el modelo relacional, véase la Figura 2.28.

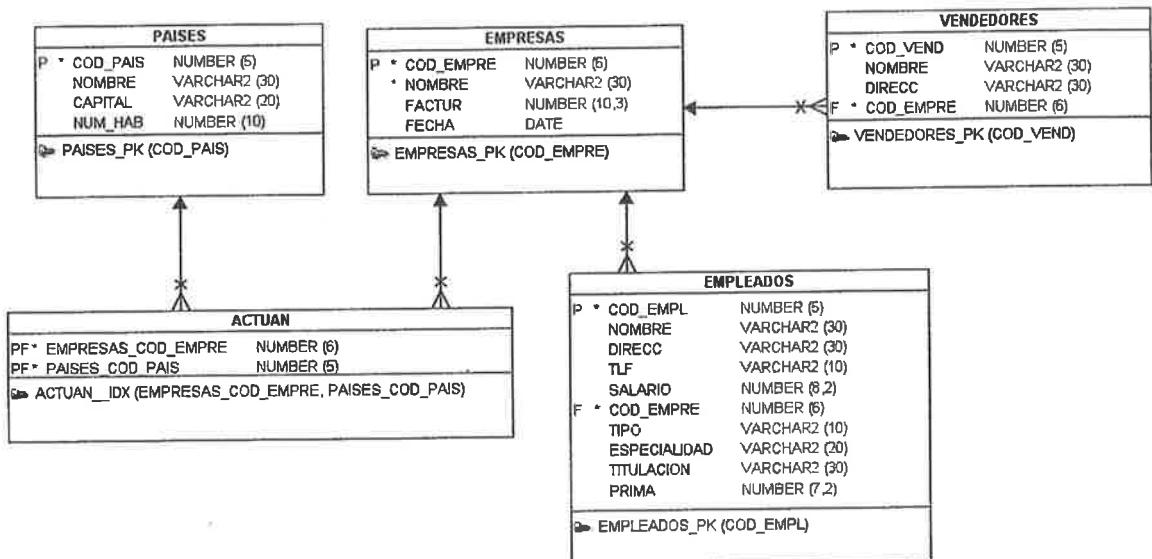


Figura 2.28 Representación del modelo relacional del ejemplo.

Observa cómo queda el modelo relacional y cómo quedan las relaciones y la propagación de claves. Las columnas NOT NULL aparecen marcadas con un punto rojo, las PRIMARYS KEYS aparecen con una P al lado de la columna correspondiente, y las FOREIGN KEYS aparecen marcadas con una F. Las columnas con restricción UNIQUE aparecerán marcadas con una U.

Desde el modelo relacional también es posible cambiar las propiedades de las tablas, para añadir más atributos o más restricciones, también se podrán añadir más tablas y más relaciones. Para ello se utilizarán los botones de la barra de botones del modelo relacional (véase la Figura 2.29). Las ventanas para acceder a las propiedades de las tablas o para añadir más tablas son similares a las ventanas de propiedades de las entidades.



Figura 2.29. Barra de botones del modelo relacional.

Lo siguiente es crear las DDL, es decir, la definición de las tablas, para ello se pulsa el botón *Generar DDL* de la barra de botones. En la ventana que se muestra se elige la base de datos y el modelo relacional, y se pulsa al botón *Generar*. A continuación aparece la ventana para añadir opciones en la generación de DDL (véase la Figura 2.30). Se pulsa el botón *Aceptar* y se visualiza el script de creación de las tablas (véase la Figura 2.31) se pulsa el botón *Guardar*.

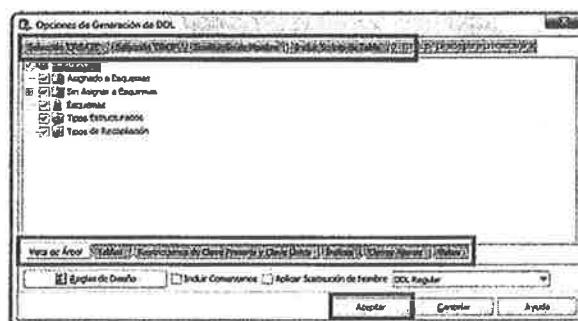


Figura 2.30. Opciones de generación DDL



Figura 2.31. Script DDL generado.

Para crear las tablas en el esquema de un usuario se abre el archivo creado con SQLDEVEOPER desde el menú *Archivo/Abrir*. Se pulsa el botón *Ejecutar Script*, de la barra de botones de la hoja de trabajo de SQL, pedirá una conexión y se elige la conexión del usuario HR (Véase la Figura 2.32).

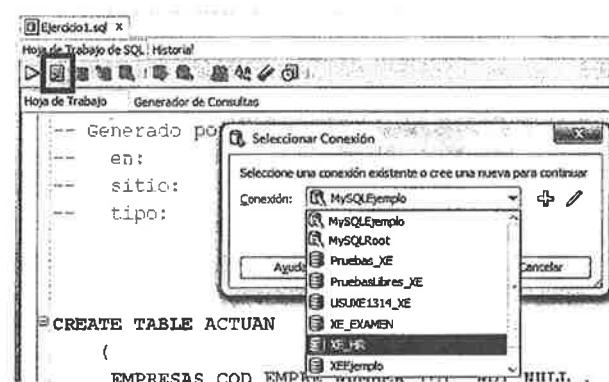


Figura 2.32. Ejecución de un script DDL generado

NOTA: Con las versiones nuevas de SQL Developer (*sqldeveloper-4.0.x*), es posible añadir atributos en las relaciones entre las entidades, es decir en el modelo lógico. Con las versiones anteriores se ha de hacer manualmente en el modelo relacional.

CREAR DOMINIOS

El dominio dentro de la estructura del modelo relacional es el conjunto de valores que puede tomar un atributo, es decir indica qué valores pueden ser asumidos por los atributos de las tablas. Normalmente un dominio se define a través de la declaración de un tipo de dato para el atributo, por ejemplo una cadena de 20 caracteres, o un numérico de 4 dígitos. Pero también es posible definir dominios más complejos y precisos.

Data Modeler dispone de un administrador de dominios, se abre pulsando el menú *Herramientas/Data Modeler/Administración de Dominios*. En la ventana que aparece (véase la Figura 2.33) se muestran opciones similares a la creación de atributos de los modelos lógico y relacional:

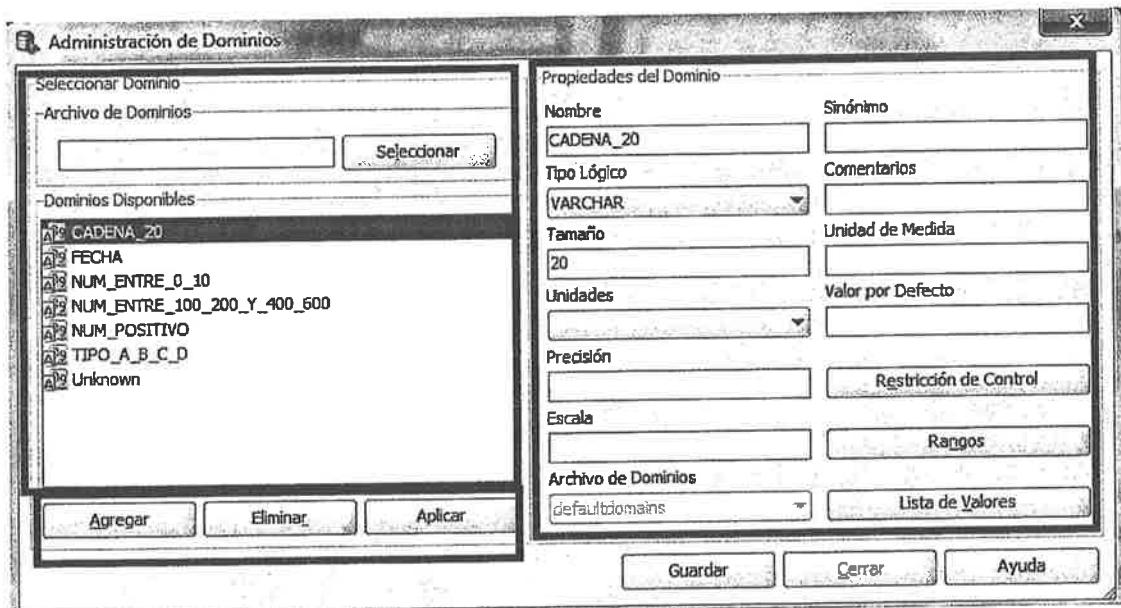


Figura 2.33. Administración de dominios.

- En el cuadro *Seleccionar Dominio*, se puede elegir un archivo de dominio donde se describen los dominios creados. Por defecto se carga *defaultdomains.xml*, que se encuentra dentro de la carpeta del *sqldeveloper*: (*sqldeveloper\sqldeveloper\extensions\oracle.datamodeler\types*). También se muestran los dominios creados.

- En *Propiedades del Dominio* se indicará, el nombre del dominio, el *Tipo Lógico* asociado al dominio, si será un dominio numérico, o de caracteres o de fecha, o cualquier otro tipo. También se indica el *Tamaño* (el máximo tamaño de datos para este tipo), La *Precisión*: para los datos numéricos se indica el número máximo de dígitos, y en *Escala* se indican los dígitos decimales. Se puede añadir un valor por defecto para el dominio. En *Restricción de Control* se pueden ver y editar las restricciones CHECK general y específica para el dominio. Desde *Rangos* se podrán añadir restricciones para que el dominio se encuentre entre un rango de valores (por ejemplo que una cantidad tome valores comprendidos entre 100 y 200, y, 400 y 600), y desde *Lista de Valores* se añaden las restricciones para que un dominio tome solo los valores de la lista de valores (por ejemplo que el tipo de producto puede ser A, B, C o D).

En la Figura 2.34 se muestra la creación de una lista de valores, y de un rango de valores.

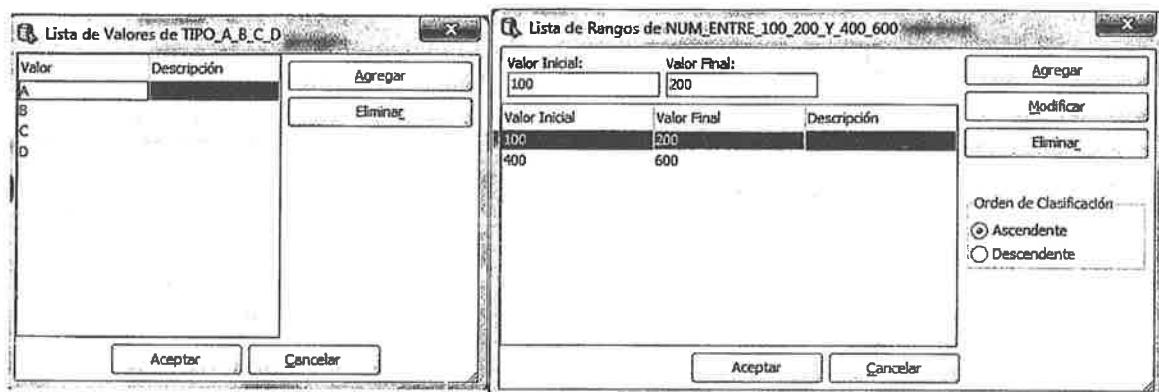


Figura 2.34. Creación de un dominio de lista de valores y de rango de valores.

A la hora de crear la DDL de una tabla con dominios asociados a los atributos en sus tipos y con restricciones de control, se añadirán estas restricciones en la definición de la tabla todas ellas de tipo check. En el ejemplo se ha creado la tabla pruebas con varios dominios:

```
CREATE TABLE PRUEBAS
(
    NOTA NUMBER (2) CHECK ( NOTA BETWEEN 0 AND 10 ) ,
    NUM_100A200_400A600 NUMBER (3) CHECK ( NUM_100A200_400A600 BETWEEN 100
        AND 200 OR NUM_100A200_400A600 BETWEEN 400 AND 600 ),
    TIPO CHAR (1) CHECK ( TIPO IN ('A', 'B', 'C', 'D') ),
    NOMBRE VARCHAR2 (20)
);
```

ACTIVIDAD 2.2

Pasa el esquema E/R de la Figura 2.35 a un modelo lógico con *Data Modeler*, realiza la ingeniería al modelo relacional. Añade los atributos de las relaciones N:M en el modelo relacional (recuerda que las relaciones N:M al pasárselas al modelo relacional se convierten en tablas).

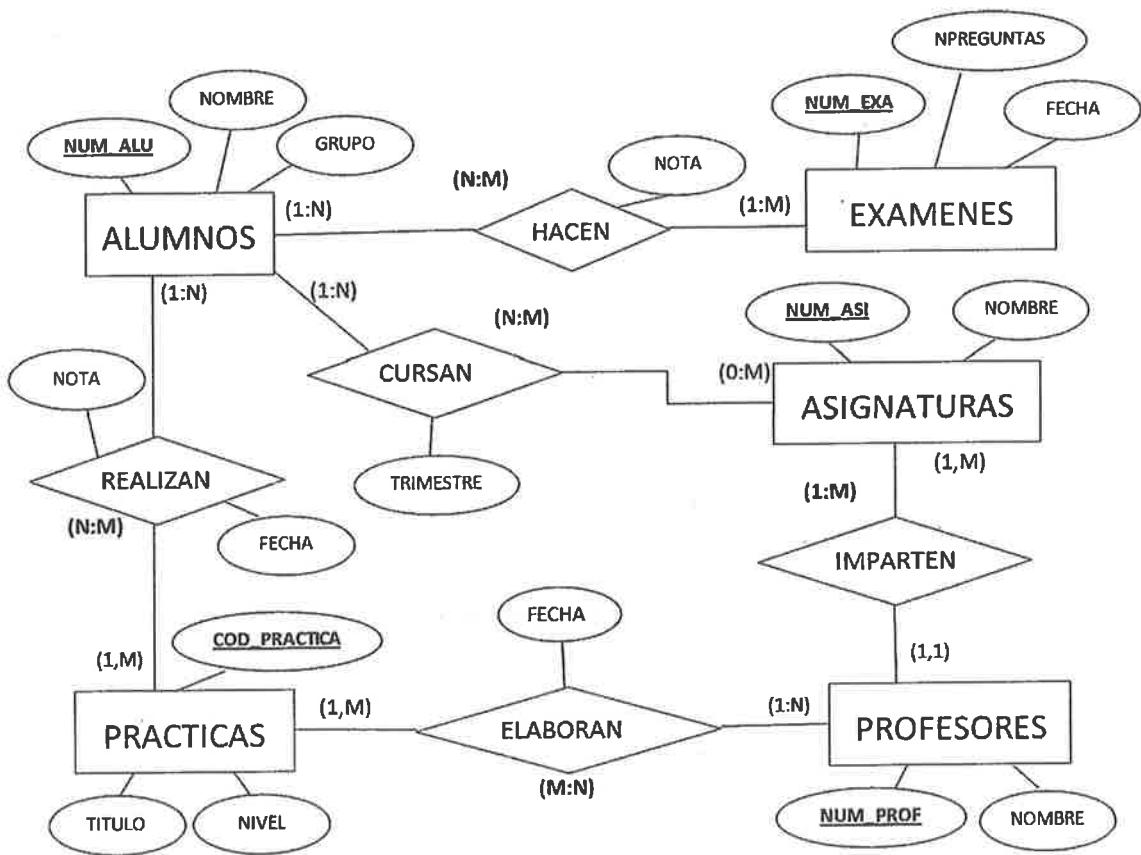


Figura 2.35. Esquema E/R de la Actividad 2.2.

Crea los siguientes dominios y asígnalos a los atributos correspondientes, los dominios se pueden asignar en el modelo lógico o en el relacional:

- Para los atributos: **NOMBRE** y **TITULO**, un dominio de tipo VARCHAR de 30 caracteres.
- Para el atributo **GRUPO**, un dominio de tipo VARCHAR de 8 caracteres.
- Para el atributo **NIVEL**, un dominio de tipo CHAR, de longitud 1 y que tome los valores A, B, C, D, y E.
- Todos los atributos clave tendrán un dominio numérico de 5 dígitos.
- Para los atributos **NOTA** define un dominio cuyo valor esté comprendido entre 0 y 10.
- Para el atributo **NPREGUNTAS** define un dominio numérico de 3 dígitos entre 1 y 500.
- Para el atributo **TRIMESTRE** define un dominio numérico cuyo valor sea 1, 2, 3 y 4.

2.4.2. Plugin ER Master de Eclipse

Otra herramienta de código abierto utilizada para crear modelos de datos es *Eclipse* con su plugin para modelar esquemas E/R **ERMaster**, algunas de las características que **ERMaster** proporciona son las siguientes:

- Soporta todas las bases de datos principales (Oracle, MySQL, DB2, SQLITE, PostgreSQL, SQL Server y HSQLDB)
- Posibilidad de importar la base de datos existente (ingeniería inversa)
- Exportar a varios formatos (Java, Excel, PNG y HTML)
- Generación DDL
- Permite ver los modelos físicos y lógicos

Toda la información del plugin se encuentra en la página: <http://ermaster.sourceforge.net/>. Para instalarlo se abre el menú **Help/Install New Software**, en la ventana que se muestra en *Work with* se teclea <http://ermaster.sourceforge.net/update-site/> se pulsa *Add*, y en el siguiente cuadro de diálogo *OK*. Se localiza el plugin, se seleccionan los elementos (véase la Figura 2.36) y se pulsa *Next*. Se aceptan los términos de la licencia, se pulsa *Finish* y comienza la instalación. Cuando termina pide reiniciar.

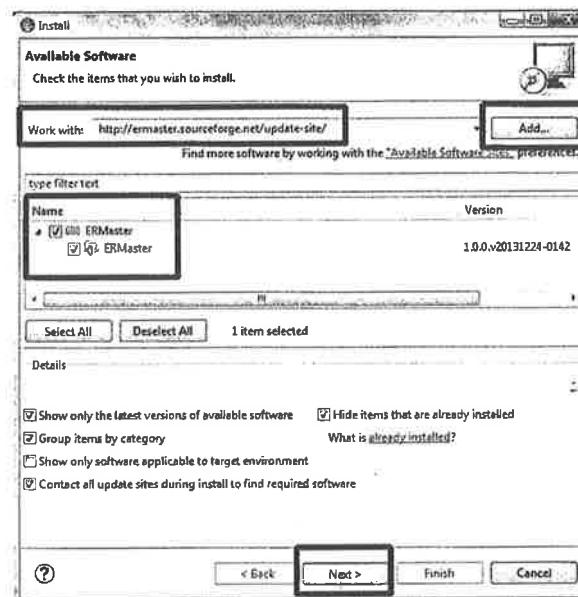


Figura 2.36. Instalación Plugin E/R Master.

2.4.2.1. Creación de modelos de datos con ER Master

Para crear modelos de datos con ER Master, se necesita tener un proyecto creado para almacenarlos, así pues se crea un nuevo proyecto Java, desde el menú **File/New/Java Project**. A continuación se abre el menú **File/New/Other**, y en la ventana que se muestra se elige **ER Master**.

Se pulsa *Next* se escribe el nombre para el modelo y se selecciona la BD donde se va a crear el modelo. Para este ejemplo se selecciona la BD MySQL, estos datos se podrán cambiar cuando se necesite. Seguidamente se pulsa *Finish* y se abre la ventana de diseño. Dentro de la ventana de diseño se puede observar la barra de botones, la paleta de diseño a la izquierda y la vista de edición donde se añadirán las entidades y relaciones, véase la Figura 2.37.



Figura 2.37. Ventana de diseño E/R Master

IMPORTAR ESQUEMAS

Antes de crear nuestro propio modelo de datos veamos cómo se importa el esquema del usuario HR a ER Master. Desde el menú contextual de la vista de edición se selecciona **Importar/Database**, se muestra la ventana de conexión a la base de datos, en ella se escribe el nombre del servidor: *localhost*, el puerto de Oracle, por defecto es *1521*, el nombre de la BD que es *XE*, el nombre y la clave del usuario *HR*. Véase la Figura 2.38.

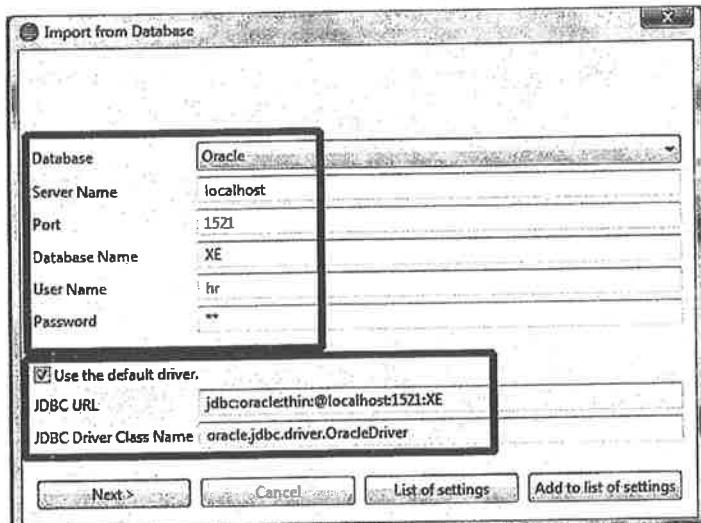


Figura 2.38. Ventana de conexión a Oracle para importar el esquema de HR.

Al pulsar *Next* pide el *path* donde se encuentra el driver de la base de datos para poder realizar la conexión. Es necesario tener los *drivers de la base de datos* para conectarnos, en Oracle se utiliza *classes12.jar*, y en MySQL *mysql-connector-java-5.1.18-bin.jar*, estos archivos o conectores se encuentran en la carpeta de recursos del capítulo.

En la siguiente ventana se selecciona el esquema a importar, en este ejemplo solo el de HR, se pulsa *Next*, y a continuación pide que se seleccionen los objetos a importar, seleccionamos todos los objetos y se pulsa OK, véase la Figura 2.39

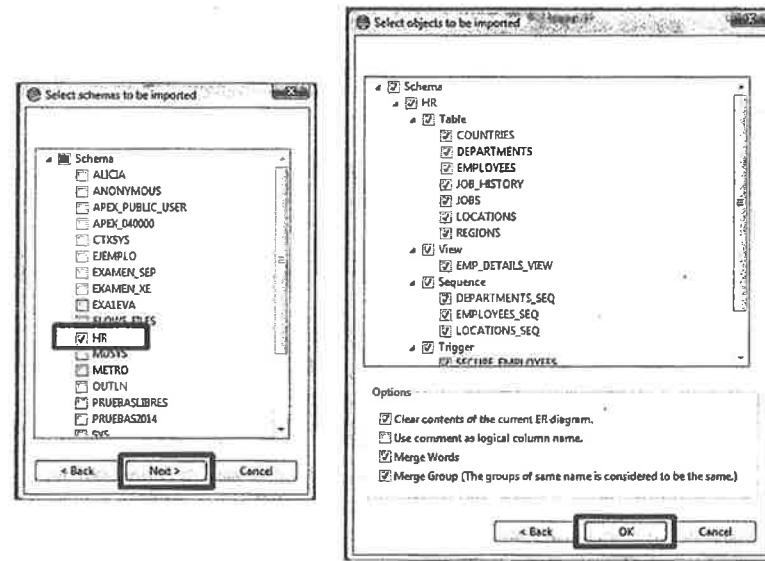


Figura 2.39. Selección de esquema y de objetos a importar.

Finalmente se muestra el esquema de HR, todas las tablas, vistas, si se observa el *Outline* se puede ver también el detalle de los objetos importados, diccionario, tablas vistas, disparadores e índices.

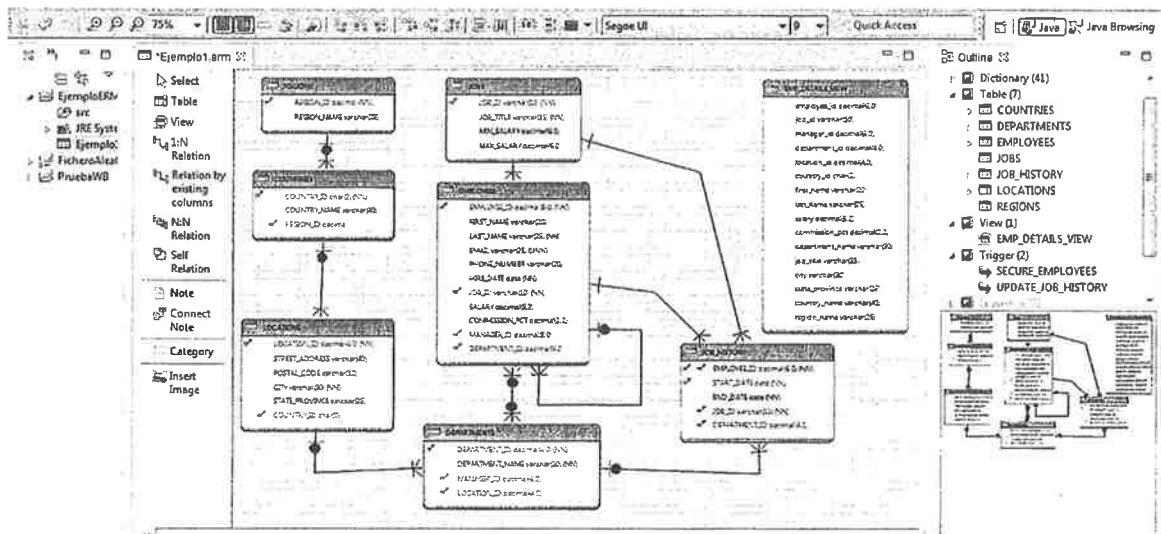


Figura 2.40. Vista del esquema de objetos de HR en Eclipse.

NOTAS: Para cambiar la base de datos asociada a un modelo abrimos el menú contextual sobre la vista de edición y se selecciona *Options*, en la ventana que aparece se podrá seleccionar una nueva base de datos.

NOTAS: También desde este menú contextual se podrá cambiar el modo de visualización desde la opción *Display*, con *View Mode* se cambia la visualización del modelo (modo Lógico, Físico, o ambos). También se puede cambiar la notación (*Notation*) o el diseño (*Design*).

EXPORTAR ESQUEMA A OTRA BD

Una vez creado el esquema este se puede exportar a un archivo DDL, a una hoja de cálculo Excel, a HTML, a una imagen, o a una clase Java entre otras opciones, se hace desde el menú contextual de la vista de edición. Pedirá una carpeta donde almacenar el archivo generado. Normalmente se seleccionará la carpeta del proyecto. Si se exporta a java se selecciona la carpeta *src* que es donde se almacenan las clases por defecto. Al exportar a Java es posible que salga algún error pues las clases se guardan en un paquete. El nombre del paquete arrastra el nombre de la carpeta *src* donde se guardan, se soluciona renombrando el paquete, es decir desde el menú contextual sobre el paquete, se elige *Refactor/Rename* y se elimina el nombre de la carpeta (por ejemplo cambiar *src.Ejemplo* por *Ejemplo*).

ACTIVIDAD 2.3

Exporta el esquema de HR, a las distintas opciones de exportación. Asegúrate de guardar los archivos generados en la carpeta del proyecto.

Guarda la clase java dentro de la carpeta *src* del proyecto. Renombra luego el paquete que se crea.

Si se desea pasar el modelo a otra base de datos y a otro usuario, por ejemplo a MySQL, a la BD prueba, creada al principio de la unidad con dos tablas empleados y departamentos. Se abre el menú contextual sobre la vista de edición y se selecciona *Options*, se elige MySQL. Observa que los tipos de datos cambian al cambiar de base de datos. En Oracle se utiliza VARCHAR2 para las cadenas, en MySQL VARCHAR. En Oracle para los campos numéricos utiliza NUMBER(precisión, escala), MySQL utiliza decimal, integer y otros tipos.

Una vez cambiada la BD se pulsa al botón *Exportar a BD*  de la barra de botones, se muestra la ventana para conectarnos a MySQL, similar a la de Oracle, pero ahora escribiremos en *Server Name* localhos, en *Port* 3306 (puerto asignado a MySQL), *Database Name* es prueba (la base de datos de MySQL donde se crearán las tablas); en *User Name*, escribimos root, y sin *Password* (véase la Figura 2.41). Se marca la casilla *Use the default driver*, se pulsa *Next* y pide donde se encuentra el driver para conectarse, en este caso es *mysql-connector-java-5.1.18-bin.jar*.

¡IMPORTANTE!

Al exportar a la base de datos:

- La BD debe estar creada en MySQL, y en ORACLE debe existir el usuario.
- Se borrará todo lo que hubiera en la BD de MySQL.
- Las BD deben de estar iniciadas.

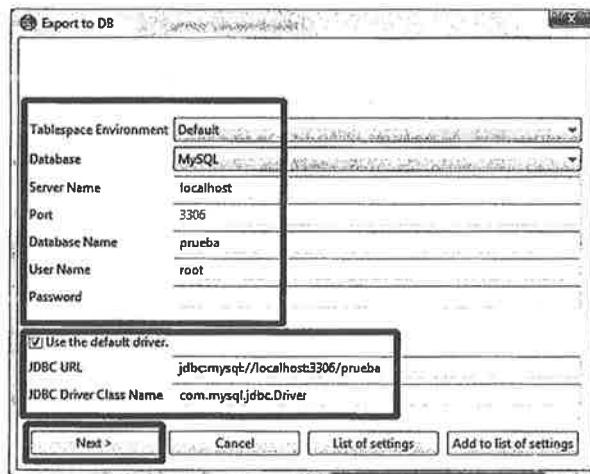


Figura 2.41. Conexión a la BD prueba de MySQL.

Seguidamente se muestra el script de creación de las tablas, este es editable, se pulsa *Execute*, y se crean las tablas en la BD. Aparecerá un error en la creación de los trigger y de la vista, no se realiza bien la exportación, pues bien, se lanza de nuevo la exportación, se suprimen las líneas de los apartados /* *Create Triggers* */ y /* *Create Views* */ y se ejecuta de nuevo el script.

Para ver si se han creado bien las tablas, abrimos el administrador de MySQL: <http://localhost/phpmyadmin/>, seleccionamos la BD y comprobamos que se han creado las tablas y sus relaciones (desde el menú *Más/Diseñador*) Véase la Figura 2.42.

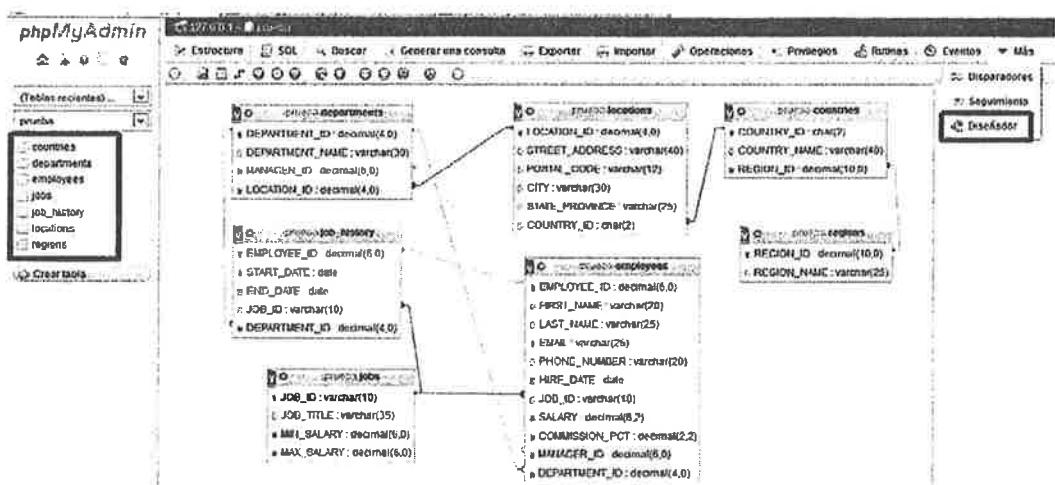


Figura 2.42. BD exportada a MySQL.

NOTA: Al generar el script de creación de tablas en MySQL de un esquema exportado de ORACLE, si las restricciones ORACLE tienen nombre que incluyen el carácter guión bajo (por ejemplo PK_EMPLE), este no se exporta y los nombres de restricciones darán error al ejecutar el script, habría que corregir el script o no poner guión bajo en los nombres de restricciones.

CREAR MODELOS

A continuación se crea el modelo representado en la Figura 2.20. Desde el menú *File/New/Other* seleccionamos *ER Master*, seleccionamos la carpeta del proyecto de trabajo, le damos un nombre y elegimos la BD, por ejemplo MySQL.

A diferencia de Data Modeler con ER Master no se pueden crear jerarquías (caso de EMPLEADOS, ASESORES y NORMALES), se necesita saber con antelación la transformación que se hará de la jerarquía, así pues la jerarquía se representará como una tabla con todos los atributos, más el atributo *Tipo* para distinguir el tipo de empleados.

Para crear las tablas y relaciones pulsaremos los botones de la *Paleta de Diseño*: *Create Table* para crear tablas; *Create Relation 1:N*, para crear relaciones de este tipo, primero se marca la tabla origen de donde sale el 1; *Create relation by existing columns*, para crear relaciones que llevan asociadas columnas; *Create N:N Relation*, para relaciones M:M; y *Self Relation* para relaciones reflexivas. Además se podrán añadir notas e imágenes entre otros elementos.

Al crear una tabla se muestra la ventana de *Table Information*, desde donde se podrá ir añadiendo campos a la tabla.

Lo primero es escribir el nombre a la tabla y para añadir los atributos se pulsa el botón *Add*, seguidamente se abre la ventana de añadir un atributo (véase la Figura 2.43) en esta ventana se indicará si el atributo va a ser la clave principal (*Primary Key*), o si no podrá contener valor nulo (*NOT NULL*), o si no se podrá repetir su valor (*UNIQUE*), o si la columna va a ser auto incrementable (*AUTO_INCREMENT*). En ella se indicará el nombre de la columna, tanto el nombre físico como el lógico y el tipo de dato en el desplegable *Type*, donde aparecerán los tipos de datos válidos para la BD seleccionada.

NOTA: Para los ejercicios se utilizarán los tipos *numeric(precisión)* para los numéricos sin decimales, *numeric(precisión, escala)* para numéricos con decimales, *date* para los tipo fecha, *Varchar(longitud)* para las cadenas de longitud variable, y *char(longitud)* para las cadenas de longitud fija.

Una vez creados los atributos de la tabla (véase la Figura 2.44) podremos movernos por las distintas pestañas para obtener más información de la tabla. Se pulsa *OK*, y se vuelve a la ventana de diseño. Repetimos la operación con el resto de las tablas.

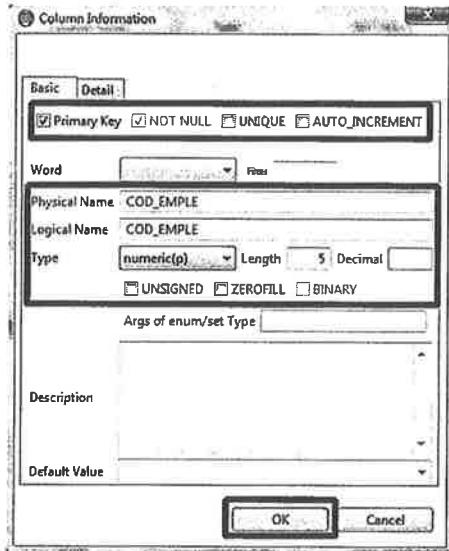


Figura 2.43. Creación de un atributo de la tabla, propiedades.

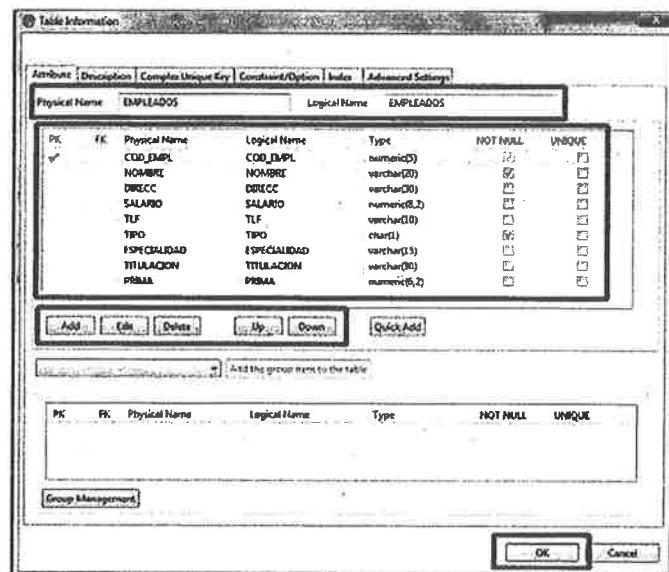


Figura 2.44. Creación de una tabla, propiedades.

Una vez creadas las entidades (tablas) se crean las relaciones. Se selecciona la relación de la paleta y se hace clic en la tabla origen, hasta la tabla destino. Se crea la relación y si se hace doble clic sobre ella se abre la ventana de información de la relación, véase la Figura 2.45, en ella se pone el nombre de la relación, se indica cómo se desea la actualización (ON UPDATE) o el borrado de registros (ON DELETE): *RESTRICT* (no actualizar o borrar si hay registros relacionados), *CASCADE* (borrar o actualizar en cascada, si se elige esta opción en el borrado, se borrarán todos los registros relacionados cuando se borre un registro padre), *NO ACTION* (equivalente a *RESTRICT*), *SET NULL* (asigna null en las columnas de las tablas relacionadas cuando se borre o actualice el registro padre) o *SET DEFAULT* (asigna el valor por defecto si las columnas lo tuviesen). En *Multiplicity* se añaden las cardinalidades, del origen al destino.

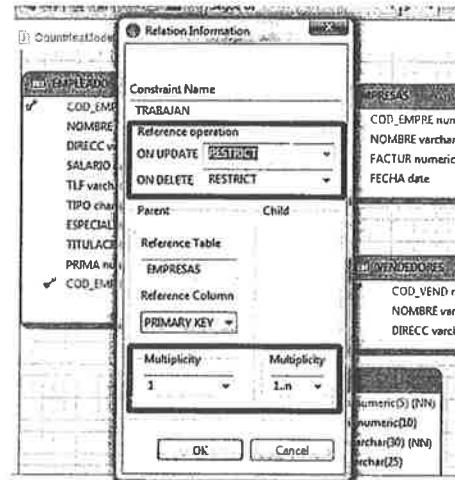


Figura 2.45. Propiedades de la relación TRABAJAN.

Observa que al crear las relaciones automáticamente se propagan las claves a la tabla destino. Observa también que en la relación M:M, automáticamente se crea la tabla resultante de la propagación de claves de las dos tablas que relaciona. En este caso a veces hay que modificar las cardinalidades de estas relaciones, en el ejemplo hay que indicar (0:M). En la Figura 2.46 se muestra el modelo final.

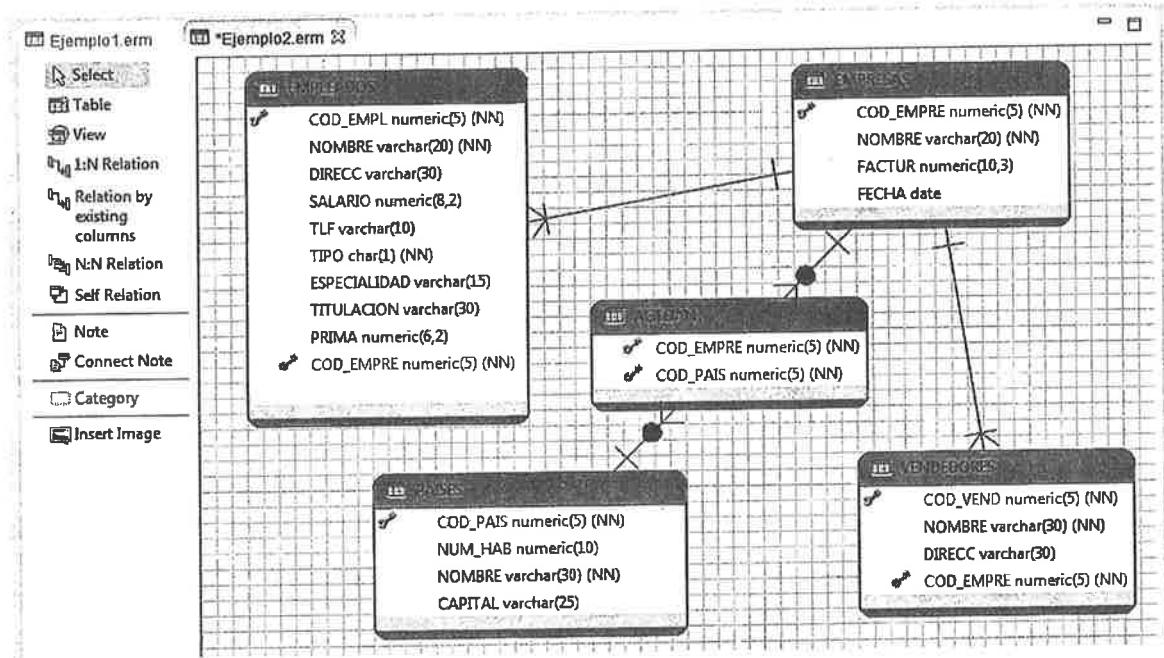


Figura 2.46. Modelo Relacional del ejemplo de la Figura 2.20.

El siguiente paso es crear este modelo de datos dentro de una base de datos MySQL, en este caso se pulsa al botón *Exportar a BD* y se procede como en los apartados anteriores.

ACTIVIDAD 2.4

- Crea en MySQL una nueva BD y exporta el modelo realizado a esa BD creada.
- Expórtalo también a Oracle al esquema de HR.
- Realiza con ER Master el modelo de datos que se representa en la Figura 2.35. Exporta el modelo a Oracle y crea una nueva BD en MySQL para exportarlo a MySQL.

2.5. COMPARACIÓN DE ENTORNOS DE DESARROLLO

Actualmente en el mercado existen multitud de entornos de desarrollo, uno de los más utilizados es ECLIPSE, sobre todo por ser gratuito y por contar con una gran cantidad de plugins gratuitos que van a facilitar cualquier tipo de desarrollo.

Las grandes empresas proponen sus soluciones comerciales de entornos de desarrollo y dependiendo de la empresa, del país, del proyecto a realizar, del lenguaje de programación a utilizar, del tipo de licencia con la que se quiera trabajar (libre o propietaria), de la vinculación de las empresas con las marcas se trabajará con unos entornos o con otros, así por ejemplo empresas vinculadas con IBM trabajarán con entornos ***IBM Rational Application Developer (RAD)*** es un entorno de desarrollo integrado creado por la división *Rational Software* de *IBM* para el diseño visual, construcción, pruebas y despliegue de servicios web, portales y aplicaciones JEE (fuente wikipedia).

Empresas vinculadas a ORACLE trabajarán probablemente con las herramientas de desarrollo Oracle como puede ser ***JDeveloper***, es un entorno de desarrollo integrado desarrollado por Oracle Corporation para los lenguajes Java, XML, SQL, PL/SQL, Javascript, PHP, Oracle ADF, UML y otros. Este software es propietario pero gratuito desde 2005 (fuente wikipedia).

Empresas vinculadas a Microsoft o proyectos que requieran las funciones de Microsoft utilizarán la herramienta ***Microsoft Visual Studio***, este es un entorno de desarrollo integrado para sistemas operativos Windows. Soporta múltiples lenguajes de programación tales como C++, C#, Visual Basic .NET, etc.). Se pueden descargar versiones de prueba de 90 días.

Existen otros entornos gratuitos como puede ser ***NetBeans*** hecho principalmente para el lenguaje de programación Java. Existe además un número importante de módulos para extender sus funciones. NetBeans IDE es un producto libre y gratuito sin restricciones de uso. Actualmente es un producto de *Sun Microsystems/Oracle Corporation* (fuente wikipedia).

ACTIVIDAD 2.5

Busca en internet distintos entornos de desarrollo y clasificalos según las siguientes características:

IDE	Plataforma (Windows, Mac, Linux,...)	Lenguajes que soporta	Libres o propietarios	Tipo de Aplicaciones	Fabricante o empresa
Eclipse					
NetBeans					
IntelliJ IDEA					
JBuilder					
Bluefish					
Visual C++					
MS Visual Studio					
Clarion					
Anjuta					
KDevelop					
JDeveloper					
Dreamweaver					

COMPRUEBA TU APRENDIZAJE

1. ¿Qué es un entorno de desarrollo?
2. ¿Qué componentes tiene un entorno de desarrollo?
3. ¿Qué es Eclipse? ¿Qué se necesita para su instalación?
4. ¿Qué son los plugins? ¿Cómo se instalan?
5. ¿Qué es SQL Developer?
6. ¿Qué es Data Modeler?
7. ¿Cómo copiarías las tablas de un usuario ORACLE a una base de datos MySQL, utilizando ER Master?
8. ¿Es posible generar clases Java asociadas a las tablas de un modelo de datos? ¿Cómo se haría?

ACTIVIDADES DE AMPLIACIÓN

1. Realiza la siguiente interfaz (Figura 2.47) utilizando *WindowBuilder* de Eclipse. Escribe tu nombre donde se indica. Añade el código para mostrar en consola un mensaje cuando se pulsa cada uno de los botones.

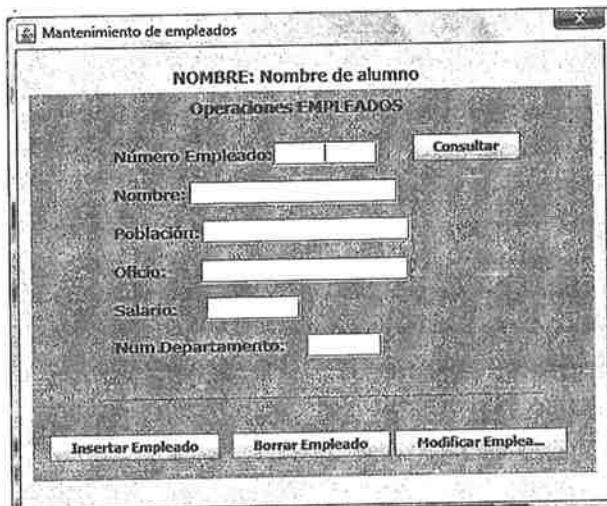


Figura 2.47. Ventana de Ejercicio 1.

2. Pasa el esquema E/R de la Figura 2.48 a un modelo lógico con Data Modeler.

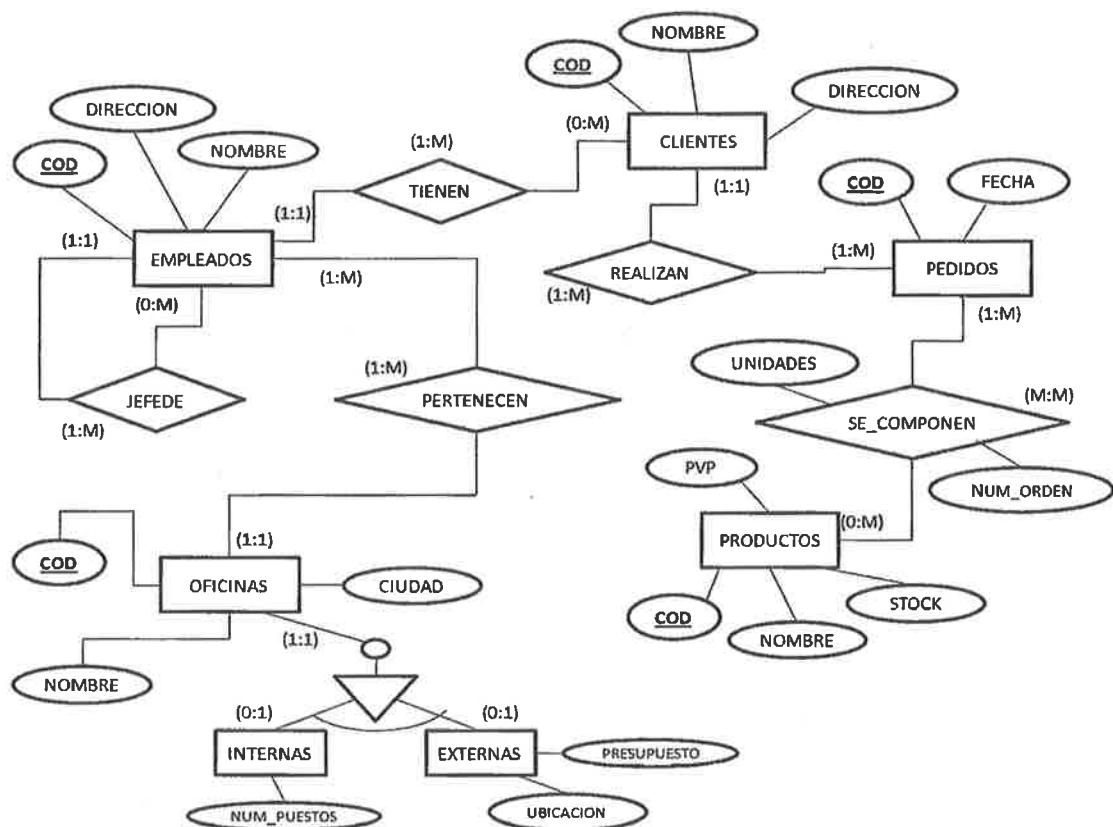


Figura 2.48. Modelo E/R del Ejercicio 2.

3. Realiza la ingeniería al modelo relacional y genera el script DDL. Ejecuta el script en el esquema de HR.
4. Crea un nuevo proyecto con Eclipse, crea un modelo con ER Master e importa las tablas creadas en el esquema de HR del ejercicio anterior a este modelo.
5. Crea una BD en MySQL y exporta el modelo creado en el ejercicio 4 a esta base de datos.
6. Exporta el modelo creado a HTML y a Java, corrige posibles errores en la generación de las clases java.
7. Crea dentro del proyecto Eclipse un nuevo modelo con ER Master y pasa el esquema E/R de la Figura 2.49 a un modelo lógico. Crea una base de datos MySQL y exporta a esa BD el modelo creado.

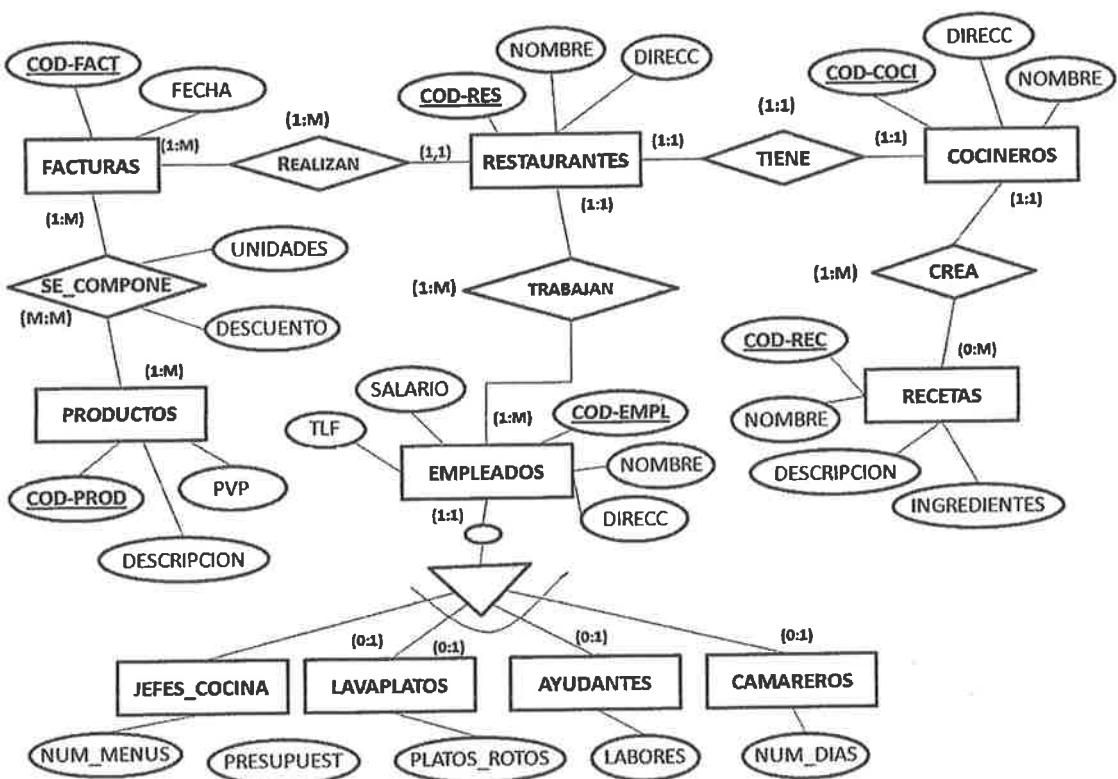


Figura 2.49. Modelo E/R del Ejercicio 7.

CAPÍTULO 3

DISEÑO Y REALIZACIÓN DE PRUEBAS

CONTENIDOS

- Técnicas de diseño de casos de prueba
- Estrategias de prueba del software
- Documentación para las pruebas
- Pruebas de código
- Herramientas de depuración
- Pruebas unitarias con JUNIT

OBJETIVOS

- Identificar los tipos y estrategia de pruebas
- Definir casos de prueba
- Utilizar herramientas de depuración en un entorno de desarrollo
- Realizar pruebas unitarias utilizando la herramienta JUNIT
- Implementar pruebas automáticas

RESUMEN DEL CAPÍTULO

En este capítulo aprenderemos a utilizar diferentes técnicas para elaborar casos de prueba. Usaremos una herramienta de depuración definiendo puntos de ruptura y examinando variables durante la ejecución de un programa. Aprenderemos a utilizar la herramienta JUNIT para elaborar pruebas unitarias para clases Java

3.1. INTRODUCCIÓN

Las pruebas de software consisten en verificar y validar un producto software antes de su puesta en marcha. Constituyen una de las etapas del desarrollo de software, y básicamente consiste en probar la aplicación construida. Se integran dentro de las diferentes fases del ciclo de vida del software dentro de la ingeniería de software.

La ejecución de pruebas de un sistema involucra una serie de etapas que se nombraron en el Capítulo 1: planificación de pruebas, diseño y construcción de los casos de prueba, definición de los procedimientos de prueba, ejecución de las pruebas, registro de resultados obtenidos, registro de errores encontrados, depuración de los errores e informe de los resultados obtenidos.

En este Capítulo se estudian dos enfoques para el diseño de casos de prueba y diferentes técnicas en cada uno para probar el código de los programas.

3.2. TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA

Un **caso de prueba** es un conjunto de entradas, condiciones de ejecución y resultados esperados, desarrollado para conseguir un objetivo particular o condición de prueba. Para llevar a cabo un caso de prueba, es necesario definir las precondiciones y post condiciones, identificar unos valores de entrada y conocer el comportamiento que debería tener el sistema ante dichos valores. Tras realizar ese análisis e introducir dichos datos en el sistema, se observa si su comportamiento es el previsto o no y por qué. De esta forma se determina si el sistema ha pasado o no la prueba¹.

Para llevar a cabo el diseño de casos de prueba se utilizan dos técnicas o enfoques: **prueba de caja blanca** y **prueba de caja negra** (véase Figura 3.1). Las primeras se centran en validar la estructura interna del programa (necesitan conocer los detalles procedimentales del código) y las segundas se centran en validar los requisitos funcionales sin fijarse en el funcionamiento interno del programa (necesitan saber la funcionalidad que el código ha de proporcionar). Estas pruebas no son excluyentes y se pueden combinar para descubrir diferentes tipos de errores.

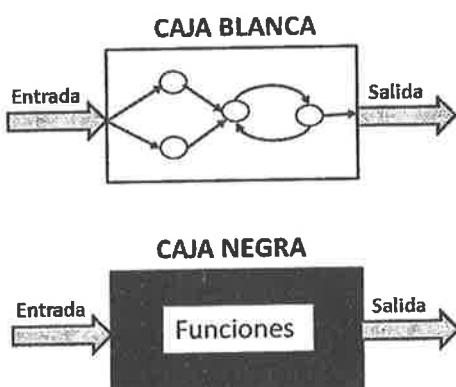


Figura 3.1. Pruebas de caja blanca y negra.

¹ GUÍA DE VALIDACIÓN Y VERIFICACIÓN. Inteco. Laboratorio Nacional de Calidad del Software.

3.2.1. Pruebas de caja blanca

También se las conoce como *pruebas estructurales* o *de caja de cristal*. Se basan en el minucioso examen de los detalles procedimentales del código de la aplicación. Mediante esta técnica se pueden obtener casos de prueba que:

- Garanticen que se ejecutan al menos una vez todos los caminos independientes de cada módulo.
- Ejecuten todas las sentencias al menos una vez.
- Ejecuten todas las decisiones lógicas en su parte verdadera y en su parte falsa.
- Ejecuten todos los bucles en sus límites.
- Utilicen todas las estructuras de datos internas para asegurar su validez.

Una de las técnicas utilizadas para desarrollar los casos de prueba de caja blanca es la prueba del camino básico que se estudiará más adelante.

3.2.2. Pruebas de caja negra

Estas pruebas se llevan a cabo sobre la interfaz del software, no hace falta conocer la estructura interna del programa ni su funcionamiento. Se pretende obtener casos de prueba que demuestren que las funciones del software son operativas, es decir, que las salidas que devuelve la aplicación son las esperadas en función de las entradas que se proporcionen.

A este tipo de pruebas también se les llama *prueba de comportamiento*. El sistema se considera como una caja negra cuyo comportamiento solo se puede determinar estudiando las entradas y las salidas que devuelve en función de las entradas suministradas.

Con este tipo de pruebas se intenta encontrar errores de las siguientes categorías:

- Funcionalidades incorrectas o ausentes.
- Errores de interfaz.
- Errores en estructuras de datos o en accesos a bases de datos externas.
- Errores de rendimiento.
- Errores de inicialización y finalización.

Existen diferentes técnicas para confeccionar los casos de prueba de caja negra, algunos son: clases de equivalencia, análisis de valores límite, métodos basados en grafos, pruebas de comparación, etc. En este capítulo se estudiarán algunas de estas técnicas.

3.3. ESTRATEGIAS DE PRUEBAS DEL SOFTWARE

La estrategia de prueba del software se puede ver en el contexto de una espiral (véase Figura 3.2):

- En el vértice de la espiral comienza la **prueba de unidad**. Se centra en la unidad más pequeña de software, el módulo tal como está implementado en código fuente.
- La prueba avanza para llegar a la **prueba de integración**. Se toman los módulos probados mediante la prueba de unidad y se construye una estructura de programa que esté de acuerdo con lo que dicta el diseño. El foco de atención es el diseño.

- La espiral avanza llegando a la **prueba de validación** (o de aceptación). Prueba del software en el entorno real de trabajo con intervención del usuario final. Se validan los requisitos establecidos como parte del análisis de requisitos del software, comparándolos con el sistema que ha sido construido.
- Finalmente se llega a la **prueba del sistema**. Verifica que cada elemento encaja de forma adecuada y se alcanza la funcionalidad y rendimiento total. Se prueba como un todo el software y otros elementos del sistema.

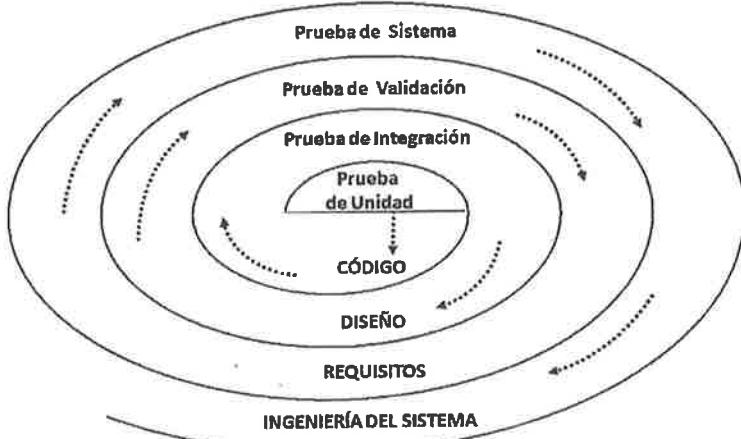


Figura 3.2. Estrategia de prueba.

3.3.1. Prueba de unidad

En este nivel se prueba cada unidad o módulo con el objetivo de eliminar errores en la interfaz y en la lógica interna. Esta actividad utiliza técnicas de caja negra y caja blanca, según convenga para lo que se desea probar. Se realizan pruebas sobre (véase Figura 3.3):

- La interfaz del módulo, para asegurar que la información fluye adecuadamente.
- Las estructuras de datos locales, para asegurar que mantienen su integridad durante todos los pasos de ejecución del programa.
- Las condiciones límite, para asegurar que funciona correctamente en los límites establecidos durante el proceso.
- Todos los caminos independientes de la estructura de control, con el fin de asegurar que todas las sentencias se ejecutan al menos una vez.
- Todos los caminos de manejo de errores.

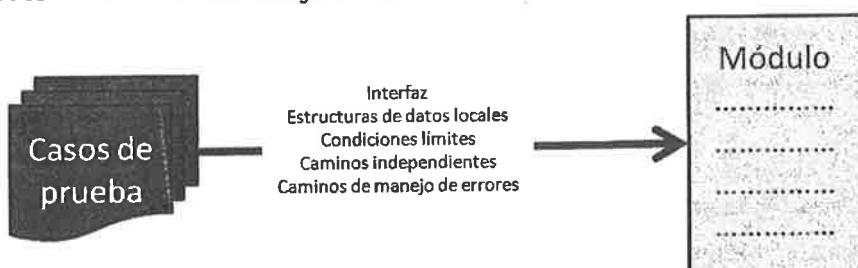


Figura 3.3. Prueba de unidad.

Algunas de las herramientas que se utilizan para pruebas unitarias son: JUnit, CPPUnit, PHPUnit, etc.

3.3.2. Prueba de integración

En este tipo de prueba se observa cómo interactúan los distintos módulos. Se podría pensar que esta prueba no es necesaria, ya que si todos los módulos funcionan por separado, también deberían funcionar juntos. Realmente el problema está aquí, en comprobar si funcionan juntos.

Existen dos enfoques fundamentales para llevar a cabo las pruebas:

- **Integración no incremental o *big bang*.** Se prueba cada módulo por separado y luego se combinan todos de una vez y se prueba todo el programa completo. En este enfoque se encuentran gran cantidad de errores y la corrección se hace difícil.
- **Integración incremental.** El programa completo se va construyendo y probando en pequeños segmentos, en este caso los errores son más fáciles de localizar. Se dan dos estrategias *Ascendente* y *Descendente*. En la integración *Ascendente* la construcción y prueba del programa empieza desde los módulos de los niveles más bajos de la estructura del programa. En la *Descendente* la integración comienza en el módulo principal (programa principal) moviéndose hacia abajo por la jerarquía de control.

La Figura 3.4 representa varios módulos y la interconexión entre ellos. El módulo principal es el que está en la raíz, M1. La figura muestra una estrategia de integración *Ascendente*. Se empieza probando los módulos de más bajo nivel en la jerarquía modular del sistema y se procede a probar la integración de abajo hacia arriba hasta llegar al programa principal M1.

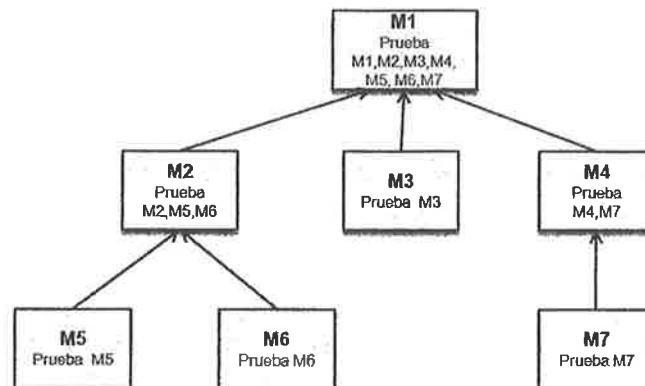


Figura 3.4. Prueba de integración *Ascendente*.

3.3.3. Prueba de validación

La validación se consigue cuando el software funciona de acuerdo con las expectativas razonables del cliente definidas en el documento de especificación de requisitos del software o ERS. Se llevan a cabo una serie de pruebas de caja negra que demuestran la conformidad con los requisitos. Las técnicas a utilizar son:

- **Prueba Alfa.** Se lleva a cabo por el cliente o usuario en el lugar de desarrollo. El cliente utiliza el software de forma natural bajo la observación del desarrollador que irá registrando los errores y problemas de uso.
- **Prueba Beta.** Se lleva a cabo por los usuarios finales del software en su lugar de trabajo. El desarrollador no está presente. El usuario registra todos los problemas que

encuentra, reales y/o imaginarios, e informa al desarrollador en los intervalos definidos en el plan de prueba. Como resultado de los problemas informados el desarrollador lleva a cabo las modificaciones y prepara una nueva versión del producto.

3.3.4. Prueba del sistema

La prueba del sistema está formada por un conjunto de pruebas cuya misión es ejercitar profundamente el software. Son las siguientes:

- **Prueba de recuperación.** En este tipo de prueba se fuerza el fallo del software y se verifica que la recuperación se lleva a cabo apropiadamente.
- **Prueba de seguridad.** Esta prueba intenta verificar que el sistema está protegido contra accesos ilegales.
- **Prueba de resistencia (Stress).** Trata de enfrentar el sistema con situaciones que demandan gran cantidad de recursos, por ejemplo, diseñando casos de prueba que requieran el máximo de memoria, incrementando la frecuencia de datos de entrada, que den problemas en un sistema operativo virtual, etc.

3.4. DOCUMENTACIÓN PARA LAS PRUEBAS

El estándar IEEE 829-1998 describe el conjunto de documentos que pueden producirse durante el proceso de prueba. Son los siguientes:

- **Plan de Pruebas.** Describe el alcance, el enfoque, los recursos y el calendario de las actividades de prueba. Identifica los elementos a probar, las características que se van a probar, las tareas que se van a realizar, el personal responsable de cada tarea y los riesgos asociados al plan.
- **Especificaciones de prueba.** Están cubiertas por tres tipos de documentos: la especificación del diseño de la prueba (se identifican los requisitos, casos de prueba y procedimientos de prueba necesarios para llevar a cabo las pruebas y se especifica la función de los criterios de pasa/no-pasa), la especificación de los casos de prueba (documenta los valores reales utilizados para la entrada, junto con los resultados previstos), y la especificación de los procedimientos de prueba (donde se identifican los pasos necesarios para hacer funcionar el sistema y ejecutar los casos de prueba especificados).
- **Informes de pruebas.** Se definen cuatro tipos de documentos: un informe que identifica los elementos que están siendo probados, un registro de las pruebas (donde se registra lo que ocurre durante la ejecución de la prueba), un informe de incidentes de prueba (describe cualquier evento que se produce durante la ejecución de la prueba que requiere mayor investigación) y un informe resumen de las actividades de prueba.

3.5. PRUEBAS DE CÓDIGO

La prueba del código consiste en la ejecución del programa (o parte de él) con el objetivo de encontrar errores. Se parte para su ejecución de un conjunto de entradas y una serie de condiciones de ejecución; se observan y registran los resultados y se comparan con los resultados esperados. Se observará si el comportamiento del programa es el previsto o no y por qué.

Para las pruebas de código se van a mostrar diferentes técnicas que dependerán del tipo de enfoque utilizado: de caja blanca, se centran en la estructura interna del programa; o de caja negra, más centrado en las funciones, entradas y salidas del programa.

3.5.1. Prueba del camino básico

La prueba del camino básico es una técnica de prueba de caja blanca que permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución. Los casos de prueba obtenidos del conjunto básico garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa².

Para la obtención de la medida de la complejidad lógica (o *complejidad ciclomática*) emplearemos una representación del flujo de control denominada *grafo de flujo* o *grafo del programa*.

NOTACIÓN DE GRAFO DE FLUJO

El grafo de flujo de las estructuras de control se representa de la siguiente forma:

ESTRUCTURA	GRAFO DE FLUJO
SECUENCIAL Instrucción 1 Instrucción 2 <u>Instrucción n</u>	
CONDICIONAL Si <condición> Entonces <instrucciones> Si no <instrucciones> Fin si	
HACER MIENTRAS Mientras <condición> Hacer <instrucciones> Fin mientras	
	REPETIR HASTA Repetir <instrucciones> Hasta que <condición>

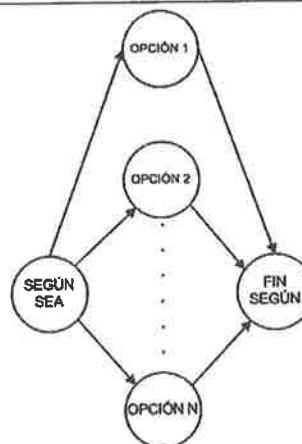
² Ingeniería del software. Un enfoque práctico. Roger S. Pressman.

CONDICIONAL MÚLTIPLE

```

Según sea <variable> Hacer
  Caso opción 1:
    <Instrucciones>
  Caso opción 2:
    <Instrucciones>
  Caso opción 3:
    <Instrucciones>
  Otro caso:
    <Instrucciones>
Fin según

```



Donde cada círculo representa una o más sentencias, sin bifurcaciones, en pseudocódigo o código fuente. A continuación construimos un grafo de flujo a partir de un diagrama de flujo (Figura 3.5).

Ejemplo 1: se muestra el diagrama de flujo y el grafo de flujo para un programa que lee 10 números de teclado y muestra cuántos de los números leídos son pares y cuántos son impares. Para comprobar si el número es par o impar utilizamos el operador % de Java (devuelve el resto de la división) que devuelve 0 si es par. La estructura principal corresponde a un MIENTRAS (o WHILE) y dentro hay una estructura SI (o IF). Véase Figura 3.5.

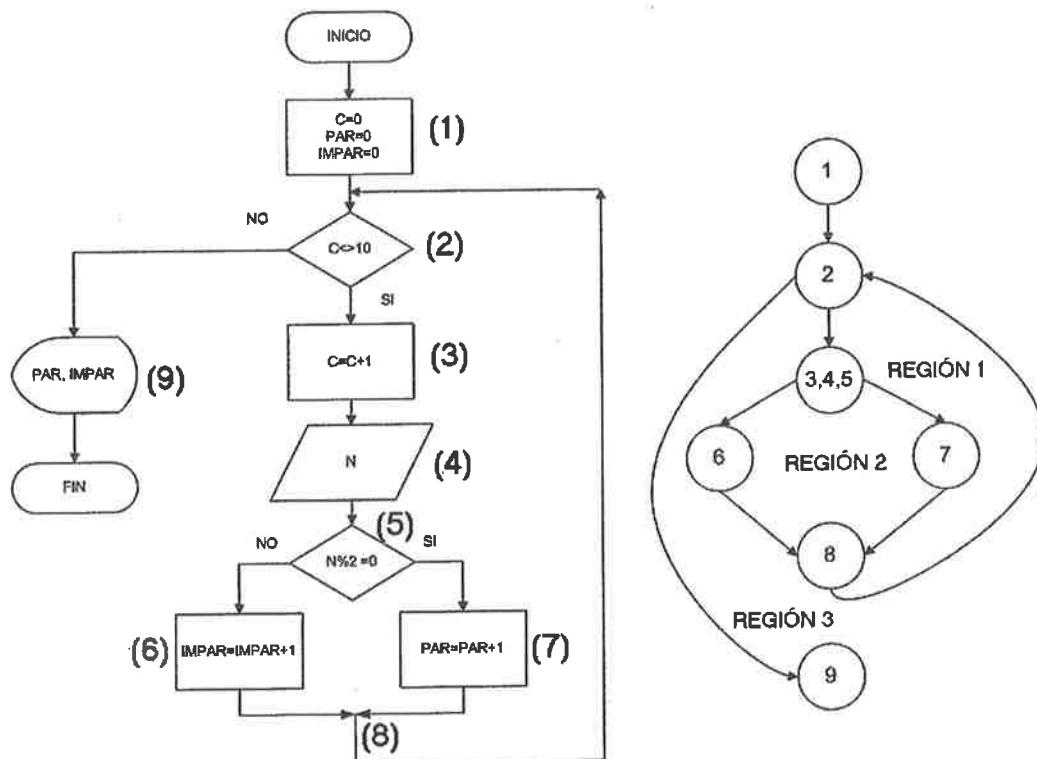


Figura 3.5. Diagrama de flujo y grafo de flujo.

Se numeran en el diagrama de flujo cada uno de los símbolos, y los finales de las estructuras de control (por ejemplo, el (9) es el final del WHILE) aunque no tengan ningún símbolo (por ejemplo, el número (8) es el final de una estructura condicional).

Cada círculo del grafo de flujo se llama *nodo*. Representa una o más sentencias procedimentales. Un solo nodo se puede corresponder con una secuencia de símbolos del proceso y un rombo de decisión. Un ejemplo es el nodo numerado como 3, 4, 5.

Las flechas del grafo de flujo se denominan *aristas* o *enlaces* y representan el flujo de control, como en el diagrama de flujo. Una arista termina en un nodo, aunque el nodo no tenga ninguna sentencia procedimental; es el caso del nodo numerado como 8.

Las áreas delimitadas por aristas y nodos se llaman *regiones*, el área exterior del grafo es otra región más. En el ejemplo se muestran 3 regiones, 8 aristas y 7 nodos.

El nodo que contiene una condición se llama *nodo predicado* y se caracteriza porque de él salen dos o más aristas. En el ejemplo se muestran 2 nodos predicado, el representado por el número 2 y el representado por 3, 4, 5. Únicamente de estos nodos pueden salir dos aristas.

Ejemplo 2: se dispone de un fichero de Alumnos con la siguiente estructura de registro: Curso, Nombre, Sexo (puede ser H o M) y Nota. El fichero está ordenado ascendente por curso. Vamos a realizar un proceso que lea los registros del fichero y muestre por cada curso el número de hombres y el número de mujeres. Se construirá el pseudocódigo y el grafo de flujo, véase Figura 3.6. En el pseudocódigo se muestran en un cuadrito las sentencias que se representan en un nodo; al lado hay un número entre paréntesis que se corresponde con su nodo en el grafo de flujo (recuerda que cada nodo puede corresponder a una o más sentencias de proceso y una condición). La estructura principal corresponde a un MIENTRAS (mientras haya registros en el fichero), dentro hay otro MIENTRAS (para tratar el mismo curso siempre y cuando haya registros) y dentro hay una estructura SI (para contar los hombres y mujeres). En este segundo ejemplo se muestran 5 regiones, 14 aristas y 11 nodos, 4 de ellos son nodos predicado.

Cuando en un diseño procedural se encuentran condiciones compuestas, es decir cuando en una condición aparecen uno o más operadores (en el ejemplo anterior hay una condición compuesta: *Haya registros y Mismo curso*) se complica la generación del grafo de flujo. En este caso se crea un nodo aparte para cada una de las condiciones; nodos 4 y 5 del ejemplo. La Figura 3.7 muestra un ejemplo de lógica compuesta:

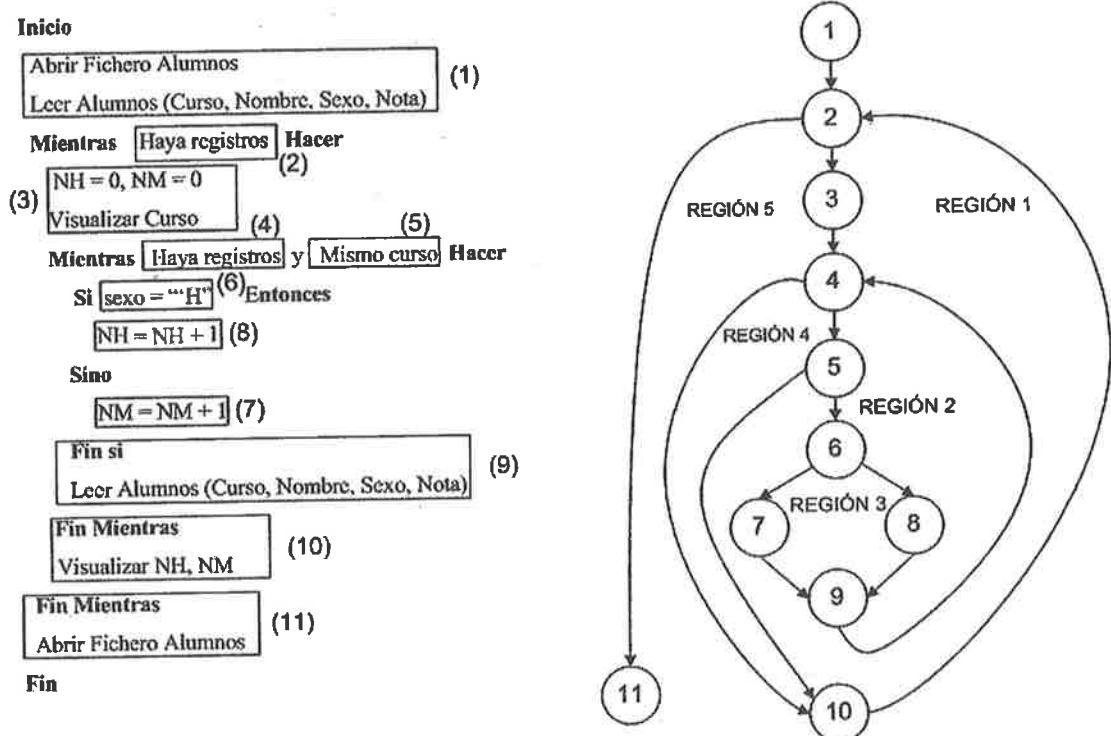


Figura 3.6. Pseudocódigo y grafo de flujo.

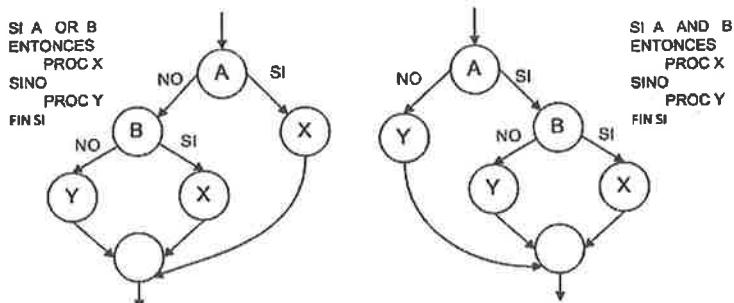


Figura 3.7. Lógica compuesta.

COMPLEJIDAD CICLOMÁTICA

La complejidad ciclomática es una métrica del software que proporciona una medida cuantitativa de la complejidad lógica de un programa³. En el contexto del método de prueba del camino básico, la complejidad ciclomática establece el número de caminos independientes del conjunto básico de caminos de ejecución de un programa, y por lo tanto, el número de casos de prueba que se deben ejecutar para asegurar que cada sentencia se ejecuta al menos una vez.

La complejidad ciclomática $V(G)$ puede calcularse de tres formas:

³ Ingeniería del software. Un enfoque práctico. Roger S. Pressman.

1. $V(G) = \text{Número de regiones del grafo.}$
2. $V(G) = \text{Aristas} - \text{Nodos} + 2.$
3. $V(G) = \text{Nodos predicado} + 1$

Para el *Ejemplo 1*, la complejidad ciclomática es 3.

1. $V(G) = \text{Número de regiones del grafo} = 3.$
2. $V(G) = \text{Aristas} - \text{Nodos} + 2 = 8 - 7 + 2 = 3.$
3. $V(G) = \text{Nodos predicado} + 1 = 2 + 1 = 3$

Para el *Ejemplo 2*, la complejidad ciclomática es 5.

1. $V(G) = \text{Número de regiones del grafo} = 5.$
2. $V(G) = \text{Aristas} - \text{Nodos} + 2 = 14 - 11 + 2 = 5$
3. $V(G) = \text{Nodos predicado} + 1 = 4 + 1 = 5$
- 4.

Se establecen los siguientes valores de referencia de la complejidad ciclomática:

Complejidad ciclomática	Evaluación de riesgo
Entre 1 y 10	Programas o métodos sencillos, sin mucho riesgo.
Entre 11 y 20	Programas o métodos más complejos, riesgo moderado.
Entre 21 y 50	Programas o métodos complejos, alto riesgo.
Mayor que 50	Programas o métodos no testeables, muy alto riesgo.

El valor de $V(G)$ nos da el número de caminos independientes del conjunto básico de un programa. Un camino independiente es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una condición. En términos del diagrama de flujo, un camino independiente está constituido por lo menos por una arista que no haya sido recorrida anteriormente a la definición del camino³.

Para el *Ejemplo 1*, un conjunto de caminos independientes será:

- Camino 1: 1 – 2 – 9
- Camino 2: 1 – 2 – 3, 4, 5 – 6 – 8 – 2 – 9
- Camino 3: 1 – 2 – 3, 4, 5 – 7 – 8 – 2 – 9

Para el *Ejemplo 2*, un conjunto de caminos independientes será:

- Camino 1: 1 – 2 – 11
- Camino 2: 1 – 2 – 3 – 4 – 10 – 2 – 11
- Camino 3: 1 – 2 – 3 – 4 – 5 – 10 – 2 – 11
- Camino 4: 1 – 2 – 3 – 4 – 5 – 6 – 8 – 9 – 4 – 10 – 2 – 11
- Camino 5: 1 – 2 – 3 – 4 – 5 – 6 – 7 – 9 – 4 – 10 – 2 – 11

OBTENCIÓN DE LOS CASOS DE PRUEBA

El último paso de la prueba del camino básico es construir los casos de prueba que fuerzan la ejecución de cada camino. Con el fin de comprobar cada camino, debemos escoger los casos de prueba de forma que las condiciones de los nodos predicho estén adecuadamente establecidas. Una forma de representar el conjunto de casos de prueba es como se muestra en la siguiente tabla. Por ejemplo, representamos los casos de prueba para el *Ejemplo 1*, en la Figura 3.8 se muestran los nodos predicho con sus condiciones para que sea más fácil obtener los casos de prueba:

Camino	Caso de prueba	Resultado esperado
1	Escoger algún valor de C tal que NO se cumpla la condición $C <> 10$ $C = 10$	Visualizar el número de pares y el de impares
2	Escoger algún valor de C tal que Sí se cumpla la condición $C <> 10$. Escoger algún valor de N tal que NO se cumpla la condición $N \% 2 = 0$ $C = 1, N = 5$	Contar números impares
3	Escoger algún valor de C tal que Sí se cumpla la condición $C <> 10$. Escoger algún valor de N tal que Sí se cumpla la condición $N \% 2 = 0$ $C = 2, N = 4$	Contar números pares

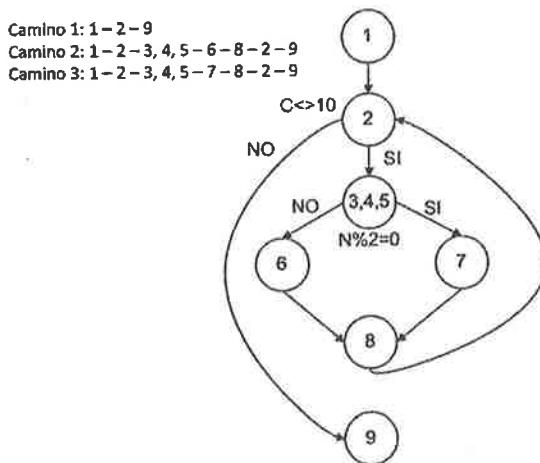


Figura 3.8. Nodos predicho del Ejemplo1 con sus condiciones.

El camino 1 no puede ser probado por sí solo, debe ser probado como parte de las pruebas de los caminos 2 y 3.

Ejemplo 3: en la Figura 3.9 se muestra una función Java y el correspondiente grafo de flujo. Calculamos la complejidad ciclomática, calculamos los caminos independientes y elaboramos los casos de prueba. El grafo de flujo tiene 8 aristas, 7 nodos, 2 nodos predicho y 3 regiones. El valor de $V(G)$ es 3, por tanto tenemos tres caminos independientes que probar:

```

static void visualizarMedia(float x, float y) {
    float resultado = 0; (1)
    if (x < 0 || y < 0) (3)
        (2) System.out.println("X e Y deben ser positivos"); (4)
    else {
        resultado = (x + y) / 2;
        System.out.println("La media es: " + resultado);
    }
} (6)
}

```

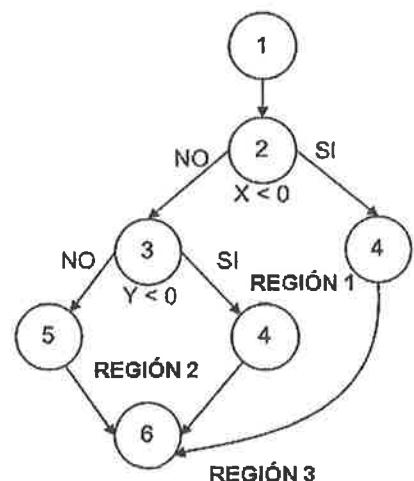


Figura 3.9. Ejemplo 3.

Los caminos independientes y los casos de prueba para cada camino se muestran en la siguiente tabla:

Camino	Caso de prueba	Resultado esperado
Camino 1: 1-2-3-5-6	Escoger algún X e Y tal que NO se cumpla la condición $X < 0 \ \ Y < 0$ $X=4, Y=5$ visualizarMedia(4,5)	Visualiza: La media es: 4.5
Camino 2: 1-2-4-6	Escoger algún X tal que SÍ se cumpla la condición $X < 0$ (Y puede ser cualquier valor) $X=-4, Y=5$ visualizarMedia(-4,5)	Visualiza: X e Y deben ser positivos
Camino 3: 1-2-3-4-6	Escoger algún X tal que NO cumpla la condición $X < 0$ y escoger algún Y que SÍ cumpla la condición $Y < 0$ $X=4, Y=-5$ visualizarMedia(4,-5)	Visualiza: X e Y deben ser positivos

3.5.2. Partición o clases de equivalencia

La partición equivalente es un método de prueba de caja negra que divide los valores de los campos de entrada de un programa en clases de equivalencia. Por ejemplo, supongamos un campo de entrada llamado *número de empleado*, definido con una serie de condiciones: numérico de 3 dígitos y el primero no puede ser 0. Entonces se puede definir una clase de equivalencia no válida: *número de empleado < 100*; y otra válida: *número de empleado comprendido entre 100 y 999*.

Para identificar las clases de equivalencia se examina cada condición de entrada (son parte del dominio de valores de la entrada y normalmente son una frase en la especificación) y se divide en dos o más grupos. Se definen dos tipos de clases de equivalencia:

- **Clases válidas:** son los valores de entrada válidos.
- **Clases no válidas:** son los valores de entrada no válidos.

Las clases de equivalencia se definen según una serie de directrices:

1. Si una condición de entrada especifica un *rango*, se define una clase de equivalencia válida y dos no válidas.
2. Si una condición de entrada requiere un *valor específico*, se define una clase de equivalencia válida y dos no válidas.
3. Si una condición de entrada especifica un *miembro de un conjunto*, se define una clase de equivalencia válida y una no válida.
4. Si una condición de entrada es *lógica*, se define una clase de equivalencia válida y una no válida.

La siguiente tabla resume el número de clases de equivalencia válidas y no válidas que hay que definir para cada tipo de condición de entrada:

Condiciones de entrada	Nº de Clases de equivalencia válidas	Nº de Clases de equivalencia no válidas
1 . Rango	1 CLASE VÁLIDA Contempla los valores del rango	2 CLASES NO VÁLIDAS Un valor por encima del rango Un valor por debajo del rango
2. Valor específico	1 CLASE VÁLIDA Contempla dicho valor	2 CLASES NO VÁLIDAS Un valor por encima Un valor por debajo
3. Miembro de un conjunto	1 CLASE VÁLIDA Una clase por cada uno de los miembros del conjunto	1 CLASE NO VÁLIDA Un valor que no pertenece al conjunto
4. Lógica	1 CLASE VÁLIDA Una clase que cumpla la condición	1 CLASE NO VÁLIDA Una clase que no cumpla la condición

Ejemplo 4: se va a realizar una entrada de datos de un empleado por pantalla gráfica, se definen 3 campos de entrada y una lista para elegir el oficio. La aplicación acepta los datos de esta manera:

- *Empleado*: número de tres dígitos que no empiece por 0.
- *Departamento*: en blanco o número de dos dígitos.
- *Oficio*: Analista, Diseñador, Programador o Elige oficio.

Si la entrada es correcta el programa asigna un salario (que se muestra en pantalla) a cada empleado según estas normas:

- S1 si el *Oficio* es Analista se asigna 2500.
- S2 si el *Oficio* es Diseñador se asigna 1500.
- S3 si el *Oficio* es Programador se asigna 2000.

Si la entrada no es correcta el programa muestra un mensaje indicando la entrada incorrecta:

- ER1 si el *Empleado* no es correcto.
- ER2 si el *Departamento* no es correcto.
- ER3 si no se ha elegido *Oficio*.

Para representar las clases de equivalencia para cada condición de entrada se puede usar una tabla. En cada fila se definen las clases de equivalencia para la condición de entrada, se añade un código a cada clase definida (válida y no válida) para usarlo en la definición de los casos de prueba:

Condición de entrada	Clases de equivalencia	Clases Válidas	COD	Clases no Válidas	COD
Empleado	Rango	100>= Empleado<=999	V1	Empleado < 100 Empleado >999	NV1 NV2
Departamento	Lógica (puede estar o no)	En blanco	V2	No es un número.	NV3
	Valor	Cualquier número de dos dígitos	V3	Número de más de 2 dígitos. Número de menos de 2 dígitos	NV4 NV5
Oficio	Miembro de un conjunto	Oficio = "Programador"	V4	Oficio = "Elige oficio"	NV8
		Oficio = "Analista"	V5		
		Oficio = "Diseñador"	V6		

A partir de esta tabla se generan los casos de prueba. Utilizamos las condiciones de entrada y las clases de equivalencia (a las que se asignó un código en la columna COD, también se podría haber asignado un número a cada clase). Los representamos en otra tabla donde cada fila representa un caso de prueba con los códigos de las clases de equivalencia que se aplican, los valores asignados a las condiciones de entrada y el resultado esperado según el enunciado del problema:

CASO DE PRUEBA	Clases de equivalencia	CONDICIONES DE ENTRADA			Resultado esperado
		Empleado	Departamento	Oficio	
CP1	V1, V3, V4	200	20	Programador	S3
CP2	V1, V2, V5	250		Analista	S1
CP3	V1, V3, V6	450	30	Diseñador	S2
CP4	V1, V2, V4	220		Programador	S3
CP5	NV1, V3, V6	90	35	Diseñador	ER1
CP6	V1, NV3, V5	100	AD	Analista	ER2
CP7	V1, V2, NV8	300		Elige oficio	ER3
CP8	V1, NV4, V6	345	123	Diseñador	ER2
....				

Al rellenar la tabla de casos de prueba se han tenido en cuenta estas dos reglas: los casos de prueba válidos (CP1, CP2, CP3 y CP4) cubren tantas clases de equivalencia válidas como sea posible y los casos de prueba no válidos (CP5, CP6, CP7 y CP8) cubren una sola clase no válida (si se prueban múltiples clases de equivalencia no válidas en el mismo caso de prueba, puede ocurrir que algunas de estas pruebas nunca se ejecuten porque la primera enmascara a las otras o termina la ejecución del caso de prueba).

Los casos de prueba se van añadiendo a la tabla hasta que todas las clases de equivalencia válidas y no válidas hayan sido cubiertas. Por ejemplo a la tabla anterior le faltan clases de equivalencia válidas: (V1, V2, V6) y (V1, V3, V5); y no válidas (NV2, V2, V4) y (V1, NV5, V6).

Ejemplo 5: tenemos una función Java que recibe un número entero y devuelve una cadena con el texto “*Par*” si el número recibido es par, o “*Impar*” si el número es impar.

```
public String parImpar(int nume) {
    String cad="";
    if(nume % 2 == 0)
        cad="Par";
    else
        cad="Impar";
    return cad;
}
```

En este ejemplo tenemos una condición de entrada que requiere un valor específico, un número entero, entonces según la segunda directriz se define una clase de equivalencia válida y dos no válidas. Como en este caso los números son tratados de forma diferente podemos crear una clase de equivalencia para cada entrada válida.

Condición de entrada	Clases de equivalencia	Clases Válidas	COD	Clases no Válidas	COD
nume	Valor Par	Cualquier número entero par	V7	Número impar Cadena	NV9 NV10
	Valor Impar	Cualquier número entero impar	V8	Número par Cadena	NV11 NV12

Los casos de prueba serían los siguientes:

CASO DE PRUEBA	Clases de equivalencia	CONDICIONES DE ENTRADA		Resultado esperado
			nume	
CP1	V7		20	Par
CP2	V8		25	Impar
CP3	NV9		45	Error, número impar
CP4	NV10		“we”	Error, es una cadena
CP5	NV11		10	Error, número par
CP6	NV12		“ad”	Error, es una cadena”

3.5.3. Análisis de valores límite

El análisis de valores límite se basa en que los errores tienden a producirse con más probabilidad (por razones que no están del todo claras) en los límites o extremos de los campos de entrada.

Esta técnica complementa a la anterior y los casos de prueba elegidos ejercitan los valores justo por encima y por debajo de los márgenes de la clase de equivalencia. Además no solo se centra en las condiciones de entrada, sino que también se exploran las condiciones de salida definiendo las clases de equivalencia de salida.

Las reglas son las siguientes:

- Si una condición de entrada especifica un *rango de valores*, se deben diseñar casos de prueba para los límites del rango y para los valores justo por encima y por debajo del rango. Por ejemplo, si una entrada requiere un rango de valores enteros comprendidos entre 1 y 10, hay que escribir casos de prueba para el valor 1, 10, 0 y 11.

2. Si una condición de entrada especifica un *número de valores*, se deben diseñar casos de prueba que ejercenten los valores máximo, mínimo, un valor justo por encima del máximo y un valor justo por debajo del mínimo. Por ejemplo, si el programa requiere de dos a diez datos de entradas, hay que escribir casos de prueba para 2, 10, 1 y 11 datos de entrada.
3. Aplicar la regla 1 para la condición de salida. Por ejemplo, si se debe aplicar sobre un campo de salida un descuento de entre un 10% mínimo y un 50% máximo (dependiendo del tipo de cliente); se generarán casos de prueba para 9,99%, 10%, 50% y 50,01% .
4. Usar la regla 2 para la condición de salida. Por ejemplo, si la salida de un programa es una tabla de temperaturas de 1 a 10 elementos, se deben diseñar casos de prueba para que la salida del programa produzca 0, 1, 10 y 11 elementos. Tanto en esta regla, como en la anterior, hay que tener en cuenta que no siempre se podrán generar resultados fuera del rango de salida.
5. Si las estructuras de datos internas tienen límites prestablecidos (por ejemplo un array de 100 elementos), hay que asegurarse de diseñar casos de prueba que ejercente la estructura de datos en sus límites, primer y último elemento.

Ejemplo 6: determina los casos de prueba para los siguientes elementos según las condiciones de entrada y de salida:

	Condiciones de entrada y salida	Casos de prueba
Código	Entero de 1 a 100	Valores: 0, 1, 100, 101
Puesto	Alfanumérico de hasta 4 caracteres	Longitud de caracteres: 0, 1, 4, 5
Antigüedad	De 0 a 25 años (Real)	Valores: 0, 25, -0.1, 25.1
Horas semanales	De 0 a 60	Valores: 0, 60, -1, 61
Fichero de entrada	Tiene de 1 a 100 registros	Para leer 0, 1, 100 y 101 registros
Fichero de salida	Podrá tener de 0 a 10 registros	Para generar 0, 10 y 11 registros (no se puede generar -1 registro)
Array interno	De 20 cadenas de caracteres	Para el primer y último elemento.

Ejemplo 7: partimos del *Empleado* (que tiene que ser un número de tres dígitos que no empiece por 0) del ejemplo 4 del epígrafe anterior. Utilizando esta técnica, para la clase de equivalencia V1 que representa un rango de valores ($100 \geq \text{Empleado} \leq 999$) se deben generar dos casos de prueba con el límite inferior y el superior del rango (para identificar estos casos de prueba utilizamos *V1a* para el límite inferior y *V1b* para el superior):

CASO DE PRUEBA	Clases de equivalencia	CONDICIONES DE ENTRADA			Resultado esperado
		Empleado	Departamento	Oficio	
CP11	V1a, V3, V4	100	20	"Programador"	S3
CP12	V1b, V2, V5	999		"Analista"	S1
CP13	NV1, V3, V6	99	30	"Diseñador"	ER1
CP14	NV2, V2, V4	1000		"Programador"	ER1

3.6. HERRAMIENTAS DE DEPURACIÓN

El proceso de depuración comienza con la ejecución de un caso de prueba, véase Figura 3.10. Se evalúan los resultados de la ejecución y fruto de esa evaluación se comprueba que hay una falta de correspondencia entre los resultados esperados y los obtenidos realmente. El proceso de depuración siempre tiene uno de los dos resultados siguientes:

1. Se encuentra la causa del error, se corrige y se elimina
2. No se encuentra la causa del error. En este caso, la persona encargada de la depuración debe sospechar la causa, diseñar casos de prueba que ayuden a confirmar sus sospechas y volver a repetir las pruebas para identificar los errores y corregirlos (pruebas de regresión, repetición selectiva de pruebas para detectar fallos introducidos durante la modificación).

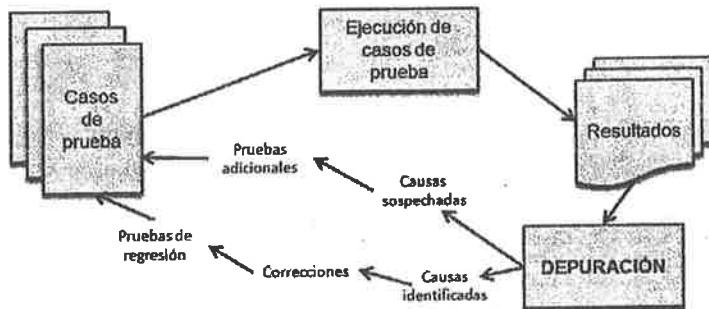


Figura 3.10. El proceso de depuración.

Al desarrollar programas cometemos dos tipos de errores: errores de compilación y errores lógicos. Los primeros son fáciles de corregir ya que normalmente usamos un IDE para codificar y al escribir las sentencias, si por ejemplo nos olvidamos el punto y coma, o usamos una variable inexistente, el entorno proporciona información de la localización del error y cómo solucionarlo, véase Figura 3.11.

Las capturas muestran partes de un código Java:

```

int n = 1;
int[] tabla = enteros(n);
a=10;|

```

En la línea 6, se muestra un cuadro de diálogo: "Syntax error, Insert ')' to complete BlockStatements".

En la línea 8, se muestra otro cuadro de diálogo: "a cannot be resolved to a variable".

Figura 3.11. Errores devueltos por el IDE al escribir código.

Los errores de tipo lógico son más difíciles de detectar ya que el programa se puede compilar con éxito (no hay errores sintácticos), sin embargo, su ejecución puede devolver resultados inesperados y erróneos. A estos errores de tipo lógico se le suele llamar *bugs*.

Los entornos de desarrollo incorporan una herramienta conocida como depurador (o debugger) para ayudarnos a resolver este tipo de errores. El depurador nos permite analizar el código del programa mientras se ejecuta. Permite establecer puntos de interrupción o de ruptura, suspender la ejecución del programa, ejecutar el código paso a paso y examinar el contenido de las variables.

En Eclipse podemos lanzar el depurador de varias formas, véase Figura 3.12: pulsando en el botón *Debug*, seleccionando el menú *Run->Debug* o mediante el menú contextual que se muestra al hacer clic con el botón derecho del ratón en la clase que se va a ejecutar y seleccionando *Debug As -> Java Application*. En cualquiera de esos casos la clase se ejecuta.

También es conveniente abrir la vista de depuración desde el menú *Window -> Open perspective-> Debug*, en la que se muestra la información relativa al programa que se está ejecutando, véase Figura 3.13. No obstante, al ejecutar el programa en modo depuración, si la

vista no está abierta, se muestra un mensaje desde el que confirmaremos su apertura. Al cambiar a la vista depuración se muestra en la barra de herramientas una serie de botones: para continuar la ejecución, suspenderla, pararla, para meterse dentro de la primera línea de un método, avanzar un paso la ejecución, avanzar el programa hasta salir del método actual, etc.

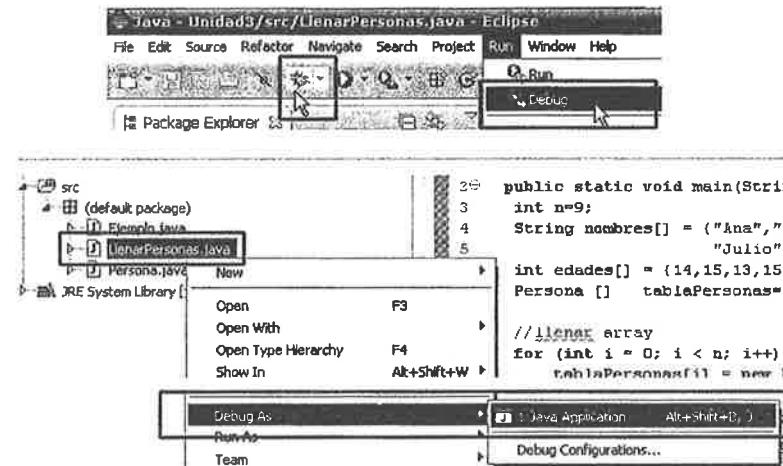


Figura 3.12. Lanzar el depurador en Eclipse.

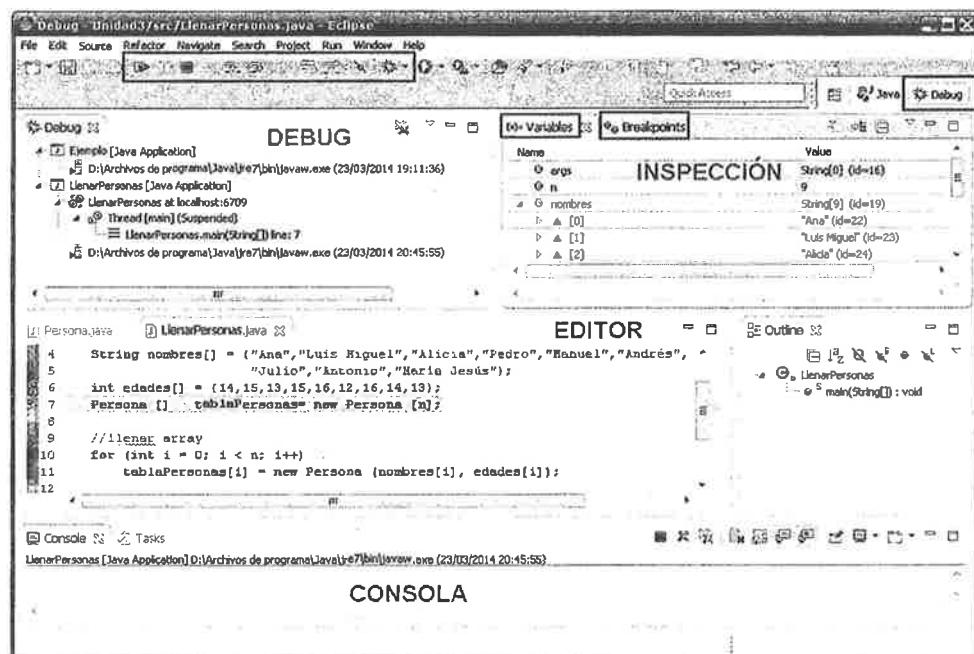


Figura 3.13. Perspectiva Debug.

En esta vista se pueden ver varias zonas:

- En la vista EDITOR se va marcando la traza de ejecución del programa mostrándose una flechita azul en el margen izquierdo de la línea que se está ejecutando.
- La vista DEBUG muestra los hilos de ejecución, en este caso solo muestra un hilo (*Thread[main]*) y debajo la clase en la que está parada la ejecución mostrando el número de línea.

- Las vistas de INSPECCIÓN permiten ver los valores de las variables y de los puntos de ruptura (o breakpoints) que intervienen en el programa en un instante determinado. Desde aquí se puede modificar el valor de las variables, basta con hacer clic en el valor y cambiarlo; el nuevo valor se usará en los siguientes pasos de ejecución. También desde la pestaña *Breakpoints* se puede activar o desactivar un breakpoint, eliminarlo, configurarlo para que la ejecución se detenga cuando se pase por él un determinado número de veces, etc.
- Por último, la vista CONSOLA muestra la consola de ejecución del programa que se está depurando. Es la vista sobre la que se redirecciona tanto la entrada como la salida estándar.

Desde el menú *Run* de la perspectiva de depuración se pueden observar varias opciones, algunas de ellas similares a los botones que aparecen de la barra de herramientas y en la barra de la vista de inspección (véase Figura 3.14):

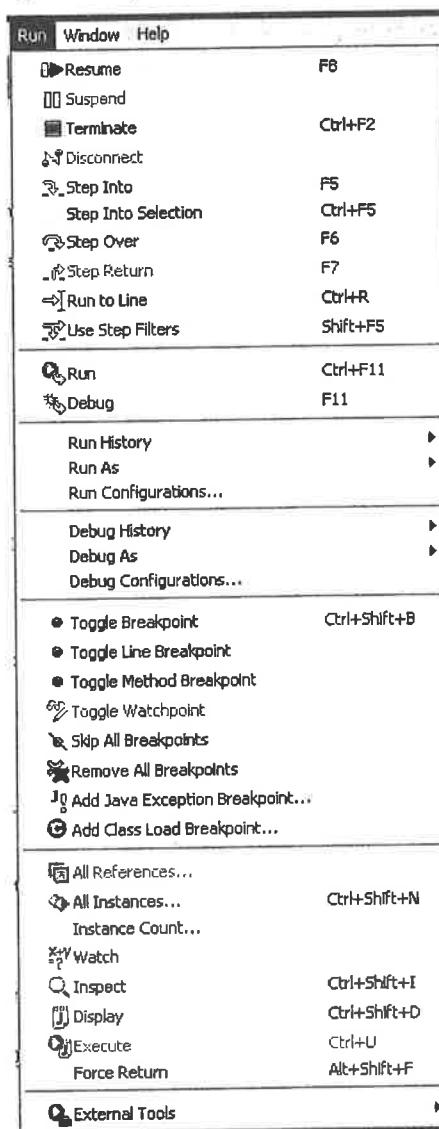


Figura 3.14. Menú *Run* de la perspectiva Debug

- ▶ **Resume** (o tecla F8). Reanuda un hilo suspendido. Se utiliza cuando no queremos analizar instrucción por instrucción y deseamos que el depurador se pare en la siguiente línea donde hay un breakpoint.
- **Suspend**. Suspende el hilo seleccionado.
- **Terminate** (o Ctrl+F2). Finaliza el proceso de depuración.
- ❖ **Step Into** (o tecla F5). Se ejecuta paso a paso cada instrucción. Si el depurador encuentra una llamada a un método o función, al pulsar *Step Into* se irá a la primera instrucción de dicho método.
- ❖ **Step Over** (o pulsar F6). Se ejecuta paso a paso cada instrucción, pero si el depurador encuentra un método, al pulsar *Step Over* se irá a la siguiente instrucción, sin entrar en el código del método.
- ❖ **Step Return** (o pulsar F7). Si nos encontramos dentro de un método, el depurador sale del método actual.
- ❖ **Run to Line** (o pulsar Ctrl+R). Se reanuda la ejecución del código a partir de la línea seleccionada.
- ❖ **Use Step Filters** (o pulsar Shift+F5). Cambia los filtros de paso de activado a desactivado. Estos filtros están definidos en el menú *Windows -> Preferences -> Java -> Debug -> Step Filtering*. Se utilizan comúnmente para filtrar tipos que no se desean recorrer durante la depuración.
- ❖ **Run** (o pulsar Ctrl+R). Ejecutar el programa.
- ❖ **Debug** (o pulsar F11). Ejecutar en modo depuración.
- ❖ **Skip All Breakpoints**. Se omiten todos los breakpoints.
- Establece u omite puntos de ruptura a nivel de línea o de método.
- ❖ **Remove All Breakpoints**. Elimina todos los puntos de ruptura.
- ❖ **Add Java exception Breakpoints**. Permite añadir una excepción Java como breakpoint.
- ❖ **All instances**. Abre un cuadro de diálogo emergente que muestra una lista de todas las instancias del tipo Java seleccionado.
- ❖ **Watch**. Permite crear nuevas expresiones basadas en las variables del programa.
- ❖ **Inspect**. Crea una nueva expresión para la variable seleccionada y la agrega a la vista de inspección.
- ❖ **Display**. Muestra el resultado de evaluar la expresión seleccionada.

3.6.1. Puntos de ruptura y seguimiento

Partimos de la siguiente clase para empezar a utilizar el depurador:

```
public class LlenarNumeros {
    public static void main(String[] args) {
        int n = 5;
        int[] tabla = new int[n];
```

```

        tabla = llenar(n);
        int suma = sumar(tabla);
        System.out.println("La suma es: " + suma);
    }

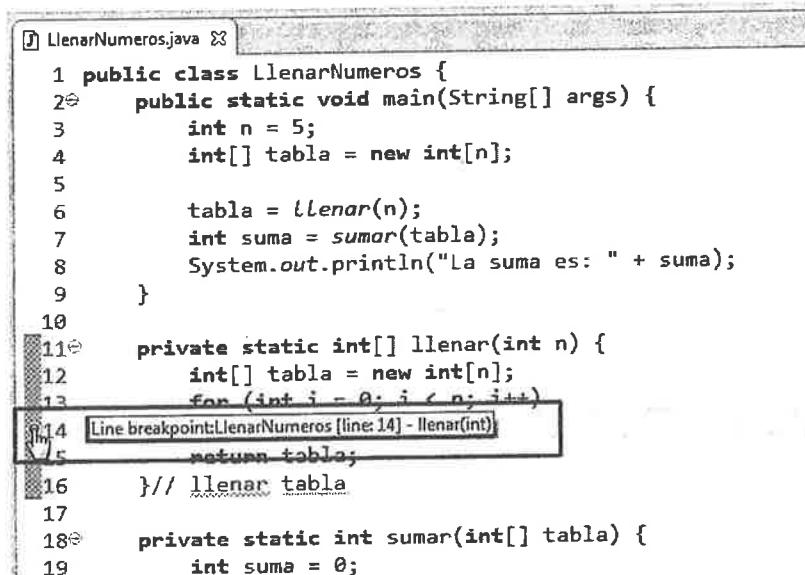
private static int[] llenar(int n) {
    int[] tabla = new int[n];
    for (int i = 0; i < n; i++)
        tabla[i] = i * 10;
    return tabla;
} // llenar tabla

private static int sumar(int[] tabla) {
    int suma = 0;
    int n=tabla.length;
    for (int i = 0; i < n; i++)
        suma = suma + tabla[i];
    return suma;
} // sumar tabla
} //fin clase

```

Donde se definen dos métodos además del método *main()*. El primer método llamado *llenar()* recibe el parámetro entero *n* y devuelve un array con *n* enteros. El segundo método llamado *sumar()*, recibe un array de enteros, suma sus elementos y devuelve la suma.

En primer lugar para empezar el proceso de depuración abrimos la vista de depuración (menú *Window -> Open perspective-> Debug*). Una vez ahí, para empezar a depurar una clase establecemos un punto de ruptura o breakpoint. Se trata de seleccionar una línea en nuestro código donde queramos que la ejecución se detenga, así podremos ver los valores que tienen las variables en ese momento. Para poner un breakpoint hacemos doble clic en el margen izquierdo del editor, justo en la línea donde queremos que se detenga la ejecución, aparecerá un circulito a la izquierda . Véase Figura 3.15 donde se establece un punto de ruptura en la línea 14 (*tabla[i] = i * 10*), donde se va llenando el array.



```

1 public class LlenarNumeros {
2     public static void main(String[] args) {
3         int n = 5;
4         int[] tabla = new int[n];
5
6         tabla = llenar(n);
7         int suma = sumar(tabla);
8         System.out.println("La suma es: " + suma);
9     }
10
11     private static int[] llenar(int n) {
12         int[] tabla = new int[n];
13         for (int i = 0; i < n; i++)
14             tabla[i] = i * 10; // Line breakpoint:LlenarNumeros [line:14] - llenar(int)
15         return tabla;
16     } // llenar tabla
17
18     private static int sumar(int[] tabla) {
19         int suma = 0;

```

Figura 3.15. Estableciendo un breakpoint.

Una vez establecido el punto de ruptura ejecutamos el programa en modo depuración, para ello pulsamos en el botón *Debug* . El programa se ejecutará de forma normal hasta que la ejecución llegue al punto de ruptura establecido, en ese momento se detendrá.

En la ventana *Debug* aparece la pila de llamadas, véase Figura 3.16, donde se ven cada uno de los hilos de ejecución, en este caso solo hay uno (*Thread main[]*). Debajo de esta línea se muestra la clase con el método donde está ahora la ejecución parada, clase *LlenarNumeros*, método *llenar()*; se muestra también el número de línea donde está detenida la ejecución, en este caso la línea 14.

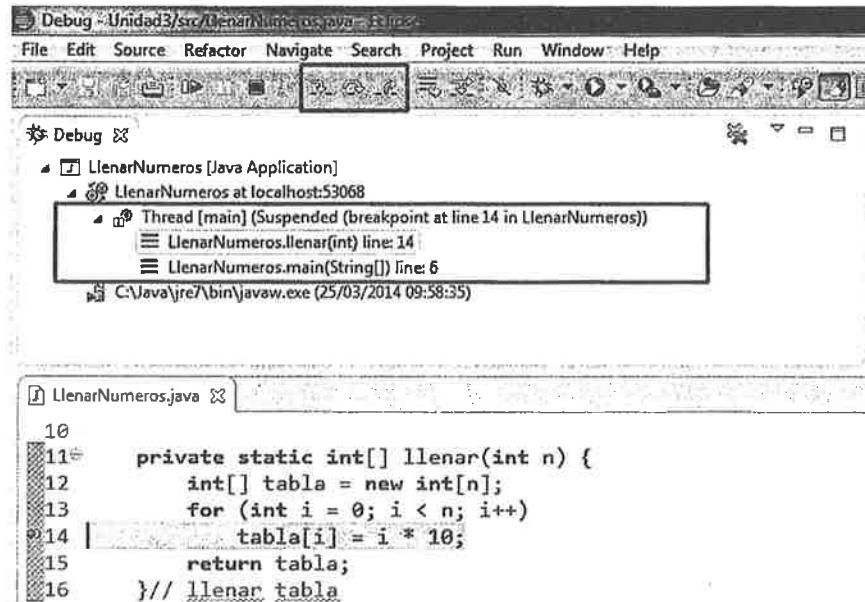


Figura 3.16. Ventana *Debug*.

La siguiente línea muestra quién ha llamado a este método, en este caso la llamada se realiza desde la clase *LlenarNumeros* y dentro del método *main()* en la línea 6, en esta línea está la sentencia *tabla = llenar(n)*. Al hacer clic en estas líneas se muestra la línea de código que se está ejecutando.

A continuación podemos usar los botones:

- *Step Into* para ejecutar el programa paso a paso instrucción por instrucción, o
- *Step Over* para ejecutar paso a paso cada instrucción pero si encuentra un método saltarlo,
- o usar *Step Return* , en este caso si nos encontramos dentro de un método, al pulsarlo el depurador hace avanzar la ejecución del programa saliendo del método.

En cualquier momento podemos finalizar el depurador pulsando en el botón *Terminate*. Una flechita al lado de la línea 13 nos indica la línea que se está ejecutando. Para quitar el punto de ruptura de alguna línea hacemos doble clic sobre el circulito.

Podemos establecer puntos de ruptura condicionales, por ejemplo, con el punto de ruptura establecido en la línea 14 podemos determinar que la ejecución se detenga cuando el valor de la *i* sea 3, véase Figura 3.17.

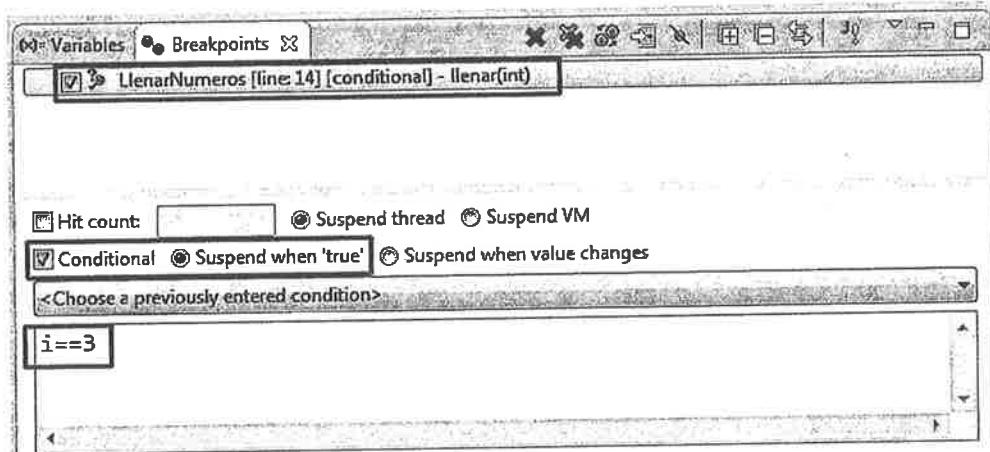


Figura 3.17. Punto de ruptura condicional.

Desde la vista de INSPECCIÓN y desde la pestaña *Breakpoints* marcamos la casilla *Conditional*, seleccionamos *Suspend when 'true'* y escribimos la condición a evaluar (en este caso que la *i* sea igual a 3, *i==3*) que debe devolver un valor booleano. Entonces cuando la *i* sea 3 se detiene la ejecución y podemos empezar a usar los botones para ejecutar instrucción por instrucción.

Para eliminar todos los breakpoints podemos usar el botón . El botón elimina el breakpoint seleccionado.

3.6.2. Examen y modificación de variables

Desde la vista de INSPECCIÓN y desde la pestaña de *Variables* podemos inspeccionar las variables definidas en el punto en el que el programa está detenido en este momento, véase Figura 3.18. En este punto se muestra el valor de la variable *n*, los elementos de la tabla inicializados a 0 y el valor inicial de la *i* que es 0. En la parte inferior se muestra el valor de la variable seleccionada, en este caso se muestran los valores de la tabla [0, 0, 0, 0, 0].

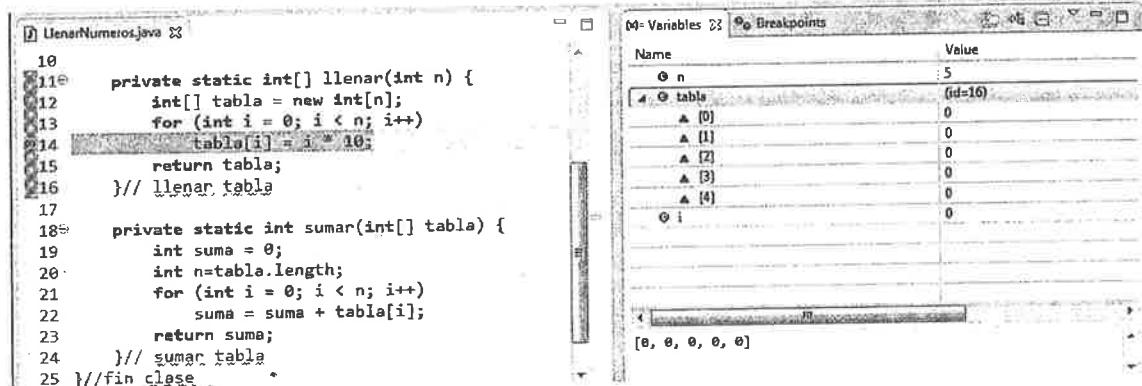


Figura 3.18. Pestaña de variables.

Desde aquí se puede modificar el valor de una variable tan solo haciendo clic sobre él y escribiendo el valor deseado. La Figura 3.19 muestra un momento en la ejecución del programa donde se han modificado dos valores de la tabla (posiciones 0 y 1). Se muestra marcado en amarillo (en la figura, es la línea que contiene [3]) por donde va actualmente la ejecución del programa.

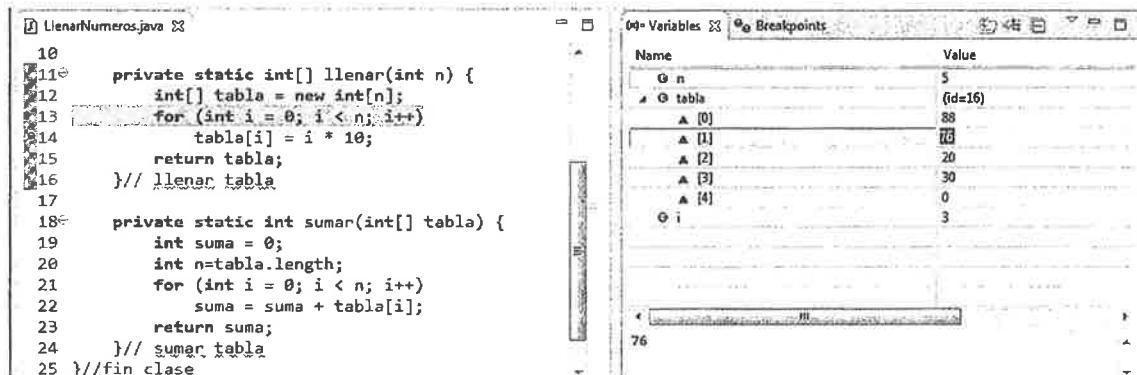


Figura 3.19. Pestaña de variables con variables modificadas.

Otra forma de ver el contenido de la variable es pasando el puntero del ratón por ella, se abre una ventanita mostrándonos la información, véase Figura 3.20.

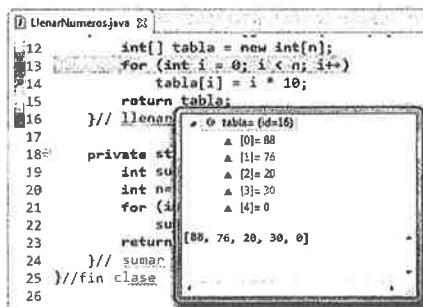


Figura 3.20. Muestra el valor de una variable.

Como resumen podemos decir que el depurador nos permite ejecutar un programa de forma controlada con el fin de probarlo, encontrar la causa de un error o incluso conocer mejor su funcionamiento. Proporciona las siguientes funciones:

- Ejecutar el programa paso a paso. Tras cada paso, el usuario recupera el control.
- Detener la ejecución del programa cuando alcance una determinada línea del código o cuando se cumpla una condición.
- A cada paso de ejecución se puede conocer el valor de las variables o expresiones.

Para las pruebas de este epígrafe y el siguiente se ha utilizado el paquete Eclipse estándar versión kepler: *eclipse-standard-kepler-SR2-win32.zip*.

3.7. PRUEBAS UNITARIAS CON JUNIT

Hasta ahora hemos estado haciendo pruebas de forma manual a partir de una especificación o de un código. En este apartado aprenderemos a utilizar una herramienta para implementar pruebas que verifiquen que nuestro programa genera los resultados que de él esperamos.

JUnit es una herramienta para realizar pruebas unitarias automatizadas. Está integrada en Eclipse, por lo que no es necesario descargarse ningún paquete para poder usarla. Las pruebas unitarias se realizan sobre una clase para probar su comportamiento de modo aislado independientemente del resto de clases de la aplicación. Aunque esto no siempre es así porque una clase a veces depende de otras clases para poder llevar a cabo su función.

3.7.1. Creación de una clase de prueba

Para empezar a usar JUnit creamos un nuevo proyecto en Eclipse y creamos la clase a probar, en este caso se llama *Calculadora*:

```
public class Calculadora {
    private int num1;
    private int num2;

    public Calculadora(int a, int b) {
        num1 = a;
        num2 = b;
    }
    public int suma() {
        int resul = num1 + num2;
        return resul;
    }
    public int resta() {
        int resul = num1 - num2;
        return resul;
    }
    public int multiplica() {
        int resul = num1 * num2;
        return resul;
    }
    public int divide() {
        int resul = num1 / num2;
        return resul;
    }
}
```

A continuación, hay que crear la clase de prueba. Con la clase *Calculadora* seleccionada pulsamos el botón derecho del ratón y seleccionamos *New->JUnit Test Case*, véase Figura 3.21.

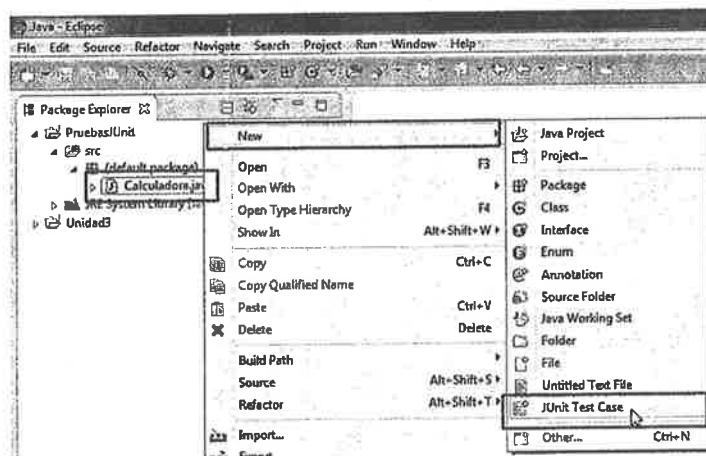


Figura 3.21. Opción New->JUnit Test Case.

También se puede hacer desde el menú *File->New->JUnit Test Case*. En cualquier caso se abre una ventana de diálogo. Desde aquí debemos seleccionar *New JUnit 4 test*, el resto de opciones dejamos los valores por defecto, como nombre de clase se generará el nombre *CalculadoraTest*. Pulsamos el botón *Next*. A continuación hemos de seleccionar los métodos que queremos probar, marcamos los 4 métodos y pulsamos *Finish*. Se abre una ventanita

indicándonos que la librería *JUnit 4* no está incluida en nuestro proyecto, pulsamos el botón *OK* para que se incluya, véase Figura 3.22,

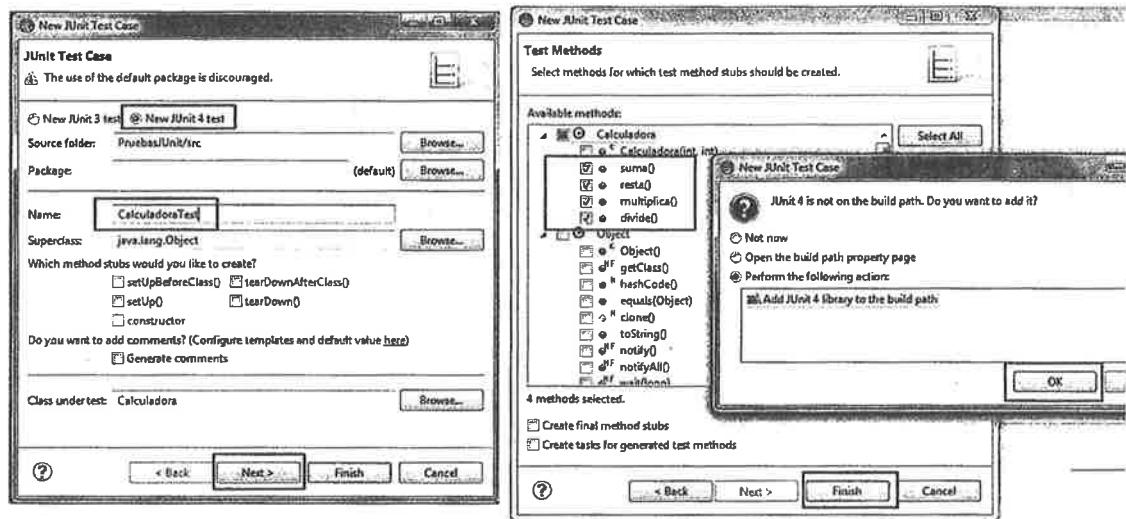


Figura 3.22. Creación de la clase de prueba.

Para que la clase de prueba se incluya en un paquete diferente al de la clase a probar se escribe el nombre del paquete en el campo *Package* (en el ejemplo la clase de prueba y la clase a probar están dentro del mismo paquete: *default package*).

La clase de prueba se crea automáticamente, se observan una serie de características:

- Se crean 4 métodos de prueba, uno para cada método seleccionado anteriormente.
- Los métodos son públicos, no devuelven nada y no reciben ningún argumento.
- El nombre de cada método incluye la palabra *test* al principio *testSuma()*, *testResta()*, *testMultiplica()* y *testDivide()*.
- Sobre cada uno de los métodos aparece la anotación *@Test* que indica al compilador que es un método de prueba.
- Cada uno de los métodos de prueba tiene una llamada al método *fail()* con un mensaje indicando que todavía no se ha implementado el método. Este método hace que el test termine con fallo lanzando el mensaje.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class CalculadoraTest {

    @Test
    public void testSuma() {
        fail("Not yet implemented");
    }

    @Test
    public void testResta() {
        fail("Not yet implemented");
    }

    @Test
    public void testMultiplica() {
        fail("Not yet implemented");
    }

    @Test
    public void testDivide() {
        fail("Not yet implemented");
    }
}
```

```

    @Test
    public void testMultiplica() {
        fail("Not yet implemented");
    }

    @Test
    public void testDivide() {
        fail("Not yet implemented");
    }
}

```

3.7.2. Preparación y ejecución de las pruebas

Antes de preparar el código para los métodos de prueba veamos una serie de métodos de JUnit para hacer las comprobaciones, todos estos métodos devuelven un tipo *void*:

MÉTODOS	MISIÓN
<code>assertTrue(boolean expresión)</code> <code>assertTrue(String mensaje, boolean expresión)</code>	Comprueba que la expresión se evalúe a <i>true</i> . Si no es <i>true</i> y se incluye el String, al producirse error se lanzará el <i>mensaje</i>
<code>assertFalse(boolean expresión)</code> <code>assertFalse(String mensaje, boolean expresión)</code>	Comprueba que la expresión se evalúe a <i>false</i> . Si no es <i>false</i> y se incluye el String, al producirse error se lanzará el <i>mensaje</i>
<code>assertEquals(valorEsperado, valorReal)</code> , <code>assertEquals(String mensaje, valorEsperado, valorReal)</code>	Comprueba que el <i>valorEsperado</i> sea igual al <i>valorReal</i> . Si no son iguales y se incluye el String, entonces se lanzará el <i>mensaje</i> . <i>ValorEsperado</i> y <i>valorReal</i> pueden ser de diferentes tipos
<code>assertNull(Object objeto)</code> , <code>assertNull(String mensaje, Object objeto)</code>	Comprueba que el <i>objeto</i> sea null. Si no es null y se incluye el String al producirse error se lanzará el <i>mensaje</i> .
<code>assertNotNull(Object objeto)</code> , <code>assertNotNull(String mensaje, Object objeto)</code>	Comprueba que el <i>objeto</i> no sea null. Si es null y se incluye el String, al producirse error se lanzará el <i>mensaje</i>
<code>assertSame(Object objetoEsperado, Object objetoReal)</code> <code>assertSame(String mensaje, Object objetoEsperado, Object objetoReal)</code>	Comprueba que <i>objetoEsperado</i> y <i>objetoReal</i> sean el mismo objeto. Si no son el mismo y se incluye el String, al producirse error se lanzará el <i>mensaje</i>
<code>assertNotSame(Object objetoEsperado, Object objetoReal)</code> <code>assertNotSame(String mensaje, Object objetoEsperado, Object objetoReal)</code>	Comprueba que <i>objetoEsperado</i> no sea el mismo objeto que <i>objetoReal</i> . Si son el mismo y se incluye el String, al producirse error se lanzará el <i>mensaje</i>
<code>fail()</code> <code>fail(String mensaje):</code>	Hace que la prueba falle. Si se incluye un String la prueba falla lanzando el <i>mensaje</i>

Más información sobre estos métodos la podemos encontrar en la siguiente dirección web:
<http://junit.sourceforge.net/javadoc/index.html?org/junit/>.

Vamos a crear el código de prueba para el método `testSuma()` que probará el método `suma()` de la clase `Calculadora`. Lo primero que hacemos es crear una instancia de la clase `Calculadora`, llamamos al método `suma()` llevando los valores a sumar, por ejemplo 20 y 10, y comprobamos los resultados con el método `assertEquals()`. En el primer parámetro de este último método escribimos el resultado esperado al realizar el método `suma()`, en este caso es 30, y como segundo parámetro asignamos el resultado obtenido al llamar a dicho método:

```
@Test
public void testSuma() {
    Calculadora calcu = new Calculadora(20, 10);
    int resultado = calcu.suma();
    assertEquals(30, resultado);
}
```

Si pulsamos ahora el botón *Run* para ejecutar el test, se mostrarán algunos errores ya que no se han implementado todos los test de pruebas. También se puede ejecutar la clase de prueba pulsando sobre la clase con el botón derecho del ratón y seleccionando *Run As-> JUnit Test*. En ambos casos se abre la pestaña de JUnit donde se muestran los resultados de ejecución de las pruebas, véase Figura 3.23.

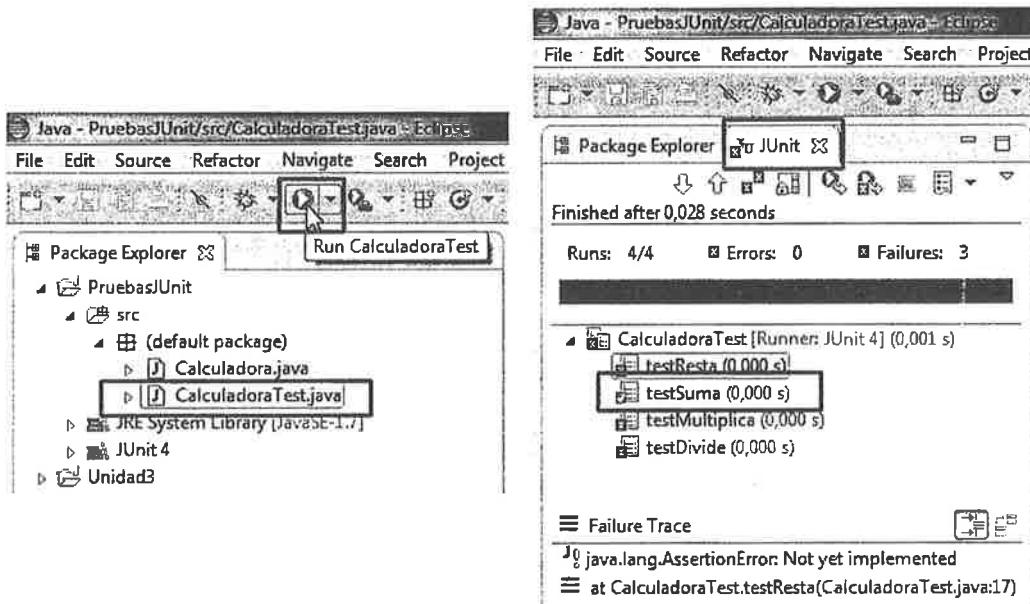


Figura 3.23. JUnit con el resultado de las pruebas.

Al lado de cada prueba aparece un ícono con una marca: una marca de verificación verde indica prueba exitosa , un aspa azul indica fallo; y un aspa rojo indica error . Más adelante veremos la diferencia entre fallo y error.

El resultado de la ejecución de la prueba muestra: *Runs: 4/4 Errors: 0 Failures: 3*; esto nos indica que se han realizado 4 pruebas, ninguna de ellas ha provocado error y 3 de ellas han provocado fallo.

En el contexto de JUnit un fallo es una comprobación que no se cumple, un error es una excepción durante la ejecución del código. En esta prueba solo se ha realizado satisfactoriamente la prueba con el método `testSuma()` que muestra un ícono con una marca de verificación al lado

, el resto de pruebas han fallado, muestran el icono con aspa azul (recordemos que todos los métodos inicialmente incluyen el método `fail()` que hace fallar la prueba).

Rellenamos el resto de los métodos de prueba escribiendo en los métodos `assertEquals()` el valor esperado y el resultado de realizar la operación con los números 20 y 10:

```

@Test
public void testResta() {
    Calculadora calcu = new Calculadora(20, 10);
    int resultado = calcu.resta();
    assertEquals(10, resultado);
}

@Test
public void testMultiplica() {
    Calculadora calcu = new Calculadora(20, 10);
    int resultado = calcu.multiplica();
    assertEquals(200, resultado);
}

@Test
public void testDivide() {
    Calculadora calcu = new Calculadora(20, 10);
    int resultado = calcu.divide();
    assertEquals(2, resultado);
}

```

Ahora al ejecutar la clase de prueba el resultado que muestra es: *Runs: 4/4 Errors: 0 Failures: 0*; nos indica que se han realizado 4 pruebas, ninguna ha provocado error y ninguna ha provocado fallo, véase Figura 3.24.

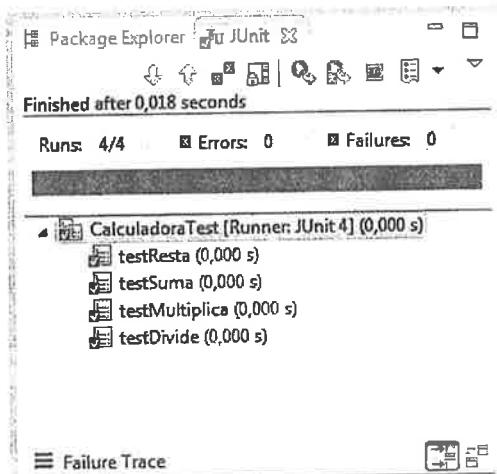


Figura 3.24. JUnit con el resultado satisfactorio de las pruebas.

Para ver la diferencia entre un fallo y un error cambiamos el código de dos de los métodos de prueba. Para hacer que el método `multiplica()` produzca un fallo hacemos que el valor esperado no coincida con el resultado; se incluye un String en el método `assertEquals()` para que si se produce el fallo se lance el mensaje. Para que el método `divide()` produzca un error, al crear el

objeto calculadora asignamos el valor 0 al segundo parámetro (será el denominador de la división, al dividir por 0 se produce una excepción).

En la Figura 3.25 se muestra la vista JUnit con el resultado de la ejecución, el botón *Filter Stack Trace* muestra la traza completa de ejecución.

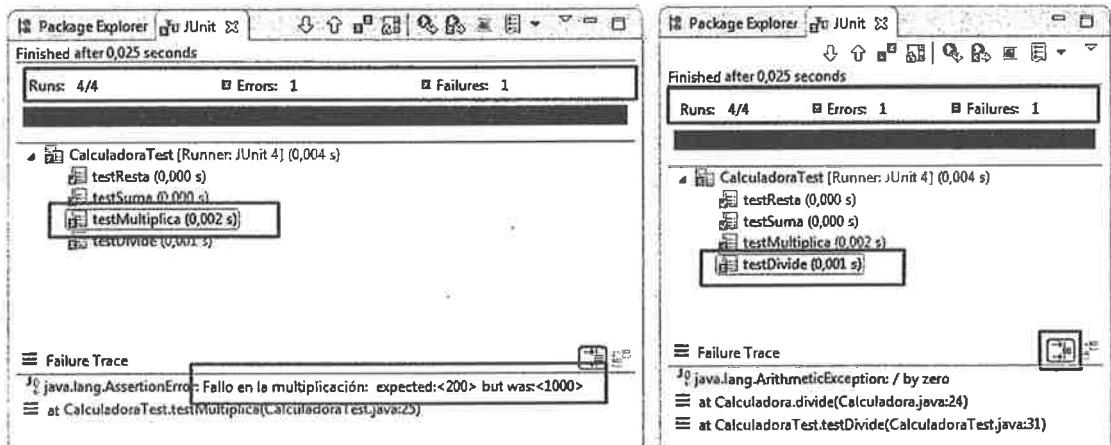


Figura 3.25. Pruebas con 1 fallo y 1 error.

Al pulsar en los test que han producido fallos o errores se muestra la traza de ejecución. El código de los métodos es el siguiente:

```
@Test
public void testMultiplica() {
    Calculadora calcu = new Calculadora(20, 50);
    int resultado = calcu.multiplica();
    assertEquals("Fallo en la multiplicación: ", 200, resultado);
}

@Test
public void testDivide() {
    Calculadora calcu = new Calculadora(20, 0);
    int resultado = calcu.divide();
    assertEquals(2, resultado);
}
```

El siguiente test comprueba que la llamada al método *divide()* devuelve la excepción *ArithmetcException* al dividir 20 entre 0; por tanto sale por la cláusula *catch*. Si no se lanza la excepción, se lanza el método *fail()* con un mensaje indicando que se ha producido un fallo al probar el test. La prueba tiene éxito si se produce la excepción y falla en caso contrario:

```
@Test
public void testExcepcion() {
    try {
        Calculadora calcu = new Calculadora(20, 0);
        int resultado = calcu.divide();
        fail("FALLO, Debería haber lanzado la excepción");
    } catch (ArithmetcException e) {
        // PRUEBA satisfactoria
    }
}
```

ACTIVIDAD 3.1

Modifica el método *resta()* de la clase *Calculadora* y añade los métodos *resta2()* y *divide2()* que se exponen a continuación. Crea después los test para probar los 3 métodos. Los métodos son:

```
public int resta() {
    int resul;
    if(resta2())resul = num1 - num2;
    else          resul= num2- num1;
    return resul;
}

public boolean resta2() {
    if (num1 >= num2) return true;
    else                  return false;
}

public Integer divide2() {
    if(num2==0) return null;
    int resul = num1 / num2;
    return resul;
}
```

Utiliza los métodos *assertTrue()*, *assertFalse()*, *assertNull()*, *assertNotNull()* o *assertEquals()* según convenga.

Para probar un método que puede lanzar excepciones se utiliza el parámetro **expected** con la anotación **@Test**. Por ejemplo, para hacer que el método *divide()* lance la excepción *ArithmeticeException* si el denominador es 0, se puede utilizar la instrucción **throw** de la siguiente manera:

```
public int divide0() {
    if(num2==0)
        throw new java.lang.ArithmeticeException("División por 0");
    else{
        int resul= num1 / num2;
        return resul;
    }
}
```

En la clase de prueba, para poder verificar que se lanza esa excepción se utilizó el parámetro **expected** de la siguiente manera:

```
@Test(expected = java.lang.ArithmeticeException.class)
public void testDivide0() {
    Calculadora calcu = new Calculadora(20, 0);
    Integer resultado = calcu.divide0();
}
```

La prueba fallará, si no se produce la excepción.

En la vista de JUnit se muestran varios botones:

- ⇒ *Next Failed Test*: navega a la siguiente prueba que ha producido fallo o error.
- ⇒ *Previous Failed Test*: navega a la anterior prueba que ha producido fallo o error.

- *Show Failures Only*: muestra solo las pruebas que han producido fallo o error.
- *Scroll Lock*: activa o desactiva el scroll lock.
- *Rerun Test*: vuelve a ejecutar las pruebas.
- *Rerun Test - Failures First*: vuelve a ejecutar las pruebas, ejecutando en primer lugar los fallos y errores.
- *Stop JUnit Test Run*: detiene la ejecución de las pruebas.
- *Test Run History*: muestra el historial de las pruebas realizadas anteriormente.

3.7.3. Tipos de anotaciones

En todos los métodos de prueba anteriores se repetía la línea *Calculadora calcu = new Calculadora(20, 10)*; esta sentencia de inicialización se puede escribir una sola vez dentro de la clase.

JUnit dispone de una serie de anotaciones que permiten ejecutar código antes y después de las pruebas:

- **@Before**: si anotamos un método con esta etiqueta, el código será ejecutado antes de cualquier método de prueba. Este método se puede utilizar para inicializar datos, por ejemplo, en una aplicación de acceso a base de datos se puede preparar la base de datos, si vamos a usar un array para las pruebas se puede inicializar aquí. Puede haber varios métodos en la clase de prueba con esta anotación.
- **@After**: si anotamos un método con esta etiqueta el código será ejecutado después de la ejecución de todos los métodos de prueba. Se puede utilizar para limpiar datos. Puede haber varios métodos en la clase de prueba con esta anotación.

La clase *CalculadoraTest* incluyendo dos métodos con las anotaciones **@Before** y **@After** quedaría de la siguiente manera, en el primer método *creaCalculadora()* inicializamos el objeto calculadora y en el segundo *borraCalculadora()* lo limpiamos:

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class CalculadoraTest2 {
    private Calculadora calcu;
    private int resultado;

    @Before
    public void creaCalculadora() {
        calcu = new Calculadora(20, 10);
    }

    @After
    public void borraCalculadora() {
        calcu = null;
    }

    @Test
    public void testSuma() {
```

```

        resultado = calcu.suma();
        assertEquals(30, resultado);
    }

    @Test
    public void testResta() {
        resultado = calcu.resta();
        assertEquals(10, resultado);
    }

    @Test
    public void testMultiplica() {
        resultado = calcu.multiplica();
        assertEquals(200, resultado);
    }

    @Test
    public void testDivide() {
        resultado = calcu.divide();
        assertEquals(2, resultado);
    }
}
}

```

Otras anotaciones a destacar son `@BeforeClass` y `@AfterClass`, tienen algunas diferencias respecto a las anteriores:

- `@BeforeClass`: solo puede haber un método con esta etiqueta. El método marcado con esta anotación es invocado una vez al principio del lanzamiento de todas las pruebas. Se suele utilizar para inicializar atributos comunes a todas las pruebas o para realizar acciones que tardan un tiempo considerable en ejecutarse.
- `@AfterClass`: solo puede haber un método con esta anotación. Este método será invocado un sólo vez cuando finalicen todas las pruebas.

En este caso los métodos anotados con `@BeforeClass` y `@AfterClass` deben ser *static* y por tanto los atributos a los que acceden también. Los métodos añadidos anteriormente quedarían así:

```

import static org.junit.Assert.*;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;

public class CalculadoraTest3 {
    private static Calculadora calcu;
    private int resultado;

    @BeforeClass
    public static void creaCalculadora() {
        calcu = new Calculadora(20, 10);
    }

    @AfterClass
    public static void borraCalculadora() {
        calcu = null;
    }
}

```

```
* * * * *
```

}

3.7.4. Pruebas parametrizadas

Supongamos que queremos ejecutar una prueba varias veces con distintos valores de entrada, por ejemplo, queremos probar el método *divide()* con diferentes valores.

JUnit nos permite generar parámetros para lanzar varias veces una prueba con dichos parámetros. Para poder hacer esto seguiremos estos pasos:

1. Debemos añadir la etiqueta `@RunWith(Parameterized.class)` a la clase de prueba, con esto indicamos que la clase va a ser usada para realizar una batería de pruebas. En esta clase se debe declarar un atributo por cada uno de los parámetros de la prueba y un constructor con tantos argumentos como parámetros en cada prueba. Para probar el método *divide()* o cualquiera de los demás métodos definiremos 3 parámetros, dos de ellos para los números con los que se realiza la operación y el tercero para recoger el resultado; el constructor tendrá el siguiente aspecto: `public CalculadoraTest4(int nume1, int nume2, int resul)`.
2. Lo siguiente es definir un método anotado con la etiqueta `@Parameters`, que será el encargado de devolver la lista de valores a probar. En este método se definirán filas de valores para *num1*, *num2* y *resul* (en el mismo orden en que están definidos en el constructor). Por ejemplo, un grupo de valores de prueba sería `{ 20, 10, 2 }`, que para la división equivale a la operación *resul=num1/num2*, es decir $2=20/10$ (sería un caso de prueba correcto).

En el siguiente ejemplo para probar el método *divide()* usamos tres casos de prueba, dos de ellos correctos y uno con resultado erróneo. La clase, *CalculadoraTest4*, quedaría así:

```
import static org.junit.Assert.*;
import java.util.*;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class CalculadoraTest4 {
    private int nume1;
    private int nume2;
    private int resul;

    public CalculadoraTest4(int nume1, int nume2, int resul) {
        this.nume1 = nume1;
        this.nume2 = nume2;
        this.resul = resul; //resultado de la operación
    }

    @Parameters
    public static Collection<Object[]> numeros() {
        return Arrays.asList(new Object[][] {
            { 20, 10, 2 }, { 30, -2, -15 }, { 5, 2, 3 }
        });
    }
}
```

```

    }

    @Test
    public void testDivide() {
        Calculadora calcu = new Calculadora(numel, nume2);
        int resultado = calcu.divide();
        assertEquals(resul, resultado);
    }
}

```

La ejecución produce la salida mostrada en la Figura 3.26, al lado del método de prueba se muestra entre corchetes la prueba de que se trata. En este caso no se prueban todos los métodos a la vez, ya que la lista de valores para las pruebas se ha preparado para la operación de dividir.

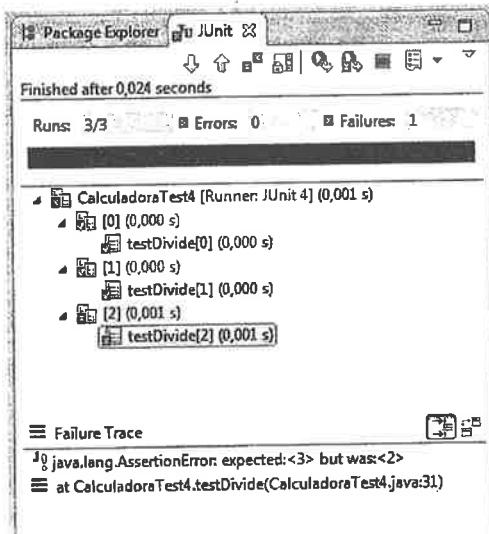


Figura 3.26. Batería de pruebas en JUnit.

ACTIVIDAD 3.2

Realiza pruebas parametrizadas para los métodos *suma()*, *resta()* y *multiplica()* de la clase *Calculadora*.

3.7.5. Suite de pruebas

A veces nos interesa ejecutar varias clases de prueba una tras otra. Es el caso del ejemplo anterior que no podemos ejecutar las pruebas parametrizadas para todos los métodos en una única clase. JUnit proporciona el mecanismo llamado **Test Suites** que agrupa varias clases de prueba para que se ejecuten una tras otra.

Supongamos que se han creado pruebas parametrizadas para los métodos *suma()*, *resta()* y *multiplica()*; y el nombre para las clases de prueba es *CalculadoraTestSuma*, *CalculadoraTestResta* y *CalculadoraTestMultiplica*. Entonces para crear la suite de pruebas pulsamos en la opción de menú: *File->New->Other->Java->JUnit->JUnit Test Suite*, y a continuación en el botón *Next*, véase Figura 3.27. Desde la siguiente pantalla elegimos las clases que formarán parte de la suite de pruebas, marcamos la opción *New JUnit 4 suite* y le damos un nombre, en el ejemplo *VariasPruebas*, a continuación pulsamos en *Finish*.

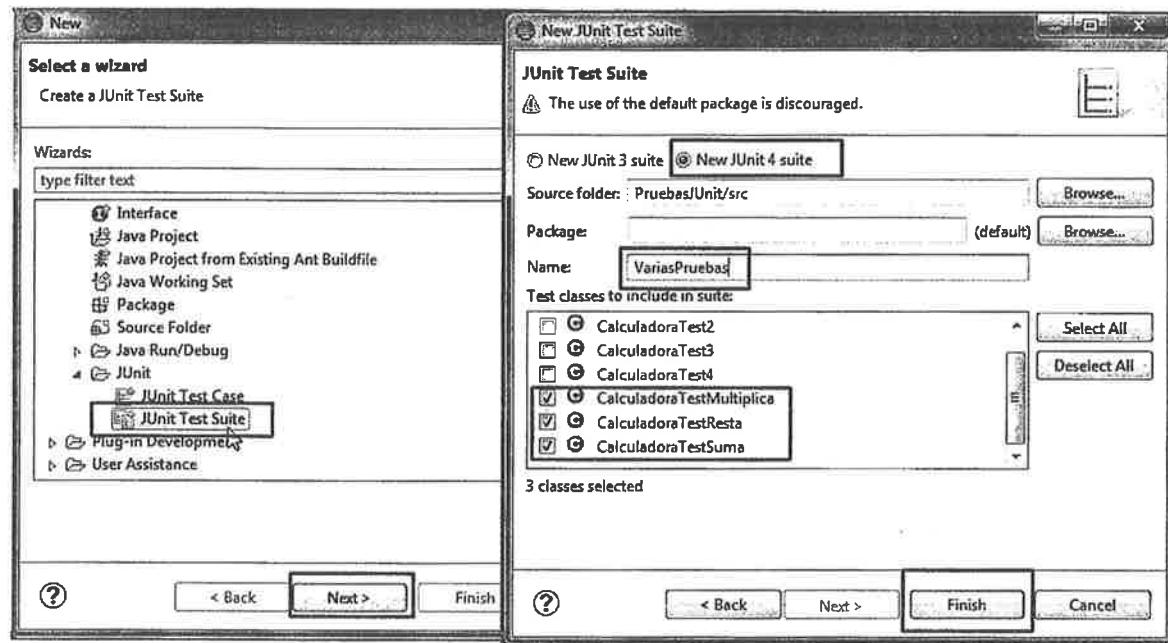


Figura 3.27. Creación de la suite de pruebas.

Se genera la siguiente clase:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({
    CalculadoraTestMultiplica.class,
    CalculadoraTestResta.class,
    CalculadoraTestSuma.class
})

public class VariasPruebas {
```

Donde hay que destacar dos anotaciones. La primera `@RunWith(Suite.class)` indica a JUnit que la clase es una suite de pruebas y en la segunda anotación `@SuiteClasses()` se indican las clases que forman parte del conjunto de pruebas y que son las que se van a ejecutar. Dentro de la clase no se genera ninguna línea de código.

La Figura 3.28 muestra la ejecución de la suite de pruebas.

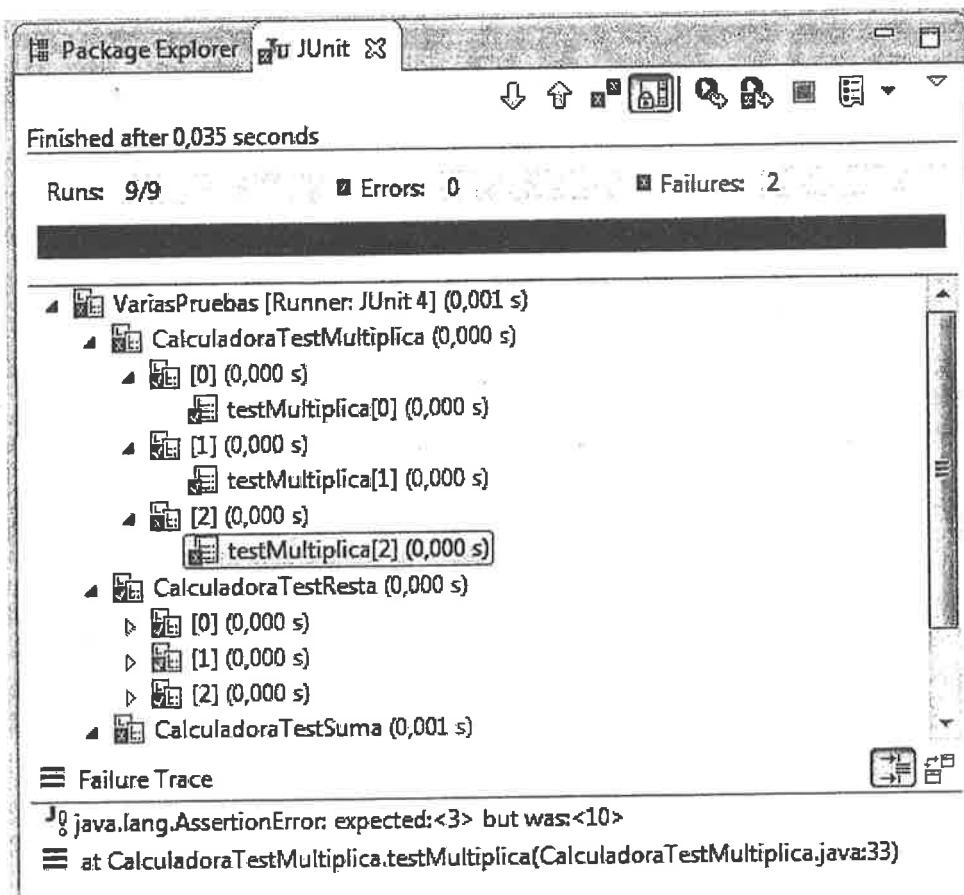


Figura 3.28. Ejecución de la suite de pruebas.

¡INTERESANTE!

DBUnit es una extensión de JUnit que nos permite realizar test unitarios de clases que interactúan con bases de datos. Utiliza ficheros XML para cargar los datos de prueba en la base de datos. Una vez cargadas las tablas con los datos de prueba se lanza la prueba. Al finalizar la misma se restauran los datos en la base de datos para que el siguiente test de prueba se encuentre la base de datos en el estado original. Con esto se evita que datos corruptos procedentes de pruebas fallidas queden en la base de datos ocasionando problemas a las siguientes pruebas.

Los componentes principales son:

IDatabaseConnection: interfaz que representa una conexión DBUnit a una base de datos.

IDataSet: interfaz que representa una colección de tablas.

DatabaseOperation: clase abstracta que representa la operación que se va a realizar sobre la base de datos antes y después de cada prueba.

COMPRUEBA TU APRENDIZAJE

1. ¿Qué estrategias se siguen para probar el software? Si las pruebas de unidad funcionan, ¿es necesario hacer la prueba de integración?
2. Considérese una aplicación bancaria, donde el usuario puede conectarse al banco a través de Internet y realizar una serie de operaciones bancarias. Una vez que ha accedido al banco con las consiguientes medidas de seguridad puede realizar una serie de operaciones. La operación que se va a gestionar requiere la siguiente entrada:
 - *Código del banco*: puede estar en blanco o puede ser un número de 3 dígitos. En este último caso, el primero de ellos tiene que ser mayor que 1.
 - *Código de sucursal*: número de cuatro dígitos. El primero de ellos mayor de 0.
 - *Número de cuenta*: número de cinco dígitos.
 - *Clave personal*: valor alfanumérico de cinco posiciones.
 - *Orden*: puede estar en blanco o ser uno de los valores siguientes: “Talonario” o “Movimientos”.

El programa responde de la siguiente manera:

- Si *Orden* tiene el valor “Talonario”, el usuario recibe un talonario de cheques.
- Si *Orden* tiene el valor “Movimientos”, el usuario recibe los movimientos del mes en curso.
- Si *Orden* está en blanco, el usuario recibe los dos documentos.
- Si ocurre algún error en la entrada de datos, el programa muestra un mensaje de error sobre el dato implicado.

Se pide definir las clases de equivalencia, casos de prueba válidos y casos de prueba no válidos que cubran una sola clase no válida.

3. Rellena en la siguiente tabla los casos de prueba tomando como referencia las reglas del análisis de valores límite:

Condiciones de entrada y de salida	Casos de prueba
Una variable toma valores comprendidos entre -4 y 4 (enteros)	
El programa lee un fichero que contiene de 1 a 100 registros	
El programa deberá generar de 1 a 5 listados	
El número de alumnos para calcular la nota media es 35	
La función deberá devolver un array de enteros, de 1 a 10 valores	

Realiza después un programa Java para probar la función que devuelve el array de enteros. Utiliza los casos de prueba que hayas definido.

4. A partir del diagrama de flujo mostrado en la Figura 3.29 construye el grafo de flujo. Indica el número de nodos, aristas, regiones, nodos predicado, la complejidad ciclomática y el conjunto de caminos independientes.

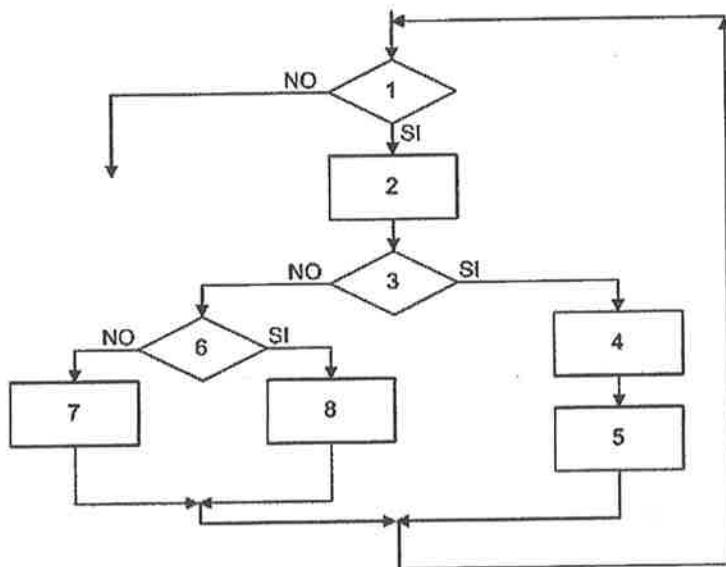


Figura 3.29. Ejercicio 4.

5. Realiza el grafo de flujo, calcula la complejidad ciclomática, define el conjunto básico de caminos, elabora los casos de prueba para cada camino y evalúa el riesgo para la siguiente función Java:

```

static int Contador1(int x, int y) {
    Scanner entrada = new Scanner(System.in);
    int num, c = 0;
    if (x > 0 && y > 0) {
        System.out.println("Escribe un número");
        num = entrada.nextInt();
        if (num >= x && num <= y){
            System.out.println("\tNúmero en el rango");
            c++;
        }
        else
            System.out.println("\tNúmero fuera de rango");
    }
    else
        c = -1;
    entrada.close();
    return c;
} // 
```

6. Desarrolla una batería de pruebas para probar el método *DevuelveFecha()* de la clase *Fecha* que se expone a continuación. El método recibe un número entero y devuelve un String con un formato de fecha que dependerá del valor de dicho número. Si el número recibido es distinto de 1, 2 y 3 el método devuelve ERROR. La clase es la siguiente:

```
import java.text.SimpleDateFormat;
```

```

import java.util.Date;

public class Fecha {
    SimpleDateFormat formato;
    Date hoy;

    public Fecha() {
        hoy = new Date();
    }

    public String DevuelveFecha(int tipo) {
        String cad = "";
        switch (tipo) {
            case 1: {
                formato = new SimpleDateFormat("yyyy/MM");
                cad = formato.format(hoy);
                break;
            }
            case 2: {
                formato = new SimpleDateFormat("MM/yyyy");
                cad = formato.format(hoy);
                break;
            }
            case 3: {
                formato = new SimpleDateFormat("MM/yy");
                cad = formato.format(hoy);
                break;
            }
            default: {
                cad = "ERROR";
            }
        }
        return cad;
    }
}

```

7. Escribe una clase de pruebas para probar el método *calculo()* de la clase *Factorial*. En el método se comprueba si el número es menor que 0, en este caso se lanza la excepción *IllegalArgumentException* con el mensaje *Número n no puede ser < 0*. Si el valor del factorial calculado es menor que 0 es que ha ocurrido un error de desbordamiento, en este caso se lanza la excepción *ArithmaticException* y se lanza el mensaje *Overflow, número n demasiado grande*. La clase a probar es la siguiente:

```

public class Factorial {
    public static int calculo(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("Número " + n +
                " no puede ser < 0");
        }
        int fact = 1;
        for (int i = 2; i <= n; i++)
            fact *= i;
    }
}

```

```

        if (fact < 0) {
            throw new ArithmeticException("Overflow, número "+
                n +" demasiado grande");
        }
        return fact;
    }
}

```

8. Se trata de implementar la entrada de datos desde una web para la petición de un CD de música. El cliente introduce el código del CD y una cantidad. El código deberá comenzar por un dígito, seguido de un guión y 7 dígitos más. Solo se podrá pedir un máximo de 3 CDs. Si el código del CD no está en el catálogo se produce un error (ER1) y si la cantidad solicitada supera el stock de dicho CD se produce otro error (ER2). En el caso de que todo vaya bien se enviará el CD al solicitante con el precio mostrado en la web, salvo que tenga oferta; en este caso se le aplicará al precio la oferta.

Realiza la tabla con las condiciones de entrada y las clases de equivalencia válidas y no válidas. Muestra varios casos de prueba, en los que se valide todo correctamente y que den error y no se lleve a cabo el pedido, en este caso indica por qué no se lleva a cabo el pedido.

ACTIVIDADES DE AMPLIACIÓN

1. Realiza el grafo de flujo, calcula la complejidad ciclomática, define el conjunto básico de caminos, elabora los casos de prueba para cada camino y evalúa el riesgo para la siguiente función Java:

```

static int Contador2(int x, int y) {
    Scanner entrada = new Scanner(System.in);
    int num, c = 0;
    if ( x > 0 && y > 0) {
        System.out.println("Escribe un número");
        num = entrada.nextInt();
        while (num != 0) {
            if ( num >= x && num <= y ) {
                System.out.println("\tNúmero en el rango");
                c++;
            } else
                System.out.println("\tNúmero fuera de rango");
            System.out.println("Escribe un número");
            num = entrada.nextInt();
        }//fin while
    }
    else
        c = -1;
    entrada.close();
    return c;
}

```

2. Escribe una clase de pruebas para probar los métodos de la clase *TablaEnteros*. En esta clase de prueba crea un método con la anotación `@BeforeClass` en el que inicialices un array de enteros para usarlo en las pruebas de los métodos. El método *sumaTabla()* suma los elementos del array y devuelve la suma. El método *mayorTabla()* devuelve el elemento mayor de la tabla. Y el método *posicionTabla()* devuelve la posición ocupada por el elemento cuyo valor se envía.

En el constructor se comprueba si el número de elementos de la tabla es nulo o 0, en este caso se lanza la excepción *IllegalArgumentException* con el mensaje *No hay elementos*. El método *posiciónTabla* también lanza la excepción, *java.util.NoSuchElementException*, en el caso de que no se encuentre el elemento en la tabla. Hay que añadir otros dos métodos de prueba para probar estas excepciones. La clase a probar es la siguiente:

```
public class TablaEnteros {
    private Integer[] tabla;

    TablaEnteros(Integer[] tabla) {
        if (tabla == null || tabla.length == 0)
            throw new IllegalArgumentException("No hay elementos");
        this.tabla = tabla;
    }

    //devuelve la suma de los elementos de la tabla
    public int sumaTabla() {
        int suma = 0;
        for (int i = 0; i < tabla.length; i++)
            suma += tabla[i];
        return suma;
    }

    //devuelve el mayor elemento de la tabla
    public int mayorTabla() {
        int max = -999;
        for (int i = 0; i < tabla.length; i++)
            if (tabla[i] > max)
                max = tabla[i];
        return max;
    }

    //devuelve la posición de un elemento cuyo valor se pasa
    public int posicionTabla(int n) {
        for (int i = 0; i < tabla.length; i++)
            if (tabla[i] == n)
                return i;
        throw new java.util.NoSuchElementException("No existe:" + n);
    }
}
```


CAPÍTULO 4

OPTIMIZACIÓN Y DOCUMENTACIÓN

CONTENIDOS

- Control de versiones. Subversión.
- Cliente tortoise. Servidor Visual SVN
- Control de versiones en ECLIPSE y en SQL DEVELOPER
- Etiquetas de documentación.
- Documentación de clases con Javadoc
- Refactorización, cuando refactorizar, refactorización de código en ECLIPSE

OBJETIVOS

- Describir las características de las herramientas de control de versiones
- Utilizar herramientas de control de versiones integradas en los entornos de desarrollo
- Identificar los patrones de refactorización más usuales
- Aplicar patrones de refactorización en ECLIPSE
- Documentar clases utilizando herramientas de documentación integrada en ECLIPSE

RESUMEN DEL CAPÍTULO

En este capítulo se estudiará qué es el control de versiones y para qué se utiliza en la elaboración de proyectos de software. Se utilizarán herramientas cliente y herramientas servidor, además se realizarán ejercicios utilizando estas herramientas en ECLIPSE y SQL DEVELOPER. También se aprenderá a documentar clases java utilizando Javadoc de ECLIPSE, se realizará la documentación de proyectos ya elaborados. Y por último, se aprenderá el concepto de refactorización, se aprenderá por qué es necesario refactorizar y se realizará la refactorización de proyectos ya elaborados en ECLIPSE.

4.1. INTRODUCCIÓN

El objetivo del capítulo es que el alumno maneje herramientas de control de versiones, herramientas para documentar los programas y herramientas de refactorización. El uso de este tipo de herramientas aseguran por un lado que los proyectos mantengan una trazabilidad de todos los cambios que han sufrido a lo largo de su vida, y quién o quienes han realizado los cambios, cuestión de vital importancia cuando se trabaja en equipos de software.

Por otro lado, el uso de este tipo de herramientas garantiza que los proyectos estén bien documentados informando de lo que hace cada una de las clases y métodos que los forman, y bien refactorizados simplificando el código de los programas y favoreciendo su lectura, entendimiento y fácil mantenimiento. Cuestiones muy importantes para conseguir ser unos buenos programadores.

4.2. CONTROL DE VERSIONES

Se puede definir control de versiones como la capacidad de recordar todos los cambios que se realizan tanto en la estructura de directorios como en el contenido de los archivos. Esto es de mucha utilidad cuando se desea recuperar un documento, o una carpeta, o un proyecto en un momento concreto de su desarrollo. También es muy útil cuando se necesita mantener un cierto control de los cambios que se realizan sobre documentos, archivos o proyectos que comparten varias personas o un equipo de trabajo, se hace necesario saber qué cambios se hacen, quién los hace y cuándo se realizan.

TERMINOLOGÍA

Antes de entrar en la instalación y manejo de herramientas de control de versiones veamos los términos que se utilizan y son comunes para la mayoría de estas herramientas:

- **Repositorio.** Lugar donde se almacenan todos los datos y los cambios. Puede ser un sistema de archivos en un disco duro, un banco de datos, un servidor, etc.
- **Revisión o versión.** Una revisión es una versión concreta de los datos almacenados. Algunos sistemas identifican las revisiones con un número contador (como *Subversion*). Otros identifican las revisiones mediante un código de detección de modificaciones (en Git usa SHA1). La última versión se la identifica como la cabeza o **HEAD**.
- **Etiquetar o Rotular (tag).** Cuando se crea una versión concreta en un momento determinado del desarrollo de un proyecto se le pone una etiqueta, de forma que se pueda localizar y recuperar en cualquier momento. Las etiquetas permiten identificar de forma fácil revisiones importantes en el proyecto (por ejemplo una versión publicada). En algunos sistemas se considera una etiqueta cuando se crea una rama en la que los ficheros no evolucionan, es decir se congela.
- **Tronco (trunk),** es el tronco o la línea principal de desarrollo de un proyecto.
- **Rama o ramificar (branch).** Las ramas son copias de archivos, carpetas o proyectos. Cuando se crea una rama se crea una bifurcación del proyecto y se crean dos líneas de desarrollo. Son motivos habituales de creación de ramas la creación de nuevas funcionalidades o la corrección de errores.
- **Desplegar (Checkout).** Crear una copia de trabajo del proyecto, o de archivos y carpetas del repositorio en el equipo local. Por defecto se obtiene la última versión, aunque también se puede indicar una versión concreta. Con el *checkout* se vincula la carpeta de

trabajo del equipo local con el repositorio, y se crean los metadatos de control de versiones (carpetas y archivos ocultos que se crean `.svn`).

- **Confirmar (commit o check-in).** Se realiza commit cuando se confirman los cambios realizados en local para integrarlos al repositorio.
- **Exportación (export).** Similar a *Checkout*, pero en esta ocasión no vincula la copia con el repositorio. Es una copia limpia sin los metadatos de control de versiones.
- **Importación (import).** Es la subida de carpetas y archivos del equipo local al repositorio. Se puede hacer en cualquier momento desde el sistema de archivos.
- **Actualizar (update).** Se realiza una actualización cuando se desea integrar los cambios realizados en el repositorio en la copia de trabajo local. Los cambios pueden ser realizados por personas del equipo de trabajo.
- **Fusión (merge).** Una fusión consiste en unir los cambios realizados sobre uno o varios archivos en una única revisión. Se suele realizar cuando hay varias líneas de desarrollo separadas en ramas y en alguna etapa se necesitan fusionar los cambios hechos entre ramas o en una rama con el tronco principal, o viceversa.
- **Conflicto.** Ocurre cuando dos usuarios crean una copia local (*Checkout*) de la misma versión de un archivo, uno de ellos realiza cambios y envía los cambios (*commit*) al repositorio, y el otro no actualiza (*update*) esos cambios y realiza cambios sobre el archivo e intenta enviar luego sus cambios al repositorio. Entonces se produce el conflicto y el sistema no es capaz de fusionar los cambios. Este usuario deberá resolver el conflicto combinando los cambios o eligiendo uno de ellos.
- **Resolver conflicto.** La actuación del usuario para atender un conflicto entre diferentes cambios al mismo documento.

Para trabajar en proyectos utilizando un sistema de control de versiones, lo primero que hay que hacer es crearse una copia en local de la información del repositorio con *checkout*, de esta manera se vincula la copia con el repositorio, a continuación el usuario realizará sus modificaciones y una vez finalizadas sube las modificaciones al repositorio con *commit*. Si la copia del usuario ya está vinculada al repositorio, antes de modificar y realizar cambios tiene que hacer *Update*, para asegurarse que los cambios se realizan sobre la versión última del repositorio.

4.2.1. Subversión. Ciclo de vida de subversión

Subversión es una herramienta multiplataforma (Win32, Linux, Mac, etc.) de código abierto para el control de versiones. Usa una base de datos central, el *repositorio*, que contiene los archivos cuyas versiones y respectivas historias se controlan. El repositorio actúa como un servidor de ficheros, con la capacidad de recordar todos los cambios que se hacen tanto en sus directorios como en sus ficheros.

Cuando un proyecto de software es desarrollado por un equipo de personas es indispensable llevar un control bajo un método de trabajo, seguir unas normas y aplicar *las buenas prácticas* en el uso de las herramientas. Así el proyecto debe verse como un árbol que tiene su tronco (*trunk*) donde está la línea principal de su desarrollo; que tiene sus ramas (*branches*) en la que se añadirán nuevas funciones o se corregirán errores; y que además tiene sus etiquetas (*tags*) para marcar situaciones importantes, o versiones finalizadas. Así la estructura de carpetas recomendada en la creación de proyectos utilizando estas herramientas y la funcionalidad que se le debe dar a cada carpeta dentro del repositorio son las siguientes:

- **Trunk (tronco):** base común para guardar las carpetas del proyecto o trabajo a controlar. Es donde está la versión básica, es decir, la rama de desarrollo principal.
- **Tags (etiquetas):** una etiqueta es una copia del proyecto, de una carpeta o de un archivo que se hace con el objetivo de obtener una versión que no se va a modificar. Deben ser copias del tronco (*trunk*). Útil para crear versiones ya finalizadas, aquí se guardarán las versiones cerradas de los proyectos.
- **Branches (ramas):** en las ramas se desarrollan versiones que luego se van a publicar. Es una copia del *trunk*, de un proyecto, de una carpeta o de un archivo con la intención de modificar sobre ella, para conseguir un producto final diferente y alternativo al original. Es la ramificación del código, es decir, modificaciones de versiones cerradas.

El diagrama de la Figura 4.1 muestra el ciclo de vida de subversión.

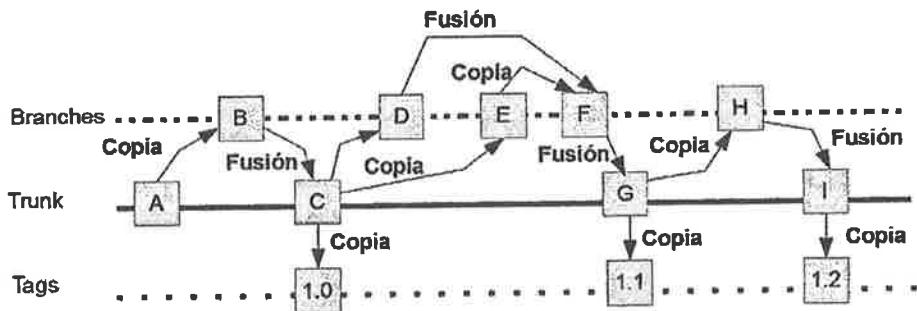


Figura 4.1. Ciclo de vida de subversión.

Donde:

- A. Partimos del desarrollo inicial, en el trunk.
- B. Se crea una rama porque hay que añadir una nueva funcionalidad.
- C. Mientras tanto se ha corregido un bug en el tronco principal (C). Una vez que está lista la rama se fusiona en el trunk. Cuando esté lista esta versión libre de bugs y con nueva funcionalidad se crea la primera versión disponible para el público (*Tag 1.0*).
- D. Despues de salir la primera versión, se han detectado nuevos bugs, y se necesita añadir nuevas funcionalidades. Entonces se crea una rama para desarrollar una nueva funcionalidad.
- E. Se ha creado otra rama porque se necesitan otras funcionalidades.
- F. Se realiza la fusión de las dos ramas, primero se realiza una copia de una de ellas y luego se fusiona con la otra.
- G. Se incorporan las nuevas funcionalidades al tronco principal. Una vez que está lista esta revisión se crea una nueva versión para el público (*Tag 1.1*).
- H. Se han detectado nuevos bugs y se necesita añadir nuevas funcionalidades. Se crea una nueva rama para desarrollar una nueva funcionalidad.

- I. Se incorporan las nuevas funcionalidades al tronco principal. Se crea una nueva versión para el público (*Tag 1.2*).

4.2.2. Cliente TortoiseSVN

TortoiseSVN es un cliente gratuito de código abierto para el sistema de control de versiones *Subversion*. Al instalarse aparece *integrado en la shell de Windows*. Los menús contextuales de *TortoiseSVN* también funcionan en otros administradores de archivos, y en la ventana *Fichero/Abrir* que es común a la mayoría de aplicaciones estándar de *Windows*. Desde la URL <http://tortoisessvn.net/downloads.html> se puede descargar la última versión de la herramienta, existe una versión para sistemas operativos de 32 bits y otra para 64, también existe el instalador del idioma. En esta unidad se trabajará con la versión 1.8.4, primero se instala la herramienta y a continuación el idioma.

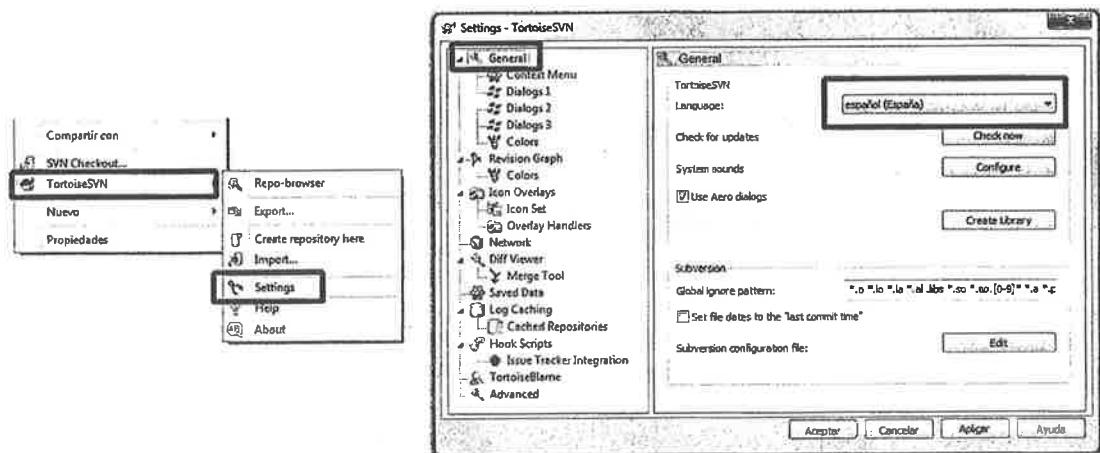


Figura 4.2. Configuración de idioma en *TortoiseSVN*.

La instalación es sencilla, se aceptan los términos de la licencia, se indica la carpeta donde se va a instalar y se pulsa a instalar. A continuación se instala el idioma. Para configurar el idioma se pulsa el botón derecho del ratón desde el escritorio, o desde un directorio del sistema de archivos para que aparezca el menú contextual, se selecciona *TortoiseSVN/Settings*, y en el cuadro de diálogo que aparece dentro de *General* se selecciona el idioma en *Language* (véase la Figura 4.2). Observa que ahora las opciones de *TortoiseSVN* aparecen en español (véase la Figura 4.3).

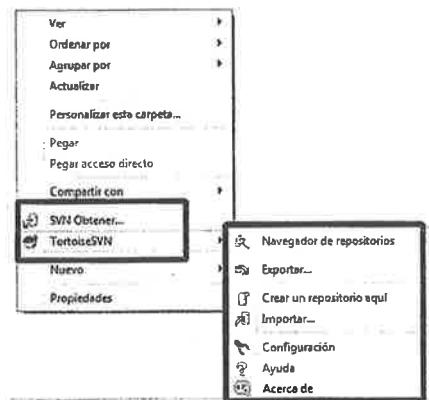


Figura 4.3. Opciones de *TortoiseSVN* en español.

Una vez instalada la herramienta veamos cómo crear un repositorio privado en el propio PC y controlar los archivos y documentos almacenados en ese repositorio.

Para crear el repositorio se crea una carpeta en el disco duro, se selecciona y desde el menú contextual se elige *TortoiseSVN*, se accede a la opción *Crear un repositorio aquí*. Seguidamente se hace clic en el botón *Crear estructura de carpetas* (al pulsar este botón se crea la estructura *branches, tags y trunks*) y luego se pulsa al *Navegador de repositorios* para que se muestre el navegador, véase la Figura 4.4. Observa la estructura de archivos creada y la revisión inicial que es la 1 (creación del repositorio), esta revisión se incrementará automáticamente cada vez que se hace un cambio en el repositorio. También podemos acceder a las diferentes revisiones pulsando el botón *HEAD* que aparece en la parte superior derecha de la ventana.

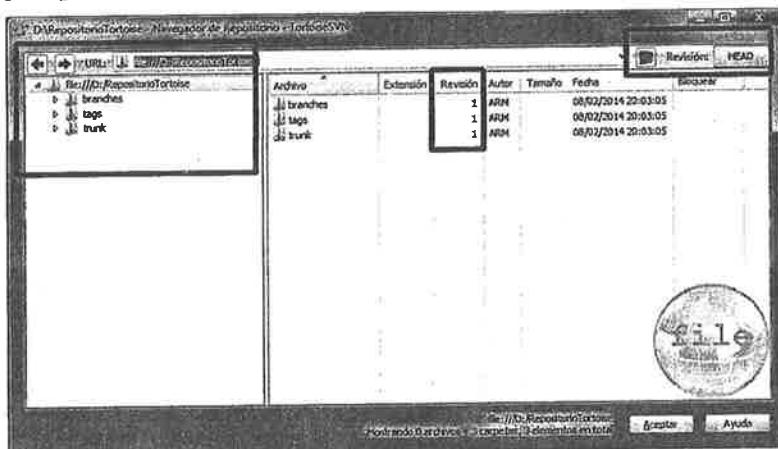


Figura 4.4. Navegador de repositorios.

Se podrán crear todos los repositorios que se deseen y desde el navegador se podrán administrar. Observa que se crea la estructura de carpetas *trunk-tags-branches*, esa estructura puede variar dependiendo de la organización que se deseé. Se puede organizar un repositorio que contiene muchos proyectos por ramas, o por proyectos, véase la tabla:

REPOSITORIO ORGANIZADO POR RAMAS	REPOSITORIO ORGANIZADO POR PROYECTOS
/trunk/proyecto1	/ proyecto1/trunk
/trunk/ proyecto2	/ proyecto1/branches
/branches/ proyecto1	/ proyecto1/tags
/branches/ proyecto2	/ proyecto2/trunk
/tags/ proyecto1	/ proyecto2/branches
/tags/ proyecto1	/ proyecto2/tags

4.2.2.1. Operaciones con Tortoise

Para probar la herramienta realiza una copia de varios archivos y carpetas de tu disco duro para trabajar con ellos, o bien puedes utilizar los archivos y carpetas de la carpeta de recursos de la unidad.

Se puede trabajar con *TortoiseSVN* dentro y fuera del navegador de repositorios. Dentro del navegador y desde el menú contextual de *TortoiseSVN* podremos seleccionar las operaciones a realizar (véase la Figura 4.5). Desde aquí se podrán subir archivos y carpetas al repositorio, con las opciones *Añadir archivo* y *Añadir carpeta*. O crear, renombrar o eliminar carpetas y archivos.

También se podrán realizar copias de trabajo vinculadas al repositorio con *Obtener (checkout)* o realizar simples copias sin vincular al repositorio con *Exportar*. Con *Mostrar el registro* de revisiones se mostrarán todas las operaciones realizadas (como las subidas y eliminaciones de archivos y carpetas), con *Gráfico de revisiones* se visualizará un gráfico de la situación seleccionada.

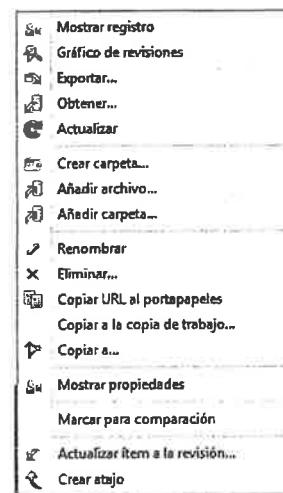


Figura 4.5. Operaciones de TortoiseSVN.

ACTIVIDAD 4.1

Desde el navegador de repositorios, selecciona la carpeta *trunk* y sube una carpeta y dos archivos, añade un mensaje a las subidas. Observa las revisiones que se han ido generando. Accede a cada una de ellas desde el botón *HEAD*, muestra el registro de revisiones (véase la Figura 4.6), marca cada una de las revisiones y observa en la pantalla el mensaje añadido para esa revisión y los archivos implicados. Visualiza el gráfico de revisiones.

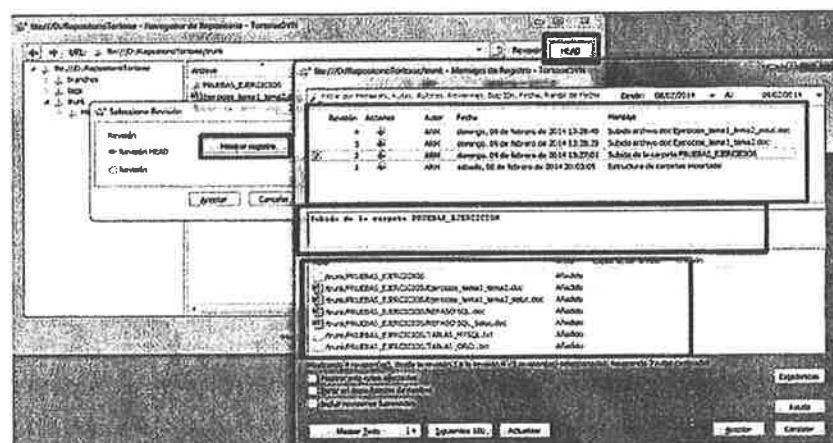


Figura 4.6. Registro de revisiones de TortoiseSVN.

El proceso de trabajo fuera del navegador de repositorios consistirá en hacer *CHECKOUT* para obtener una copia de trabajo en una carpeta local, realizar cambios en esa carpeta local, y confirmar los cambios *COMMIT* para subirlos al repositorio. Y siempre antes de realizar cambios es aconsejable actualizar *UPDATE* con lo que

hay en el repositorio, para asegurarse de trabajar con la última revisión. Luego se realizan los cambios, y una vez finalizados se confirman, véase la Figura 4.7.

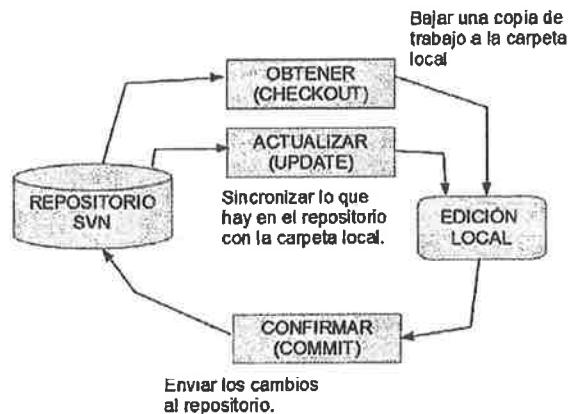


Figura 4.7. Operaciones básicas de control de versiones.

ACTIVIDAD 4.2

Desde el sistema de archivos, crear una carpeta de trabajo para operar con el repositorio, por ejemplo *TrabajoTortoise*.

Desde el menú contextual hacer *CHECKOUT (SVN Obtener)* de los archivos y carpetas subidos en la actividad anterior. En la ventana que se muestra se elige la URL del repositorio, también se puede obtener una revisión concreta (véase la Figura 4.8). En el ejercicio elegir la carpeta *trunk* para descargarnos su contenido.

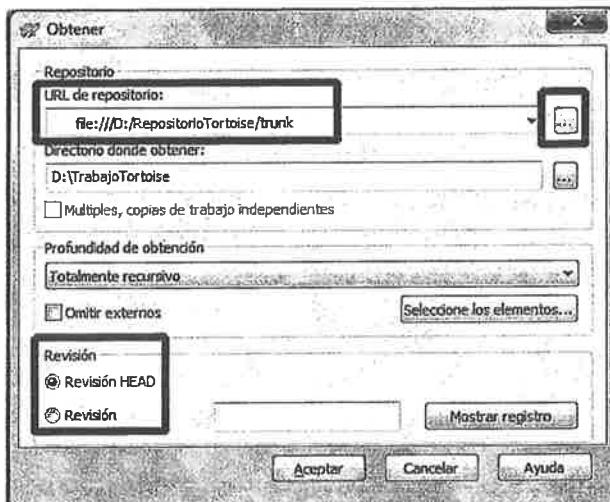


Figura 4.8. Hacer Checkout de la carpeta *trunk*.

Observa que se crea la carpeta oculta **.svn** (si no ves la carpeta oculta, activa la opción de *Mostrar archivos, carpetas y unidades ocultas* desde el menú de Windows *Organizar*, accede a *Opciones de carpeta y búsqueda*, selecciona la pestaña *Ver* y dentro de *Archivos y carpetas ocultos* marca la opción de mostrar).

Realiza cambios sobre algún documento, borra alguno de los documentos, copia un nuevo documento a esa carpeta y confirma los cambios, botón derecho del ratón, elegir *SVN Confirmar*.

En la ventana de confirmar añade un mensaje y marca los cambios. Observa el estado de los cambios, aparece *modificado* en el documento modificado, *falta* en el documento que se ha borrado y *no versionado* en el documento que hemos añadido (véase la Figura 4.9). Pulsa *Aceptar*, y a continuación se muestra la ventana con información de los cambios, la nueva revisión y si se desea *Fusionar* los cambios o *Aceptar*, se pulsa *Aceptar*.

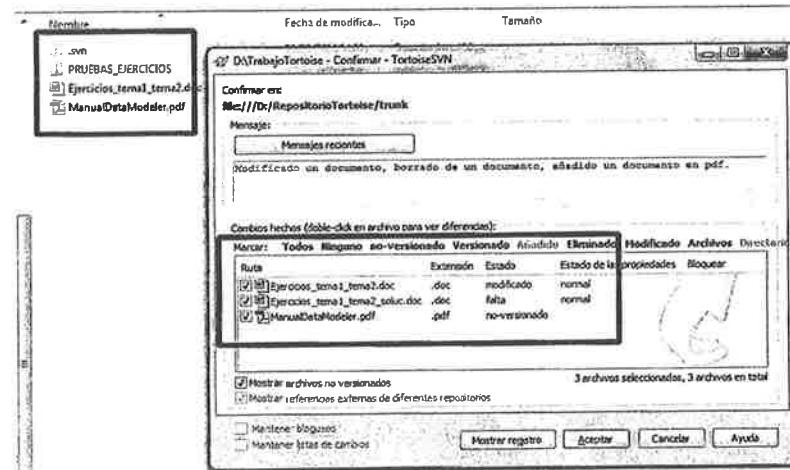


Figura 4.9. Ventana de confirmar cambios.

Muestra el registro (*TortoiseSVN/Mostrar registro*) y observa las modificaciones.

Abre el navegador de repositorios (*TortoiseSVN/Navegador de repositorios*) y observa las revisiones creadas, accede al registro de revisiones. Observa que el archivo borrado permanece en las revisiones iniciales, muy útil si se desea recuperar de nuevo.

RECUERDA: Se debe actualizar *SVN Actualizar* (Update) nuestro directorio de trabajo siempre antes de hacer cualquier cambio, para asegurarnos de que estamos modificando la última versión del repositorio. Al actualizar se harán todos los cambios necesarios (añadir/borrar/modificar ficheros y directorios) para que los datos sean idénticos a los del repositorio (en ese momento).

OBTENIENDO INFORMACIÓN DEL ESTADO DEL REPOSITORIO

Mientras se trabaja con la copia de trabajo es necesario saber qué archivos se han cambiado, añadido, borrado, renombrado, o incluso qué archivos han sido cambiados y confirmados por los demás. Para ello aparecerán unos iconos superpuestos que dan información del estado en *Subversion* de los archivos y carpetas (véase la Figura 4.10). Esta configuración se puede cambiar desde la *TortoiseSVN/Configuración/Iconos Sobrepuestos*.

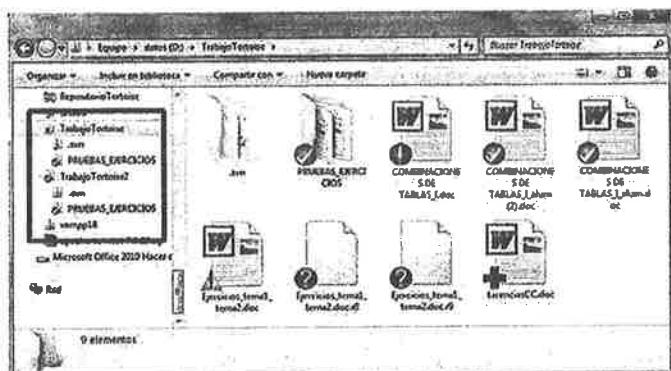


Figura 4.10. Iconos sobrepuertos con información de subversión.

Los iconos más comunes son los siguientes:

	Una copia de trabajo recién obtenida y sincronizada indica estado de <i>Subversion</i> normal.
	Al editar un archivo, el estado cambia a modificado y el ícono cambia entonces a una marca de exclamación roja.
	Este ícono muestra que algunos archivos o carpetas dentro de la carpeta actual se han marcado para ser eliminados del control de versiones, o bien que falta un archivo que está bajo el control de versiones dentro de una carpeta.
	El signo más indica que el archivo o carpeta está programado para ser añadido al control de versiones, se le ha indicado que se añadirá desde el menú <i>TortoiseSVN/Añadir</i> .
	Si durante una actualización ocurre un conflicto, el ícono cambia a un signo de exclamación amarillo.
	Si se ha bloqueado un archivo, y el estado de <i>Subversion</i> es normal, este ícono recordará que se debe liberar el bloqueo si no se está utilizando para permitir a los demás que puedan confirmar sus cambios en el archivo.
	Este ícono muestra los archivos y carpetas que no están bajo el control de versiones pero tampoco han sido ignorados. Este ícono superpuesto es opcional. Suelen aparecer cuando hay conflictos, marca con el ícono a las versiones del archivo.

ACTIVIDAD 4.3

Observa los iconos superpuestos asociados a los archivos y carpetas.

Prueba a copiar nuevos archivos en la copia de trabajo y programarlos para subirlos al repositorio con el menú *TortoiseSVN/Añadir*.

Prueba a realizar modificaciones sobre un archivo de la copia de trabajo, y sin validarla comprueba el archivo con la versión anterior del repositorio, desde el menú *TortoiseSVN* sobre el archivo, y elige *Comparar con la revisión Anterior*. Observa las diferencias entre uno y otro archivo.

RESOLVER CONFLICTOS

Una vez que se hayan realizado los cambios en la carpeta de trabajo se subirán al repositorio con la opción *SVN Confirmar*. Es conveniente añadir comentarios para recordar el momento del cambio. Si al confirmar se producen *conflictos* es porque alguien realizó cambios en el mismo fichero y lo subió al repositorio antes que nosotros, *TortoiseSVN* avisará de que la operación ha fallado y no subirá los cambios. Aconseja que se actualice a la copia del repositorio o que se cancele la operación. Si se elige la opción de actualizar, *TortoiseSVN* creará una copia de cada archivo en la carpeta de trabajo, y el archivo pasará al estado *En conflicto*. Para resolver el conflicto se seleccionarán *Resolver* del menú *TortoiseSVN*.

Para simular una situación de conflicto, crea una carpeta de trabajo nueva, por ejemplo *TrabajoTortoise2* y haz *CHECKOUT (Obtener)* de los archivos y carpetas del repositorio subidos en las actividades anteriores, antes asegúrate de que las dos carpetas de trabajo tengan la misma revisión, es decir, confirma los cambios de la carpeta *TrabajoTortoise* y actualiza (*commit* y *update*) antes de bajar los datos del repositorio a la nueva carpeta *TrabajoTortoise2*.

Realiza cambios en un archivo de una de las carpetas de trabajo, y confírmalos en el repositorio, por ejemplo en *TrabajoTortoise*. Realiza cambios en el mismo archivo pero en la otra copia de trabajo, *TrabajoTortoise2*. Al validar *TortoiseSVN* muestra una ventana indicando que la confirmación falló, que el archivo a modificar está desactualizado porque no coinciden las revisiones, e indica que se debe de actualizar, véase la Figura 4.11. Al pulsar *Aceptar* pedirá: que se actualice y se vuelve a intentar la subida; o que no se actualice la copia y se cancele. Se pulsa la opción de actualizar y se muestra una ventana en la que se indica que se ha actualizado a la versión del repositorio, pero que el archivo está en conflicto (véase la Figura 4.12).

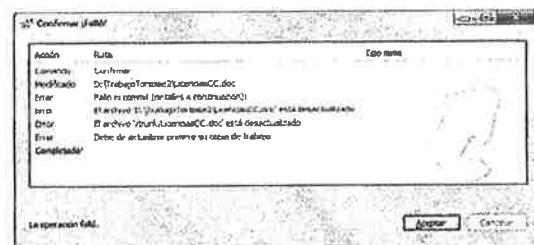


Figura 4.11. Fallo en la validación por desactualización de archivo.

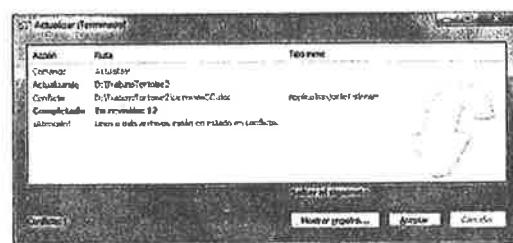


Figura 4.12. Actualización y archivo en conflicto.

Con los archivos en conflicto la validación no podrá ser realizada, así pues se pulsa *Cancelar* para salir de la validación, el conflicto se tiene que resolver de forma manual. Observa que en la carpeta de trabajo (la última en validar) se han creado dos archivos con el icono superpuesto ?, indicando que hay un conflicto entre archivos, aparecen dos, observa que la herramienta los etiqueta añadiendo como extensión el número de revisión, por ejemplo *.r11* y *.r12*, que son las revisiones en las que está el conflicto. Véase la Figura 4.13.

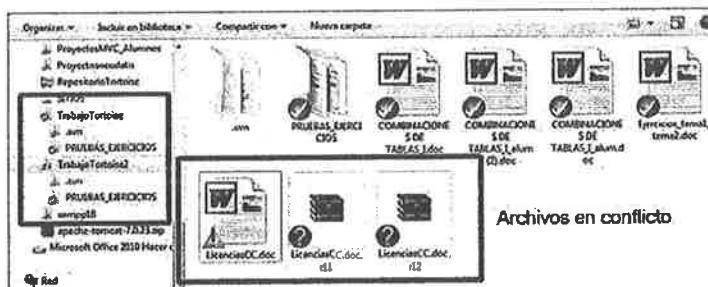


Figura 4.13. Archivos creados en el conflicto

Para resolver el conflicto se abre el menú *TortoiseSVN* (botón derecho dentro de la carpeta con el conflicto) y se elige *Resolver*, se selecciona el archivo y se pulsa *Aceptar*. El archivo se quedará en estado modificado pendiente de la confirmación al repositorio. Ahora se añadirían los cambios y se validaría. Se perderían los cambios realizados por el primero que validó.

4.2.2.2. Creación de ramas y etiquetas con Tortoise

Como se vio al principio, una de las características de los sistemas de control de versiones es la posibilidad de crear líneas separadas de desarrollo para realizar cambios, corregir errores o añadir funcionalidades, esto se conoce como crear una *rama*. Tan pronto como la nueva característica es lo suficientemente estable, la rama de desarrollo se *fusiona* de nuevo en la rama principal (*trunk*, troncal o tronco).

Otra característica de estos sistemas es la posibilidad de marcar revisiones particulares (por ejemplo, una versión lanzada a producción), para que se pueda en cualquier momento recrear un cierto entorno o compilación. Este proceso se conoce como *etiquetar*.

En realidad al crear una rama o etiqueta, no se hacen copias completas del repositorio, sino que se crean vínculos internos, apuntando a una revisión y árbol específicos. Como resultado, las ramas y las etiquetas son muy rápidas de crear, y casi no conllevan espacio extra en el repositorio.

CREAR ETIQUETAS

Para crear una etiqueta, nos posicionamos en la carpeta o archivos del repositorio local de trabajo que queramos etiquetar, y hacemos clic con el botón derecho. Después se selecciona el elemento *Rama/etiqueta* del menú de *TortoiseSVN*. En la ventana que aparece se indica el origen y el destino de la etiqueta a crear, y sobre qué elementos se crea la etiqueta o copia.

Antes de crear una rama o una etiqueta hay que crear en el repositorio (desde el *Navegador del repositorio*) una carpeta en *branches* o en *tags* (dependiendo de si se crea rama o etiqueta) para guardar los archivos y que no cuelguen directamente de *branches* o en *tags*. En el ejemplo se crea la carpeta *PRUEBAS_EJERCICIOS* dentro de *tags*.

Al crear la etiqueta se muestra la ventana de la Figura 4.14 en la que se indica lo siguiente:

- En *CT/URL*: se indica la carpeta que se desea etiquetar del repositorio local, por ejemplo *trunk/PRUEBAS_EJERCICIOS* (véase Figura 4.14).
- En *Ruta Destino*: se indica la carpeta destino de la etiqueta. Para ello se selecciona la carpeta creada dentro de *tags* (*tags/PRUEBAS_EJERCICIOS*) y se añade un nombre de versión, por ejemplo *Versión1*, que creará una carpeta nueva donde se guardarán todos los archivos a etiquetar (véase la Figura 4.14). En el *Navegador del repositorio*

- se debe mostrar la carpeta creada dentro de *tags*, y dentro de la *Versión1* aparecerán los archivos etiquetados, véase la Figura 4.15.

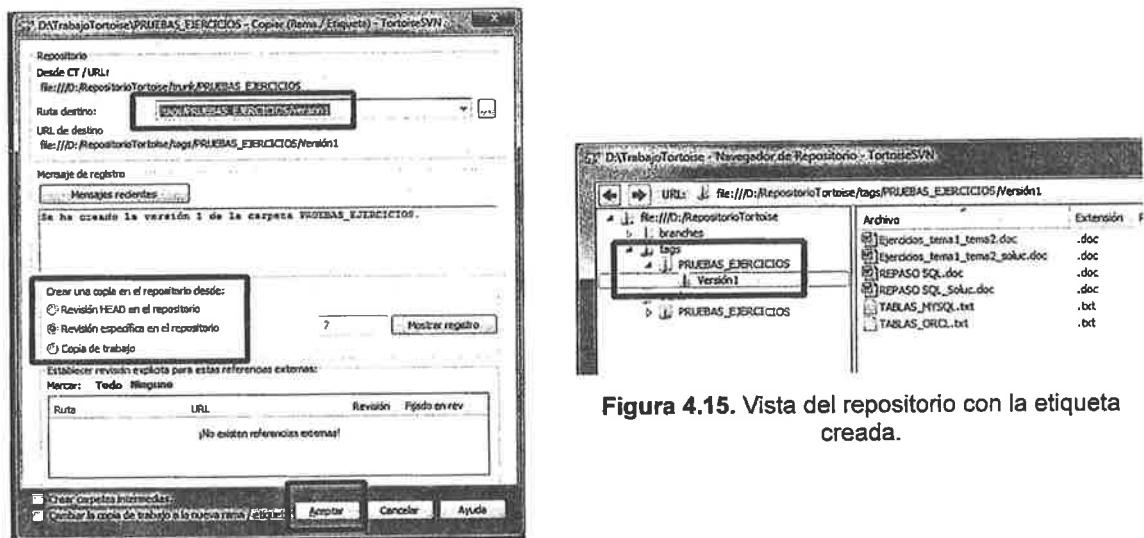


Figura 4.14. Creación de etiqueta.

Figura 4.15. Vista del repositorio con la etiqueta creada.

Para elegir el origen de la copia se dispone de tres opciones:

- Revisión HEAD en el repositorio:** la nueva etiqueta/rama se copia directamente en el repositorio desde la revisión HEAD del repositorio.
- Revisión específica en el repositorio:** la nueva etiqueta/rama se copia directamente en el repositorio, eligiendo una versión anterior. Con el botón *Mostrar registro* se pueden ver las revisiones y seleccionarlas. De nuevo no se transfiere datos desde la copia local de trabajo.
- Copia de trabajo:** la nueva etiqueta/rama es una copia idéntica de la copia local de trabajo. Si se han cambiado algunos archivos a una revisión anterior en la copia de trabajo, o si se han hecho cambios locales, esto es exactamente lo que irá a la copia.

CREAR RAMAS

El proceso es similar a crear una etiqueta. Primero hay que crear las carpetas destino dentro de *branches* desde el *Navegador del repositorio*, poniendo el mismo nombre de la carpeta a enramar. Aunque *Subversion* por sí mismo no hace ninguna distinción entre etiquetas y ramas, la forma en la que normalmente se usan ya se explicó en los anteriores apartados.

Las etiquetas se usan típicamente para crear una copia estática de un proyecto en una etapa concreta. Trabajar en una revisión etiquetada (es decir dentro de tag) no es una buena idea. Pero si se necesita hacer cambios de una versión etiquetada, la forma correcta de manejar la situación es crear primero una nueva rama desde la etiqueta, hacer los cambios en la rama, y luego crear una nueva etiqueta para esta rama. Recuerda que si se modifica una copia de trabajo creada desde una rama y se confirman los cambios, los cambios irán a la rama y no al tronco.

ACTIVIDAD 4.4

Crea una rama con una carpeta que tengas en el repositorio (por ejemplo con la carpeta **PRUEBAS_EJERCICIOS**). Recuerda que antes de crear la rama hay que crear la carpeta dentro de *branches* con el mismo nombre desde el *navegador de repositorios*. Al crear la rama, en *Ruta Destino* indica la carpeta creada del repositorio y añade el nombre de la rama, llámalo **RAMA1**. Añade también un comentario. Una vez creada la rama se debe ver como muestra la Figura 4.16.



Figura 4.16. Vista del repositorio con la rama creada.

Crea una copia de trabajo en tu disco de la rama para realizar modificaciones (*SVN Obtener*).

Añade algún archivo y modifica algún documento.

Confirma los cambios en la rama (*SVN Confirmar*) y añade un comentario.

FUSIONAR RAMAS

Los cambios se realizarán siempre en las ramas, una vez realizados los fusionamos con las copias de trabajo, y desde las copias de trabajo asociadas al tronco se confirman los cambios a *trunk*. *La fusión siempre se realiza sobre copias de trabajo*.

Para fusionar los cambios realizados en una rama creada de una carpeta de *trunk*, en la copia de trabajo asociada a *trunk*, se abre la carpeta sobre la que se creó la rama y dentro de esa carpeta que se ha modificado en la rama, se abre el menú *TortoiseSVN* y se elige *Fusionar*. Aparecen dos opciones de fusión (véase la Figura 4.17):

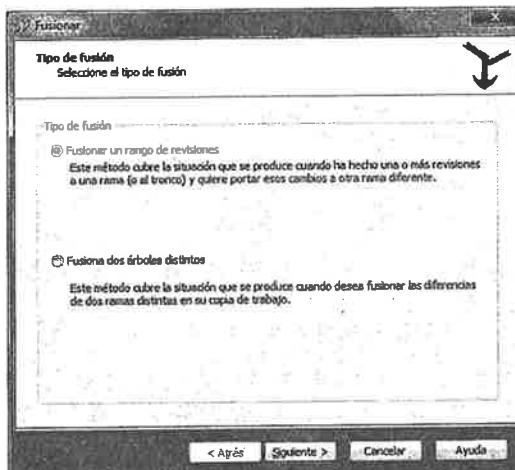


Figura 4.17. Opciones de fusión.

- Fusionar un rango de revisiones.** Este método se elige cuando se ha hecho una o más revisiones en una rama y se desea recoger los cambios a una rama diferente o a la carpeta de trabajo (véase la Figura 4.18). En el campo *URL desde* se escribirá la URL completa de la carpeta de la rama que contiene los cambios que se desea cargar en la copia de trabajo. También se podrá elegir un rango de revisiones.



Figura 4.18. Fusionar un rango de revisiones.



Figura 4.19. Fusionar dos ramas.

- Fusionar dos árboles diferentes.** Este método se utiliza cuando se desea fusionar las diferencias de dos ramas distintas en la copia de trabajo. Lo que se le indica a *Subversion* es que haga los cambios necesarios para ir desde la *URL y revisión inicial del rango a fusionar*, por ejemplo la versión actual (HEAD) del trunk, hasta la *URL y revisión del fin del rango a fusionar*, por ejemplo los cambios de una rama, y que aplique esos cambios a la copia de trabajo (que es la asociada a *trunk*) (véase la Figura 4.19). El resultado final es conseguir que la copia del tronco sea igual a la de la rama.

En la ventana del asistente de fusión se puede indicar opciones para la fusión, aunque la mayoría de las veces se utilizarán las opciones por defecto (véase la Figura 4.20). Algunas de las opciones son *Profundidad de fusión* donde se indica cuántos niveles debe bajar la fusión en la copia de trabajo, por defecto es *Copia de trabajo*, que utiliza la configuración de profundidad existente, es la opción que debe de hacerse. Se puede especificar la forma en la que se manejan los cambios en los finales de línea y en los espacios en blanco. El comportamiento por defecto es tratar todos los espacios en blanco y las diferencias en los finales de líneas como cambios reales que deben fusionarse. La casilla *Forzar la fusión* se usa para evitar conflictos de árbol en los que un borrado entrante afecta a archivos que, o están modificados localmente o ni siquiera están versionados.

NOTA: Si al hacer la fusión aparecen muchos errores o muchos conflictos es posible volver a la situación anterior seleccionando *Revertir* *Revertir...* del menú *TortoiseSVN*.

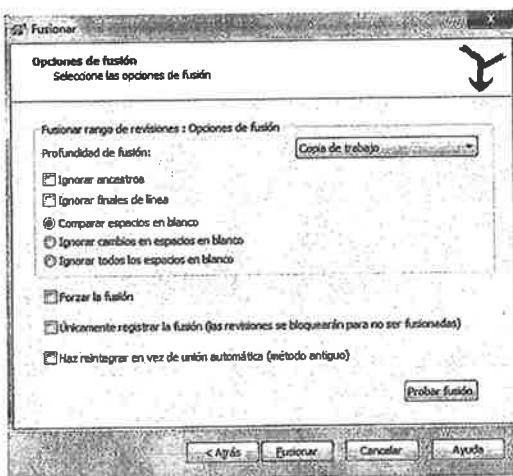


Figura 4.20. Opciones de fusión.

ACTIVIDAD 4.5

Antes de fusionar actualiza la carpeta de trabajo con la versión del repositorio. Se supone que las revisiones de *trunk* son anteriores a las de la rama.

Fusiona la rama creada en la actividad anterior con la opción *Fusionar un rango de revisiones*. Sigue el asistente y antes de aceptar la fusión pulsa el botón *Probar fusión* para comprobar posibles conflictos. Si ocurren conflictos en algún archivo marca la opción de posponer la resolución para los archivos en conflicto.

Confirma los cambios en el repositorio añadiendo comentarios.

Realiza más cambios sobre la rama, añadiendo un archivo, modificando otro y eliminando otro.

Confirma los cambios de la rama en el repositorio, añade un comentario.

Realiza la fusión con la opción *Fusionar dos árboles diferentes*, desde la revisión que se encuentra en *trunk* hasta la revisión de la rama.

Confirma los cambios añadiendo un comentario.

4.2.3. Servidor subversión, VisualSVN Server

VisualSVN Server es un servidor de subversión que se instala y administra fácilmente en Windows, se instala como un servidor web e incluye un servidor *Apache*. El programa se descarga de <http://www.visualsvn.com/server/>. Proporciona una consola de administración de repositorios muy útil llamada *Management Console*, que permite administrar el servidor de forma muy sencilla. Además los repositorios de *VisualSVN Server* están almacenados en el formato estándar y el servidor es accesible por los clientes estándar como *TortoiseSVN*. *VisualSVN Server* es un servidor muy completo y asequible para pequeñas empresas y usuarios corporativos, se puede integrar perfectamente en una red corporativa basada en Windows.

Para instalar el servidor lanzamos el ejecutable, se puede encontrar en la carpeta de recursos del capítulo, la versión utilizada en este capítulo es *VisualSVN-Server-2.7.3*. El asistente es muy sencillo de manejar, se aceptan los términos de la licencia, se selecciona la opción *VisualSVN Server and Management Console* y la *Standard Edition*. Y en la configuración del Server indicamos lo siguiente: en *Location*, la carpeta de instalación de la aplicación, en *Repositories*, se indica donde se almacenarán los repositorios, y en *Server Port*, el puerto que utilizará para conexiones seguras, por defecto aparece 443, pero si ya tenemos una instalación Apache (caso de tener instalado XAMPP por ejemplo), conviene cambiar este puerto para no provocar conflictos de puertos, así pues escribimos 4430, véase la Figura 4.21. Se pulsa *Next* y continúa la instalación.

Si todo va bien al final de la instalación selecciona la opción *Start VisualSVN Server*, y pulsa *Finish* para terminar. Se abre la ventana de administración (véase la Figura 4.22) desde ahí se podrán crear usuarios y repositorios.

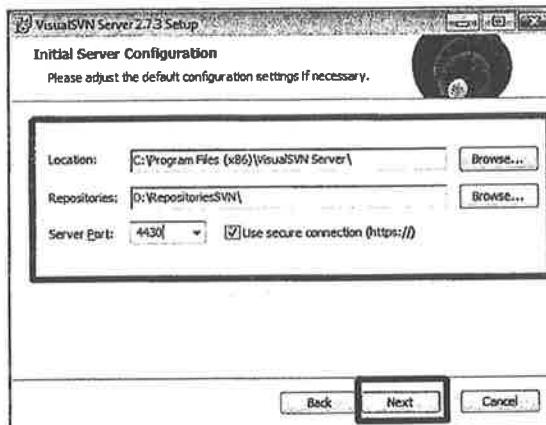


Figura 4.21. Configuración de VisualSVN Server.

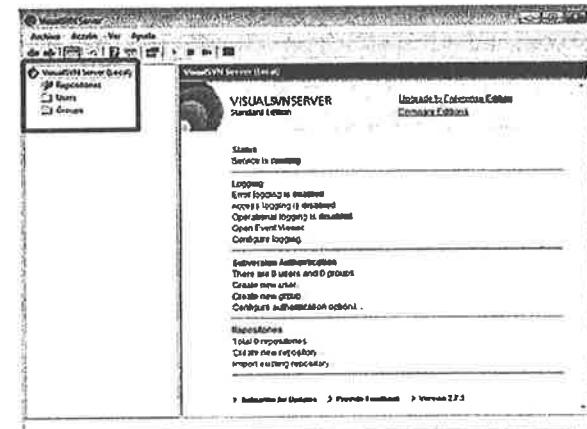


Figura 4.22. Consola de administración de VisualSVN.

Para crear un repositorio se pulsa el botón derecho sobre *Repositories* y se selecciona *Create new Repository...*, pedirá teclear el nombre del repositorio, se pulsa siguiente y en la siguiente ventana se indica si el repositorio se crea vacío o con la estructura *trunks-branches-tags*. En nuestro caso elegir la opción *trunks-branches-tags*.

A continuación se indica el permiso para los usuarios: que nadie tenga acceso, o todos los usuarios tienen acceso de lectura/escritura, o si se pulsa a *Custom* se puede configurar el acceso a unos cuantos. En principio se deja la opción por defecto, es decir, que todos tengan acceso.

Finalmente se pulsa *Create*, y en la ventana que se muestra aparece la URL que utilizaremos para conectarnos con el repositorio, a través de clientes como TortoiseSVN, o vía web desde el navegador. Para conectarnos a él utilizamos este formato:

https://NOMBRE_EQUIPO:4430/svn/Nombre_Repository. Salimos de la creación y en la consola se mostrará el repositorio creado, véase la Figura 4.23.

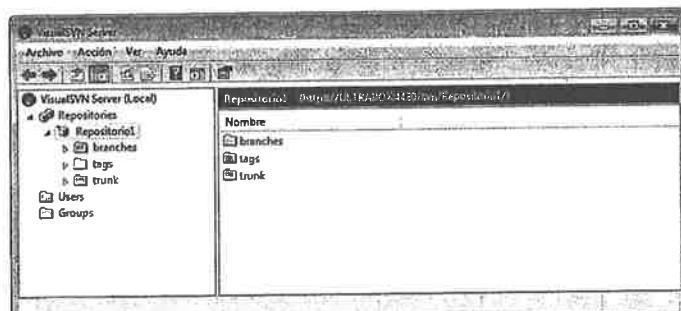


Figura 4.23. Consola de administración y repositorio creado.

El siguiente paso es crear un usuario para que este se conecte al repositorio. Para ello nos posicionamos en la carpeta *Users*, se pulsa el botón derecho del ratón y se elige *Create User*. Se escribe el nombre de usuario y su contraseña, por ejemplo se crea el usuario *usu1* y password *usu1*. Este usuario se utilizará a lo largo de la unidad.

Para probar si el servidor está funcionando nos conectamos a él desde el navegador, se escribe la URL (botón derecho sobre el repositorio, elegir *Copy URL to Clipboard*) en el navegador. Pedirá autenticación, se escribe el usuario y la contraseña, y finalmente se mostrará el repositorio creado, véase la Figura 4.24.

Igualmente para conectar este repositorio al navegador de repositorios del cliente *TortoiseSVN* pedirá el nombre de usuario, véase la Figura 4.25.

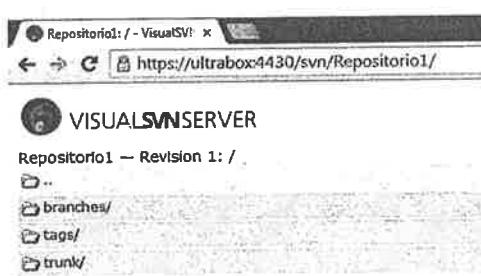


Figura 4.24. Conexión al repositorio desde el navegador.

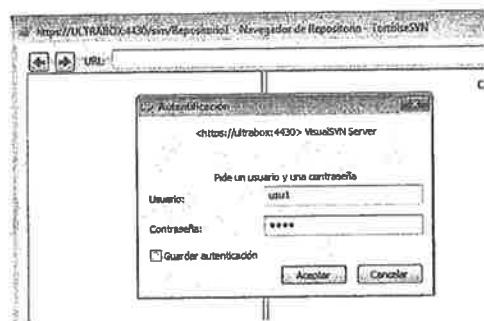


Figura 4.25. Conexión al repositorio desde el cliente TortoiseSVN.

4.2.4. Subversión en SQL Developer

En este apartado se estudiará subversión con la versión *sqldeveloper-4.0.2.15.21* que se encuentra en la carpeta de recursos del capítulo (se ha decidido estudiar esta versión porque añade más funcionalidades a subversión que la versión estudiada en el capítulo 2 -*sqldeveloper-3.2.10.09.57-*, sobre todo para la resolución de conflictos). La instalación se realiza como en el Capítulo 2.

Desde SQL Developer se pueden crear repositorios locales para almacenar proyectos y también se pueden crear conexiones con servidores de repositorios. Para ello, se accede al menú *Equipo/Subversión/* y se elige *Crear Conexión* para conectarnos con un servidor de repositorios como *VisualSVN Server*, o para crear un repositorio local dentro del equipo (En la antigua versión se accede desde el menú *Ver/Equipo/Navegador de versiones*).

Para crear una conexión con el Repositorio1 anterior, en el menú *Equipo/Subversión/* se elige *Crear Conexión*, a continuación se elige la conexión manual, y en la ventana que aparece (véase la Figura 4.26) teclearemos los datos del servidor, la URL y el usuario con su clave, al aceptar aparece una ventana indicando una excepción, se acepta.

La vista del Navegador de versiones se muestra en la Figura 4.27. Para abrir el navegador de versiones se accede al menú *Equipo/Versiónes*.

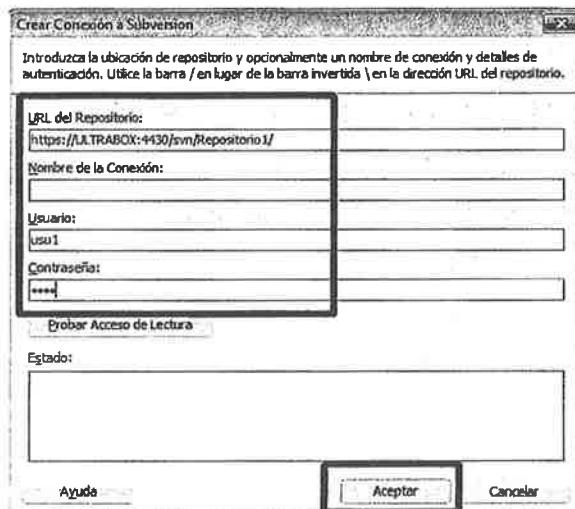


Figura 4.26. Conexión al repositorio desde Sql Developer.

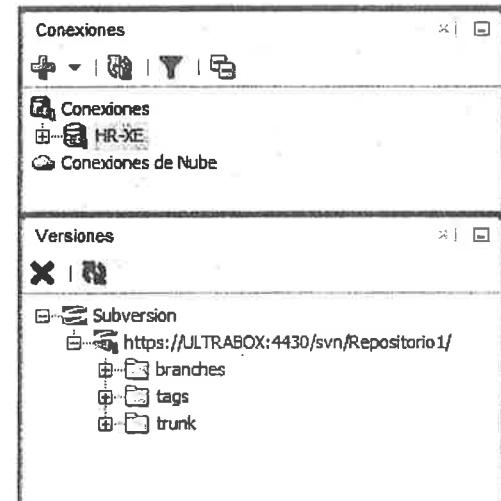


Figura 4.27. Navegador de versiones de Sql Developer.

ACTIVIDAD 4.6

Crea un repositorio local en tu disco duro con Sql Developer. Llámalo *RepositorioSqldeveloper*

AÑADIR UN PROYECTO AL REPOSITORIO

Para añadir un proyecto a un repositorio primero lo abrimos con Data Modeler desde el menú *Archivo/Data Modeler/Abrir*. Una vez abierto, con el explorador de Data Modeler (*Ver/Data Modeler/Explorador*) lo importamos al repositorio, para ello seleccionamos el menú *Equipo/Subversión/Importar Archivos* (si la opción no aparece activa asegúrate de mostrar el modelo lógico o relacional y posicionarte en uno de ellos). Se ejecuta el *Asistente de importación a Subversión*, se selecciona el repositorio y la carpeta dentro del repositorio donde se desea importar el proyecto (véase la Figura 4.28). Se pulsa siguiente y se selecciona la ubicación donde se encuentra el proyecto a importar, se añade un comentario para describir el momento de la subida (véase la Figura 4.29). Se pulsa siguiente, se dejan las opciones por defecto y se pulsa terminar.

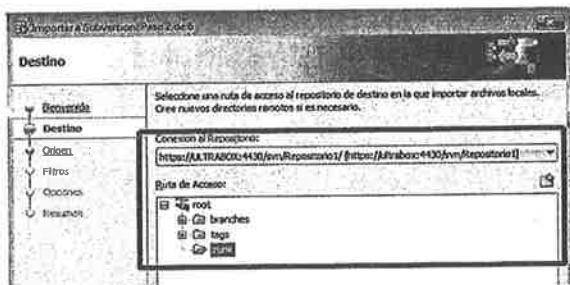


Figura 4.28. Importación a subversión, destino de la importación.

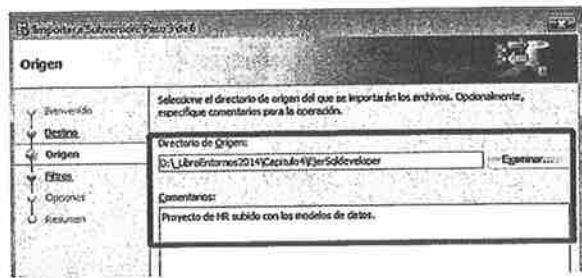


Figura 4.29. Importación a subversión, origen de la importación.

En la *Consola de SVN* se visualizarán los mensajes de importación, y el número de versión actual y confirmada, la número 3 en el ejemplo (véase la Figura 4.30). Observa también la carpeta que contiene el proyecto que se acaba de importar, se crearán dos nuevas carpetas una con el sufijo *.svn-import-backup*, y otra con *.svn-import-workarea* (véase la Figura 4.31).

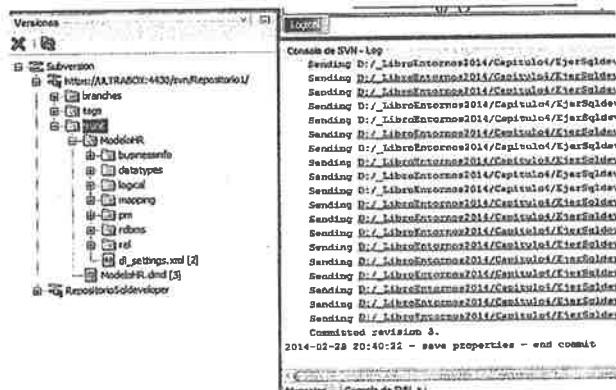


Figura 4.30. Importación a subversión, completada.

Para hacer *checkout* de una carpeta del repositorio a una carpeta local de trabajo, se selecciona la carpeta del repositorio, y se elige *Desproteger* en el menú contextual, o también desde el menú *Equipo/Subversión*, y en el cuadro de diálogo (véase la Figura 4.32) se seleccionará la carpeta destino y la versión a desproteger.

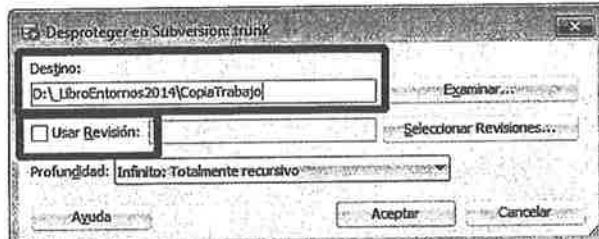


Figura 4.32. Hacer checkout del repositorio.

Desde el menú *Equipo/Data Modeler* (véase la Figura 4.33) se podrá acceder a todas las operaciones de control de versiones sobre los diseños de Data Modeler, como confirmar, actualizar, resolver conflictos, ver el historial de versiones, ver cambios pendientes o ver el árbol de versiones.

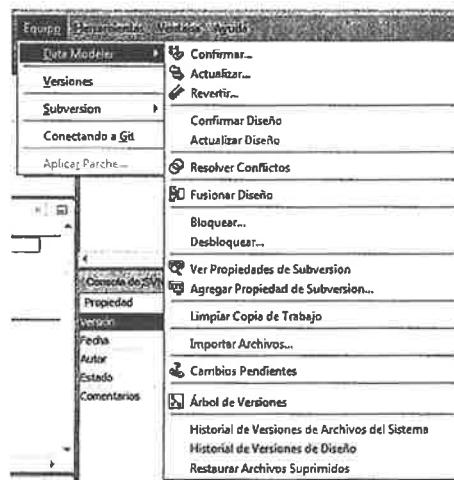


Figura 4.33. Menú de Subversión de Data Modeler.

ACTIVIDAD 4.7

A. Importar un proyecto al repositorio, realizar cambios y validar.

Abre con Data Modeler el modelo de datos *ProfeCentros* que se encuentra en la carpeta de recursos.

En el repositorio *Repository1*, crea una carpeta dentro de *trunk* con nombre *ProyectoCentros* e importa el proyecto a esa carpeta. Añade un comentario indicando que se ha subido el proyecto.

Abre el proyecto y realiza los siguientes cambios:

- En el modelo lógico crea la entidad PRUEBA1 con dos atributos COD numérico (5) campo identificativo, y NOMBRE varchar (20). Véase la Figura 4.34.
- En el modelo relacional crea la tabla TABLA1 con dos columnas NUM numérico (5) campo clave, y DENOMINACION varchar (20). Véase la Figura 4.34.

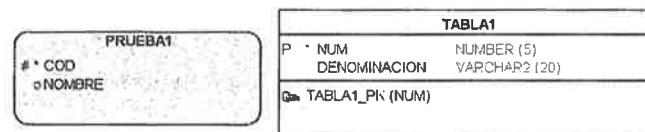


Figura 4.34. Entidad y tabla a añadir al proyecto.

Guarda los cambios del diseño, y visualiza los cambios pendientes de validar en el repositorio, desde el menú *Equipo/Data Modeler/Cambios pendientes*. Véase la Figura 4.35. Utiliza el botón *Refrescar* si no se ven los cambios.

Observa los cambios salientes pendientes de confirmar.

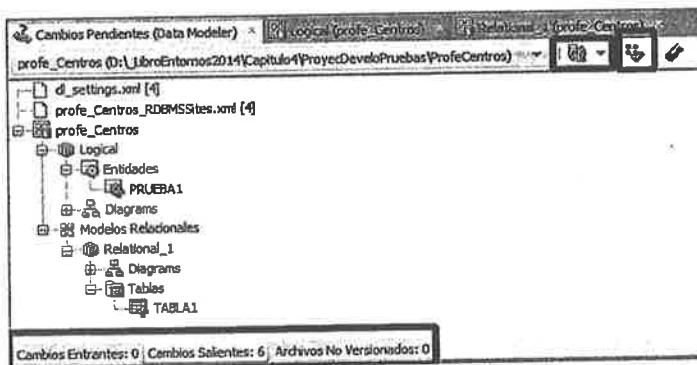


Figura 4.35. Cambios pendientes de validar.

NOTA: En *Cambios Entrantes* se mostrarán los cambios que han hecho otros usuarios, o que se han hecho desde otras copias de trabajo sobre el repositorio, estos cambios se deberán actualizar en nuestra copia de trabajo antes de validar los cambios salientes.

Los *Cambios Salientes* son los realizados en la actual copia de trabajo, y los *Archivos No Versionados*, son archivos que se han añadido al proyecto pero que aún no se han confirmado en el repositorio.

Confirma primero los cambios del modelo lógico (marca *Logical* y pulsa el botón *Confirmar* desde la ventana de cambios pendientes), añadiendo un mensaje a la validación, indicando que se confirman cambios en el modelo lógico. Visualiza el historial de versiones (*Equipo/Data Modeler/Historial de versiones de diseño*).

A continuación confirma los cambios del modelo relacional, añadiendo un mensaje a la validación, indicando que se confirman cambios en el modelo relacional. En la Figura 4.36 se muestran las dos validaciones.

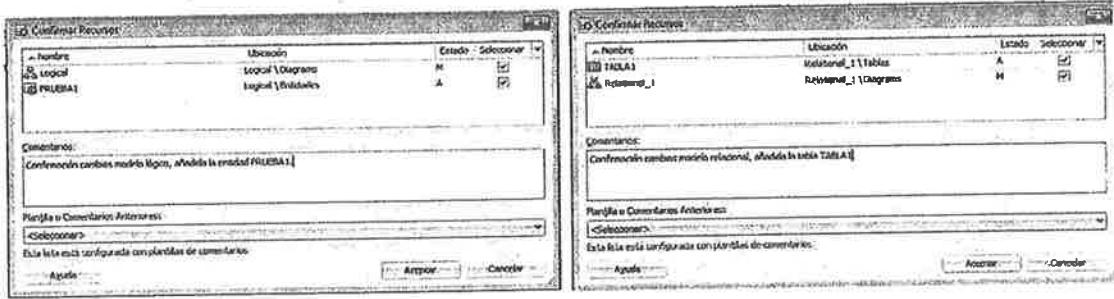


Figura 4.36. Confirmación de cambios modelo lógico y relacional, por separado.

Finalmente, confirma todos los cambios realizados en el proyecto y observa todas las validaciones realizadas en el historial de versiones, que se debe mostrar como en la Figura 4.37.

Historial de Versiones			
(1) profe_Centros (D:\LibroEntornos2014\Capítulo4\Proyectos\ProfeCentros) D:\LibroEntornos2014\Capítulo4\Proyectos\ProfeCentros\profe_Centros.dmd			
Revisión	Fecha	Autor	Comentario
Almacenamiento			
9	Thu Mar 06 15:02:43 ... user1		Validación final de cambios del proyecto.
	[Thu Mar 06 14:54:27 ...] 2011		Confirmación cambios modelo relacional, añadido la tabla TABLA1.
7	Thu Mar 06 14:51:20 ... user1		Confirmación cambios modelo lógico, añadida la entidad PRUEBA1.
6	Thu Mar 06 11:58:03 ... user1		
5	Thu Mar 06 11:57:31 ... user1		Subida del directorio ProfeCentros al repositorio.

Acción	Objetos cambiados	Ubicación	Copiar R...	Copiar Ruta de Acceso	Confirmación cambios modelo relacional, añadido la tabla TABLA1.
H	M -> Relacional_1	Modelos Relacionales...			
A	TABLA1	Modelos Relacionales...			

Figura 4.37. Historial de versiones y cambios confirmados.

B. Crear una copia de trabajo, realizar cambios y validar

Cierra el proyecto y crea ahora una carpeta de trabajo en el equipo, hacer *checkout* (elemento *Desproteger* del menú *Equipo/Subversión*) de la carpeta del repositorio *ProyectoCentros* a esa carpeta de trabajo. Hay que posicionarse en la carpeta a desproteger.

A continuación abre el proyecto de esa carpeta de trabajo y realiza cambios, por ejemplo, realiza una copia (copiar y pegar) de la entidad y la tabla creada en el apartado anterior.

Observa los cambios pendientes que se han generado (*Equipo/Data Modeler/Cambios pendientes*), pulsa el botón *Refrescar* si no se muestran. Valida esos cambios y añade un mensaje de validación. Cierra este proyecto.

C. Abrir el proyecto importado al inicio y actualizar

Carga el proyecto importado en el apartado A y abre el menú *Equipo/Data Modeler/Cambios pendientes*. Selecciona la pestaña de *Cambios Entrantes*, en esta pestaña se deben mostrar los cambios realizados en el apartado B, y que no aparecen en esta copia de trabajo. Véase la Figura 4.38.

Antes de realizar cambios en la copia actual es necesario actualizarla con la última versión del repositorio, es decir, hay que incorporar estos cambios entrantes, ya que si se realizan nuevos cambios se producirán conflictos. Así pues se pulsa al botón *Actualizar Copia de Trabajo*, de esta manera nuestra copia se sincroniza con la versión actual del repositorio, y se podrán realizar cambios sin peligros de conflictos.

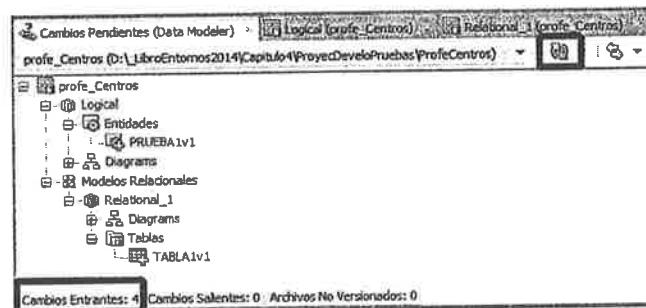


Figura 4.38. Cambios entrantes, pendientes de actualizar.

4.2.5. Subversión en Eclipse

Para utilizar Eclipse como cliente de *VisualSVN* es necesario instalar el plugin *Subversive SVN*. Para ello desde *Eclipse Marketplace* (menú *Help*) se busca SVN y de los resultados de búsqueda se selecciona *SVN Team Provider* (véase la Figura 4.39). Se pulsa a instalar, se seleccionan y confirman los elementos a instalar, se aceptan los términos de la licencia y comienza la instalación. El *Subversive* es un plugin que se instala en Eclipse y permite interactuar directamente nuestros proyectos de eclipse con los repositorios locales y remotos que trabajan con *Subversion*.

Para conectar los proyectos a un repositorio es necesario instalar los *conectores de Subversive*. La ventana para seleccionar los conectores se visualizará inmediatamente al crear un proyecto o al cargar un proyecto. Si esta ventana (Figura 4.40) no aparece, se debe comprobar que el conector no está instalado desde el menú *Window/Preferences/Team/SVN/SVN Connector*, pestaña *SVN Connector*. En ese caso es necesario instalar los conectores.

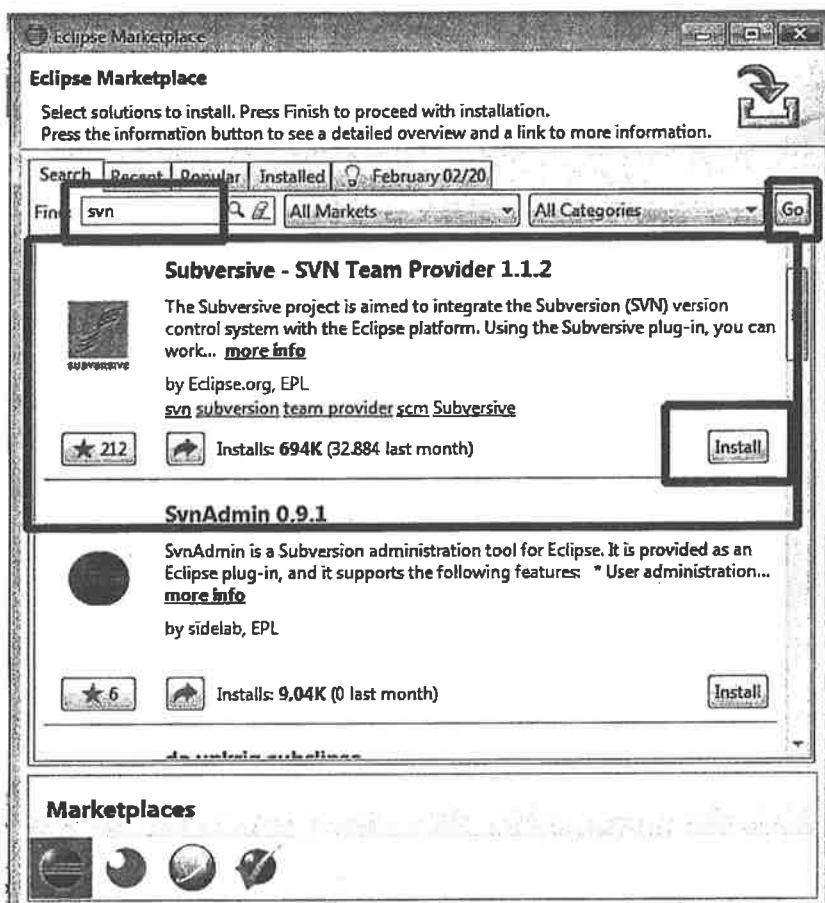


Figura 4.39. Plugin Subversive SVN para Eclipse.

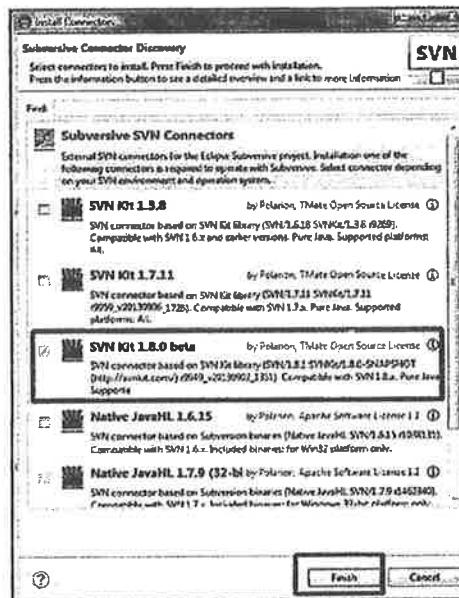


Figura 4.40. Conectores Subversive.

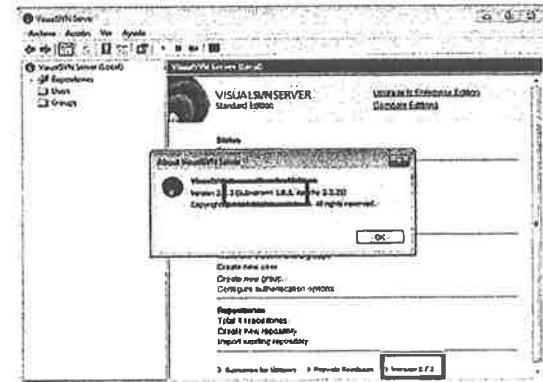


Figura 4.41. Versión Subversive instalada con VisualSVN.

Para saber qué versión de conector instalar es necesario ver la versión de *Subversive* que se ha instalado. Desde la administración de consola del servidor *VisualSVN*, haz clic en el enlace de la versión de *VisualSVN*, y se visualizará en un cuadro de diálogo la versión de subversión y apache que se instala, véase la Figura 4.41.

NOTA: si no se instala el conector, hay que entrar en el menú *Help/Install new Software/* y en *Work with* escribir la URL:

<http://community.polarion.com/projects/subversive/download/eclipse/3.0/kepler-site/>

O también instalar el plugin que se encuentra en la carpeta de recursos desde el menú *Help/Install New Software*. El plugin se llama *Subversive-1.1.0.I20130527-1700.zip*

Para obtener información de subversive y de los conectores consulta las siguientes URLs: <http://www.eclipse.org/subversive/latest-releases.php>

Polarion.com desarrolla los conectores *Subversive SVN Connector*. Desde el sitio de Polarion se podrá obtener toda la información y ver las instrucciones de instalación.

<http://www.polarion.com/products/svn/subversive/download.php>

AÑADIR UN PROYECTO AL REPOSITORIO

Una vez que ya se tiene el plugin y el conector instalado lo siguiente es añadir un proyecto al repositorio. Pero antes de subir un proyecto hay que crear un repositorio nuevo llamado *Repositorio2*, desde el *VisualSVN Server*, con la estructura *trunk-tags-branches*. Para trabajar con un repositorio limpio.

Se crea un proyecto Java, o se importa un proyecto de los que hay en la carpeta de recursos (por ejemplo *AleatorioEmple*). Nos posicionamos sobre el proyecto y hacemos clic con el botón derecho del ratón, se selecciona la opción **Team/Share Project**.

En el primer cuadro de diálogo se elige el tipo de repositorio al que nos vamos a conectar, se elige *SVN*, se pulsa *Next* y pide la información del repositorio (véase la Figura 4.42). En la pestaña *Advanced* se desmarca la casilla para poder cargar el proyecto dentro de *trunk*.

En la pestaña **General** se indica la *URL* del repositorio, por ejemplo: <https://ULTRABOX:4430/svn/Repositorio2/trunk>. Pulsa el botón *Browse* para visualizar el repositorio, y elegir la carpeta *Trunk*, pedirá la conexión de un usuario. Nos conectamos con *usu1*. En *Authentication* también añadiremos los datos del usuario (*usu1/usu1*). Se pulsa el botón *Next*. Y seguidamente a *Finish*. A continuación aparece una ventana donde se puede añadir un comentario para la subida del proyecto, y donde se ven los archivos que se van a subir. Si alguno de ellos no se desea subir se puede desmarcar. Se puede elegir también una plantilla para los comentarios. Se añade un comentario, se pulsa *OK*, y el proyecto ya está en el repositorio.

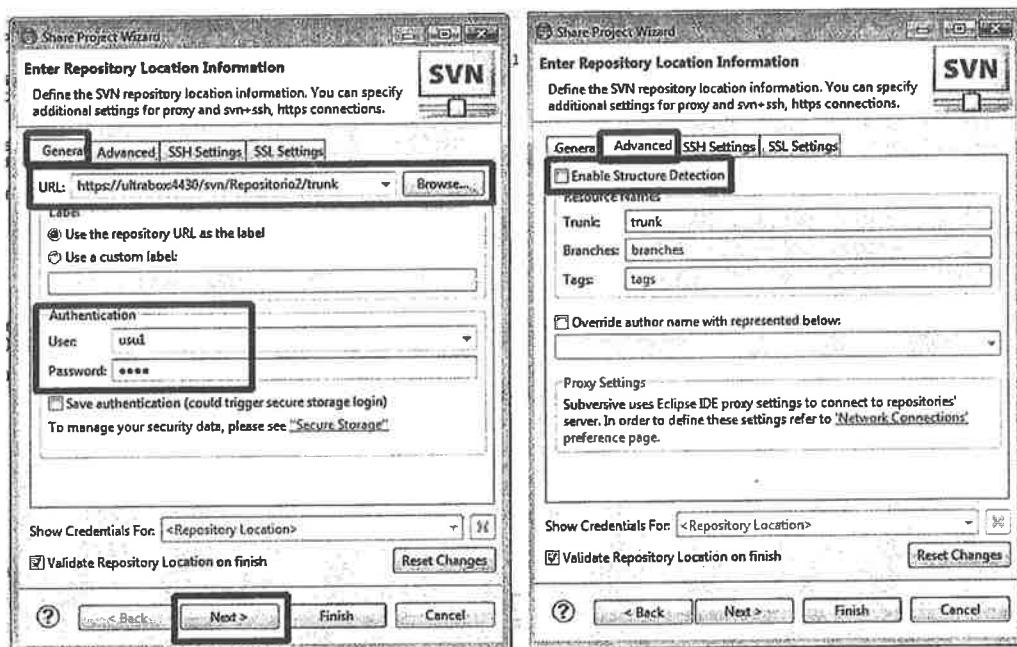


Figura 4.42. Configuración de la conexión al repositorio en Eclipse.

NOTA: si al intentar conectar con un repositorio no se visualiza su estructura revisa que los servicios del *VisualSVN Server* estén iniciados. Desde *Panel de Control / Herramientas administrativas/Servicios*.

Observa el proyecto subido al repositorio. Observa que cada nodo del proyecto aparece con un ícono indicando que está en un repositorio, aparece también el número de revisión al lado, y en la carpeta del proyecto se muestra la URL donde se encuentra. Véase la Figura 4.43.

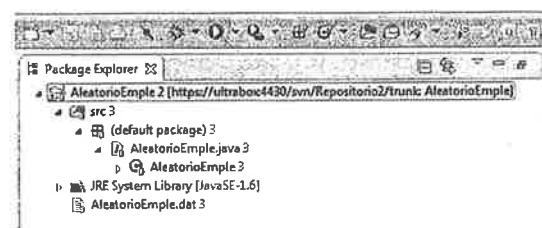


Figura 4.43. Vista del proyecto subido al repositorio.

Si ahora se cambia de perspectiva a la perspectiva *SVN Repository Exploring*. Podemos observar el repositorio, la estructura del repositorio, el proyecto almacenado, el navegador del repositorio, las revisiones, el usuario que hace los cambios, etc. Véase la Figura 4.44.

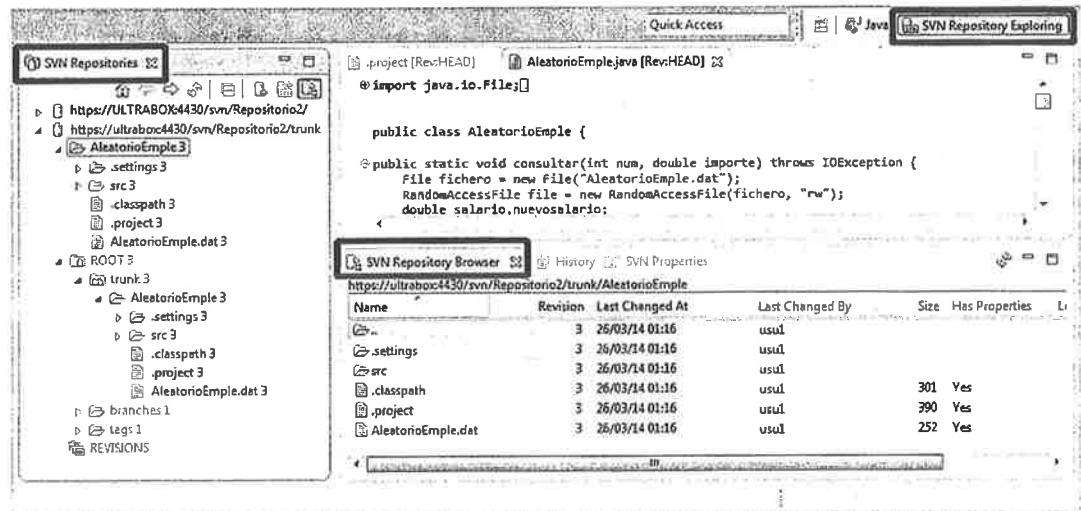


Figura 4.44. Perspectiva SVN Repository Exploring.

OPERACIONES CON SVN ECLIPSE

Las operaciones que se pueden realizar con *SVN Eclipse* son las mismas que las realizadas desde cualquier otro cliente. Nos posicionamos en el proyecto, o en un nodo del proyecto y desde el menú contextual se verán activas las opciones de SVN (véase más abajo la Figura 4.45).

Para obtener una copia de trabajo haremos *Check Out*, que almacenará la copia en el workspace actual, si se elige *Check Out As*, almacenará la copia en la carpeta que se indique. Para ver el historial de revisiones elegiremos *Show History*.

En la perspectiva Java se realizan los cambios en el proyecto. Si se realiza algún cambio aparecerá un símbolo > en los elementos cambiados. Por ejemplo añade una línea de comentario a la clase, guarda el proyecto y automáticamente los archivos implicados aparecerán marcados con un símbolo mayor >. Si ahora se crea una clase, aparecerá una interrogación junto a los nuevos archivos, eso indica que aún no han sido añadidos al repositorio.

Si ahora se visualiza el menú contextual (botón derecho *Team*) sobre el elemento se verán todas las operaciones de SVN (véase la Figura 4.46) como son hacer *Commit* o *Update*, crear etiquetas y ramas, hacer fusión, visualizar el historial, o bloquear y desbloquear el elemento.

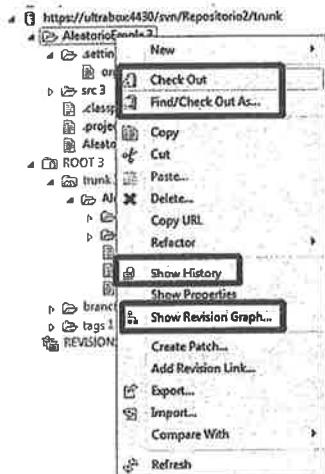


Figura 4.45. Menú contextual del proyecto, en la vista SVN Repository Exploring.

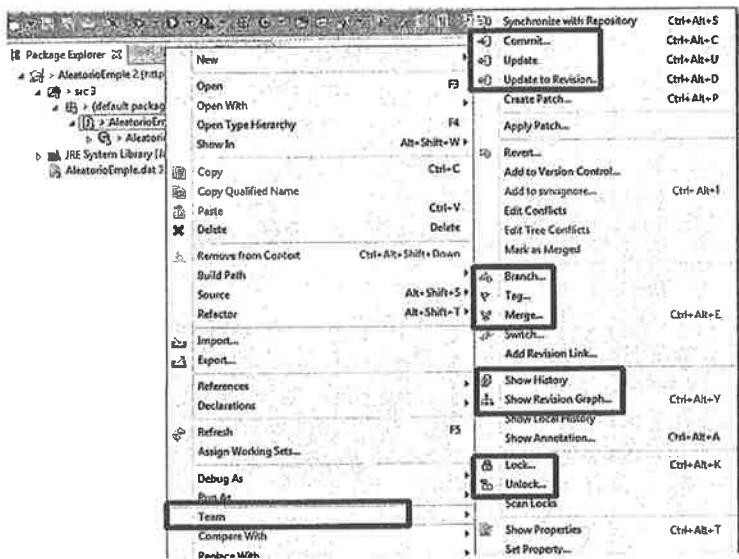


Figura 4.46. Menú contextual Team SVN del proyecto en la vista Java.

Cuando se realizan cambios se puede acceder a la pestaña **Synchronize** (si no se muestra, hay que ir al menú *Window>Show View/Others*, se elige *Team/Synchronize*), en la barra de botones de esta pestaña se podrán ver los cambios entrantes y los salientes. También se puede ver desde la perspectiva **Team Synchronizing** (véase la Figura 4.47). Esta perspectiva nos muestra los cambios locales que van a subir el servidor, y viceversa, los cambios del servidor que nosotros deberíamos agregar localmente. Mediante esta perspectiva mantenemos la comunicación entre nuestro proyecto local de trabajo y el proyecto que está en el servidor.

Desde la barra de botones de esta perspectiva podremos hacer update, validar, o ver los conflictos. Cuando hay **cambios salientes** aparece una flecha negra hacia afuera, si aparece el signo + es que el archivo es nuevo y no está versionado (véase la Figura 4.47). Lo siguiente que habría que hacer es validar, se pulsa al botón que aparece en la barra de herramientas de validar los cambios salientes.

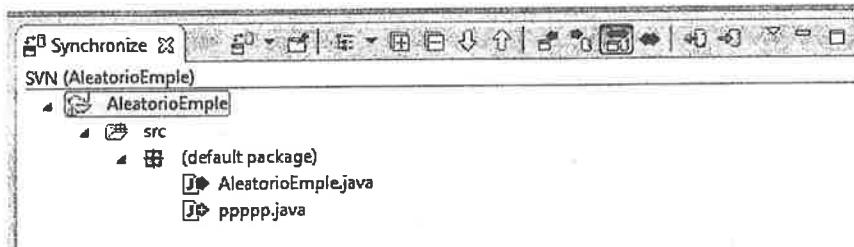


Figura 4.47. Perspectiva Team Synchronizing, cambios salientes.

Cuando hay **cambios entrantes** aparece una flecha hacia adentro en azul, si aparece el signo + es que el archivo es nuevo (véase la Figura 4.48). Lo siguiente que habría que hacer es actualizar esos cambios entrantes, se pulsa al botón que aparece en la barra de herramientas para actualizar, y luego validar los cambios salientes.

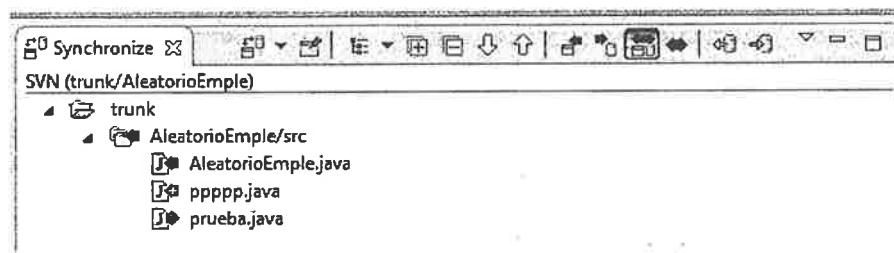


Figura 4.48. Perspectiva Team Synchronizing, cambios entrantes.

SOLUCIÓN DE CONFLICTOS

Un conflicto se va a producir cuando *dos usuarios modifiquen el mismo archivo del repositorio, y las mismas líneas del archivo*. Uno de ellos confirma los cambios, y a continuación el otro usuario los confirma también, el servidor va a detectar un conflicto pues se quieren validar cambios en una copia que no ha sido actualizada con la versión del repositorio, que fue validada por el primer usuario.

Para provocar un conflicto crea dos carpetas (por ejemplo *SVN_trabajo1* y *SVN_trabajo2*) y realiza dos *CheckOut As* desde la perspectiva *SVN Repository Exploring* del mismo proyecto, cambia de nombre al proyecto, llámalo por ejemplo *AleatorioEmple1* y *AleatorioEmple2*, y almacena las copias de trabajo, una en cada carpeta.

Los proyectos se abrirán automáticamente en el Eclipse, cada uno de ellos está en una carpeta diferente. Añadimos los siguientes cambios en la clase *AleatorioEmple.java* y los guardamos en local:

En *AleatorioEmple1* añadimos la declaración `int dato=0;` debajo de `char aux;`

En *AleatorioEmple2* añadimos la declaración `String cad="HOLA";` debajo de `char aux;`

Se debe validar ahora los cambios del proyecto *AleatorioEmple1*, hacer commit, añadir el comentario “*Se ha añadido la declaración int dato=0*”. Si ahora se validan los cambios del proyecto *AleatorioEmple2*, se visualizará una ventana indicando que hay un conflicto y que no se puede hacer COMMIT.

A continuación abrimos la vista o también la pestaña *Synchronize*, nos posicionamos sobre el proyecto *AleatorioEmple2*, pulsamos al botón derecho, se elige *Team/Synchronize with Repository*, observa que los conflictos aparecen marcados con una doble flecha o rombo en rojo, y en la parte inferior se indican los cambios entrantes, los salientes y los conflictos (véase la Figura 4.49).

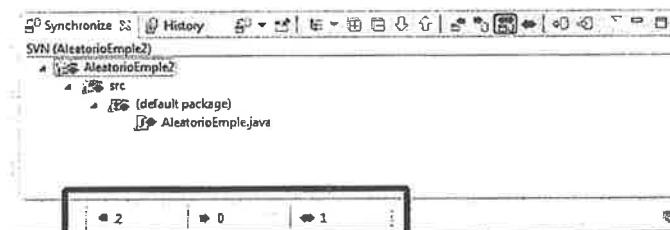


Figura 4.49. Perspectiva Team Synchronizing, cambios entrantes.

Si hacemos doble clic en el conflicto, es decir en la clase *AleatorioEmple.java*, se muestra una ventana en la que se comparan las dos versiones (véase la Figura 4.50), en la parte izquierda se muestra el archivo de la copia local y en la derecha el archivo del repositorio, aparecen marcadas con borde rojo las líneas donde ha aparecido el conflicto. En este caso indicamos que se copie la línea del repositorio (derecha) a la copia local pulsando al cuadrado rojo. Debe quedar como se muestra en la Figura 4.51.

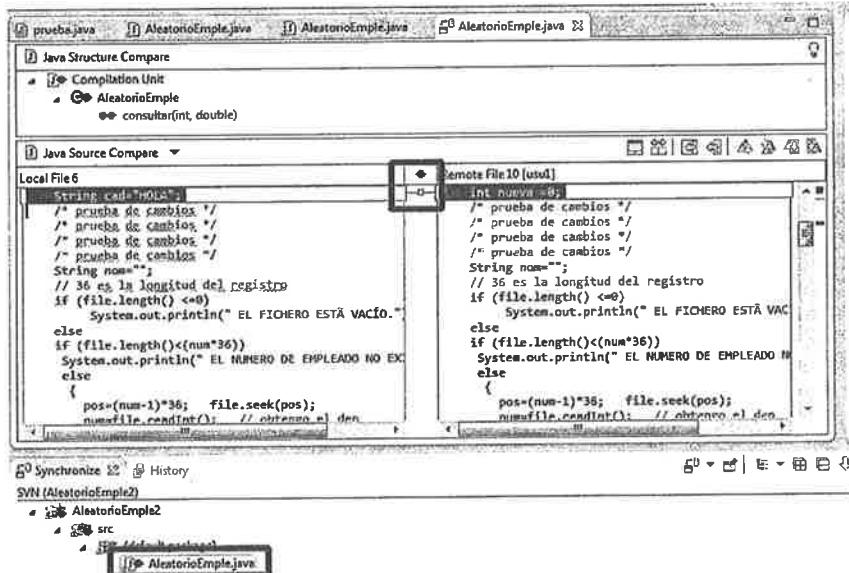


Figura 4.50. Vista del conflicto entre la versión del repositorio (dcha.) y la copia de trabajo (izq.).

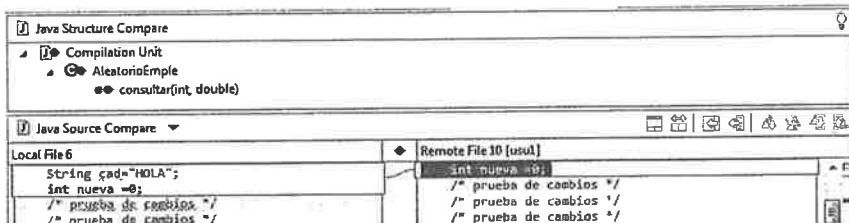


Figura 4.51. Cambios copiados del repositorio (dcha.) a la copia de trabajo (izq.).

Para resolver finalmente el conflicto, una vez que se han recogido los cambios en la copia local, pulsamos el botón derecho sobre el archivo en conflicto, elegimos *Team/Mark as Merged*, y deberán desaparecer los rombos rojos, y aparecerá una flecha negra, de cambio saliente. Al hacer esto ya podremos hacer commit.

¡IMPORTANTE!

Para resolver un conflicto, una vez recogidos los cambios en la copia local es necesario hacer *Mark as Merged*. Hasta que no se haga esto Subversive sigue creyendo que hay conflicto con lo que no va a dejar validar.

Cuando tenemos cambios en la copia local que no se recogen en el repositorio los veremos marcados con una línea negra, y cuando son cambios que están en el repositorio y no los tenemos recogidos en local aparecerán marcados con una línea azul, pero estos no aparecen en conflicto porque son cambios en líneas diferentes. Si se cliquea en el cuadradito que aparece en el medio (en rojo o azul dependiendo del conflicto) lo pasa a nuestro archivo local, y listo, el archivo ya estaría sincronizado, ya solo falta hacer *Mark as Merged*. Y lo siguiente es hacer commit.

ACTIVIDAD 4.8

Antes de realizar nuevos cambios, comprueba que los dos proyectos tienen la misma versión y están actualizados con el repositorio. Hay que actualizar *AleatorioEmple1* que no contiene los cambios añadidos al repositorio del proyecto *AleatorioEmple2*, para actualizar: botón derecho sobre el proyecto elegir *Team/Update*. Una vez que los dos proyectos estén actualizados y marcan la misma revisión, realiza los siguientes cambios en la misma clase *AleatorioEmple.java*:

En *AleatorioEmple1* añade el método prueba, en cualquier sitio y guarda los cambios en local:

```
///////////////////////////////
public static void prueba()
{
    // Se ha añadido un método
}
/////////////////////////////
```

En *AleatorioEmple2* añade la declaración `int prueba=0;` debajo de `public static void consultar(...)` y guarda los cambios en local.

Valida los cambios de *AleatorioEmple1*. Con el proyecto *AleatorioEmple2* visualiza la pestaña *Synchronize* y observa los cambios. Aparecen conflictos, si se hace doble clic sobre el conflicto se muestra la versión del repositorio y la local.

Observa que ahora los conflictos no aparecen marcados en rojo porque no son las mismas líneas la modificadas, ahora se muestran en color negro, los cambios que se han realizado en *AleatorioEmple2* (cambios salientes) y en azul los cambios realizado en *AleatorioEmple1* y que fueron validados (cambios entrantes). Véanse las Figuras 4.52 y 4.53.

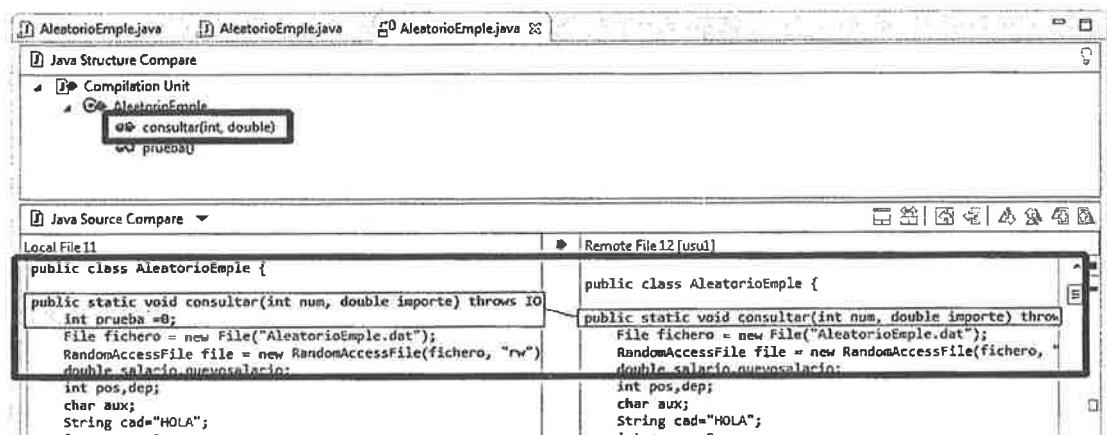


Figura 4.52. Cambios salientes de *AleatorioEmple2* (izq.) al repositorio (dcha.).

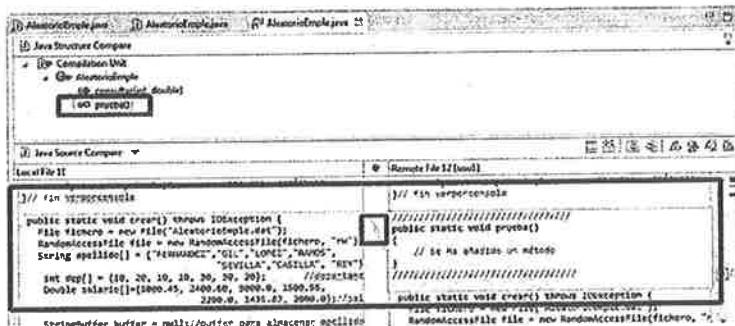


Figura 4.53. Cambios entrantes del repositorio (dcha.) a la copia local *AleatorioEmpleo2* (izq.).

Pulsa al cuadrado azul para pasar los cambios a la copia local, guarda el proyecto, pulsa el update y a continuación valida los cambios al repositorio , haz commit.

Finalmente para que los dos proyectos queden en la misma versión, haz un update en cada uno de ellos, desde la carpeta del proyecto, botón derecho del ratón **Team/Update**.

RECUERDA:

Se recomienda antes de trabajar con un proyecto hacer un *sincronize*, y un *update*, para asegurarnos de estar trabajando con la última versión del repositorio.

CREAR RAMAS

Para crear una rama se hace desde la perspectiva **SVN Repository Exploring**. Hay que situarse en el raíz del repositorio, por ejemplo <https://ULTRABOX:4430/svn/Repositorio2/>, si no hay creada una conexión a la raíz del repositorio, se crea desde **New/Repository Location**. O botón derecho del ratón y se selecciona **New/Repository Location**. Todo desde la vista **SVN Repository Exploring**.

Para subir un proyecto a una rama, nos posicionamos en el elemento **Branches** de la raíz del repositorio, y creamos una carpeta dentro de la rama (botón derecho **New/Folder**), véase la Figura 4.54. A continuación nos posicionamos en **ROOT** (seguimos situados en la raíz <https://ULTRABOX:4430/svn/Repositorio2/>) desplegamos y buscamos el proyecto a subir a la rama. Nos posicionamos sobre él pulsamos al botón derecho del ratón y elegimos **New/Branch**. Y en la ventana que se muestra se elige la carpeta que creamos anteriormente en la rama (véase Figura 4.55). Toda la operación se hace desde la vista **SVN Repository Exploring**, y desde la conexión con la raíz del repositorio.

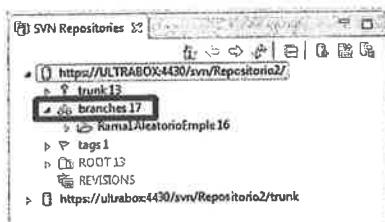


Figura 4.54. Creación de una carpeta dentro de la rama del repositorio.

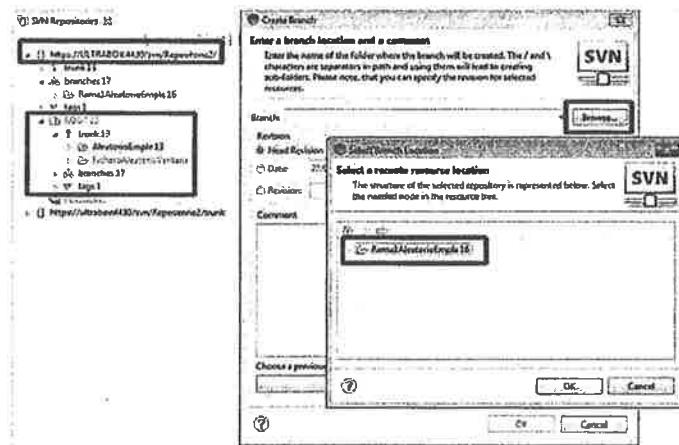


Figura 4.55. Copiar el proyecto de trunk a la carpeta de la rama.

A partir de ahora los *CheckOut* se harán desde las ramas. Se cargarán en el disco duro, se modificarán las copias locales de las ramas, y se validarán con la correspondiente rama del repositorio.

Para añadir los cambios de las ramas a la versión del trunk, se hace un *Merge/Reintegrate* (*Team/Merge* pestaña *Reintegrate*) desde la copia de trabajo del trunk con la copia del repositorio de la rama. Nos posicionamos en la copia de trabajo del proyecto (se supone que está sincronizado y actualizado con la versión del repositorio). Pulsamos el botón derecho del ratón y se elige *Team/Merge* pestaña *Reintegrate* (véase Figura 4.56). A continuación se muestra la vista *Synchronize* con los cambios detectados, se aceptan los cambios y luego se hace commit de la copia local del proyecto de trunk con la del repositorio. Finalmente, se hace update para tener sincronizadas la versión de la copia de trabajo y la versión del trunk del repositorio.

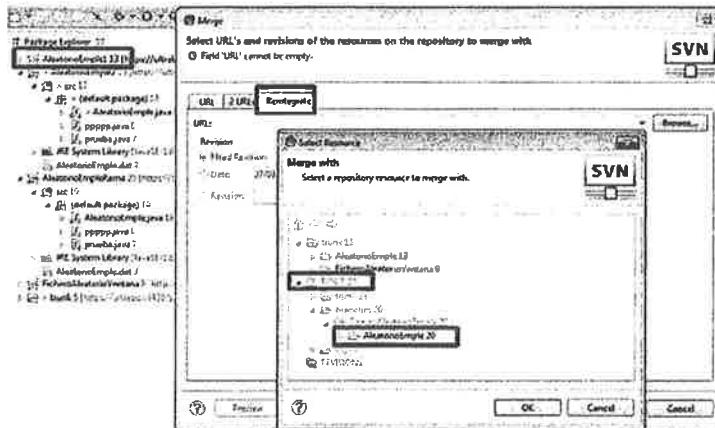


Figura 4.56. Merge/Reintegrate copia de trunk, reintegrarla con la copia de la rama.

ACTIVIDAD 4.9

Busca en la carpeta de recursos de la unidad el proyecto *FicheroAleatorioVentana*. Cópialo en el workspace de trabajo y ábrelo.

Sube el proyecto al repositorio, colócalo dentro de *trunk* y añade un comentario de subida.

Crea una rama llamada *RamaFicheroAleatorioVentana*. Añade a la rama una copia del proyecto.

Haz un *Chek Out As* de esa rama en una carpeta de trabajo, puede ser la misma del workspace, ponle un nombre diferente al proyecto para distinguirlo.

Realiza cambios en la copia de la rama, por ejemplo, añade un método de prueba y añade alguna línea comentada, etc. Valida esos cambios.

Guarda los cambios que tiene la rama en la versión del *trunk*, recuerda que hay que hacerlo desde una copia de trabajo del *trunk*, esta copia debe de estar sincronizada con su copia del *trunk*, es decir, realiza un *Merge/Reintegrate*.

Valida los cambios y sincroniza ambas copias de trabajo, para que todas ellas se queden en la misma versión.

CONFLICTOS EN LA REINTEGRACIÓN DE RAMAS

Para integrar los cambios realizados en las copias de trabajo de las ramas, a las copias de trabajo del *trunk*, el número de revisión de las ramas debe de ser mayor que la de *trunk*, pues se supone que los cambios se realizan en la rama.

Si se intenta integrar cambios que modifican las mismas líneas en los dos proyectos, se van a producir conflictos. En el fichero local aparecerán unas líneas como estas:

```
<<<<< .working
public static void prueba2_metodo_rama()
{
=====
public static void prueba2Cambio_metodo_rama()
{ int a=0;
>>>>> .merge-right.r24
}
```

En estas líneas se indica el conflicto encontrado, lo que hay en la copia de trabajo (*.working*) y lo que hay en la revisión 24 en esas mismas líneas.

Además se generarán varios archivos: uno de texto con la extensión *.java.working* (por ejemplo *AleatorioEmple.java.working*) que contiene una copia de trabajo de la clase en conflicto. Y otros dos con las dos versiones estables a las que poder restaurar para solucionar el conflicto, la versión de la izquierda que llevará el nombre *.merge-left.r*, es el último *Update* exitoso que se hizo en el *trunk*, por ejemplo *AleatorioEmple.java.merge-left.r19*, y la versión de la derecha que llevará el nombre *.merge-right.r*, que es la versión del último *Update* exitoso de la rama, por ejemplo *AleatorioEmple.java.merge-right.r24* (el número indica la versión).

Para resolver este conflicto podemos proceder de dos maneras:

- Editar el conflicto (en la ventana *Synchronizing*, botón derecho del ratón sobre el archivo en conflicto, elegir *Edit Conflicts*), se muestra la ventana con los archivos en conflicto, y marcadas en rojo las líneas conflictivas (véase la Figura 4.57). Se pulsa al cuadrito rojo, se hace *Update* pulsa el botón , se marca el fichero como “*mergeado*” *Mark as Merged* (botón derecho sobre el fichero y elegir *Team/Mark as Merged*), observa que desaparecen los ficheros que se crearon en el conflicto. Finalmente se pulsa al botón de commit  y conflicto resuelto.

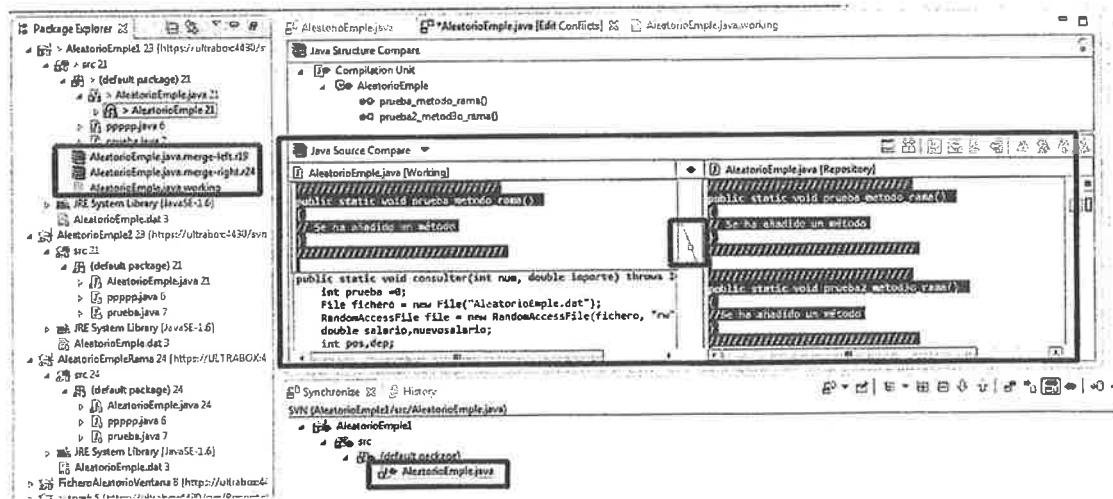


Figura 4.57. Conflicto después de hacer un Merge/Reintegrate.

- Otra posible solución es quedarse con una de las versiones en conflicto. Para ello se borra el fichero Java que da el conflicto. A la versión elegida se la quita la extensión `.merge-right.r` o `.merge-left.r`, (botón derecho sobre el fichero, se elige **Refactor/Rename**, y se cambia el nombre). Y ya solo queda marcarlo como “mergeado” **Mark as Merged** (botón derecho sobre el fichero y elegir **Team/Mark as Merged**), y hacer commit.

ACTIVIDAD 4.10

Realiza cambios en el proyecto subido en la Actividad 4.9. En la copia de trabajo de la rama, cambia el nombre al método creado, añade un parámetro, y cambia una de las líneas de comentario creadas. Valida los cambios.

Reintegra los cambios de la rama con la copia de trabajo del *trunk*.

Resuelve el conflicto surgido.

4.3. DOCUMENTACIÓN

Por documentación se entiende el texto escrito que acompaña a los proyectos de cualquier tipo, en nuestro caso de software. La documentación es un requisito importante en un proyecto comercial, el cliente siempre va a solicitar que se documenten las distintas fases del proyecto. Podemos distinguir los siguientes tipos de documentación:

- Documentación de las especificaciones:** este documento tiene como objeto asegurar que tanto el desarrollador como el cliente tienen la misma idea sobre las funcionalidades del sistema. Es muy importante que esto quede claro ya que si no el desarrollo software no será aceptable. En ingeniería del software existe una normativa referente a este tipo de documento la *norma IEEE 830* que recoge recomendaciones para la documentación de requerimientos de software, se indica que las especificaciones deben describir la siguiente documentación:

- *Introducción.* En ella se definen los fines y los objetivos del software.
- *Descripción de la información.* Se realiza una descripción detallada del problema, incluyendo el hardware y el software necesario.
- *Descripción funcional.* Descripción de cada función requerida en el sistema, incluyendo diagramas.
- *Descripción del comportamiento.* Comportamiento del software ante sucesos externos y controles internos.
- *Criterios de validación.* Documentación sobre límites de rendimiento, clases de pruebas, respuesta esperada del software, consideraciones especiales.
- **Documentación del diseño:** en la fase de diseño se decide la estructura de datos a utilizar, la forma en que se van a implementar las distintas estructuras, el contenido de las clases, sus métodos y sus atributos, los objetos a utilizar. También se definen las funciones, sus datos de entrada y salida, qué tarea realizan, etc.
- **Documentación del código fuente:** durante la fase de implementación, cuando se está programando, es necesario comentar convenientemente cada una de las partes que tiene el programa. Estos comentarios se incluyen en el código fuente con el objeto de clarificar y explicar cada elemento del programa, se deben comentar las clases, los métodos, las variables y en definitiva todo elemento que se considere importante.
- **Documentación de usuario final:** que es la documentación que se entrega al usuario tanto especializado como no especializado. En ella se describirá cómo utilizar las aplicaciones del proyecto. Este tipo de documentación puede ser redactado por cualquier persona, no necesariamente tiene que ser alguien involucrado en el proyecto.

DOCUMENTACIÓN DEL CÓDIGO FUENTE

Documentar el código de los programas es necesario para explicar claramente lo que hace el programa, de esta manera todo el equipo de desarrollo sabrá lo que se está haciendo y por qué. En un programa bien documentado es mucho más fácil reparar errores y añadirle nuevas funcionalidades para adaptarlo a nuevos escenarios que si carece de documentación. Hay dos reglas que no se deben olvidar nunca:

- Todos los programas tienen errores y descubrirlos solo es cuestión de tiempo y de que el programa tenga éxito y se utilice frecuentemente.
- Todos los programas sufren modificaciones a lo largo de su vida, al menos todos aquellos que tienen éxito.

Normalmente los programas que tienen éxito serán modificados en el futuro, bien por el autor del programa, o por otro programador del equipo. Por esta razón es necesario tener bien documentado el código, para poder hacer modificaciones de forma sencilla. Esta documentación va a permitir que otro programador ajeno localice enseguida los cambios a realizar.

A la hora de documentar interesa que se explique lo que hace una clase o un método y por qué y para qué se hace. Así se indicará entre otras cosas, de qué se encarga una clase, un paquete, un método o una variable. Cuál es el uso esperado de esa variable o de ese método. Qué algoritmo se utiliza para resolver algún problema. Qué se debería revisar y modificar si hubiese tiempo para ello.

Para documentar proyectos existen muchas herramientas, normalmente cada lenguaje dispone de su propia herramienta como PHPDoc, phpDocumentor, Javadoc o JSDoc el javadoc para JavaScript. En esta unidad nos ocuparemos de la herramienta Javadoc para documentar programas java.

¡IMPORTANTE!

Citas a tener en cuenta sobre documentación de código:

"If your program isn't worth documenting, it probably isn't worth running" (Si su programa no merece la pena documentarlo, probablemente no merece la pena ejecutarlo). J. Nagler. 1995.

"Do not document bad code - rewrite it" (No documentar código malo – reescribirlo). R. Caron. 2000.

"Write the documentation before you write the code." (Escribe la documentación antes de escribir el código). S.W. Ambler. 2000.

4.3.1. Uso de JavaDoc en Eclipse

JavaDoc es la utilidad de Java para extraer y generar documentación directamente del código en formato HTML. Para que la documentación sea en verdad útil debemos escribir los comentarios del código de acuerdo a las recomendaciones de Javadoc. La documentación y el código se van a incluir dentro del mismo fichero. Veamos a continuación las recomendaciones sobre los comentarios y la documentación del código fuente.

Los tipos de comentarios para generar la documentación son:

- **Comentarios de línea:** comienzan con los caracteres "://" y terminan con la línea.
- **Comentarios tipo C:** comienzan con los caracteres "/*", y terminan con los caracteres "*/". Pueden agrupar a varias líneas.
- **Comentarios de documentación Javadoc:** estos comentarios se colocan entre los delimitadores /** ... */, agrupan varias líneas, y cada línea irá precedida por un *, y lo más importante es que estos **deben colocarse antes de la declaración de una clase, un campo, un método o un constructor**. Dentro de estos delimitadores se podrá escribir etiquetas HTML. Los comentarios Javadoc están formados por dos partes una descripción seguida de un bloque de **tags**. Un ejemplo de documentación puede ser el siguiente:

```
/**  
 * <h2>Este es un ejemplo de documentación</h2>.  
 * Puedo añadir etiquetas HTML, para <b>mejorar</b> la presentación.  
 * Por ejemplo añado unas viñetas.  
 * <ul>  
 *   <li>Inserción de registros</li>  
 *   <li>Borrado y Modificación de registros</li>  
 * </ul>  
 *  
 * @author ARNM 2014  
 * @version v1.2014  
 *
```

```
*/
class prueba { ...
```

USO DE ETIQUETAS DE DOCUMENTACIÓN

Se pueden usar tags para documentar ciertos aspectos concretos como la versión de la clase, el autor, los parámetros utilizados, o los valores devueltos. Las etiquetas de Javadoc van precedidas por @, estas son las más usadas:

ETIQUETA	DESCRIPCIÓN
@author	Autor de la clase. Solo para las clases.
@version	Versión de la clase. Solo para clases.
@see	Referencia a otra clase, ya sea del API, del mismo proyecto o de otro. Por ejemplo: @see cadena @see paquete.clase#miembro @see enlace
@param	Descripción de parámetro. Una etiqueta por cada parámetro.
@return	Descripción de lo que devuelve. Solo si no es void. Podrá describir valores de retorno especiales según las condiciones que se den, dependiendo del tipo de dato.
@throws	Descripción de la excepción que puede propagar. Habrá una etiqueta throws por cada tipo de excepción.
@deprecated	Marca el método como obsoleto. Solo se mantiene por compatibilidad.
@since	Indica el nº de versión desde la que existe el método.

Observa este ejemplo de documentación de una clase utilizando etiquetas:

```
/**
 * <h2>Clase Empleado, se utiliza para crear y leer empleados de una BD</h2>
 *
 * Busca información de javadoc en <a href="http://google.com">GOOGLE</a>
 * @see <a href="http://www.google.com">Google</a>
 * @version 1-2014
 * @author ARM
 * @since 1-1-2014
 */
public class Empleado {
    /**
     * Atributo Nombre del empleado
     */
    private String nombre;
    /**
     * Atributo apellido del empleado
     */
```

```
private String apellido;
/**
 * Edad del empleado
 */
private double salario;

/**
 * Constructor con 3 parámetros.
 * Crea objetos empleado, con nombre, apellido y salario.
 * @param nombre Nombre del empleado
 * @param apellido Apellido del empleado
 * @param salario Salario del empleado
 */
public Empleado(String nombre, String apellido, double salario){
    this.nombre=nombre;
    this.apellido=apellido;
    this.salario=salario;
}

//Métodos públicos

/**
 * Sube el salario al empleado.
 * @see Empleado
 * @param subida
 *
 */
public void subidasalario (double subida){
    salario=salario + subida;
}

//Métodos privados
/**
 * Comprueba que el nombre no este vacío
 * @return <ul>
 *         <li>true: el nombre es una cadena vacía</li>
 *         <li>false: el nombre no es una cadena vacía</li>
 *     </ul>
 */
private boolean comprobar(){
    if(nombre.equals("")){
        return false;
    }
    return true;
}
```

RECUERDA:

Los comentarios de documentación Javadoc deben colocarse ANTES de las declaraciones de las clases, de los métodos, o de los atributos.

GENERAR LA DOCUMENTACIÓN

La mayor parte de los entornos de desarrollo incluyen un botón o un enlace para configurar y ejecutar Javadoc. Para ejecutar Javadoc desde Eclipse, se abre el menú Project y se elige Generate Javadoc. En la ventana que se muestra pedirá la siguiente configuración (véase la Figura 4.58):

- En *Javadoc command* se tiene que indicar dónde se encuentra el archivo ejecutable de Javadoc el *javadoc.exe*. Se pulsa el botón *Configure* para buscarlo dentro de la carpeta donde se encuentra instalado el JDK. Y dentro de esa carpeta se elige la carpeta bin.
- En los dos cuadros inferiores se elegirá el proyecto y las clases a documentar.
- Se selecciona la visibilidad de los elementos que se van a documentar. Con *Private* se documentarán todos los miembros públicos, privados y protegidos.
- Por último, se indica la carpeta de destino donde se almacenará el código HTML (conviene poner un nombre diferente al que viene por defecto pues con doc hay veces que no se cargan los estilos).



Figura 4.58. Generación de documentación con Javadoc.



Figura 4.59. Configuración de la salida de Javadoc

Se pulsa *Next*, y en la siguiente ventana (véase la Figura 4.59) se indica el título del documento html que se genera, y se eligen las opciones para la generación de las páginas HTML. Como mínimo se selecciona la barra de navegación y el índice. En la Figura 4.60 se muestra la documentación generada.



Figura 4.60. Documentación generada.

ACTIVIDAD 4.11

Realiza pruebas de documentación en los proyectos de las actividades anteriores.

Añade autor y versión a las clases.

Añade descripciones a las clases, los métodos y sus parámetros utilizando etiquetas HTML.

Genera la documentación cambiando las distintas opciones.

4.4. REFACTORIZACIÓN

La refactorización es una técnica de la ingeniería de software que permite la optimización de un código previamente escrito, por medio de cambios en su estructura interna sin que esto suponga alteraciones en su comportamiento externo (véase la Figura 4.61); dicho de otro modo, la refactorización no busca ni arreglar errores ni añadir nueva funcionalidad, sino mejorar la comprensión del código para facilitar así nuevos desarrollos, la resolución de errores o la adición de alguna funcionalidad al software. (<http://es.wikibooks.org/wiki/Refactorización/Definición>)

La refactorización tiene como objetivo limpiar el código para que sea más fácil de entender y de modificar, permitiendo una mejor lectura para comprender qué es lo que se está realizando. Despues de refactorizar el proyecto seguirá ejecutándose igual y obteniendo los mismos resultados.

¿Qué hace la refactorización?:

- Limpia el código, mejorando la consistencia y la claridad.
- Mantiene el código, no corrige errores ni añade funciones nuevas.
- Va a permitir facilitar la realización de cambios en el código
- Se obtiene un código limpio y altamente modularizado.

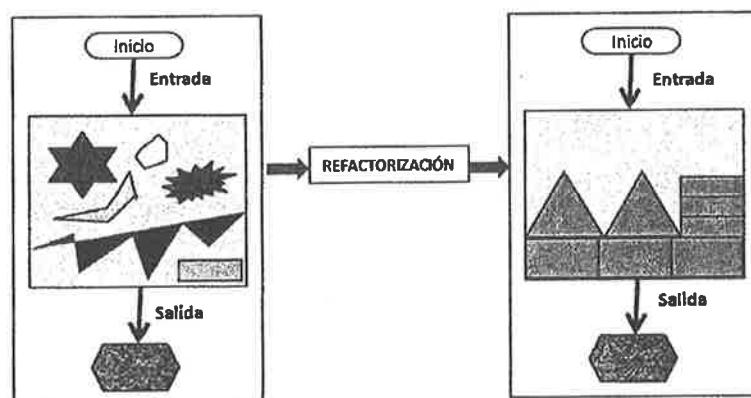


Figura 4.61. Refactorización antes y después

4.4.1. Cuándo refactorizar. **Malos olores (bad smells)**

La refactorización se debe ir haciendo mientras se va realizando el desarrollo de la aplicación, *Piattini y Garcia (2003)* analizan los síntomas que indican la necesidad de refactorizar, a los que *Martin Fowler* y otros autores (1999) llamaron **bad smells** (*malos olores*). Estos síntomas son los siguientes:

- **Código duplicado (Duplicated code).** Es la principal razón para refactorizar. Si se detecta el mismo código en más de un lugar, se debe buscar la forma de extraerlo y unificarlo.
- **Métodos muy largos (Long method).** Cuanto más largo es un método más difícil es de entender. Un método muy largo normalmente está realizando tareas que deberían ser responsabilidad de otros. Se deben identificar y descomponer el método en otros más pequeños. En la programación orientada a objetos cuanto más corto es un método más fácil es reutilizarlo.
- **Clases muy grandes (Large class).** Si una clase intenta resolver muchos problemas, tendremos una clase con demasiados métodos, atributos o incluso instancias. La clase está asumiendo demasiadas responsabilidades. Hay que intentar hacer clases más pequeñas, de forma que cada una trate con un conjunto pequeño de responsabilidades bien delimitadas.
- **Lista de parámetros extensa (Long parameter list).** En la programación orientada a objetos no se suelen pasar muchos parámetros a los métodos, sino solo aquellos mínimamente necesarios para que el objeto involucrado consiga lo necesario. Tener demasiados parámetros puede estar indicando un problema de encapsulación de datos o la necesidad de crear una clase de objetos a partir de varios de esos parámetros, y pasar ese objeto como argumento en vez de todos los parámetros. Especialmente si esos parámetros suelen tener que ver unos con otros y suelen ir juntos siempre.
- **Cambio divergente (Divergent change):** una clase es frecuentemente modificada por diversos motivos, los cuales no suelen estar relacionados entre sí, a lo mejor conviene eliminar la clase. Este síntoma es el opuesto del siguiente.
- **Cirugía a tiro pistola (Shotgun surgery):** este síntoma se presenta cuando después de un cambio en una determinada clase, se deben realizar varias modificaciones adicionales en diversos lugares para compatibilizar dicho cambio.
- **Envidia de funcionalidad (Feature envy):** se observa este síntoma cuando tenemos un método que utiliza más cantidad de elementos de otra clase que de la propia. Se suele resolver el problema pasando el método a la clase cuyos elementos utiliza más.
- **Clase de solo datos (Data class):** Clases que solo tienen atributos y métodos de acceso a ellos (“get” y “set”). Este tipo de clases deberían cuestionarse dado que no suelen tener comportamiento alguno.
- **Legado rechazado (Refused bequest):** este síntoma lo encontramos en subclases que utilizan solo unas pocas características de sus superclases. Si las subclases no necesitan o no requieren todo lo que sus superclases les proveen por herencia, esto suele indicar que como fue pensada la jerarquía de clases no es correcto. La delegación suele ser la solución a este tipo de inconvenientes.

El proceso de refactorización presenta algunas ventajas, entre las que se encuentran el mantenimiento del diseño del sistema, incremento de facilidad de lectura y comprensión del código fuente, detección temprana de fallos, aumento en la velocidad en la que se programa.

En cambio existen áreas conflictivas en la refactorización, tanto *Fowler*, como *Piattini y García* coinciden en que las áreas conflictivas de la refactorización son las bases de datos, y los cambios de interfaces. Un cambio de base de datos es muy costoso pues los sistemas están muy acoplados a las bases de datos, y sería necesaria una migración tanto de estructura como de datos.

4.4.2. Refactorización en Eclipse

Eclipse tiene diversos métodos de refactorizar o *refactoring*. Dependiendo de dónde invoquemos a la refactorización tendremos un menú contextual u otro con sus diferentes opciones de refactorización (véase la Figura 4.62). Para refactorizar, elegiremos la opción *Refactor* del menú contextual.

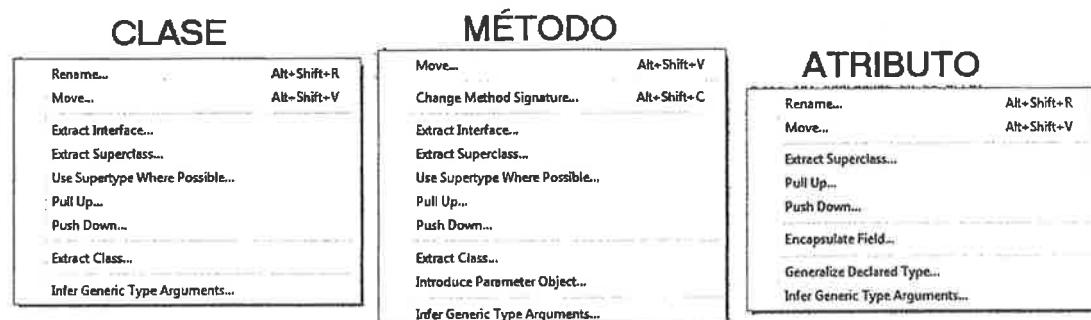


Figura 4.62. Menú Refactor asociado a una clase, un método y un atributo.

MÉTODOS DE REFACTORIZACIÓN

Los *métodos de refactorización* son las prácticas para refactorizar el código, utilizando las herramientas podremos plantear casos para refactorizar y se mostrarán las posibles soluciones en las que podremos ver el antes y el después de refactorizar. A los métodos de refactorización también se les llama *patrones de refactorización* o *catálogos de refactorización*.

Para refactorizar se selecciona el elemento (puede ser una clase, una variable, una expresión, un bloque de instrucciones, un método, etc.), se pulsa al botón derecho del ratón, se selecciona *Refactor*, y seguidamente se selecciona el *método de refactorización*. A continuación se muestran algunos de los métodos más comunes:

- **Rename.** Es una de las opciones más utilizadas. Cambia el nombre de variables, clases, métodos, paquetes, directorios y casi cualquier identificador Java. Tras la refactorización, se modifican las referencias a ese identificador.
- **Move.** Mueve una clase de un paquete a otro, se mueve el archivo .java a la carpeta, y se cambian todas las referencias. También se puede arrastrar y soltar una clase a un nuevo paquete, se realiza una refactorización automática.

- **Extract Constant.** Convierte un número o cadena literal en una constante. Al hacer la refactorización se mostrará dónde se van a producir los cambios, y se puede visualizar el estado antes de refactorizar y después de refactorizar. Tras la refactorización, todos los usos del literal se sustituyen por esa constante. El objetivo es modificar el valor del literal en un único lugar.
- **Extract Local Variable.** Asignar una expresión a variable local. Tras la refactorización, cualquier referencia a la expresión en el ámbito local se sustituye por la variable. La misma expresión en otro método no se modifica. En la figura se muestran los cambios que se producirán al extraer una variable local, se muestran dónde se realizarán los cambios, y el detalle antes y después (Figura 4.63).

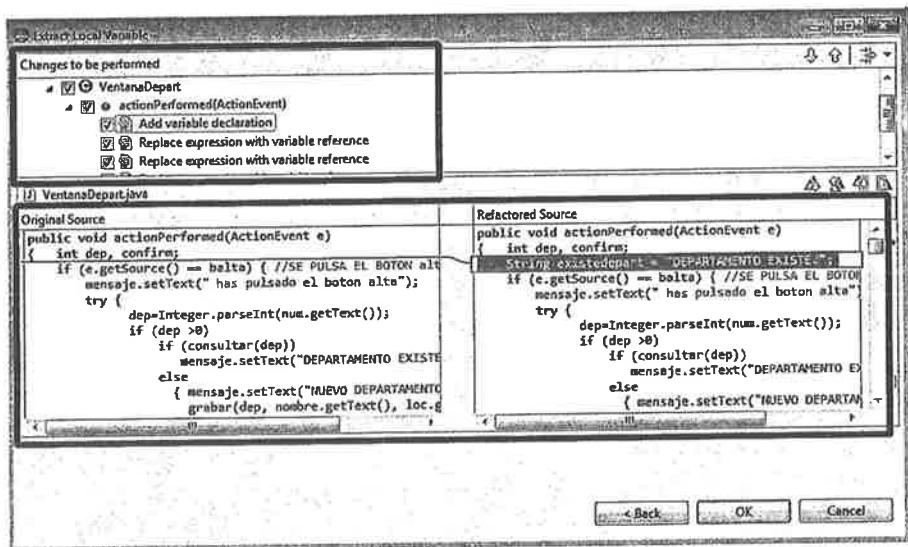


Figura 4.63. Extraer una variable local.

- **Convert Local Variable to Field.** Convierte una variable local en un atributo privado de la clase. Tras la refactorización, todos los usos de la variable local se sustituyen por ese atributo.

ACTIVIDAD 4.12

Abre el proyecto *FicheroAleatorioVentana* que se encuentra en la carpeta de recursos. Realiza los siguientes cambios en la clase *VentanaDepart.java*:

Extrae una variable local llamada *existedepart* de la cadena: "*DEPARTAMENTO EXISTE*".

Extrae una constante local llamada *NOEXISTEDEPART* de la cadena: "*DEPARTAMENTO NO EXISTE*".

Convierte la variable local creada en un atributo de la clase.

Extrae una variable local llamada *depar_error* de la cadena: "*DEPARTAMENTO ERRÓNEO*", y conviértela en un atributo de la clase.

- **Extract Method.** Nos permite seleccionar un bloque de código y convertirlo en un método. El bloque de código no debe dejar llaves abiertas. Eclipse ajustará automáticamente los parámetros y el retorno de la función. Esto es muy útil para utilizarlo cuando se crean métodos muy largos, que se podrán dividir en varios métodos. También es muy útil extraer un método cuando se tiene un grupo de instrucciones que se repiten varias veces.

Al extraer el método hay que indicar el modificador de acceso: público, protegido, privado, o sin modificador.

ACTIVIDAD 4.13

Extraer métodos dentro la clase VentanaDepart.java.

Dentro del método `public void actionPerformed(ActionEvent e)`, extrae varios métodos, uno para cada una de estas operaciones insertar departamento, consultar, borrar y modificar .

Extraer los métodos de la instrucciones que van dentro de los distintos `if (e.getSource())`, que preguntan por *balta*, *consu*, *borra* y *modif*, llamarlos *altadepart*, *consuldepart*, *borradepart* y *modifdepart*.

- **Change Method Signature.** Este método permite cambiar la firma de un método. Es decir, el nombre del método y los parámetros que tiene. De forma automática se actualizarán todas las dependencias y llamadas al método dentro del proyecto. En la Figura 4.64, se muestra la ventana para cambiar la firma de un método, en ella se indicará el nuevo nombre del método, el tipo de dato que devuelve, los nuevos parámetros, se pueden editar los parámetros y cambiarlos, o también asignar un valor por defecto.

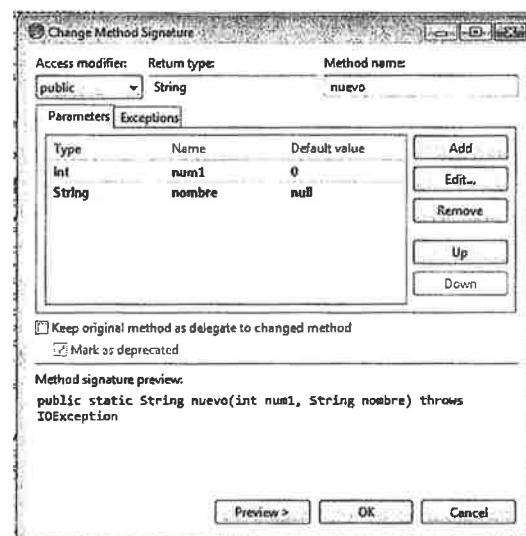


Figura 4.64. Cambiar la firma de un método.

NOTA:

Si al refactorizar cambiamos el tipo de dato de retorno del método, aparecerán errores de compilación, por lo que debemos modificarlo manualmente.

ACTIVIDAD 4.14

Prueba a cambiar la firma de los métodos creados anteriormente.

Añade un parámetro de tipo String, valor por defecto “PRUEBA”.

Cambia el tipo de dato devuelto en los métodos, haz que retornen un entero.

Comprueba y corrige los errores de retorno.

- **Inline.** Nos permite ajustar una referencia a una variable o método con la línea en la que se utiliza y conseguir así una única línea de código. Cuando se utiliza, se sustituye la referencia a la variable o método con el valor asignado a la variable o la aplicación del método, respectivamente.

Por ejemplo, dentro de la clase *FicheroAleatorioVentana* nos encontramos con esta declaración:

```
File fichero = new File("AleatorioDep.dat");
RandomAccessFile file;
file = new RandomAccessFile(fichero, "rw");
```

Posicionamos el cursor en la referencia al método o variable, en este caso la variable *fichero*. Seleccionamos la opción “**Inline**” y el resultado es:

```
RandomAccessFile file;
file = new RandomAccessFile(new File("AleatorioDep.dat"), "rw");
```

- **Member Type to Top Level.** Convierte una clase anidada en una clase de nivel superior con su propio archivo de java. Si la clase es estática, la refactorización es inmediata. Si no es estática nos pide un nombre para declarar el nombre de la clase que mantendrá la referencia con la clase inicial.

ACTIVIDAD 4.15

Crea esta clase anidada dentro de la clase *FicheroAleatorioVentana*:

```
class claseAnidada {
    void entrada() {
        System.out.println( "Método entrada." );
    }

    String salida (int d) {
```

```

        System.out.println( "Salida." );
        return "Salida el " + d;
    }
} // fin clase anidada

```

Crea un objeto de esta clase y llama a los métodos dentro del método *verporconsola*. Añade este código en el método *verporconsola*:

```

FicheroAleatorioVentana fa = new FicheroAleatorioVentana();
claseAnidada ej = new claseAnidada();
ej.entrada();
System.out.println( "Llamo a Salida: " + ej.salida(10) );

```

Prueba la ejecución (pulsa el botón *Ver por consola* de la ventana de ejecución).

Convertir la clase anidada en una de nivel superior con su archivo asociado (*Move Type to new file*). Observa que se ha creado una nueva clase con su fichero java.

Probar la ejecución.

- **Extract Interface.** Este método de refactorización nos permite escoger los métodos de una clase para crear una *Interface*. Una *Interface* es una especie de plantilla que define los métodos acerca de lo que puede o no hacer una clase. La *Interface* define los métodos pero no los desarrolla. Serán las clases que implementen la *Interface* quien desarrolle los métodos.

Por ejemplo, se define la interface *Animal*, y los métodos *comer* y *respirar* porque todos los animales comen y respiran. Sin embargo cada animal va a comer y respirar de forma diferente, por eso en cada animal se deben desarrollar esos métodos.

Clase Interface	Clase que implementa la Interface
<pre> interface Animal { void comer(); int respirar(); } </pre>	<pre> class Perro implements Animal{ public void comer(){ //se define como come el perro } public int respirar(){ // se define cómo respira el perro } public String ladrar(){ // método exclusivo del perro } } </pre>

ACTIVIDAD 4.16

Crea la interface *InterfaceVentanaDepart* (véase la Figura 4.65) que contenga los métodos creados en la Actividad 4.13. Al crear la interface solo se pueden añadir los métodos públicos. Realiza los cambios que se necesiten para crearla.

Observa en el *Package Explorer* (véase la Figura 4.66) cómo se visualiza la clase creada, observa también el cambio producido en la declaración de la clase *VentanaDepart*.

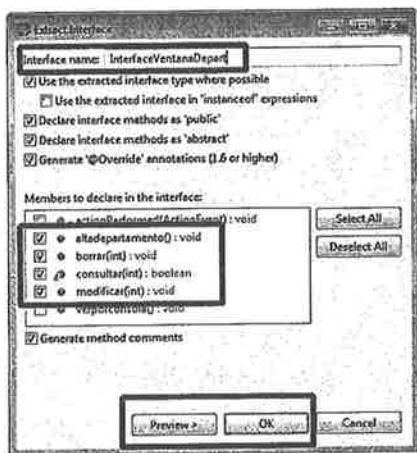


Figura 4.65. Creación de una Interface.

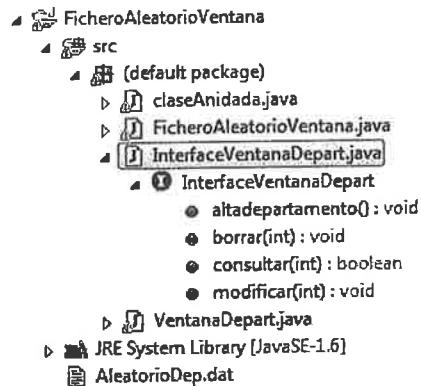


Figura 4.66. Vista de la Interface.

- **Extract Superclass.** Este método permite extraer una superclase. Si la clase ya utilizaba una superclase, la recién creada pasará a ser su superclase. Se pueden seleccionar los métodos y atributos que formaran parte de la superclase. En la superclase, los métodos están actualmente allí, así que si hay referencias a campos de clase original, habrá fallos de compilación.

ACTIVIDAD 4.17

A partir de la clase *VentanaDepart* crea la superclase *SuperclaseDepart*, que incluya sólo los métodos de *grabar* y *visualizar*.

Observa la clase generada y los constructores generados. Observa que la declaración de la clase *VentanaDepart* ha cambiado, ahora es extends de *SuperclaseDepart*.

Observa los errores generados, dos se producen por hacer referencias a campos de la clase original (*nombre* y *loc*), el tercero por definir un objeto de la clase *claseAnidada* con el parámetro (*this*) de la clase actual.

Soluciona los errores moviendo las declaraciones de los campos a la superclase, y casteando el argumento del objeto de la clase *claseAnidada* a *InterfaceVentanaDepart*.

- **Convert Anonymous Class to Nested.** Este método de refactorización permite convertir una clase anónima a una clase anidada de la clase que la contiene. Una clase anónima es una clase sin nombre de la que solo se crea un único objeto, de esta clase no se pueden definir constructores. Se utilizan con frecuencia cuando se crean ventanas, para gestionar los eventos de los distintos componentes de la interfaz gráfica.

Una clase anónima se puede definir de las siguientes maneras:

- Se utiliza la palabra new seguida de la definición de la clase anónima entre llaves {...}.
- Palabra new seguida del nombre de la clase de la que hereda (sin extends) y la definición de la clase entre llaves {...}.
- Palabra new seguida del nombre de la interface (sin implements) y la definición de la clase anónima entre llaves {...}.

Este es un ejemplo de una clase anónima que implementa el método *actionPerformed*, que es el único método de la interface *ActionListener* (esta clase detecta los eventos de acción, es decir, cuando se pulsa un botón, o cuando se pulsa INTRO, o cuando se cambia el elemento de una lista desplegable o se elige un elemento de menú).

```
 JButton btnDepart = new JButton("Operaciones Departamentos");
btnDepart.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        OperacionesDepart opDepart=new OperacionesDepart();
        opDepart.setVisible(true);
    }
});
```

Se convierte la clase anónima *new ActionListener* a clase anidada llamada *OperacionesDep* dentro de la clase donde está. Nos posicionamos sobre la clase anónima, botón derecho *Refactor/Convert Anonymous Class to Nested Class*. Se elige el modificador de acceso y en *Type name* se escribe el nombre. Si se selecciona final se indica que no se podrán crear clases derivadas de esta clase (véase la Figura 4.67)

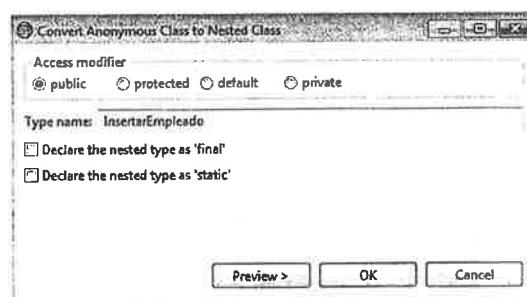


Figura 4.67. Convertir clase anónima a anidada.

Después de la refactorización la nueva clase quedará así:

```
private final class OperacionesDep implements ActionListener {
    public void actionPerformed(ActionEvent arg0) {
        OperacionesDepart opDepart=new OperacionesDepart();
        opDepart.setVisible(true);
    }
}
```

Y la ejecución del evento quedará así:

```
 JButton btnDepart = new JButton("Operaciones Departamentos");
btnDepart.addActionListener(new OperacionesDep());
```

NOTA: `@Override`: informa al compilador que el elemento está pensado para sobrescribir a un elemento declarado en una superclase.

OTRAS OPERACIONES DE REFACTORIZACIÓN

Eclipse permite visualizar el *histórico de refactorizaciones* realizado sobre un proyecto. Para ver el histórico se abre el menú *Refactor/History*. En la ventana que se muestra se pueden observar todos los cambios realizados y el detalle de los cambios (véase la Figura 4.68). Se puede elegir uno de los cambios pulsar el botón *Remove* si se desea borrar del histórico de refactorización. Eclipse también permite crear un *Script* con todos los cambios realizados en la refactorización y guardarlo en un fichero XML (menú *Refactor/Create Script*) o también cargar un script de refactorización (menú *Refactor/Apply Script*).

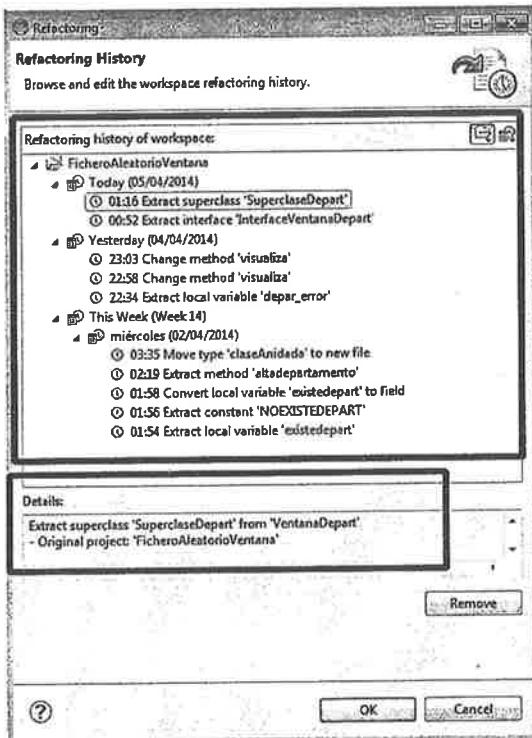


Figura 4.68. Histórico de refactorizaciones.

COMPRUEBA TU APRENDIZAJE

1. ¿Qué es el control de versiones?
2. ¿Qué es SUBVERSIÓN? ¿Cómo funciona el ciclo de vida en SUBVERSIÓN?
3. En control de versiones qué significa:
 - Repositorio.
 - Revisión, versión.
 - Hacer Checkout.
 - Hacer Commit.
 - Hacer Import.
 - Actualizar.
 - Crear una Rama.
 - Crear una etiqueta.
 - Crear un conflicto.
4. ¿Qué es documentar el código de un programa?
5. ¿Qué tipos de documentación se realizan en los proyectos?
6. ¿Qué es JavaDoc? ¿Para qué se utiliza?
7. ¿Qué es la refactorización?
8. ¿Cuándo hay que refactorizar?
9. ¿Qué son los **bad smells**?
10. Cita métodos de refactorización en eclipse.

ACTIVIDADES DE AMPLIACIÓN

1. REFACTORIZAR Y DOCUMENTAR

Copia el proyecto Eclipse con nombre *EjercicioNeodatis*, que se encuentra en la carpeta de recursos *ProyectosEjercicios_Eclipse*. Ábrelo con Eclipse y realiza las siguientes operaciones. Al hacer los cambios la ejecución del proyecto debe funcionar correctamente:

- Dentro de la clase *OperacionesEmple.java*, convierte en clases anidadas todas las clases anónimas creadas con los *ActionListener* de los botones. Que recojan los métodos *actionPerformed* asociados a los botones insertar, consultar, borrar y modificar.
- Dentro de la clase *OperacionesDepart* extrae los siguientes métodos, deben quedar dentro de esta clase, como métodos de la clase:

Dentro del *actionPerformed* del botón *btnInsertarDepartamento*, extrae el método *Insertardep* de las líneas que están dentro del *try{ }catch (NumberFormatException e)*.

Dentro del *actionPerformed* del botón *btnBorrarDepartamento*, extrae el método *borrardep* de las líneas que están dentro del *try{ }catch (NumberFormatException e)*.

Lo mismo con los *actionPerformed* de los botones *btnConsultar* y *btnModificarDepartamento*, extrae los métodos *consultardep* y *modificarddep*.

- Extrae la interfaz *Interfazdepart* con los métodos creados en el apartado anterior. Realiza los cambios que se necesiten.
- En la clase *Consultas* convierte la variable *BBDD* a global para toda la clase. Cambia el método *consuldepart*, añadirle 2 parámetros de tipo int, con valor por defecto a 0, y haz que devuelva un string. Corrige los errores que se produzcan. Extrae una clase que incorpore todos los *JButton* y *JLabel* de la clase. Llamarla *Etiquetas*.
- Cambia el nombre al proyecto y ponle tu nombre.
- Añade a la clase *Principal* las etiquetas autor, con vuestro nombre y versión por ejemplo: *Versión 1 2014*. Y el siguiente texto que aparezca en etiquetas de título , h2 de html:

"Proyecto pruebas de refactorización 2014, Este proyecto realiza el mantenimiento de la BD neodatis Empleados.dat. La base de datos está formada por las clases persistentes de Empleados y Departamentos. Un departamento tiene muchos empleados."

En el título del documento ponéis: *Documentación EJERCICIO REFACTORIZACIÓN*.

Añade las siguientes líneas de documentación entre etiquetas h2 de html, en las siguientes clases:

- *OperacionesDepart*: Esta clase se encarga del mantenimiento de los datos de departamentos, se realizan altas, bajas modificaciones y consultas.
- *OperacionesEmple*: Esta clase se encarga del mantenimiento de los datos de los empleados, se realizan altas, bajas modificaciones y consultas.
- *Consultas*: Esta clase se encarga de realizar consultas estadísticas de empleados y departamentos. También obtiene un listado de empleados y un listado de departamentos.

Genera la documentación sólo de esas clases con javadoc, con índice, y la barra de navegación. Y guárdala dentro de la carpeta del proyecto.

2. Crea con *VisualSVN Server* el repositorio *Ejercicios*, con la estructura *trunks-branches-tags*. Crea el usuario *ejercicios* con clave *ejercicios*.

3. CONTROL DE VERSIONES. TORTOISE.

- Conéctate a ese repositorio con el cliente *Tortoise*. Crea dentro de *trunk* la carpeta *Ejercicios1*, y sube varios archivos dentro de esa carpeta. Añade algún comentario.
- Crea en tu disco duro una carpeta de trabajo y realiza un CHECKOUT de la carpeta *Ejercicios1* de *trunk*.
- Dentro de *branches* crea la carpeta *Ejercicios1* y crea la rama *revisión1* con los datos de la carpeta *Ejercicios1* de *trunk*.
- Crea en tu disco duro una carpeta de trabajo y realiza un CHECKOUT de la rama. Añade algún archivo y modifica algún documento. Confirma cambios y añade un comentario.
- Fusiona la rama con el tronco.

4. CONTROL DE VERSIONES. SQLDEVELOPER.

- Crea una nueva carpeta de trabajo y copia la carpeta *Almacenes*, que se encuentra en la carpeta de recursos *ModelosEjercicios_Developer*.
- Abre el proyecto con *datamodeler* de *sqldeveloper*. Crea una conexión con el repositorio creado *Ejercicios*. Crea dentro de *trunk* la carpeta *Ejercicios2*, y sube el proyecto a esa carpeta. Añade algún comentario.
- Realiza cambios en el modelo lógico, y en el modelo relacional. Valida por separado, primero lógico y luego relacional. Añade un comentario en cada caso.
- Crea en tu disco duro una carpeta de trabajo y realiza un CHECKOUT del proyecto. Realiza cambios en el modelo lógico y relacional. Confirma cambios y añade un comentario.
- Abre el proyecto inicial, el que se ha subido en el primer paso. Realiza alguna modificación. Comprueba los cambios que hay.
- Soluciona los conflictos si surgen.

5. CONTROL DE VERSIONES. ECLIPSE.

- Crea una nueva carpeta de trabajo y copia la carpeta *EjemploMantBD_SQLite*, que se encuentra en la carpeta de recursos *ProyectosEjercicios_Eclipse*.
- Abre el proyecto con Eclipse. Crea una conexión con el repositorio creado *Ejercicios*. Crea dentro de *trunk* la carpeta *Ejercicios3*, y sube el proyecto a esa carpeta. Añade algún comentario.
- Crea una nueva carpeta de trabajo y realiza un CHECKOUT del proyecto subido, cámbiale de nombre, por ejemplo *EjemploMantBD_SQLite2*.
- Realiza cambios en *EjemploMantBD_SQLite* y valida.
- Realiza cambios en *EjemploMantBD_SQLite2* y valida. ¿Qué ocurre?
- Resuelve el conflicto.

CAPÍTULO 5

ELABORACIÓN DE DIAGRAMAS DE CLASES

CONTENIDOS

- Lenguaje UML
- Clases. Atributos, métodos y visibilidad
- Objetos. Instanciación
- Relaciones. Herencia, composición y agregación
- Herramientas de diseño de diagramas
- Generación de código a partir del diagrama de clases
- Ingeniería inversa

OBJETIVOS

- Identificar los conceptos básicos de la programación orientada a objetos
- Elaborar e interpretar diagramas de clase sencillos
- Utilizar herramientas para desarrollar diagramas de clases
- Generar código a partir de diagramas de clases
- Realizar ingeniería inversa a partir de código

RESUMEN DEL CAPÍTULO

En este capítulo introducimos los conceptos básicos que forman el fundamento para la comprensión de la tecnología orientada a objetos. Se hará una introducción al lenguaje UML y sus diagramas. Se estudiarán los diagramas de clases y se utilizarán distintas herramientas para crear diagramas de clases, para generar código a partir de diagramas y para realizar la ingeniería inversa, a partir de proyectos java.

5.1. INTRODUCCIÓN

En el diseño orientado a objetos, un sistema se entiende como un conjunto de objetos que tienen propiedades y comportamientos.

Un objeto consta de una estructura de datos y de una colección de métodos u operaciones que manipulan esos datos. Los datos definidos dentro de un objeto son sus atributos. Las operaciones definen el comportamiento del objeto y cambian el valor de uno o más atributos. Los objetos se comunican unos con otros a través del paso de mensajes.

Una clase no es más que una plantilla para la creación de objetos. Cuando se crea un objeto (instanciación) se ha de especificar de qué clase es el objeto instanciado, para que el compilador comprenda las características del objeto.

Para el análisis y diseño orientado a objetos se utiliza UML (*Unified Modeling Language - Lenguaje de Modelado Unificado*). Es un lenguaje de modelado basado en diagramas que sirve para expresar modelos (un modelo es una representación de la realidad donde se ignoran detalles de menor importancia). Se ha convertido en el estándar de facto de la mayor parte de las metodologías de desarrollo orientado a objetos que existen hoy en día.

5.2. CONCEPTOS ORIENTADOS A OBJETOS

El paradigma OO se basa en el concepto de **objeto**. Un objeto es aquello que tiene estado (propiedades más valores), comportamiento (acciones y reacciones a mensajes) e identidad (propiedad que lo distingue de los demás objetos). La estructura y comportamiento de objetos similares están definidos en su clase común; los términos instancia y objeto son intercambiables. Una clase es un conjunto de objetos que comparten una estructura y comportamiento común.

Clases y objetos pueden parecer conceptos similares pero existe una clara diferencia conceptual entre ellos. Las clases son un concepto estático definido en el programa fuente, son una abstracción de la esencia de un objeto, mientras que los objetos son entes dinámicos que existen en tiempo y espacio, y que ocupan memoria en la ejecución de un programa.

En el enfoque OO las propiedades del objeto son claves. Los principios del modelo OO son: abstracción, encapsulación, modularidad, jerarquía y polimorfismo, fundamentalmente, y en menor grado tipificación (typing), concurrencia y persistencia. Booch [1986] dice que si un modelo que se dice OO no contiene alguno de los primeros cuatro elementos, entonces no es OO. Propiedades:

- **Abstracción.** Denota las características esenciales de un objeto, donde se capturan sus comportamientos. El objetivo es obtener una descripción formal. La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella podemos llegar a componer un conjunto de clases que permitan modelar la realidad o el problema que se quiere resolver.
- **Encapsulación.** La encapsulación es el proceso de ocultar todos los detalles de un objeto que no contribuyen a sus características esenciales, es decir, separar el aspecto externo del objeto accesible por otros objetos, del aspecto interno del mismo que será inaccesible para los demás. O lo que es lo mismo: la encapsulación consiste en ocultar los atributos y métodos del objeto a otros objetos, estos no deben estar expuestos a los objetos exteriores. Una vez encapsulados, pasan a denominarse atributos y métodos privados del objeto.

- **Modularidad.** Es la propiedad de una aplicación o de un sistema que ha sido descompuesto en un conjunto de módulos o partes más pequeñas coherentes e independientes. Estos módulos se pueden compilar por separado, pero tienen conexiones con los otros módulos.
- **Jerarquía o herencia.** La programación orientada a objetos introduce la posibilidad de extender clases, produciendo nuevas definiciones de clases que heredan todo el comportamiento y código de la clase extendida. La clase original se denomina clase padre, base o superclase. La nueva clase que se define como una extensión se denomina clase hija, derivada o subclase. La extensión de una clase se denomina herencia, porque la nueva clase hija hereda todos los métodos y atributos de la clase padre que se extiende. Cada subclase estaría formada por un grupo de objetos más especializados con características comunes que compartirían datos y operaciones. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen.
- **Polimorfismo.** Consiste en reunir con el mismo nombre comportamientos diferentes. Es la propiedad por la cual un mismo mensaje puede originar conductas completamente diferentes al ser recibido por diferentes objetos. De un modo más preciso: dos instancias u objetos, pertenecientes a distintas clases, pueden responder a la llamada a métodos del mismo nombre, cada uno de ellos con distinto comportamiento encapsulado, pero que responden a una interfaz común (marcada a través del mecanismo de la herencia).
- **Tipificación.** Es la definición precisa de un objeto, de tal forma que objetos de diferentes tipos no puedan ser intercambiados o, cuando mucho, puedan intercambiarse de manera muy restringida.
- **Concurrencia.** Es la propiedad que distingue un objeto que está activo de uno que no lo está. El objeto activo está haciendo algo, se utilizan sobre todo en la programación concurrente o *multihilo*.
- **Persistencia.** Es la propiedad de un objeto a través de la cual su existencia trasciende el tiempo (es decir, el objeto continua existiendo después de que su creador ha dejado de existir) y/o el espacio. Se refiere a objetos de clases asociadas a Bases de Datos Orientadas a Objetos (BDOO) o a Bases de Datos Objeto Relacionales (BDOR).

Actualmente las metodologías más importantes de análisis y diseño de sistemas han confluido en lo que se es el UML, bajo el respaldo del *Object Management Group*.

5.3. QUÉ ES UML

El Lenguaje de Modelado Unificado (*UML - Unified Modeling Language*) es un lenguaje gráfico para visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de software. Este lenguaje se puede utilizar para modelar tanto sistemas de software, como de hardware, como organizaciones del mundo real. Para ello utiliza una serie de diagramas en los que se representan los distintos puntos de vista de modelado. Podemos decir que UML es un lenguaje que se utiliza para documentar.

Existen dos grandes versiones de UML:

- UML 1.X (comprende UML 1.1, 1.2, 1.3, 1.4, 1.5): desde finales de los 90 se empezó a trabajar con el estándar UML. En los años sucesivos fueron apareciendo nuevas versiones que introducían mejoras o ampliaban a las anteriores.

- UML 2.X (comprende UML 2.1 hasta UML 2.5, 2.6, etc.): en torno a 2005 se difundió una nueva versión de UML a la que podemos denominar UML 2.X. Comprenden varias revisiones.

UML 2.0 define 13 tipos de diagramas (véase la Figura 5.1), divididos en 3 categorías: 6 tipos de diagramas representan la estructura estática de la aplicación o del sistema, 3 representan tipos generales de comportamiento y 4 representan diferentes aspectos de las interacciones:

- **Diagramas de estructura (parte estática del modelo):** incluyen el diagrama de clases, diagrama de objetos, diagrama de componentes, diagrama de estructura compuesta, diagrama de paquetes y diagrama de implementación o despliegue. Se centran en los elementos que deben existir en el sistema modelado.
- **Diagramas de comportamiento (parte dinámica del modelo):** incluyen el diagrama de casos de uso (usado por algunas metodologías durante la recopilación de requisitos), diagrama de actividad y diagrama de estado. Se centran en lo que debe suceder en el sistema.
- **Diagramas de interacción:** todos derivados del diagrama de comportamiento más general. Incluyen el diagrama de secuencia, diagrama de comunicación, diagrama de tiempos y diagrama de vista de interacción. Se centran en el flujo de control y de datos entre los elementos del sistema modelado.

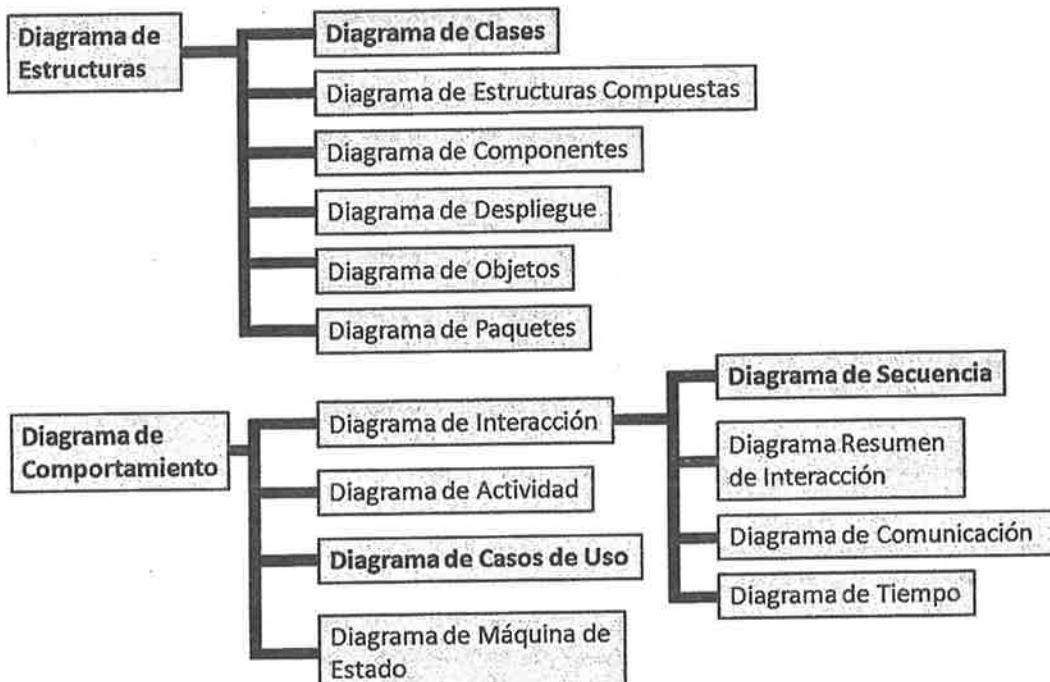


Figura 5.1. Diagramas UML 2.0

5.3.1. Tipos de diagramas

Cada diagrama UML representa alguna parte o punto de vista del sistema. Los diagramas más utilizados son los siguientes:

- **Diagramas de clase.** Los diagramas de clases muestran las diferentes clases que componen un sistema y cómo se relacionan unas con otras. En la Figura 5.2 se muestran las clases *Departamentos* y *Empleados*, y su relación o asociación. Un departamento tiene muchos empleados, y un empleado pertenece a un departamento. Un objeto Departamentos va a tener a muchos objetos empleados.

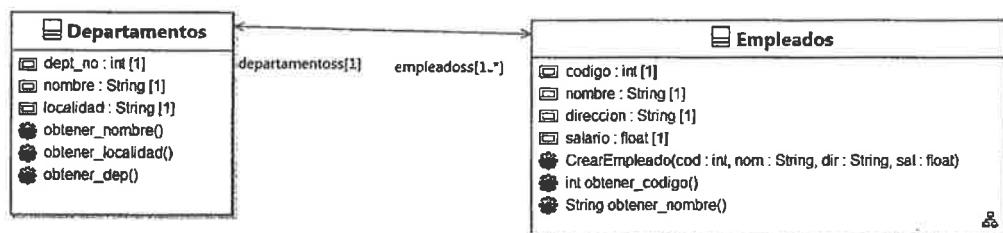


Figura 5.2. Diagrama de clase Departamentos-Empleados.

- **Diagramas de objeto.** Representan objetos y sus relaciones. Muestra una serie de objetos (instancias de las clases) y sus relaciones en un momento particular de la ejecución del sistema. Es un diagrama de instancias de las clases mostradas en el diagrama de clases. Son útiles para la comprensión de los diagramas de clases.
- **Diagramas de casos de uso.** Se utiliza para entender el uso del sistema, muestran un conjunto de actores, las acciones (casos de uso) que se realizan en el sistema, y las relaciones entre ellos. En la Figura 5.3 se muestra un diagrama de casos de usos en el que el actor operador realiza operaciones sobre los datos de los empleados de una base de datos.

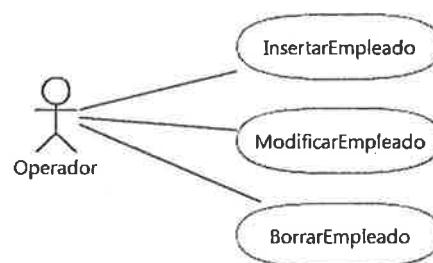
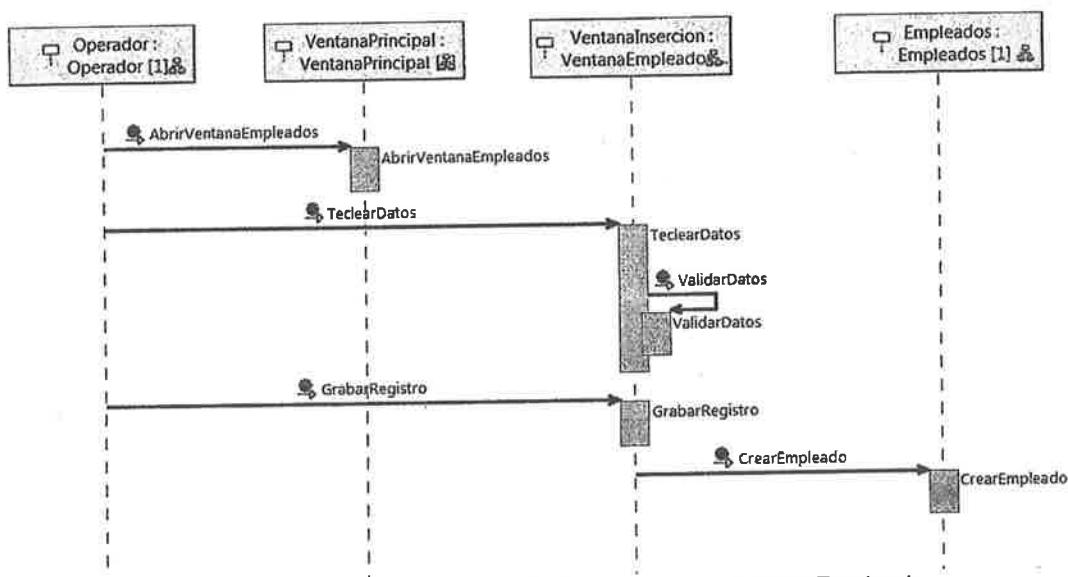


Figura 5.3. Diagrama de casos de uso.

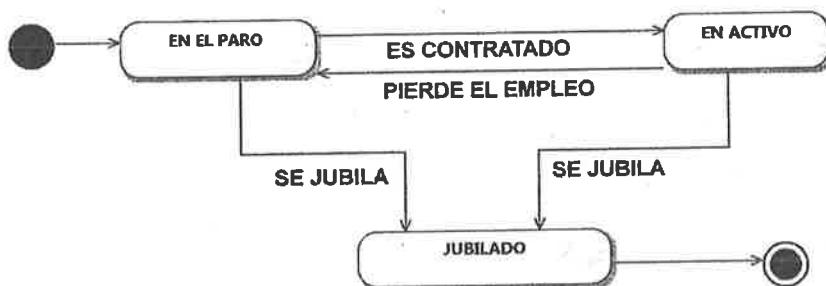
- **Diagramas de secuencia.** Son una representación temporal de los objetos y sus relaciones. Enfatiza la interacción entre los objetos y los mensajes que intercambian entre sí junto con el orden temporal de los mismos.

En el ejemplo (Figura 5.3) se muestra el diagrama de secuencia de la inserción de datos de un empleado en la base de datos. El operador indica a la ventana principal (imaginamos que en la aplicación aparecerá una ventana principal que dará paso a las ventanas para operar con empleados y departamentos) que quiere realizar operaciones con empleados, y pide abrir la ventana de empleados. En la ventana de empleados se teclearán los datos, se validarán los datos, y el operador indicará si se graba el registro en la BD. La orden de grabar el registro creará un nuevo objeto *Empleado* en la base de datos con los datos tecleados.

Figura 5.4. Diagrama de secuencia del caso de uso *InsertarEmpleado*.

- Diagramas de estado.** Se utiliza para analizar los cambios de estado de los objetos. Muestra los estados, eventos, transiciones y actividades de los diferentes objetos. En la Figura 5.5 se muestran los estados de un objeto *trabajador*. Los estados son los rectángulos redondeados. Las transiciones son las flechas de un estado a otro. Las condiciones que desencadenan las transiciones están escritas al lado de las flechas. El estado inicial (círculo relleno) se representa para iniciar la acción. El círculo relleno dentro de otro círculo se utiliza para indicar el final de la actividad.

Un trabajador cuando es contratado pasa al estado EN ACTIVO, si pierde el empleo pasa al estado EN EL PARO. Si se jubila pasa al estado JUBILADO.

Figura 5.5. Diagrama de estados de un objeto *Trabajador*.

- Diagramas de actividad.** En UML, un diagrama de actividad se utiliza para mostrar la secuencia de actividades. Los diagramas de actividades muestran el flujo de trabajo desde un punto de inicio hasta el punto final detallando las decisiones que surgen en la progresión de los eventos contenidos en la actividad. En el ejemplo se muestra el diagrama de actividad de la inserción de un empleado en la BD (véase la Figura 5.6)

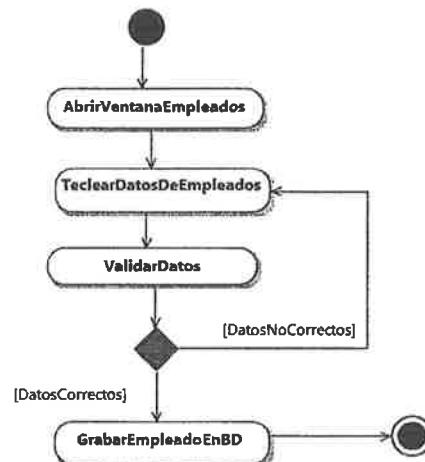


Figura 5.6. Diagrama de actividad de Insertar un empleado a la BD.

- Diagramas de despliegue.** Especifica el hardware físico sobre el que el sistema de software se ejecutará y también especifica cómo el software se despliega en ese hardware. Está compuesto de nodos. Un nodo es una unidad material capaz de recibir y de ejecutar software. La mayoría de nodos son ordenadores. Los vínculos físicos entre nodos también pueden describirse en el diagrama de despliegue, corresponden a las ramas de la red. Los nodos contienen software en su forma física, conocida como *artefact*. Los archivos ejecutables, las bibliotecas compartidas y los scripts son ejemplos de formas físicas de software.

En la Figura 5.7 se muestra el diagrama de despliegue del sistema de gestión de empleados y departamentos. La arquitectura de este sistema está basada en un servidor y dos puestos clientes conectados al servidor mediante enlaces directos que representan la red. El servidor contiene tres ejecutables (*artefactos*) el encargado de la gestión de empleados, el encargado de la gestión de departamentos y el gestor de la BD.

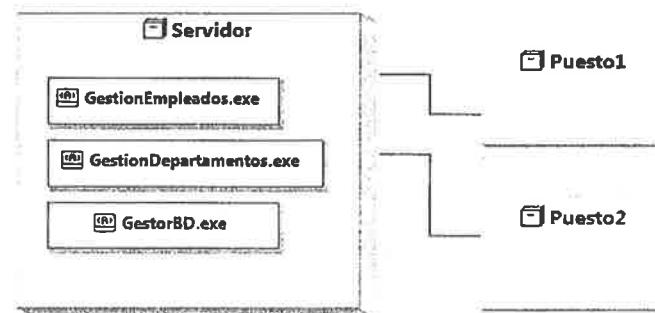


Figura 5.7. Diagrama de despliegue del sistema de gestión de empleados y departamentos.

- Diagramas de paquetes.** Los diagramas de paquetes se usan para reflejar la organización de paquetes y sus elementos. Cuando se utiliza para representar elementos de clase, los diagramas de paquete proporcionan una visualización de los espacios de nombres. El uso más común de los diagramas de paquete es organizar diagramas de casos de uso y diagramas de clases, aunque el uso de diagramas de paquetes no se limita a estos elementos UML.

En el ejemplo de la Figura 5.8, se muestra un diagrama de clases formado por 5 clases, las clases *Departamentos* y *Empleados* agrupadas en el paquete *Datos*, y las clases *VentanaPrincipal*, *VentanaEmpleados* y *VentanaDepartamentos* agrupadas en el paquete *Ventanas*. En la Figura 5.9 se muestra el diagrama de paquetes en el que se relacionan los dos paquetes mediante una asociación de *importación*, las clases del paquete ventanas importarán a las clases del paquete *Datos*.

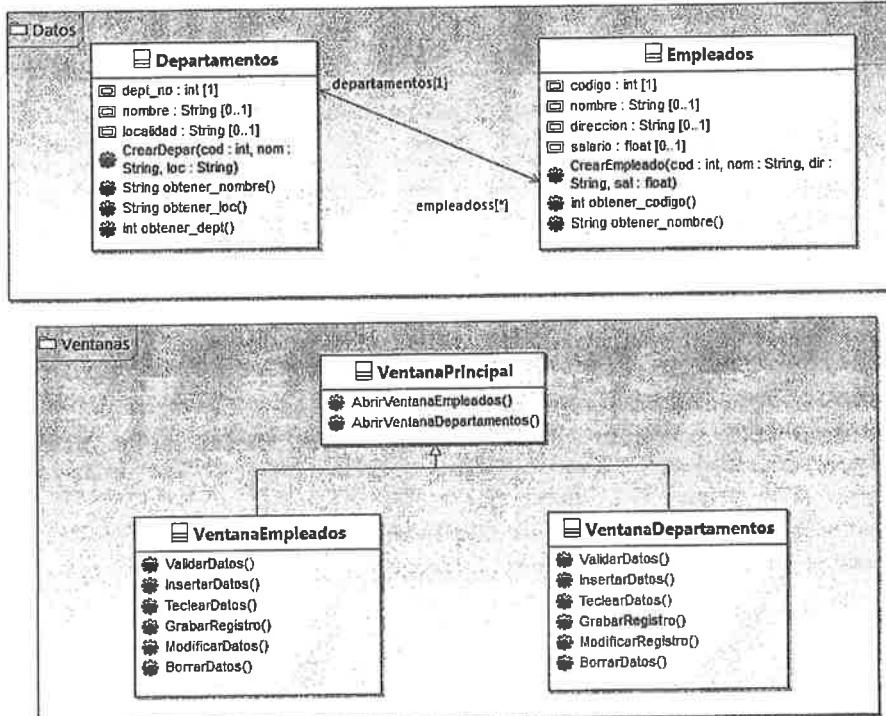


Figura 5.8. Diagrama de clases, agrupadas en paquetes.

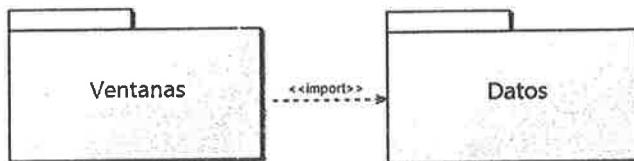


Figura 5.9. Diagrama de paquetes.

5.4. DIAGRAMAS DE CLASES

Un diagrama de clases es un tipo de diagrama de estructuras (estático) que describe la estructura de un sistema mostrando sus clases y las asociaciones entre ellas. Sirve para visualizar las relaciones entre las clases que componen el sistema. Un diagrama de clases está compuesto por los siguientes elementos:

- Clases: atributos, métodos y visibilidad
- Relaciones: asociación, herencia, agregación, composición, realización y dependencia

5.4.1. Clases

Las clases son la unidad básica que encapsula toda la información de un objeto (un objeto es una instancia de una clase). A través de ella podemos modelar el entorno en estudio (un empleado, un departamento, una cuenta corriente, un artículo, etc.). En UML, una clase se representa por un rectángulo que posee tres divisiones (véase la Figura 5.10):

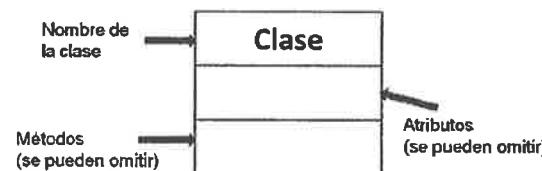


Figura 5.10. Formato de una clase.

- La parte superior: contiene el nombre de la clase
- El medio: contiene los atributos (o variables de instancia) que caracterizan a la clase (pueden ser *private*, *protected*, *package* o *public*).
- La parte inferior: contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto con su entorno (dependiendo de la visibilidad: *private*, *protected*, *package* o *public*).

En la representación de una clase los atributos y métodos pueden omitirse. Por defecto la visibilidad de los atributos debe ser *private* y de los métodos *public*. En la Figura 5.11 se representa la misma clase con distintas herramientas.

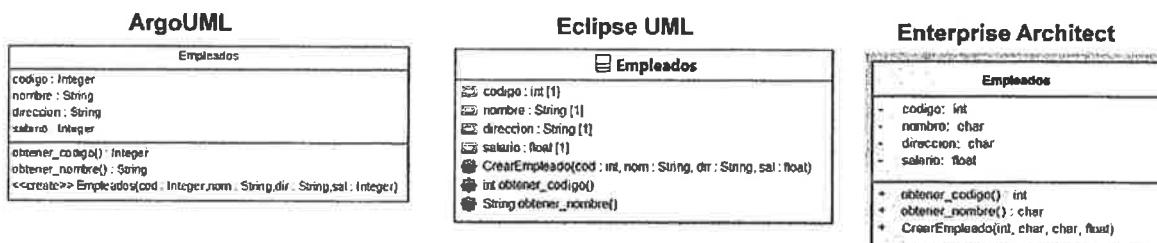


Figura 5.11. Representación de la clase *Empleados* con tres herramientas.

LOS ATRIBUTOS

Un atributo representa alguna propiedad de la clase que se encuentra en todas las instancias de la clase. Los atributos pueden representarse solo mostrando su nombre, o mostrando su nombre y su tipo, e incluso su valor por defecto. Ejemplo de atributos son: Nombre, Salario, Código, Teléfono, etc. Al crear los atributos se indicará el tipo de dato, los tipos básicos UML son Integer, String y Boolean. También se pueden indicar los tipos de cualquier lenguaje de programación.

Al crear el atributo se indicará su visibilidad con el entorno, la visibilidad está estrechamente relacionada con el encapsulamiento. Se distinguen los siguientes tipos:

- **public:** el atributo será público, visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados. También se representan con un signo +.

- **private:** el atributo solo será accesible desde dentro de la clase (solo sus métodos pueden acceder al atributo). También se representan con un signo ~.
- **protected:** el atributo no será accesible desde fuera de la clase, pero si podrá ser accedido por métodos de la clase, además de las subclases que se deriven. También se representa con el carácter almohadilla #.
- **pakage:** el atributo empaquetado es visible en las clases del mismo paquete. Se representa con el carácter tilde ~.

NOTA: Resumen de los tipos de encapsulamiento o visibilidad:

Público (public)	+	elemento no encapsulado visible para todos
Protegido (protected)	#	elemento encapsulado visible en la clase y las subclases de la clase
Privado (private)	-	elemento encapsulado visible solo en la clase
Paquete (pakage)	~	elemento encapsulado visible solo en las clases del mismo paquete

LOS MÉTODOS

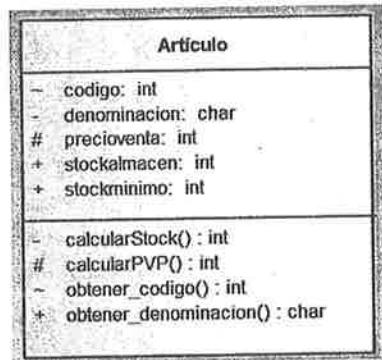
Un método, también llamado operación, es la implementación de un servicio de la clase que muestra un comportamiento común a todos los objetos. Definen la forma de cómo la clase interactúa con su entorno. Igual que los atributos los métodos pueden ser:

- **public:** el método es accesible desde todos lados (desde dentro y fuera de la clase). Se representan con un signo +.
- **private:** sólo los métodos de la clase pueden acceder a él. Se representa con un signo -.
- **protected:** el método puede ser accedido por métodos de la clase además de los métodos de las subclases que se deriven. Se representa con la almohadilla #.
- **pakage:** el método empaquetado o paquete solo es visible en las clases del mismo paquete. Se representa con el carácter tilde ~.

ACTIVIDAD 5.1

Dada la clase *Artículo*, que se muestra en la Figura 5.12, indica cuáles son los métodos y los atributos públicos, privados, protegidos y empaquetados.

Figura 5.12. Representación de una clase con varios tipos de atributos y métodos.



5.4.2. Relaciones

En el mundo real muchos objetos están vinculados o relacionados entre sí, los vínculos se corresponden con asociaciones entre los objetos, por ejemplo el vínculo existente entre un alumno y el curso en el que está matriculado; o el vínculo entre un profesor y el centro en el que trabaja. En UML, estos vínculos se describen mediante asociaciones, de igual modo que los objetos se describen mediante clases.

Las asociaciones tienen un nombre y como ocurre con las clases, este es un reflejo de los elementos de la asociación. También poseen una cardinalidad llamada **multiplicidad** que representa el número de instancias de una clase que se relacionan con las instancias de otra clase, esta multiplicidad es similar a la cardinalidad utilizada en el modelo *Entidad/Relación*. La multiplicidad situada en un extremo de una asociación indica a cuántas instancias de la clase situada en ese mismo extremo está vinculada una instancia de la clase situada en el extremo opuesto.

En uno de los extremos de la asociación, es posible especificar la multiplicidad mínima y la máxima con el fin de indicar el intervalo de valores al que deberá pertenecer siempre la multiplicidad. Para expresar las multiplicidades mínimas y máximas se utiliza la siguiente notación:

Notación	Cardinalidad/Multiplicidad
<i>0..1</i>	Cero o una vez
<i>1</i>	Una y solo una vez
<i>*</i>	De cero a varias veces
<i>1..*</i>	De una a varias veces
<i>M..N</i>	Entre M y N veces
<i>N</i>	N veces

En la Figura 5.13 se muestran ejemplos de dos asociaciones con sus multiplicidades. En el primer ejemplo se muestra la asociación *suministra* entre *Proveedor* y *Artículo*, un proveedor suministra 0 o muchos artículos. Un artículo es suministrado por un proveedor. En el segundo se muestra la asociación *tutoriza* entre *Tutor* y *Curso*, un tutor tutoriza a un curso, y un curso es tutorizado por un tutor, es una asociación 1 a 1.

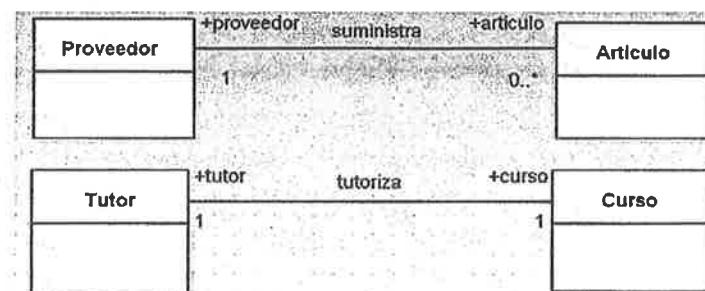


Figura 5.13. Ejemplos de asociaciones.

NOTA: Dependiendo de la herramienta de modelado, las multiplicidades destino mayores que 1, se implementa como un atributo del tipo array o bien un atributo de algún tipo colección o set.

Distinguimos los siguientes tipos de relaciones:

ASOCIACIÓN

Puede ser bidireccional o unidireccional (Figura 5.14), dependiendo de si ambas conocen la existencia la una de la otra o no. Dentro de una relación de asociación, cada clase juega un papel (rol), que se indica en la parte superior o inferior de la línea que conecta a dichas clases. La asociación cuenta con instancias que son los vínculos y que se representan como una línea que conecta dos objetos, y se pone un nombre a la relación.

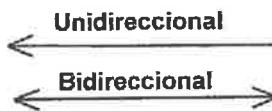


Figura 5.14. Vínculos de una asociación.

Si se convierten a Java dos clases unidas por una asociación bidireccional, cada una de las clases tendrá un objeto o un set de objetos, dependiendo de la multiplicidad entre ellas. En cambio en la asociación unidireccional, la clase destino no sabrá de la existencia de la clase origen, y la clase origen contendrá un objeto o set de objetos de la clase destino.

La **navegabilidad** entre clases nos muestra que es posible pasar desde un objeto de la clase origen a uno o más objetos de la clase destino dependiendo de la multiplicidad. En el caso de la asociación *Unidireccional* la navegabilidad va en un solo sentido, del origen al destino; el origen es navegable al destino, sin embargo, el destino no es navegable al origen.

En la Figura 5.14 se muestran ejemplos de esta asociación:

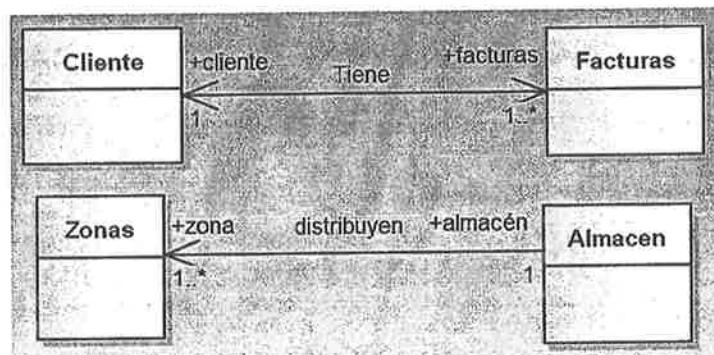


Figura 5.15. Asociaciones bidireccional y unidireccional

- La primera asociación *Tiene* muestra que un cliente tiene muchas facturas, y la factura es de un cliente, como es bidireccional ambas clases conocen su existencia, ambas clases son navegables.
- En la asociación *distribuyen*, un almacén distribuye artículos en varias zonas. La asociación es unidireccional, sólo la clase origen *Almacén* conoce la existencia de la clase destino *Zonas*. *Almacén* a *Zonas* es navegable, en cambio *Zonas* a *Almacén* no es navegable. Véase el código generado, aunque no todas las herramientas generan de la misma manera el código.

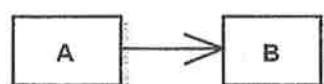
El código java generado con *Eclipse UML* correspondiente a estas asociaciones se muestra a continuación (se han eliminado las líneas de comentario), observa que se utiliza la clase *HashSet* para guardar colecciones de objetos en el caso de las multiplicidades 1..*, * o 0..*.

```

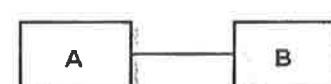
public class Cliente { // 1 cliente tiene muchas facturas (utiliza HashSet)
    public HashSet<Facturas> facturas = new HashSet<Facturas>();
    public Cliente() { // constructor
    }
    public HashSet<Facturas> getFacturas() {
        return this.facturas;
    }
    public void setFacturas(HashSet<Facturas> newFacturas) {
        this.facturas = newFacturas;
    }
}
public class Facturas { // 1 factura es de un cliente
    public Cliente cliente = null;
    public Facturas() { // constructor
    }
    public Cliente getCliente() {
        return this.cliente;
    }
    public void setCliente(Cliente newCliente) {
        this.cliente = newCliente;
    }
}
public class Zonas { // no sabe de la existencia de Almacen
    public Zonas() { // constructor
    }
}
public class Almacen { // 1 almacén distribuye en muchas zonas (HashSet)
    public HashSet<Zonas> zonas = new HashSet<Zonas>();
    public Almacen() { // constructor
    }
    public HashSet<Zonas> getZonass() {
        return this.zonas;
    }
    public void setZonass(HashSet<Zonas> newZonass) {
        this.zonas = newZonass;
    }
}

```

NOTA: A la hora de elaborar diagramas de clase no todas las herramientas utilizan la misma notación para expresar la navegabilidad. En UML2 existen varias notaciones para expresar la navegabilidad, la práctica más estándar es utilizar la siguiente notación:



A a B es navegable
B a A no es navegable



A a B es navegable
B a A es navegable

Una clase puede asociarse consigo misma creando una asociación reflexiva, similar a las relaciones reflexivas del modelo *Entidad/Relación*. Estas asociaciones unen entre si instancias de una misma clase. En la Figura 5.16 se muestran dos asociaciones reflexivas, un alumno (delegado) es delegado de muchos alumnos y un empleado (jefe) es jefe de muchos empleados:

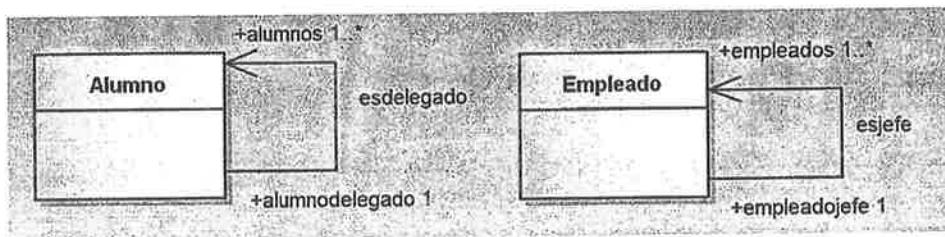


Figura 5.16. Asociaciones reflexivas

CLASE ASOCIACIÓN

Una asociación entre dos clases puede llevar información necesaria para esa asociación, a esto se le llama *clase asociación*, es como una relación M:N con atributos del modelo Entidad/Relación. En este caso esta clase asociación recibe el estatus de clase y sus instancias son elementos de la asociación, al igual que el resto, estas clases pueden estar dotadas de atributos y operaciones y estar vinculadas a otras clases a través de asociaciones.

En la Figura 5.17 se muestra la clase asociación *Compra*, un cliente compra muchos artículos, un artículo es comprado por muchos clientes, y de la compra se necesita saber la fecha de compra y las unidades compradas.

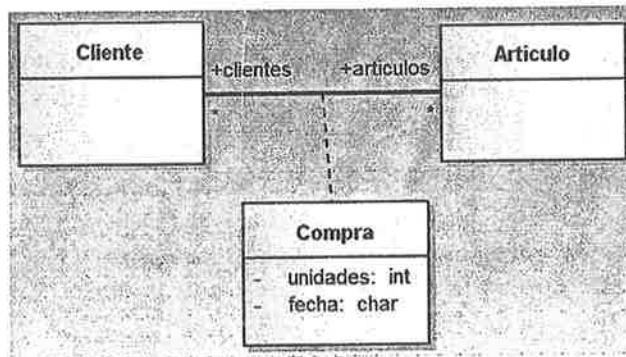


Figura 5.17. Clase asociación

HERENCIA (GENERALIZACIÓN y ESPECIALIZACIÓN)

Las clases con atributos y operaciones comunes se pueden organizar de forma jerárquica, mediante la herencia. La herencia es una abstracción importante para compartir similitudes entre clases, donde todos los atributos y operaciones comunes a varias clases se pueden compartir por medio de la superclase, una clase más general. Las clases más refinadas se conocen como las subclases. La generalización define una relación entre una clase más generalizada, y una o más versiones refinadas de ella. La generalización indica que una clase (clase secundaria o subclase) hereda los atributos y métodos de otra (clase principal o superclase). La superclase generaliza a sus subclases, y las subclases especializan a la superclase.

Para representar esta asociación se utiliza una flecha → el extremo de la flecha apunta a la superclase. En la Figura 5.18 se muestra la herencia entre la clase *Persona* (superclase) y las clases *Empleado* y *Alumno* (subclases). Todas las clases comparten los atributos y métodos de *Persona*.

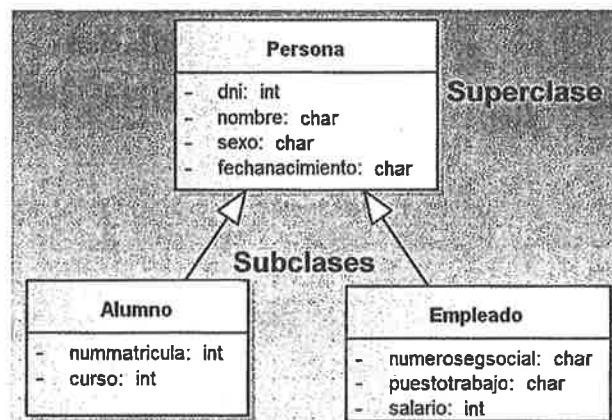


Figura 5.18. Asociación de herencia

El código java generado para estas clases sería el siguiente:

```

public class Persona {
    private int dni;
    private char nombre;
    private char sexo;
    private char fechanacimiento;
    public Persona(){}
}

public class Alumno extends Persona {
    private int nummatrícula;
    private int curso;
    public Alumno(){}
}

public class Empleado extends Persona {
    private char numerosegssocial;
    private char puestotrabajo;
    private int salario;
    public Empleado(){}
}
  
```

COMPOSICIÓN

Un objeto puede estar compuesto por otros objetos, en estos casos nos encontramos ante una asociación entre objetos llamada *Asociación de composición*. Esta asocia un objeto complejo con los objetos que lo constituyen, es decir, sus componentes. Existen dos formas de composición, fuerte o débil. La fuerte se la conoce como *composición* y la débil se conoce como *agregación*.

En la composición fuerte los componentes constituyen una parte del objeto compuesto, y estos no pueden ser compartidos por varios objetos compuestos. Por tanto, la cardinalidad máxima, a nivel del objeto compuesto, es obligatoriamente uno. La supresión del objeto compuesto comporta la supresión de los componentes.

Se representa por una línea con un rombo relleno . En la Figura 5.19 se muestran la asociación de composición entre un ordenador y sus partes. Se considera que el ordenador se compone de una placa base, una o varias memorias, un teclado y uno o varios discos.

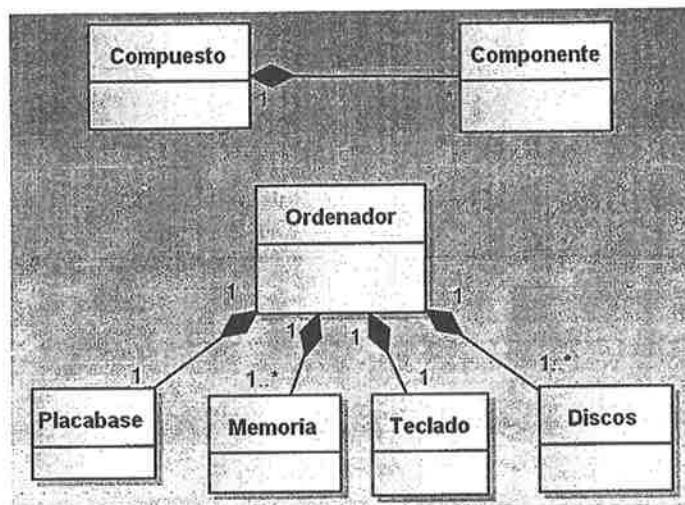


Figura 5.19. Asociación de composición.

El código java generado para la clase *Ordenador*, con los getter y los setter sería el siguiente:

```

public class Ordenador {
    public HashSet<Memoria> memorias = new HashSet<Memoria>();
    public HashSet<Disco> discos = new HashSet<Disco>();
    public Teclado teclados = null;
    public Placa placas = null;
    public Ordenador() { // constructor
    }
    public HashSet<Memoria> getMemorias() {
        return this.memorias;
    }
    public void setMemorias(HashSet<Memoria> newMemorias) {
        this.memorias = newMemorias;
    }
    public HashSet<Disco> getDiscos() {
        return this.discos;
    }
    public void setDiscos(HashSet<Disco> newDiscos) {
        this.discos = newDiscos;
    }
    public Teclado getTeclados() {
        return this.teclados;
    }
    public void setTeclados(Teclado newTeclados) {
        this.teclados = newTeclados;
    }
    public Placa getPlacas() {
        return this.placas;
    }
}
  
```

```

        }
        public void setPlacas(Placa newPlacas) {
            this.placas = newPlacas;
        }
    }
}

```

AGREGACIÓN

La agregación es la composición débil, en este caso los componentes pueden ser compartidos por varios compuestos y la destrucción del compuesto no implica la destrucción de los componentes. La agregación se da con mayor frecuencia que la composición, en las primeras fases de modelado, es posible utilizar solo la agregación y determinar más adelante qué asociaciones de agregación son asociaciones de composición.

Se representa por una línea con un rombo vacío . En la Figura 5.20 se muestran la asociación de agregación entre una clase *Equipo*, y la clase *Jugador*. Un equipo está compuesto por jugadores, sin embargo, el jugador puede jugar también en otros equipos. Si desaparece el equipo, el jugador no desaparece.

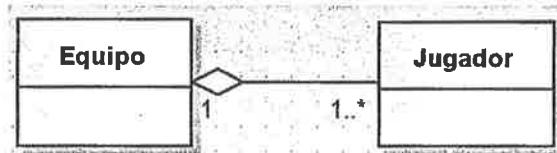


Figura 5.20. Asociación de agregación.

La siguiente tabla resume las diferencias entre la agregación y la composición:

	Agregación	Composición
Símbolo		
Varias asociaciones comparten los componentes	SI	NO
Destrucción de los componentes al destruir el compuesto	NO	SI
Cardinalidad del compuesto	Cualquiera	0..1 o 1

REALIZACIÓN

Una relación de realización es la relación de herencia existente entre una clase interfaz y la subclase que implementa esa interfaz. También se pueden encontrar relaciones de realización entre los casos de uso y las colaboraciones que los realizan.

Esta relación de herencia se representa gráficamente mediante una flecha con línea discontinua  en lugar de una línea completa.

RECUERDA:

Una **interfaz** es una clase totalmente abstracta, es decir, no tiene atributos y todos sus métodos son abstractos y públicos, sin desarrollar. Estas clases no desarrollan ningún método. Gráficamente se representan como una clase con el estereotipo «interface».

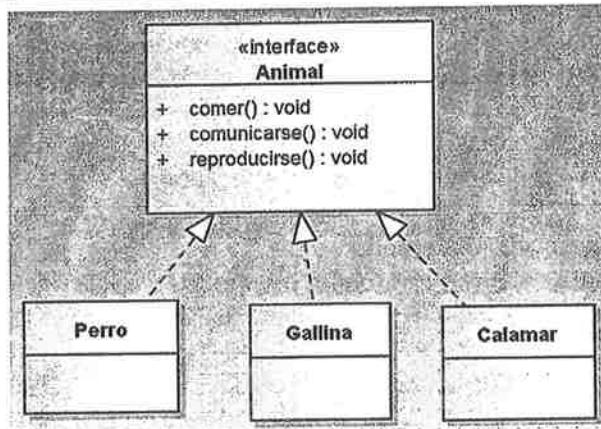


Figura 5.21. Asociación de realización.

En la Figura 5.21 se muestra una asociación de realización entre una clase interfaz *Animal*, y las clases *Perro*, *Gallina* y *Calamar*. Se considera que cualquier animal come, se comunica y se reproduce, sin embargo cada tipo de animal lo hace de manera diferente. Cada subclase implementará los métodos de la interfaz. El código java generado es el siguiente:

<pre>public interface Animal { public void comer(); public void comunicarse(); public void reproducirse(); }</pre>	<pre>public class Calamar implements Animal { public Calamar(){} public void reproducirse(){} public void comunicarse(){} public void comer(){} }</pre>
<pre>public class Perro implements Animal { public Perro(){} public void reproducirse(){} public void comunicarse(){} public void comer(){} }</pre>	<pre>public class Gallina implements Animal { public Gallina(){} public void reproducirse(){} public void comunicarse(){} public void comer(){} }</pre>

DEPENDENCIA

Es una relación que se establece entre dos clases cuando una clase usa a la otra, es decir, que la necesita para su cometido, las instancias de la clase se crean y se emplean cuando se necesitan.

Se representa con una flecha sin relleno discontinua que va desde la clase utilizadora a la clase utilizada (clase de la que depende). Con la dependencia mostramos que un

cambio en la clase utilizada puede afectar al funcionamiento de la clase utilizadora, pero no al contrario.

Ejemplo de relación de dependencia lo podemos tener con una clase *Impresora*, y una clase *Documento*, la impresora imprime documentos, entonces necesita al documento para imprimirlo. O también la relación entre un *Viajero* y su *Equipaje*, el viajero necesita su equipaje para viajar, en la Figura 5.22, se muestran la relación de dependencia entre estas dos clases, el viajero depende de la clase equipaje porque la necesita.

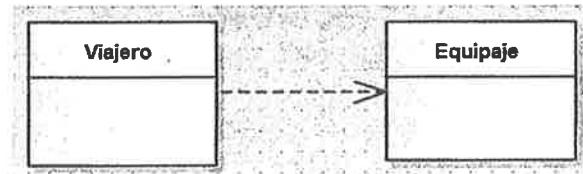


Figura 5.22. Asociaciones de dependencia.

5.4.3. Estereotipos

Los estereotipos son el mecanismo de extensibilidad que más se utiliza dentro del modelado UML, este mecanismo va a permitir definir nuevos elementos de modelado UML basándose en uno existente. Nosotros mismos podemos definir la semántica (significado, interpretación y sentido) del estereotipo. Puede ser aplicado a cualquier elemento de modelado como clases, paquetes, relaciones de herencia, o cualquier otro tipo de relación.

Cada estereotipo puede definir un conjunto de valores etiquetados y restricciones que se aplican al elemento estereotipado. El nombre del estereotipo generalmente se indica escribiéndolo entre comillas francesas « », por ejemplo «*interface*», aunque también se les puede asociar un icono. Existen muchas formas de representar un estereotipo.

SABÍAS QUE:

Otros mecanismos de extensibilidad de UML son:

Las **restricciones**: que amplían la semántica de un elemento al permitir añadir nuevas reglas, se representa como una cadena de texto entre llaves {} que especifica la condición o regla sobre el elemento.

Los **valores etiquetados**: proporcionan una forma de ampliar la especificación de un elemento al permitirnos añadir nueva información en él. Estos valores van entre llaves con la sintaxis {etiqueta=valor ...}.

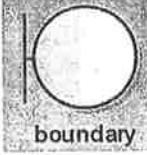
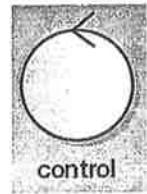
Un **perfil UML** es una colección de estereotipos, valores etiquetados y restricciones. Los perfiles se utilizan para personalizar los diagramas UML para una finalidad específica.

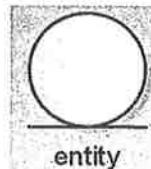
Aunque crear estereotipos no es objetivo de estudio, si conviene saber e identificar los estereotipos que proporcionan las herramientas de modelado, algunos de ellos son los siguientes:

- Estereotipos para los diagramas de clases: *Enumeration*, *Interface*, *DataType*, *Signal*, *Exception*, entre otros.
- Estereotipos para los diagramas de comportamiento: *Entity*, *Control*, y *Boundary* entre otros.

Al hacer el análisis de una aplicación podemos utilizar el método **RUP** (*Rational Unified Process*, el *Proceso Unificado Racional*) es un proceso de desarrollo de software desarrollado por la empresa *Rational Software*, actualmente propiedad de IBM. Junto a UML, constituye la metodología estándar más utilizada para el análisis, diseño, implementación y documentación de sistemas orientados a objetos.

La idea de esta técnica es considerar tres tipos distintos de clases de análisis durante la etapa del análisis del sistema, estos tres tipos distintos de clases se pueden distinguir por los estereotipos de la siguiente tabla:

Estereotipo	Icono	Semántica
«boundary»	 boundary	<p>Representa una clase mediadora entre el sistema y su entorno. Es decir son clases que hacen de interface entre el sistema y los usuarios, entre el sistema con otros sistemas, y el sistema con otros dispositivos externos. Se usan para modelar la interacción entre el sistema y los actores, esta interacción involucra recibir y presentar informaciones y peticiones desde los usuarios y sistemas externos.</p> <p>A los objetos de estas clases se les llaman <i>instancias de clases límite o frontera</i>. Estas clases representarán por ejemplo las ventanas, formularios, impresoras o dispositivos de nuestro sistema.□</p>
«control»	 control	<p>Estas clases son controladoras, sus instancias coordinan el comportamiento del sistema que corresponde a uno o más casos de uso. Los objetos control modelan la funcionalidad del sistema, representan la coordinación, secuencia, gestión de transacciones y control de otros objetos.</p> <p>Se usan para representar cálculos y derivaciones complejas, como la lógica del negocio que no se puede relacionar con ninguna entidad.</p> <p>La dinámica del sistema se modela en una clase controladora, que se encarga de delegar trabajo a otras clases. Por ejemplo, una clase que modele la gestión de artículos de un almacén.</p>

«entity»		<p>Estas clases guardan información sobre el estado interno del sistema, a corto y largo plazo, corresponde al dominio del problema. Expresan la estructura lógica de datos del sistema y están íntimamente relacionadas con el modelo de datos.</p> <p>Están manipuladas por la clase de control y aceptan información de clases límite.</p> <p>Normalmente son clases persistentes (por ejemplo asociadas a tablas de bases de datos relacionales), ejemplo de ellas pueden ser la clase artículos o la clase clientes.</p>
----------	---	---

5.5. HERRAMIENTAS PARA EL DISEÑO DE DIAGRAMAS

Hoy en día existen cientos de herramientas CASE que soportan el lenguaje UML. A la hora de elegir una herramienta hay que tener claro para qué se va a utilizar, y cuál es el objetivo que se propone, porque podemos pensar en utilizar una herramienta para que genere código java, o simplemente utilizar la herramienta para dibujar modelos y añadirlos a nuestra aplicación.

Hemos optado por elegir herramientas de software libre, en esta unidad estudiaremos el uso de varias herramientas que nos permitirán tener una visión global del funcionamiento de este tipo de software.

5.5.1. ArgoUML

ArgoUML es la herramienta de modelado UML de código abierto líder e incluye soporte para todos los diagramas UML 1.4 estándar. Se ejecuta en cualquier plataforma Java y está disponible en diez idiomas. Se descarga de la URL: <http://argouml.tigris.org/> su instalación es sencilla e intuitiva. Alguna de sus características son las siguientes:

- Está escrita en java y disponible en cualquier plataforma soportada por Java.
- Permite crear los siguientes tipos de diagramas: casos de uso, clases, secuencia, colaboración, estado, actividades y despliegue.
- Es compatible con el estándar UML 1.4.
- Tiene soporte para la creación de perfiles y la distribución de los modelos que hacen referencia a los perfiles. Se entrega con perfiles para: Java, C ++ y UML 1.4.
- ArgoUML proporciona la generación de código para Java, C ++, C #, SQL, PHP4 y PHP5.
- Proporciona generación de ficheros PNG, GIF, JPG, SVG, EPS desde los diagramas.
- Dispone de *críticas de diseño*, que analizan el diseño en el que se trabaja y sugieren posibles mejoras. Estas sugerencias van desde la indicación de errores de sintaxis, a los recordatorios para cumplir directrices de estilo.

Una vez instalada la herramienta, al abrirla se muestra la ventana inicial (véase la Figura 5.23), como todas las aplicaciones Windows dispone de la barra de menús y de la barra de herramientas. Además se distinguen cuatro paneles:



Figura 5.23. Ventana inicial de ArgoUML.

- El panel del explorador a la izquierda, donde tenemos una vista en forma de árbol de los diagramas del modelo y los elementos que lo forman.
- A la derecha el panel de diseño del diagrama, donde se colocarán los elementos que lo forman.
- En la parte inferior izquierda se encuentra el panel de críticas y consejos que ayuda a los diseñadores a resolver los problemas que vayan surgiendo durante el diseño.
- Y finalmente en la parte inferior se encuentra el panel de detalles y propiedades. Donde se configurarán las propiedades de los elementos, y además se podrá visualizar el código que se va generando.

CREACIÓN DE DIAGRAMAS DE CLASE

Al abrir ArgoUML, por defecto se abre el diseño para crear diagramas de clases. Desde la barra de botones de diseño (Figura 5.24) se podrán insertar los elementos al diagrama, basta con seleccionar el elemento y pinchar en el diseño. Para crear una asociación, se selecciona el tipo de asociación y se pincha de la clase origen a la clase destino.

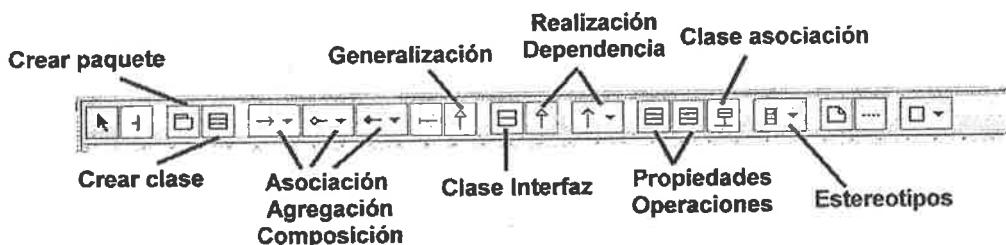


Figura 5.24. Barra de botones para creación de diagramas de clases.

Una forma rápida de crear clases y asociaciones automáticas es hacer clic sobre los iconos de asociación que se muestran al seleccionar la clase y pasar con el ratón, véase la Figura 5.25.

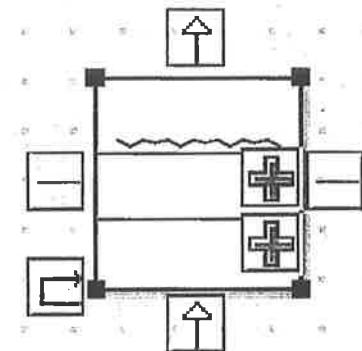


Figura 5.25. Creación de una clase.

Al crear una clase lo primero que hay que escribir es el nombre, y seguidamente añadir los atributos y operaciones, para ello se hace clic sobre los iconos de añadir de la clase o se pulsa a los botones correspondientes desde la barra de diseño.

Al crear la clase en las propiedades podemos indicar la visibilidad de la misma (**Pública**, **Paquete**, **Protegida** o **Privada**) y el tipo de clase o modificadores (véase la Figura 5.26), en los ejemplos a desarrollar no se especificará el tipo de clase. Los modificadores son los siguientes:

- ***isRoot***: para indicar que la clase no tiene antecesores, es una clase raíz.
- ***Is leaf*** (hoja): para indicar que la clase no puede en un futuro ser una especialización.
- ***Is Abstract***: para indicar si la clase es una clase abstracta.
- ***Is Active***: para indicar si la clase es una clase activa. Una clase activa es aquella cuyos objetos poseen uno o más procesos o threads (hilos) y por lo tanto pueden iniciar una actividad de control, son objetos activos. Si los objetos de la clase mantienen los datos pero no inicián actividad se dice que son objetos pasivos.



Figura 5.26. Propiedades de una clase.

Si el diseño que se está creando, es decir, las clases, los nombres, las asociaciones, etc., no cumple con las reglas establecidas para UML 1.4, aparecerán las críticas y las sugerencias en el apartado de críticas (parte inferior izquierda de la pantalla, véase la Figura 5.27). Estas críticas nos ayudarán a corregir fallos en el modelo y hacer que nuestro diseño cumpla las reglas.

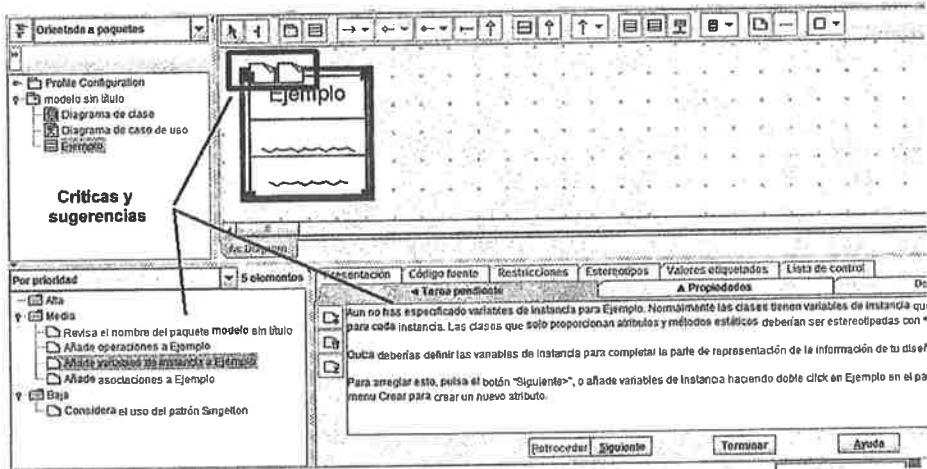


Figura 5.27. Críticas y sugerencias.

Cuando una clase tiene críticas aparecerán unos iconos en la parte superior de la clase que indicarán que se ha detectado una crítica. Si nos fijamos en el apartado de críticas observaremos que se van mostrando todas las críticas surgidas en el diseño hasta el momento, y si se hace doble clic sobre una de ellas aparecerá en el *panel de detalles y propiedades* (a la derecha de las críticas) el detalle de la crítica y la sugerencia para resolverla (véase la Figura 5.27).

Por defecto las críticas se visualizan *Por prioridad*, también se pueden visualizar *Por decisión*, *Por objetivo*, *Por causante*, *Por emisor*, o *Por tipo de conocimiento*, basta con seleccionar el tipo de visualización de la lista desplegable que aparece en la parte superior del panel de críticas.

NOTA:

Cuando estemos realizando un diseño y se desea eliminar elementos incorporados al diseño no vale con eliminarlos pulsando la tecla suprimir, esta los elimina de la vista pero permanecen en el modelo. Hay que eliminarlos pulsando al botón *Borrar del modelo* de la barra de botones de la herramienta.

Ejemplo1: Se trata de realizar un diagrama de clases para representar las relaciones entre empleados y departamentos.

- Consideraremos que un empleado trabaja en un departamento y en el departamento trabajan muchos empleados.
- Datos de los empleados son código, nombre, oficio y salario.
- Datos de los departamentos son código, nombre y localidad.
- Además un empleado puede ser jefe de varios empleados.
- Se necesita crear los métodos para asignar datos a los empleados y departamentos y devolverlos (setter y getter).

Al crear la asociación entre las clases se puede añadir el nombre de la asociación, los nombres de los roles de las clases y la multiplicidad (véase la Figura 5.28). En los nombres de los roles se indica el papel que juegan las instancias de cada una de las clases en la asociación, por ejemplo un objeto departamento desempeñará el rol de *lugardetrabajo*, y un objeto empleado el rol de *trabajador*. No es obligatorio en un diagrama de clases escribir el nombre de la asociación y los roles, si es conveniente indicar la multiplicidad.

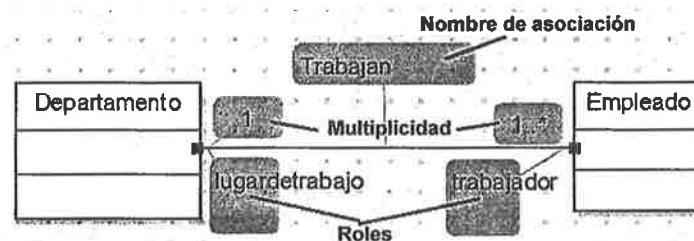


Figura 5.28. Creación de una asociación.

Al seleccionar la asociación se puede acceder al panel de propiedades (véase la Figura 5.29), desde ahí podremos cambiar el nombre de la asociación y acceder a las conexiones entre las clases desde el origen (*Departamento*) al destino (*Empleado*).



Figura 5.29. Propiedades de la asociación.

En la Figura 5.30 se muestran las propiedades de la conexión entre la clase *Departamento* y *Empleado*. Observa que la multiplicidad es 1, y que además es navegable. Para movernos a la conexión destino se pulsa el botón .

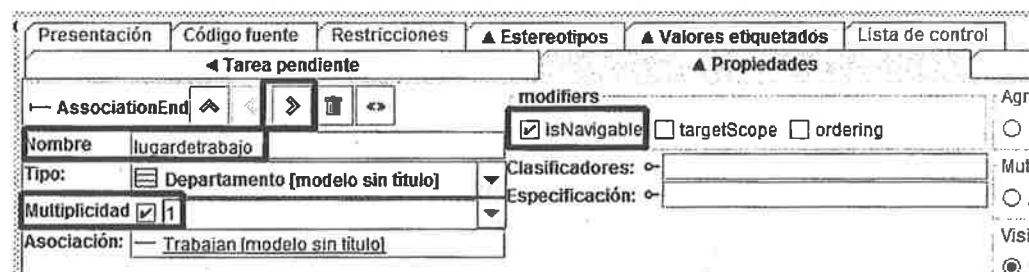


Figura 5.30. Propiedades de una conexión de la asociación.

Para añadir los atributos y las operaciones a las clases se pulsa al botón

También se pueden ir añadiendo desde la ventana de propiedades de la clase, véase la Figura 5.31, pulsando los botones correspondientes

A los atributos se les asigna el tipo de dato, UML 1.4 dispone de los tipos Integer, Boolean y String.



Figura 5.31. Añadir atributos desde las propiedades de la clase

Para añadir parámetros a las operaciones (caso de los setters) y retornar valores en las operaciones (caso de los getters) accedemos a las propiedades de la operación y pulsamos a *return* para crear valores de retorno, y al icono de parámetro para crear los parámetros, véase la Figura 5.32.

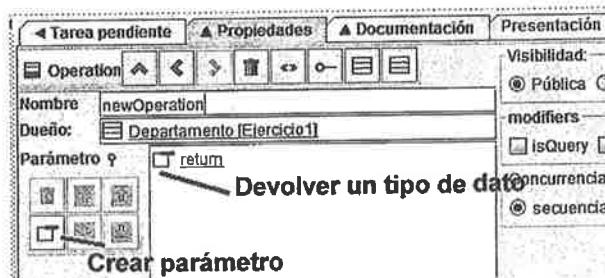


Figura 5.32. Añadir parámetros y retornar datos en las operaciones.

Finalmente para crear la asociación reflexiva de empleado con empleado, es decir un empleado puede ser jefe de muchos empleados pulsamos al botón que aparece en la esquina inferior izquierda cuando se selecciona una clase (véase la Figura 5.25).

Una vez creadas las clases con los atributos y operaciones conviene revisar si hay alguna crítica. Observa que aparece una crítica en cada clase para añadir el constructor (véase la Figura 5.33). Para ello pulsamos *Siguiente* para leer la crítica y la sugerencia de solución, y en la siguiente pantalla se pulsa *Terminar* para aceptar el cambio.

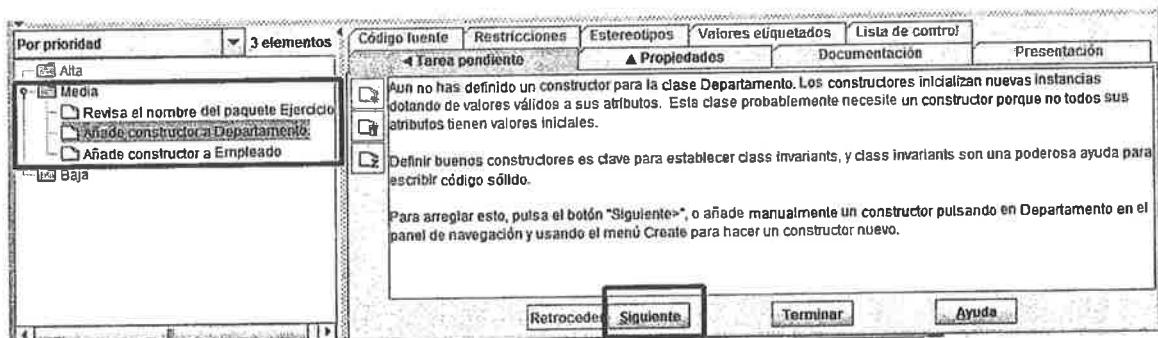


Figura 5.33. Crítica para añadir constructor a la clase.

En la Figura 5.34 se muestra el diagrama de clases del ejercicio 1.

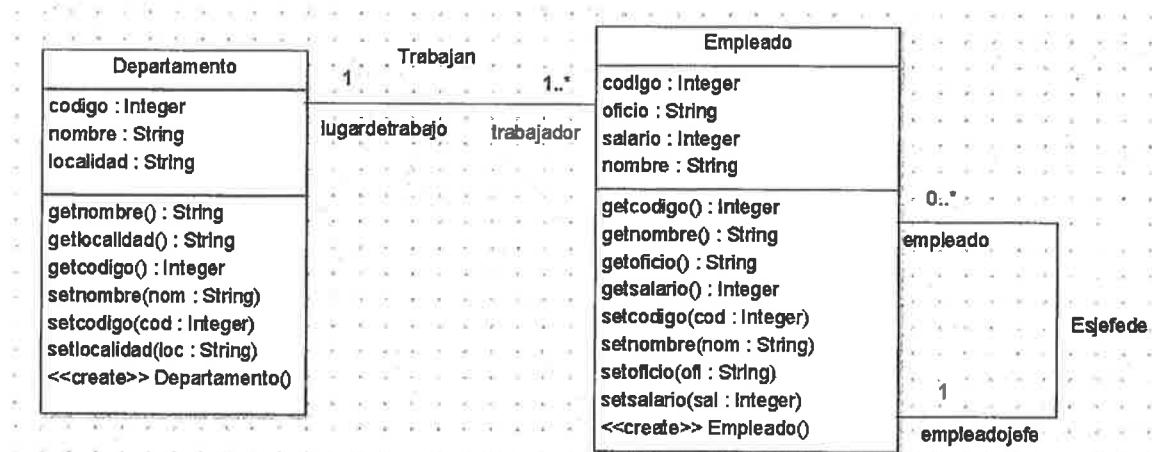


Figura 5.34. Diagrama de clases del ejercicio 1.

Para ver el código fuente generado, se selecciona la clase y en la pestaña código fuente se mostrará el código, hay que asegurarse de seleccionar el lenguaje apropiado. El código generado para este diagrama es el siguiente:

<pre> import java.util.Vector; public class Empleado { public Integer codigo; public String oficio; public Integer salario; public String nombre; public Departamento lugardetrabajo; public Empleado empleadojefe; /** * * @element-type Empleado */ public Vector empleado; public Integer getcodigo() { return null; } public String getnombre() { return null; } public String getoficio() { return null; } public Integer getsalario() { return null; } public void setcodigo(Integer cod) { } } </pre>	<pre> import java.util.Vector; public class Departamento { public Integer codigo; public String nombre; public String localidad; /** * * @element-type Empleado */ public Vector trabajador; public String getnombre() { return null; } public String getlocalidad() { return null; } public Integer getcodigo() { return null; } public void setnombre(String nom) { } public void setcodigo(Integer cod) { } } </pre>
---	--

<pre> public void setnombre(String nom) { } public void setoficio(String ofi) { } public void setsalario(Integer sal) { } public Empleado() { } } </pre>	<pre> } public void setlocalidad(String loc) { } public Departamento() { } } </pre>
--	---

Observa el código generado en cada clase:

- La clase *Empleado* tiene un objeto *Departamento* que es *Lugarde trabajo*, donde se indica el departamento donde trabaja el empleado, y un objeto *Empleado* que es *empleadojefe*, donde se indica el empleado que es su jefe. Además cuenta con el vector *empleado*, que almacenará todos los empleados de los que es jefe, caso de que el empleado sea jefe.
- La clase *Departamento* contiene un vector *trabajador* del tipo *Empleado*, que almacenará todos los empleados que trabajan en ese departamento.

Desde el menú **Generar** de la barra de menús de la herramienta se pueden generar las clases y crear los archivos correspondientes.

Ejemplo2: Se desea realizar el análisis de un sistema de gestión informática de una pequeña empresa formada por empleados y departamentos. Para ello se dispone de una base de datos donde están almacenados los datos de los empleados y los departamentos. Consideramos las clases y relaciones del ejercicio anterior (*Empleado* y *Departamento*).

Requisitos funcionales: el sistema a analizar debe permitir el acceso al operador para mantener la información de la BD, y para generar informes. El operador es un empleado y es el único autorizado a entrar en el sistema y operar. Será el encargado del mantenimiento de los datos de la base de datos. Las operaciones a realizar son las siguientes:

- El mantenimiento de datos de empleados incluye altas, bajas, modificaciones y consultas.
- El mantenimiento de datos de departamentos incluye altas, bajas, modificaciones y consultas.
- Gestión de informes.

Identificación de clases de diseño: consideraremos los tres tipos distintos de clases de análisis para la definición de las clases del ejercicio *Entity*, *Control* y *Boundary*. Así pues se crearán las siguientes clases:

- Clases del tipo *Entidad*, serán las clases persistentes utilizadas para almacenar la información, asociadas a la BD, consideraremos Departamento, Empleado y Operador, que será una especialización de Empleado.

- Clases de **Control**, su objetivo es controlar las operaciones que se hacen con los datos de la BD (altas, bajas, modificaciones, consultas). Consideramos una clase para las operaciones con datos de empleados y otra para las operaciones con departamentos.
- Clases **Interfaz**, clases que se diseñarán para la interactuación del operador con el sistema, es decir las interfaces gráficas. Se podría contar con una ventana de conexión o inicial, con un menú que dé paso al resto de ventanas de la aplicación. Estas ventanas serán una para el mantenimiento de empleados, otra para el mantenimiento de departamentos, y otra para la gestión de informes. Estas tres ventanas heredarán la ventana principal.

Identificación de paquetes. Finalmente en este último paso agruparemos las clases en paquetes, se creará un paquete para las clases *Entidad*, otro para las clases de *Control* y otro para las clases *Interfaz*, la idea es separar los datos, la lógica y la interfaz.

Para crear el diagrama primero se crean los paquetes pulsando al botón  *Crear paquete*, y las clases se van insertando en los paquetes. En la Figura 5.35 se muestra una posible solución para el ejercicio en el que:

- El paquete **datos** contiene las clases que contienen la información *Departamento*, *Empleado* y *Operador*, el *Operador* es una especialización de *Empleado*, por lo que se crea una generalización. No es necesario añadir los atributos y las operaciones, pues no se detallan en el enunciado.
- El paquete **control** contiene las clases que operarán con los datos de la base de datos, clases responsables de la gestión de información: la clase *OperacionesEmple*, operará con los datos de empleados y las operaciones serán altas, bajas, modificaciones y consultas, estas operaciones se indican en el enunciado, por lo que es necesario escribirlas. Atributos no se añaden pues no se indican en el enunciado. La clase *OperacionesDepar* hace lo mismo pero con la clase *Departamento*.
- El paquete **ventanas** contiene las clases que permitirán al actor operador interactuar con el sistema, agrupa las clases que manejan las interfaces gráficas de usuario. Consideramos una *Ventanainicial* en la que se realizará la conexión al sistema y que además contendrá los menús correspondientes para abrir el resto de ventanas. Se indican solo los métodos para conectarse y abrir las ventanas *VentanaEmpleados*, *VentanaDepartamentos* y *VentanaInformes*. Se crea una jerarquía para que estas ventanas hereden la ventana principal y sus menús. Las ventanas *VentanaEmpleados*, *VentanaDepartamentos* y *VentanaInformes* serán las encargadas de llamar a las operaciones del paquete de control para operar con los datos de la base de datos, por lo que podemos añadir algunas operaciones como *insertar_registro*, *modificar_registro*, *borrar_registro*, *crearinformeemple* o *crearinformedepart*, estas acciones se ejecutarían por ejemplo al pulsar un botón. En la Figura 5.35 se muestra el resultado final.

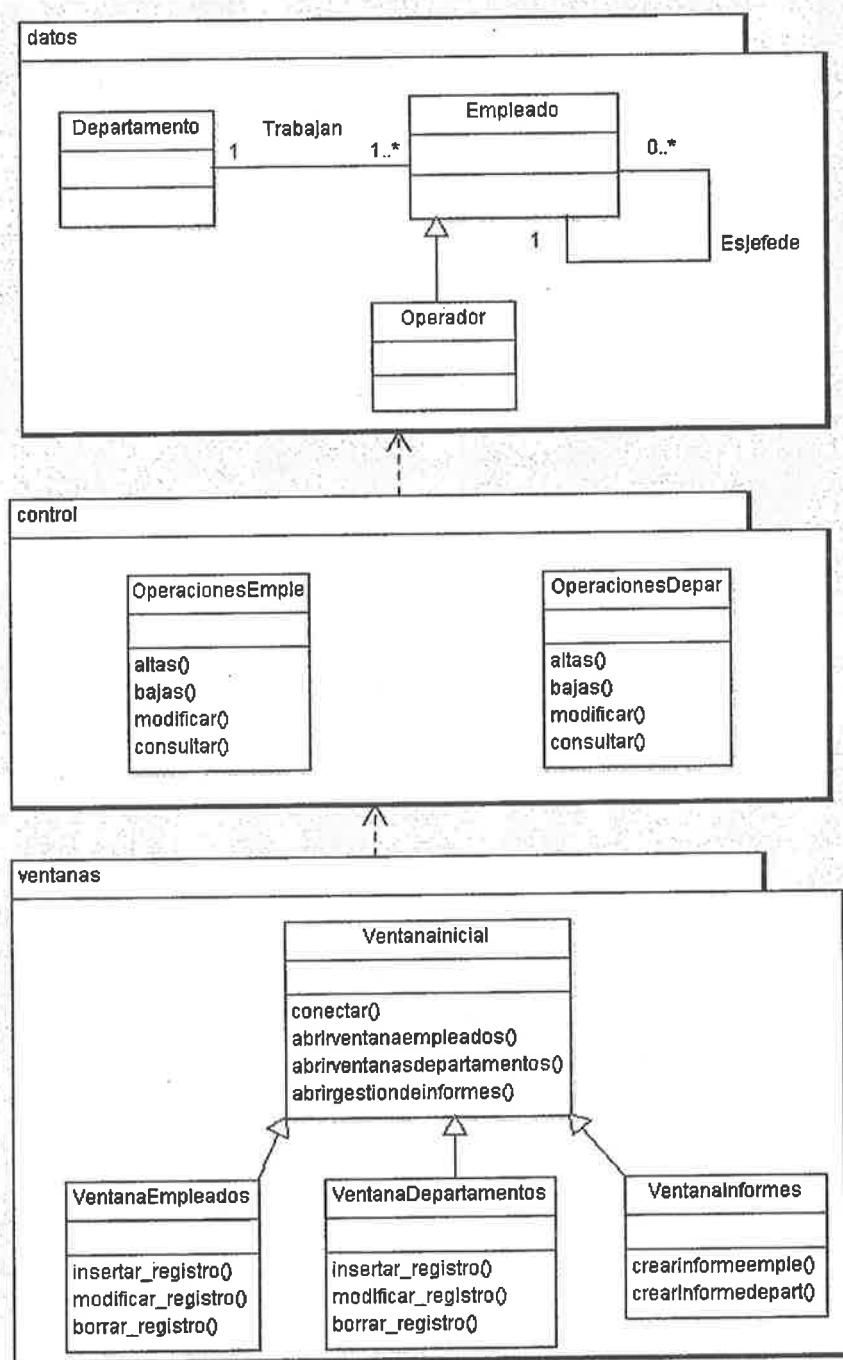


Figura 5.35. Diagrama de clases del ejercicio 2

Los paquetes están relacionados por una *asociación de dependencia*. El paquete *control* depende del paquete *datos* puesto que sus clases acceden a atributos o invocan operaciones de las clases del paquete *datos*. Igualmente el paquete *ventanas* depende del paquete *control*, pues el paquete *ventanas* invoca a operaciones del paquete *control*.

ACTIVIDAD 5.2

Se desea realizar el análisis de un sistema de gestión informática de un pequeño almacén dedicado a vender artículos a clientes. Para ello se dispone de una base de datos donde están almacenados los datos de los clientes, de los artículos, de las facturas y los datos de los detalles de las facturas. Datos de clientes son: código de cliente, nombre, teléfono y dirección. Datos de artículos son: código del artículo, denominación, pvp y stock del almacén. Datos de facturas son: número de factura y la fecha factura.

Hay que tener en cuenta que un cliente tiene muchas facturas. Cada factura está formada por varias líneas de factura o detalles que se identifican por el código de línea. Una factura tendrá varias líneas de factura.

En cada línea de factura se indica también el código de artículo y las unidades vendidas. Un artículo podrá estar en muchas líneas de factura.

Existen 2 tipos de clientes los clientes EMPRESA y los PARTICULARES, de los clientes empresa se necesita saber el CIF, y la razón social. De los particulares la cantidad de descuento.

El sistema a diseñar mostrará una ventana inicial con una serie de menús que nos permitirán abrir las ventanas para operar con los distintos datos de la base de datos. Esta ventana debe incluir los métodos para abrir las ventanas de mantenimiento de datos de clientes, de artículos y de facturas. Desde estas ventanas se podrán realizar las siguientes operaciones:

- Mantenimiento de datos de clientes que incluye altas, bajas, modificaciones, consultas y generación de informes.
- Mantenimiento de datos de artículos que incluye altas, bajas, modificaciones, consultas y generación de informes.
- Mantenimiento de datos de facturas que incluye altas, bajas, modificaciones, consultas y generación de informes.

SE PIDE:

Identificar las clases de diseño y sus relaciones: clases del tipo Entidad, de Control e Interfaz.

Identificar los atributos y las operaciones, incluir solo los atributos y operaciones que se especifican en el enunciado.

Identificar los paquetes: un paquete para las clases Entidad, otro para las clases de Control y otro para las clases Interfaz.

Realizar el diagrama de clases y de paquetes.

5.5.2. UML con Eclipse

Para poder realizar diagramas UML utilizando Eclipse instalaremos el plugin UML2. Para ello desde Eclipse, se abre el menú *Help* y se selecciona *Eclipse Market Place*. Desde ahí tecleamos *UML2* en *Find* y se pulsa el botón *Go*, para que busque el plugin. Una vez localizados los plugins se muestra una lista de ellos, hemos de elegir el plugin que se corresponda con la versión de eclipse con la que se trabaje, en nuestro caso se elige *UML Designer (Eclipse Kepler versión)*, véase la Figura 5.36. Se pulsa a *Install*, se aceptan las condiciones y comienza la instalación.

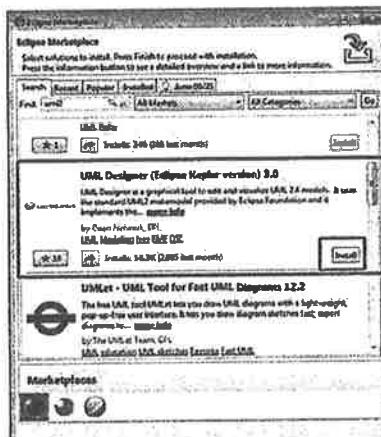


Figura 5.36. Plugin UML para Eclipse Kepler.

Una vez terminada la instalación y reiniciado, el eclipse observa que se han creado nuevas vistas, la que interesa ahora es la vista *Modeling*.

Para crear un proyecto abrimos el menú *File/New/UML Project*, se teclea el nombre, se aceptan las opciones por defecto y se crea el proyecto. En la Figura 5.37 se observa el proyecto creado.

Para crear un diagrama UML nos posicionamos sobre el proyecto, botón derecho del ratón y se elige *Create Representation*, y desde ahí se selecciona el tipo de diagrama a crear (véase la Figura 5.38).

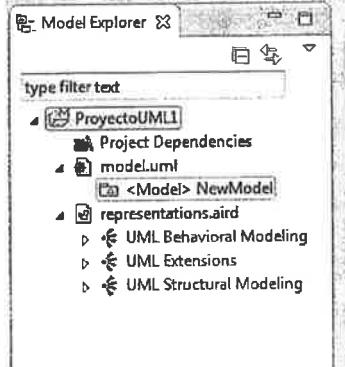


Figura 5.37. Vista de proyecto UML desde la vista *Modeling*

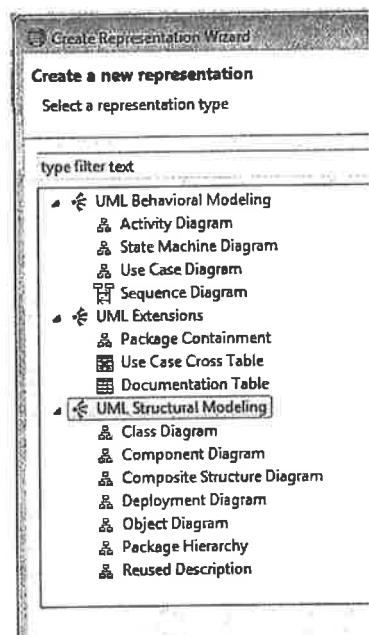


Figura 5.38. Tipos de diagrama a crear

Dentro de los tipos de diagrama nos fijamos en los siguientes:

- Diagramas de comportamiento (*UML Behavioral Modeling*) para crear los diagramas de actividad, de estado, de casos de uso y de secuencia.

- Diagramas de estructuras o estructurales (*UML Structural Modeling*), entre los que se encuentran el diagrama de clases, diagrama de componentes, diagrama de estructura compuesta, diagrama de implementación o despliegue, diagrama de objetos o diagrama de paquetes.

CREACIÓN DE DIAGRAMAS DE CLASE

Para crear un diagrama de clases se pulsa el botón derecho del ratón sobre el proyecto, se elige *Create Representation*, y dentro de *UML Structural Modeling* se selecciona *Class Diagram*. Se pulsa siguiente, se selecciona en qué modelo se creará el diagrama, se teclea un nombre y se abre la vista de diseño. En la Figura 5.39 se observan las distintas zonas de la pantalla: el explorador del modelo o la vista del proyecto, la vista de diseño donde se irán añadiendo los elementos del diagrama, la pestaña de propiedades para cambiar las características de los elementos y la paleta de elementos que se pueden añadir en el diagrama.



Figura 5.39. Ventana de trabajo de diseño UML

Desde la paleta se seleccionará el elemento a insertar al diagrama, clase, paquete, atributo, operación o tipo de relación, basta con marcar y arrastrar (véase la Figura 5.40). También se pueden seleccionar los elementos desde la barra flotante de botones.

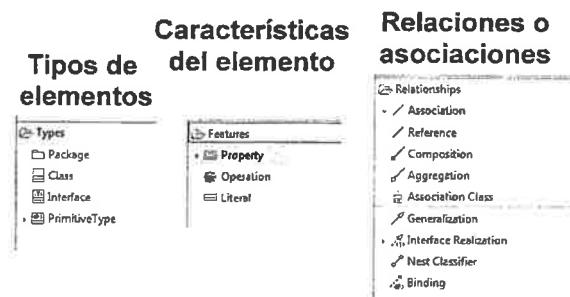


Figura 5.40. Componentes de la paleta de diseño.

Ejemplo1: Se trata de realizar un diagrama de clases para representar las relaciones entre empresa, empleados y clientes. Utilizaremos asociaciones de composición y generalización en el diagrama. Los requisitos son los siguientes:

- La empresa se compone de clientes y de empleados. Utilizaremos para estas relaciones asociaciones de composición.
- Datos de la empresa son: CIF, razón social, dirección y teléfono.
- Datos de clientes son: código de cliente, comisión, nombre de empresa, teléfono, empresa para la que trabaja y comisión.
- Datos de empleados son: código de empleado, nombre, fecha de nacimiento, teléfono, fecha de alta en la empresa y salario.
- Como los atributos nombre, fecha de nacimiento y teléfono son comunes para clientes y empleados se creará una clase persona para esos atributos, y las clases cliente y empleado heredará de la clase persona.
- Un empleado puede ser director de varios empleados. De este director se necesita saber también la categoría y la fecha de alta como director. El director heredará de empleado y además tendrá una asociación [1..*] con empleado.
- Para todas las clases crearemos dos métodos, uno para asignar datos a los atributos (como si fuese el constructor), este método tendrá tantos parámetros como atributos, y otro que se llame obtener que devolverá un objeto de la misma clase.
- El diagrama a crear debe ser como el que se muestra en la Figura 5.41.

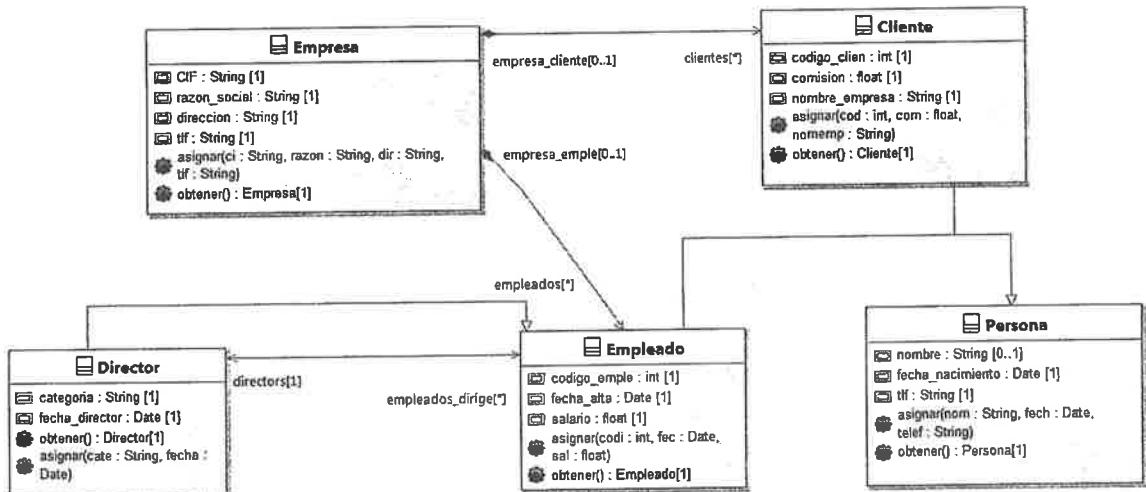


Figura 5.41. Diagrama de clases del Ejemplo1.

Al crear una clase, si se hace doble clic sobre la misma se muestra la ventana de creación de la clase, desde ella podremos teclear el nombre, elegir la visibilidad (*public*, *private*, *protected* y *package*) y marcar el tipo de clase (*Is Abstract*, si va a ser una clase abstracta. *Is leaf* -hoja- si se considera que se no puede en un futuro ser una especialización. *Is Active* para indicar si la clase es una clase activa). También se puede asignar un caso de uso a la clase.

Si se desean añadir atributos y operaciones a la clase, podemos movernos por las pestañas *Attributes* u *Operations* y utilizaremos los botones correspondientes para añadir nuevos elementos, borrarlos o cambiarlos de posición. Véase la Figura 5.42.

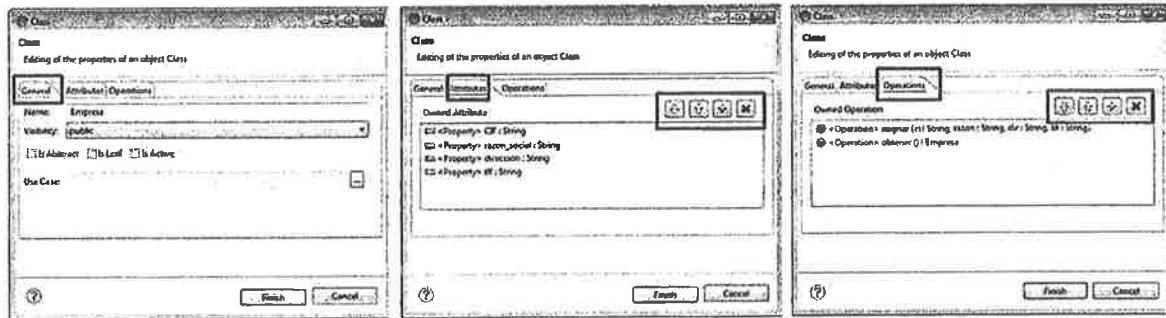


Figura 5.42. Ventana de edición de la clase, de los atributos y de las operaciones.

A la hora de añadir los atributos se indicará el nombre, la visibilidad, las propiedades del atributo y el tipo. Para los ejemplos se dejan las opciones que aparecen por defecto. En la siguiente tabla se muestran las propiedades de los atributos:

Propiedad	Valor por defecto	Descripción
Is Read Only	False	Si es true, el atributo es de solo lectura y no se puede cambiar su valor
Is Static	False	Si es true, las instancias de este tipo comparten el mismo valor para este atributo
Is Leaf	False	Si es true, no está diseñado para permitir que este atributo se redefina en los tipos derivados
Is Derived	False	Si es true, este atributo se calcula a partir de otros atributos, es un atributo calculado
Is Ordered	False	Si es true, la colección forma una lista secuencial y ordenada
Es Unique	False	Si es true, no hay valores duplicados en la colección

Al asignar un tipo de dato para los atributos, se abre la ventana para elegir el tipo de dato. Para los ejercicios que se van a realizar nos centraremos solo en los tipos de datos primitivos: int, float, String o Date, así pues se seleccionará de la lista *PrimitiveType <Primitive Type>* véase la Figura 5.43. Aparecen los tipos repetidos, se elige uno de ellos.

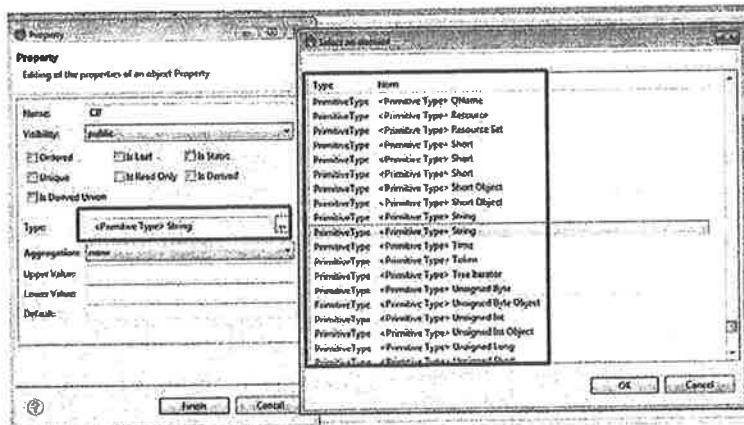


Figura 5.43. Selección del tipo de dato.

NOTA: Si no aparece ningún tipo de dato hay que importarlos, para ello se pulsa el botón derecho del ratón en la ventana de diseño y se elige *Import primitive types* del menú contextual. Se podrá seleccionar *UML Primitive types*, *Java Primitive types*, *Encore Primitive types* o *XML Primitive types*. En nuestro caso se elige *Java Primitive types*.

Al crear una operación se teclea el nombre, la visibilidad y se marca el tipo (se dejan las opciones por defecto a no ser que se indique otra cosa en el enunciado), y a continuación se añaden los parámetros si los tiene. Para ello se pulsa a la pestaña **Parameters** y al botón *añadir* (+). En la nueva ventana que aparece se escribe el nombre del parámetro, se selecciona el tipo de dato y se marca el tipo de parámetro (*in* para indicar que es un parámetro de entrada, *inout* el parámetro será de entrada-salida, *out* de salida, y *return* si la operación devuelve un valor de retorno), véase la Figura 5.44.

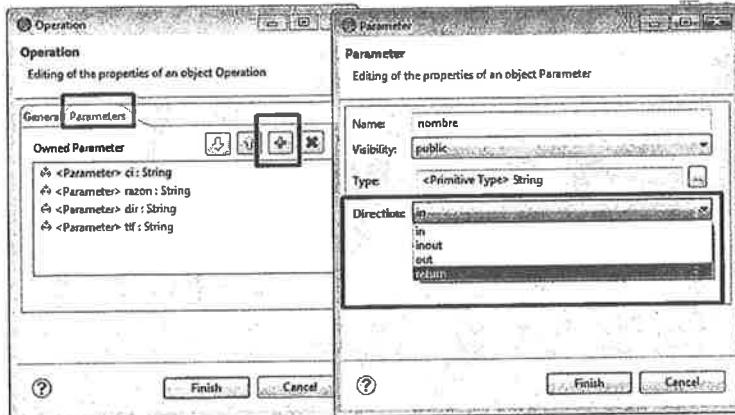


Figura 5.44. Añadir parámetros a una operación.

También se pueden cambiar las propiedades de la clase desde la pestaña de propiedades, si esta no se muestra se abre desde el menú *Window>Show view/Properties*. En la Figura 5.45 se muestra la ventana de propiedades. Desde aquí se podrán ver, añadir y modificar los atributos, las operaciones, las relaciones de la clase, la apariencia y la semántica. Cambiaremos las propiedades de la semántica, sobre todo, para indicar la multiplicidad de la clase cuando tiene asociaciones.

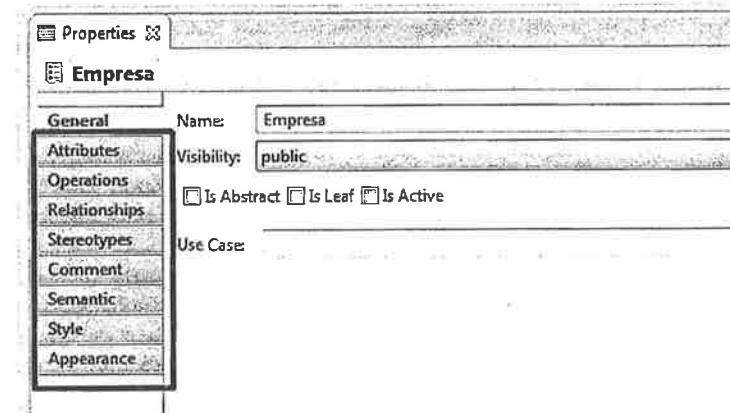


Figura 5.45. Pestaña de propiedades de una clase.

Para elaborar el ejercicio primero se crean las clases con los atributos y operaciones siguiendo lo indicado en el apartado. Una vez creadas las clases se crean las relaciones.

Entre *Empresa* y *Cliente* se crea una asociación de composición que va del compuesto (*Empresa*) al componente (*Cliente*), se selecciona la asociación de la paleta y se pincha de la clase *Empresa* a la clase *Cliente*, desde el nombre de clase origen al nombre de clase destino.

Al crear las asociaciones se generan automáticamente sus nombres, observa que llevan el nombre de la clase con la que se asocian seguida de una s. Observa también las multiplicidades de la asociación que aparecen entre [corchetes]. Véase la Figura 5.46.

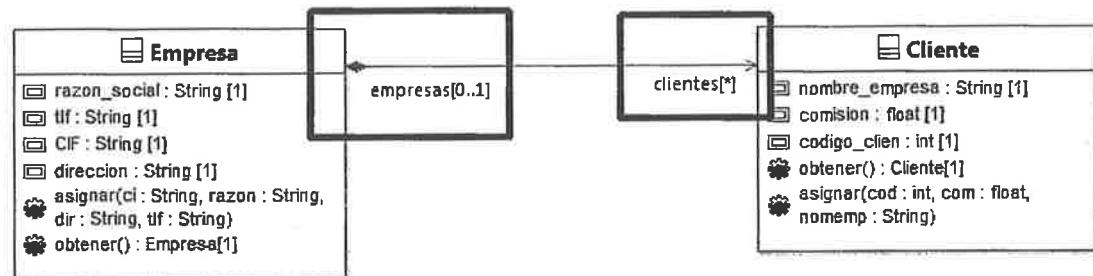


Figura 5.46. Asociación de composición Empresa-Cliente.

La multiplicidad que se ha creado del lado de la clase *Empresa* es [0..1], y del lado de la clase *Cliente* [*], esto quiere decir que 1 empresa tiene muchos clientes [*]. Esto se traduce en java a que la clase *Empresa* tendrá un *HashSet* de objetos *Cliente* llamado *clientes*, y la clase *Cliente* tendrá un objeto de la clase *Empresa* llamado *empresas* (aunque eclipse no crea esta segunda relación para asociaciones de este tipo).

Se hace lo mismo entre las clases *Empresa* y *Empleado*. Los nombres de las asociaciones se pueden cambiar para hacer más legible el diagrama. Basta con hacer dos veces clic sobre el nombre en el diagrama, o también hacer doble clic en la asociación para que aparezca la ventana de propiedades de la asociación y abrir los elementos que componen la asociación. En la Figura 5.47 se muestra la ventana de la asociación y las propiedades del extremo *empresa_cliente*. El extremo de la clase *Empresa* se llama *empresa_cliente* y el de la clase *Cliente* se llama *clientes*, véase también el diagrama de la Figura 5.41.

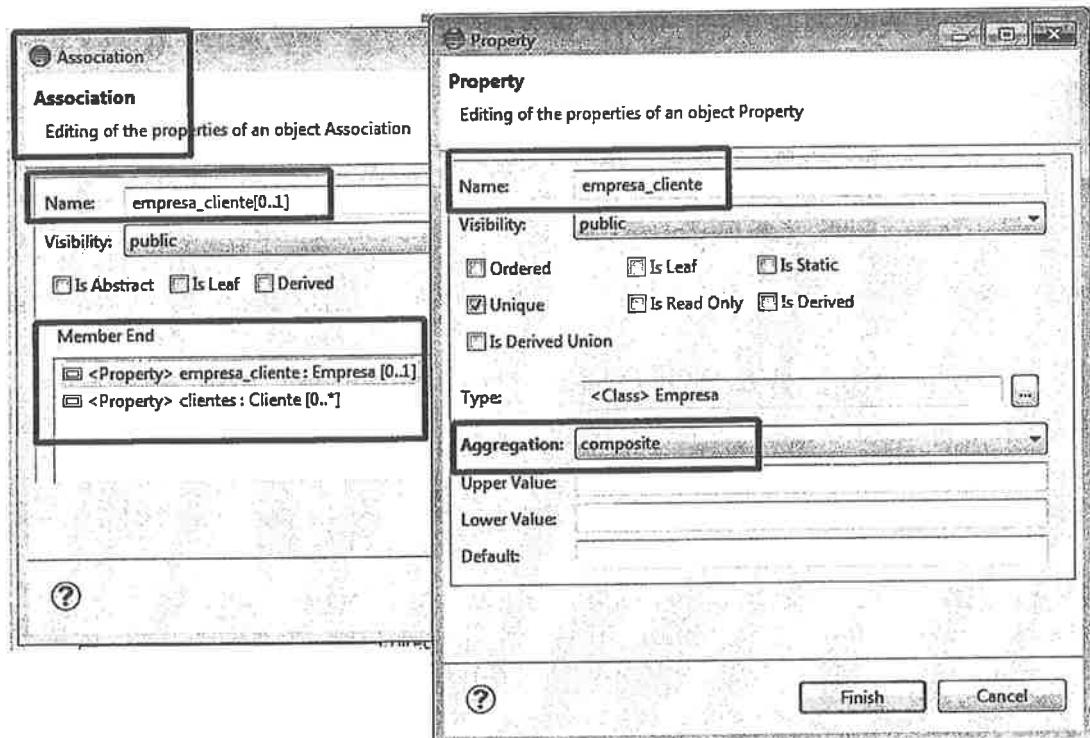


Figura 5.47. Ventana de propiedades de la asociación de composición *Empresa-Cliente*.

NOTA: Si una clase va a tener varias asociaciones con otra clase es conveniente cambiar los nombres de las asociaciones, pues esos nombres se utilizan al generar el código java a partir del modelo, y es necesario que no sean iguales para evitar errores de compilación.

Para crear las asociaciones de generalización entre *Persona-Cliente*, y *Persona-Empleado*, se selecciona la asociación de generalización de la paleta de diseño y se marca desde la clase que la hereda a la clase heredada, es decir, de *Cliente* a *Persona*, y de *Empleado* a *Persona*. En java la generalización crea una herencia, así la clase *Cliente* y *Empleado* serán *extends* de *Persona*.

Se hace lo mismo entre las clases *Director-Empleado*, la clase *Director* hereda a la clase *Empleado*, con lo que se selecciona la relación de generalización de la paleta y se arrastra de la clase *Director* a *Empleado*. En el caso de este tipo de asociaciones no se añaden nombres, en la Figura 5.48 se muestran estas asociaciones.

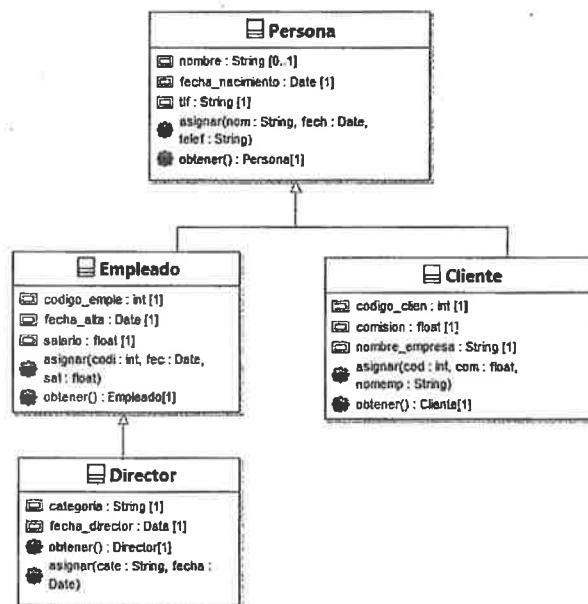


Figura 5.48. Asociaciones de generalización del ejercicio.

Finalmente se crea la asociación entre *Director* y *Empleado*. Es una asociación normal con una multiplicidad de [1..*], es decir, un *Director* dirige a muchos empleados, y un *Empleado* es dirigido por un director. Se selecciona el símbolo asociación de la paleta de diseño y se arrastra de *Director* a *Empleado*, en este caso da igual el orden, por defecto, la asociación que se crea es muchos-muchos, es decir [*] en ambos sentidos. Es necesario cambiar la multiplicidad de la asociación para dejarla en [1..*].

Para cambiar la multiplicidad de una asociación se selecciona la asociación y se abre la pestaña de *Propiedades*, si no está visible se abre el menú *Window>Show view/Properties*.

Al lado de la clase *Director* hay que poner la multiplicidad de [1], y al lado de la clase *Empleado* la de [*]. Dentro de la pestaña de propiedades destacamos los siguientes apartados:

- **General**, en este apartado se puede poner el nombre a la asociación, y se pueden ver las clases que asocia.
- **Semantic**, donde se indica la semántica de la asociación, es decir, el significado y sentido de la asociación. En este apartado es donde se cambia la multiplicidad. Está formado por la parte donde se indican las propiedades de la asociación <*Association*>. Y las partes donde se indican las propiedades <*Property*> del papel que juegan las clases que forman la asociación, en nuestro caso *Empleado* y *Director*, véase la Figura 5.49:

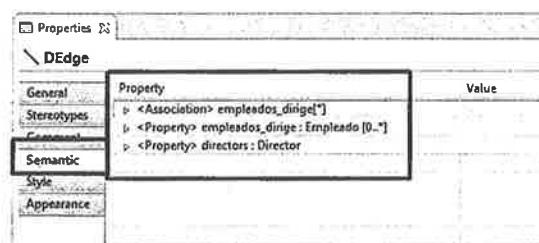


Figura 5.49. Semántica de la asociación *empleados_dirige*.

Para cambiar la multiplicidad se abren las propiedades de la asociación, y en la propiedad *Lower* se indica el valor mínimo de la participación de la clase en la asociación, y en *Upper* el valor máximo. Así en el ejercicio la clase *Empleado* en esta asociación tiene de valor mínimo [0] (también se podía haber considerado 1) y de valor máximo [*], ya que un director dirige a 0 o muchos empleados. Y la clase *Director* tiene una multiplicidad de valor mínimo [1] y de valor máximo [1] ya que es un director el que dirige. Véase la Figura 5.50 y la Figura 5.51.

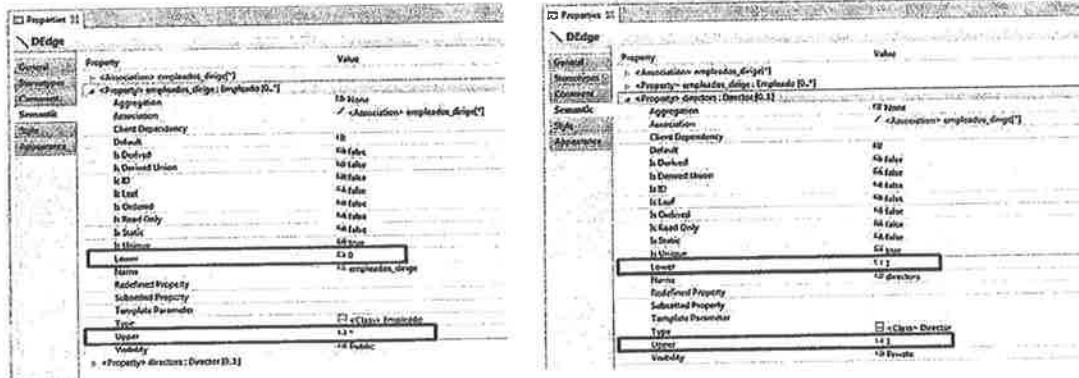


Figura 5.50. Multiplicidad de la asociación entre *Empleado-Director*.

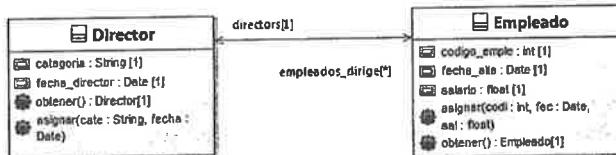


Figura 5.51. Asociación entre *Empleado-Director*

Si el modelo se convierte a Java con Eclipse, la clase *Director* tendrá un *HashSet* de objetos *Empleado* llamado *empleados_dirige*, y la clase *Empleado* tendrá un objeto de la clase *Director* llamado *directors*.

Ejemplo2: Se desea realizar el análisis de un sistema de gestión informática de una pequeña agencia de viajes que oferta viajes a sus clientes. El sistema debe proporcionar una ventana inicial con una serie de menús que abrirán paso al resto de ventanas de la aplicación que permitirán realizar las siguientes acciones:

- La gestión de las reservas de viajes para realizar reservas, modificar reservas, consultar reservas, borrar reservas y generar e imprimir facturas.
- El mantenimiento de datos de clientes, para mantener actualizados los datos se realizarán operaciones de consulta, altas, bajas y modificaciones de datos de clientes, y además debe permitir generar listados de clientes.
- Mantenimiento de datos de viajes, para mantener actualizados los datos se realizarán operaciones de consulta, altas, bajas, modificaciones e informes de viajes.

Disponemos de una base de datos donde están almacenados los datos de los clientes, los viajes, las reservas, las fechas de viaje, los datos son los siguientes:

- Datos de clientes son: código-cliente, nombre, tlf y dirección.
- Datos de viajes son: código, nombre, plazas y precio.
- Datos de las reservas son: número de reserva y estado de la reserva.
- Un cliente puede realizar muchas reservas, y una reserva es de un cliente.
- Igualmente, de un viaje se pueden realizar muchas reservas, y una reserva pertenecerá a un viaje.
- Los viajes se ofertan en varias fechas de viaje, de estas fechas se necesita saber la fecha de comienzo y la fecha de fin. Estas fechas pueden ser compartidas por varios viajes.
- También se cuenta con la información de un catálogo de viajes, datos del catálogo son: código, destino, procedencia, temporada, precio. Los viajes se crean a partir del catálogo.

Identificación de clases de diseño

Para la solución consideramos los tres tipos distintos de clases de análisis: *Entity*, *Control* y *Boundary*, y así identificamos las siguientes clases:

- Clases del tipo *Entidad*, serán las clases persistentes asociadas a la BD, consideramos Viaje, Cliente, Reserva, Catalogo y Fechas_Viaje.
- Clases de *Control*, que formarán la lógica de la aplicación, para controlar las operaciones que se hacen con los datos de la BD (altas, bajas, modificaciones, consultas). Consideramos las siguientes clases: OpeViajes, OpeCliente, OpeCatalogo, OpeReservas y OpeFechas. Estas clases contendrán operaciones como dar de alta registros, modificar, buscar o borrar.
- Clases *Interfaz*, lo forman las ventanas de la aplicación. Consideramos una ventana inicial la *VentanaPrincipal* con un menú que abrirá el resto de ventanas de la aplicación. Estas ventanas serán para el mantenimiento de viajes (*VViajes*), para la gestión de reservas (*VReservas*), y para la gestión de clientes (*VClientes*). Estas clases tendrán operaciones como insertar registro, modificar, borrar, listar, consultar. Podrían ser botones que ejecutan operaciones o que abren otras ventanas.

Identificación de paquetes

Finalmente en este último paso agruparemos las clases en paquetes, se creará un paquete para las clases *Entidad*, otro para las clases de *Control* y otro para las clases *Interfaz*, con el objetivo de separar los datos, la lógica y la interfaz.

Para elaborar este diagrama primero se crearán los paquetes y a continuación los diagramas de clase de cada paquete. En la Figura 5.52 se muestran cómo deben quedar los paquetes con sus clases, y en la 5.53 el diagrama de paquetes. El paquete *Interfaz* importa el paquete *Control*, y este el paquete *Entidad* (También se podría haber añadido una relación de dependencia entre los paquetes: *Interfaz* depende de *Control* y *Control* depende de *Entidad*).

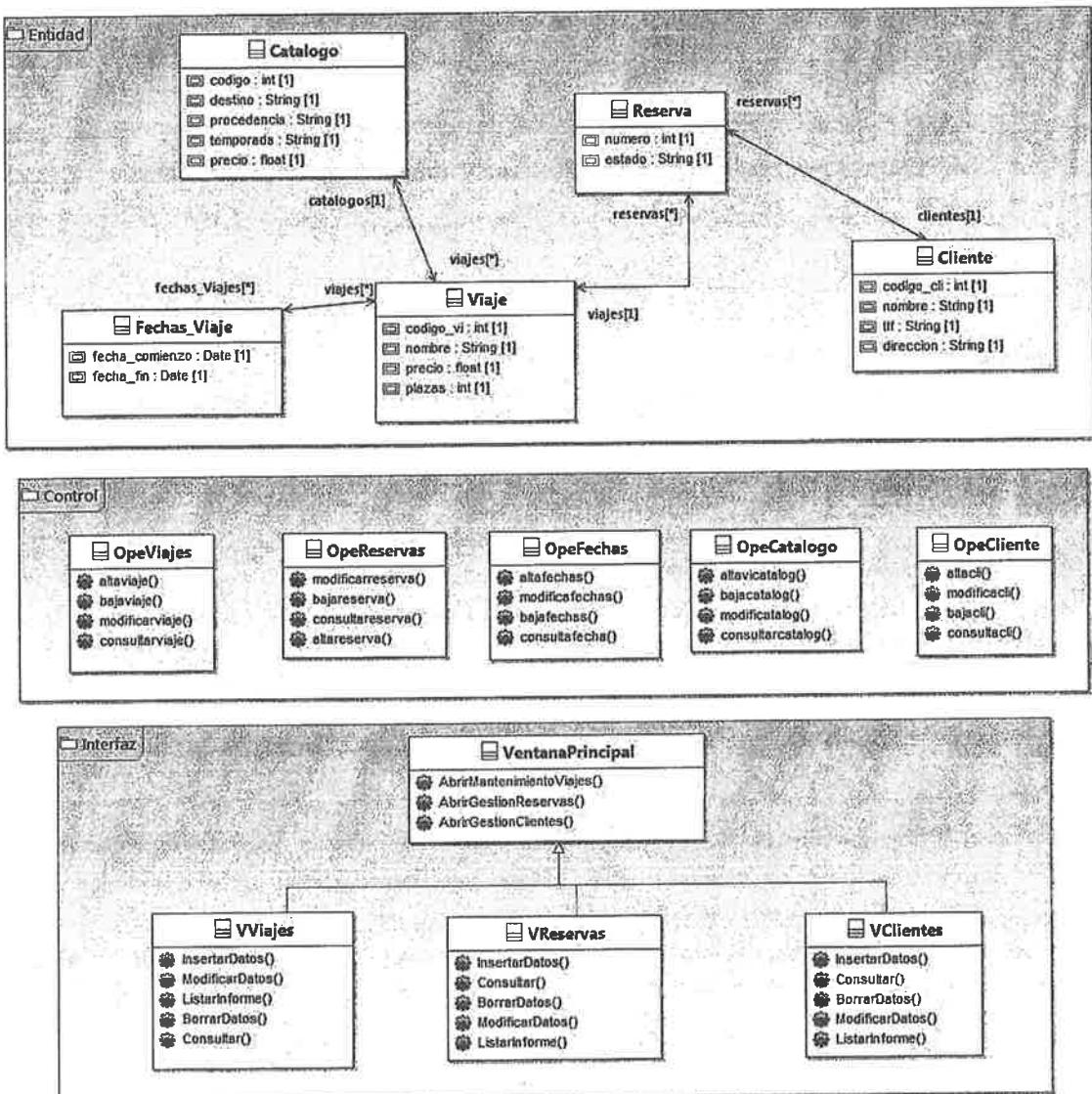


Figura 5.52. Diagrama de clases del Ejemplo2.

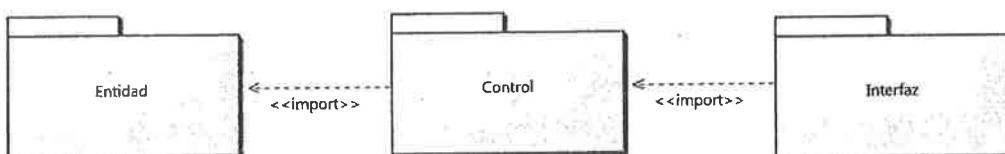


Figura 5.53. Diagrama de paquetes del Ejemplo2.

Para resolver el ejercicio seguimos estos pasos:

1. Se crea un modelo nuevo desde el menú **File/New/UML Project**.
2. Se crea un diagrama de clases, botón derecho del ratón sobre el proyecto, elige **Create Representation/Structural Modeling/Class Diagram**. Selecciona el modelo y fin.
3. En el diseño del diagrama de clases se añade primero los paquetes y se ponen los nombres. Y a continuación se añade las clases con sus atributos y operaciones dentro de cada paquete. Seguidamente se añade las relaciones y se procede como se hizo en el apartado anterior.
4. Se crea el diagrama de paquetes, botón derecho del ratón sobre el proyecto, se elige **Create Representation/Structural Modeling/Package Hierarchy**. Selecciona el modelo y fin.
5. Añade los paquetes al modelo, hay que seleccionar **Add** del apartado **Existing Elements** de la paleta, para que aparezca la ventana con los elementos que hay ya creados en el modelo (véase la Figura 5.54). Se seleccionan los tres paquetes.

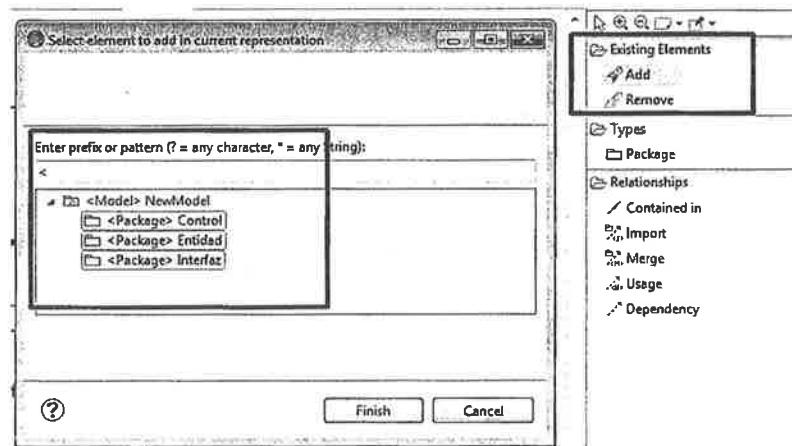


Figura 5.54. Añadir elementos existentes en el modelo.

6. Por último se añade la relación entre los paquetes, se selecciona **Import** de la paleta y se lleva de la clase que importa a la importada, es decir de *Interfaz* a *Control*, y de *Control* a *Entidad*.

5.5.3. UML con WhiteStarUML

WhiteStarUML es un proyecto de código abierto para desarrollar de una forma rápida y flexible modelos UML. En un principio esta herramienta se llamó *StarUML*, el objetivo del proyecto es construir una herramienta de modelado de software convincente y capaz de sustituir a las herramientas comerciales, está diseñado en Delphi y soporta todo tipo de diagramas UML 2.0.

Al iniciar la herramienta se muestra por defecto la hoja de diseño para la elaboración de un diagrama de clases. En esta pantalla inicial (véase la Figura 5.55) podemos observar a la izquierda la paleta de elementos para añadir al diagrama, en el centro la hoja de diseño, donde se añaden los elementos, y a la derecha se muestra el explorador del modelo (*Model Explorer*). En esta parte también se mostrarán el explorador de diagramas (*Diagram Explorer*) y las propiedades (*Properties*) del elemento seleccionado, al seleccionar la pestaña correspondiente.

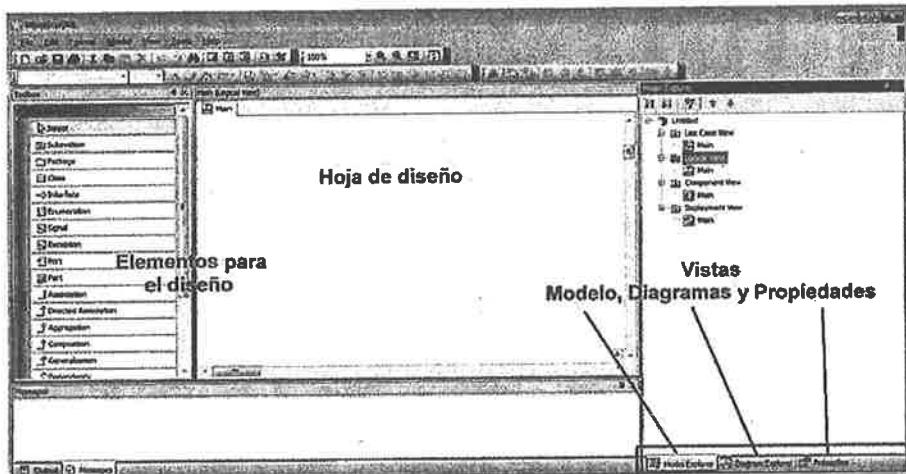


Figura 5.55. Ventana inicial de WhiteStarUML

Desde el explorador del modelo nos moveremos por cada una de las vistas de nuestro modelo: **Use Case View**, **Logical View**, **Component View** y **Deployment View**. Observa que al movernos por cada una de estas vistas y al hacer doble clic en **Main**, se abre una pestaña en la hoja de diseño y cambian los elementos de diseño de la parte izquierda de la ventana dependiendo del tipo de vista. Así pues los diagramas de clases se crearán dentro del elemento **Logical View**.

Para añadir una clase se selecciona el elemento y se hace clic en la hoja de diseño. Se pone el nombre a la clase y se añaden los atributos y operaciones (véase la Figura 5.56).

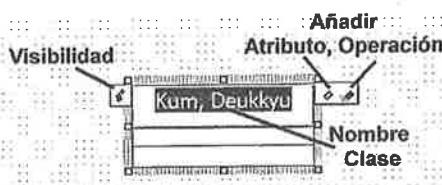


Figura 5.56. Creación de una clase.

Para borrar elementos del modelo es necesario hacerlo desde el **Model Explorer**, porque al borrar del diseño no desaparece del modelo, esto ocurría también con ArgoUML. Si se accede a las propiedades de la clase (pestaña **Properties** con la clase seleccionada) se puede decidir el tipo de clase **Root**, **Leaf**, **Active** o **Specification**, y también se podrá acceder a la ventana de propiedades para añadir los atributos y operaciones, véase la Figura 5.57.

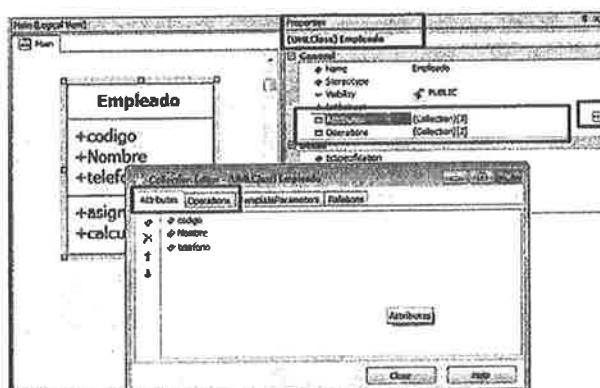


Figura 5.57. Propiedades de la clase, ventana de atributos.

Para crear una asociación se elige el tipo de asociación de la paleta y se arrastra de la clase origen a la clase destino. En la Figura 5.58 se muestra la asociación con nombre DEP_EMPLE entre departamento y empleado, un departamento tiene muchos empleados.

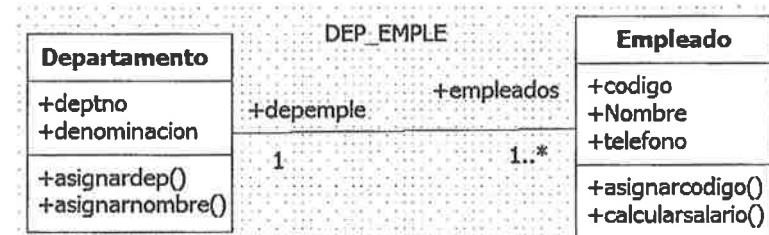


Figura 5.58. Asociación DEP_EMPLE, un departamento muchos empleados.

Para añadir la multiplicidad y los roles se accede a las propiedades de la asociación (véase la Figura 5.59). El nombre de la asociación se pone en *General/Name*. Para añadir la multiplicidad nos fijamos en *General(End1)* y *General(End2)*. Y sobre todo en el participante (*End.Participant*). Es decir en el extremo *End1* participa la clase *Departamento* entonces pondremos 1 en multiplicidad (*End.Multiplicity*), y en el nombre ponemos *depemple*, quiere decir que un empleado es de 1 departamento. En el extremo *End2*, donde participa la clase *Empleado*, la multiplicidad será *1..**, y el nombre *empleados*, quiere decir que 1 departamento va a tener muchos empleados.

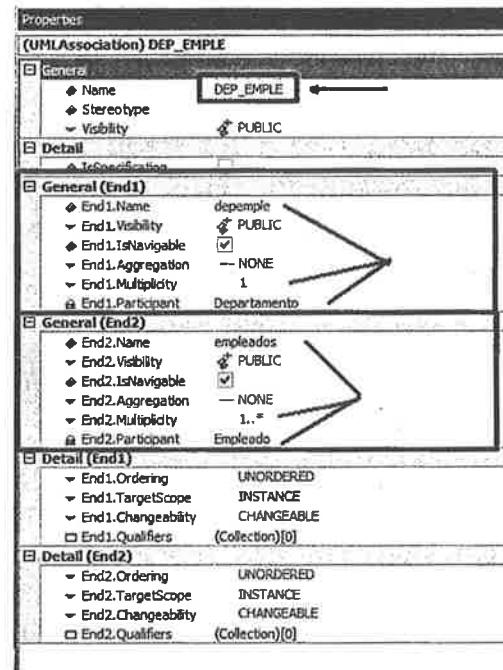


Figura 5.59. Propiedades de la asociación DEP_EMPLE.

ACTIVIDAD 5.3

Se desea realizar el análisis de un sistema de gestión informática de una pequeña academia dedicada a organizar cursos de formación. El sistema debe proporcionar una ventana inicial con una serie de menús que abrirán paso al resto de ventanas de la aplicación. Las ventanas deben permitir realizar las siguientes acciones:

- La gestión de la matrícula de alumnos en los cursos, para dar de alta, eliminar, modificar o consultar datos de matriculación.
- La gestión de cursos, para mantener actualizados los datos de cursos se realizarán operaciones de consulta, altas, bajas y modificaciones, y además generar listados de cursos, y alumnos de cursos.

Disponemos de una base de datos donde están almacenados los datos de los cursos, los alumnos, y las matrículas. Los datos son los siguientes:

- Los datos de curso son: código curso, denominación, horas y precio.
- Los datos de alumnos son: código alumno, nombre, fecha de nacimiento y nivel.
- Los datos matrícula son: el número de matrícula y la fecha.
- Un alumno se puede matricular en varios cursos es decir podrá tener varias matrículas.
- Los cursos son de dos tipos: de idiomas y de tecnologías. De los cursos de idiomas necesitamos saber datos de especialidad y laboratorio. De los de tecnología necesitamos saber el equipamiento y el presupuesto.

SE PIDE:

Identificar las clases de diseño y sus relaciones: clases del tipo Entidad, de Control e Interfaz.

Identificar los atributos y las operaciones, incluir solo los atributos y operaciones que se especifican en el enunciado.

Identificar los paquetes: un paquete para las clases Entidad, otro para las clases de Control y otro para las clases Interfaz.

Realizar el diagrama de clases y de paquetes utilizando *WhiteStarUML*, en la Figura 5.60 se muestra el resultado.

Asegúrate de que las clases se incluyan en su paquete correspondiente, se puede comprobar desde *Model Explorer*, si no se incluyen, arrástralas a su paquete.

Añade la relación de dependencia entre los paquetes.

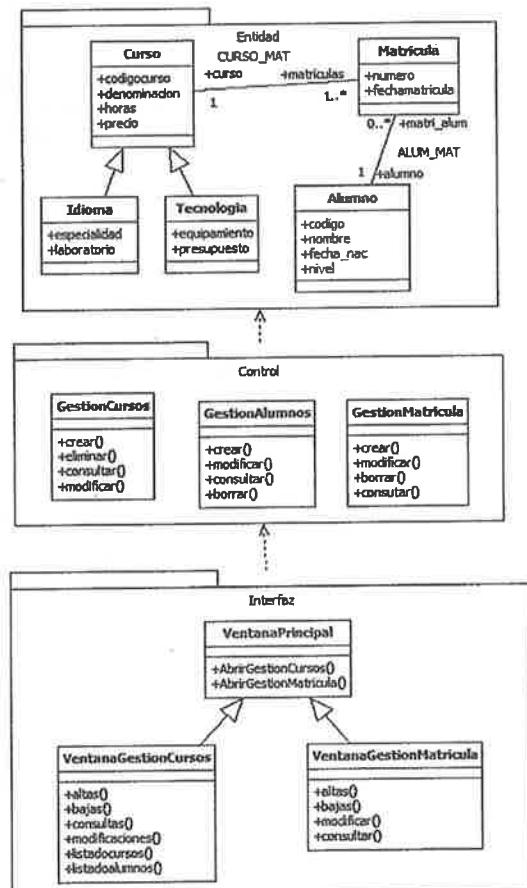


Figura 5.60. Diagrama de la actividad.

5.6. GENERACIÓN DE CÓDIGO A PARTIR DE DIAGRAMAS DE CLASES

Una vez realizado el modelo, lo que queda es generar las clases en Java. Ya vimos con *ArgoUML*, que automáticamente se podía ver el código generado seleccionando la clase y la pestaña *Código fuente*. Para generar el código y guardarlos en archivos se abre el menú *Generar*, se puede seleccionar una clase o todas, si se elige *Generar todas las clases* (véase la Figura 5.61), se seleccionan las clases y el lenguaje, se selecciona el directorio de salida, y se pulsa a *Generar*, se crearán los archivos *.java* correspondientes.



Figura 5.61. Generar las clases del modelo

Para generar las clases con *WhiteStarUML* se necesita añadir al modelo el perfil del lenguaje java, es decir *Java 1.5 Language Profile*. Para ello se abre el menú *Model/Profiles*, y en la ventana que aparece se seleccionan los lenguajes a los que se desea convertir el modelo, en nuestro caso se selecciona *Java 1.5 Language Profile*. Se pulsa *Include* (véase la Figura 5.62) y se cierra la ventana.

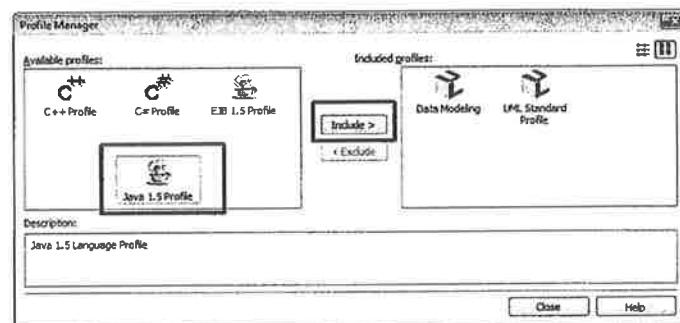


Figura 5.62. Inclusión del perfil Java para crear código java.

Una vez incluido el perfil se abre el menú *Tools/Java 1.5/Generate Code*, se muestra la ventana de la Figura 5.63, en ella se elige lo que se desea generar, se pulsa *Next* y se indica dónde se desea que se almacenen los archivos *.java*. Si observas el código generado con esta herramienta no se crean set de registros en el lado de muchos. Es decir, si un departamento tiene muchos empleados debería crearse en la clase *departamento* un set de registros de objetos *empleado*. Con esta herramienta sólo se crean objetos y no set de objetos.

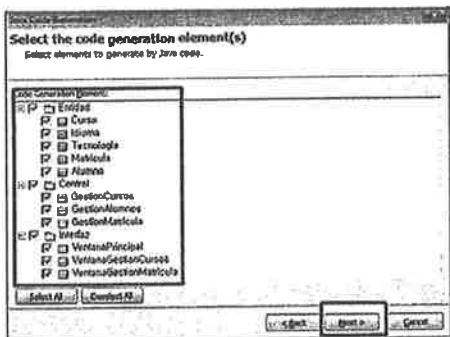


Figura 5.63. Generación de código java.

Para generar código Java en Eclipse es necesario instalarse el plugin *UML to Java Generator*, se busca desde *Eclipse Marketplace* y se instala, véase la Figura 5.64. Se marcan las condiciones, se aceptan las condiciones y se instala.

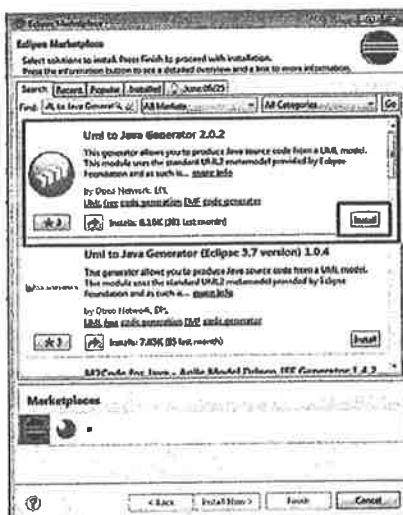


Figura 5.64. Plugin para la generación de Java en Eclipse.

Una vez instalado el plugin nos posicionamos sobre un modelo ya realizado, pulsamos el botón derecho del ratón, y en el menú contextual se elige *Run As/Acceleo UML to Java Generation* (véase la Figura 5.65).

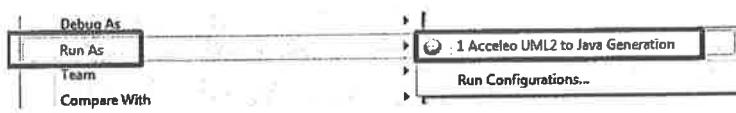


Figura 5.65. Crear un proyecto Java a partir de un modelo.

Se muestra la ventana de configuración de la generación del proyecto, en esta ventana conviene poner un nombre para saber qué proyecto se generará. Se selecciona el modelo, se indica cuál será el nombre del proyecto, el nombre de la carpeta para guardar los archivos *.java* y el nombre de la carpeta para guardar las clases (véase la Figura 5.66). Si se pulsa la pestaña *Class* se puede seleccionar si se desea que se generen automáticamente los *getters* y los *setters*, por defecto aparecen seleccionados. También se puede indicar el nombre del autor, la versión para la documentación. Se pulsa al botón *Run* y se genera el proyecto, que aparecerá en el explorador de modelos.

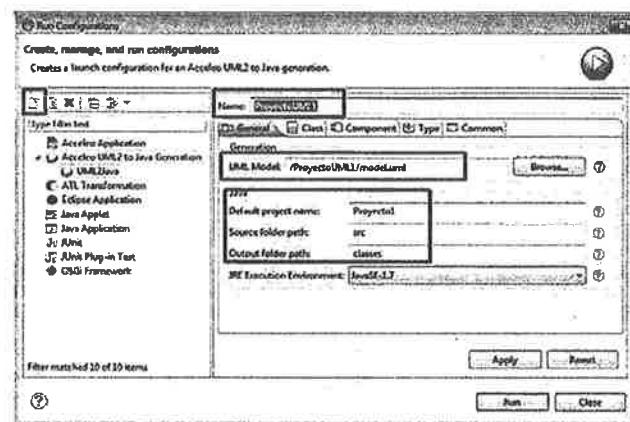


Figura 5.66. Configuración para generar un proyecto Java.

La configuración para la generación de código se guarda y queda asociada a un modelo, con lo cual cuando se va a generar un nuevo proyecto conviene crear una nueva configuración pulsando al botón (Figura 5.66) y seleccionar el modelo, porque si no siempre se lanzará el mismo proyecto.

El proyecto que se crea en java es un proyecto con errores, observa las clases del proyecto creado aquellas clases que tiene un **HashSet** tienen error en las líneas, es porque no incluye bien los import, para corregir el error se añade *import java.util.HashSet* en las clases.

ACTIVIDAD 5.4

Realiza la actividad 5.3 en Eclipse y genera el proyecto java correspondiente.

Realiza el siguiente diagrama de clases (véase la Figura 5.67) utilizando las tres herramientas estudiadas, genera el código Java correspondiente y estudia cada una de las soluciones.

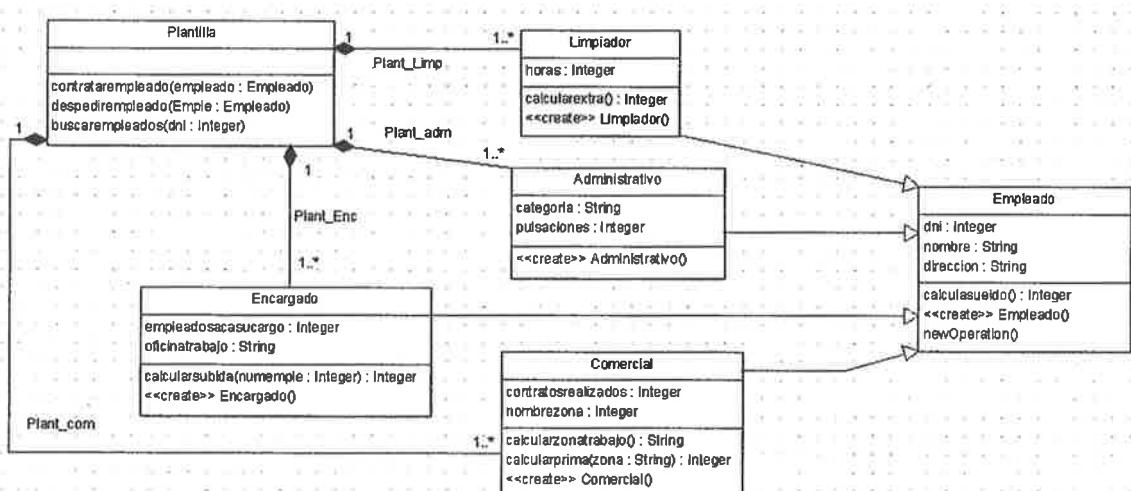


Figura 5.67. Diagrama de clases de la actividad.

5.7 INGENIERIA INVERSA

En el contexto del software, los autores E.J. Chikofsky y J.H. Cross, establecen que la ingeniería inversa es el proceso de analizar un sistema para crear una representación del mismo, pero a un nivel más elevado de abstracción. Por otro lado, P. Hall establece que la ingeniería inversa es un proceso que recorre hacia atrás el ciclo de desarrollo de software. Bajo este enfoque, es posible iniciar el proceso de abstracción a partir del código fuente y llegar hasta la fase de análisis, lo cual representa un flujo inverso al tradicional en el modelo de cascada¹.

En la práctica se han considerado dos tipos de ingeniería inversa: basada en el código fuente, y basada en el programa ejecutable. En el primer tipo, el código fuente está disponible; pero se desconocen aspectos de más alto nivel, existe una documentación pobre o existe documentación pero no está actualizada. En el segundo tipo, no existe código fuente disponible, así que los esfuerzos se concentran en descubrir el correspondiente código fuente.

En este apartado aprenderemos a crear un diagrama de clases a partir de proyectos Java ya realizados. Partimos de un proyecto Java creado desde Eclipse (que se encuentra en la carpeta de recursos del capítulo), la estructura se muestra en la Figura 5.68. Está formado por los paquetes *Interfaz* y *Lógica*. El paquete *interfaz* contiene las ventanas de la aplicación, son clases para mostrar las ventanas y para operar con los datos. La clase *Principal* es un *extends JFrame*, y las otras son *extends JPanel*. El paquete *lógica* contiene las clases persistentes *Departamento* y *Empleado*, y una clase para el tratamiento de excepciones.

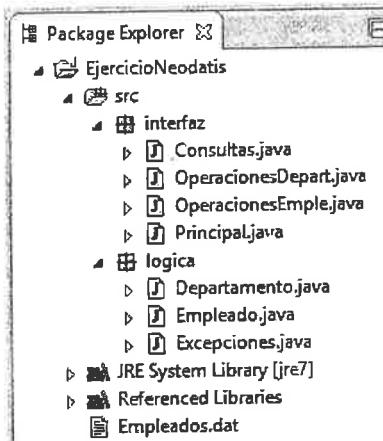


Figura 5.68. Proyecto de ejemplo en Eclipse.

Nos fijamos en las clases *Departamento* y *Empleado*, en los atributos y sobre todo en la relación entre ellas, el departamento tiene un array de empleados (*private Empleado [] empleadosdep;*), para guardar los empleados de su departamento, y el empleado tiene un objeto departamento para referenciar el departamento al que pertenece (*private Departamento dept;*). Estas relaciones entre las clases se mostrarán luego en el diagrama de clases. El código de las clases es el siguiente (no se incluyen los getters y los setters):

```
public class Departamento {
    private int dept_no;
    private String dnombre;
    private String loc;
    private Empleado [] empleadosdep;
```

¹ Fuente: [http://www.iiisci.org/journal/CV\\$riisci/pdfs/X581YP.pdf](http://www.iiisci.org/journal/CV$riisci/pdfs/X581YP.pdf).

```

public Departamento(int dept_no, String dnombre, String loc) {
    this.dept_no = dept_no;      this.dnombre = dnombre;
    this.loc = loc;             }
}

public class Empleado {
    private int emp_no;
    private String nombre;
    private String pobla;
    private String oficio;
    private Double salario;
    private Departamento dept;

    public Empleado(int emp_no, String nombre, String pobla, String oficio,
                    Double salario, Departamento dept) {
        this.emp_no = emp_no; this.nombre = nombre;
        this.pobla = pobla;   this.oficio = oficio;
        this.salario = salario; this.dept = dept;     }
}

```

INGENIERIA INVERSA EN ARGOUML

Para crear en ArgoUML el diagrama de clases del proyecto seguimos los siguientes pasos:

1. Abrir ArgoUML.
2. Pulsar en la opción de menú *Archivo->Importar desde código*. Se abre una ventana desde la que hemos de localizar la carpeta de nuestro proyecto. Se selecciona *src*. En la pestaña *General* marcamos *Importar directorio recursivamente*, para que cargue todas las clases que contiene la carpeta *src*. *Generar diagramas del código importado*, para que se creen los diagramas dentro de cada paquete, y *Disponer diagrama automáticamente*. Dentro de nivel de detalle se selecciona *Clasificadores* (se refiere a las clases) *más especificaciones*. Estas opciones son suficientes en la pestaña *General*. Y dentro de la pestaña *Java* seleccionaremos la multiplicidad de atributos y arrays para que salgan las relaciones, también se puede seleccionar el classpath de java para añadir el JDK o JRE utilizado en el proyecto. Finalmente se pulsa en el botón *Abrir*, véase Figura 5.69.

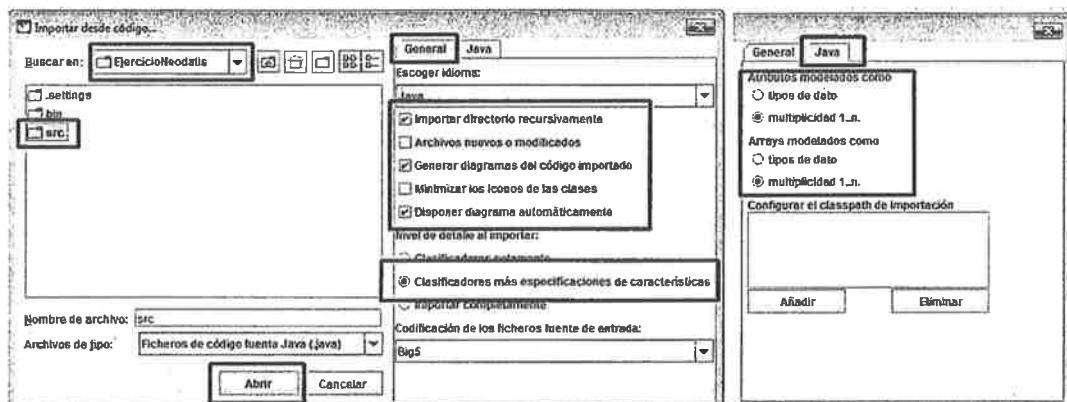


Figura 5.69. Importar código Java desde ArgoUML.

3. Comienza el proceso de generación. Una vez finalizado en el panel de la izquierda se muestran los paquetes y componentes generados. Observa los paquetes *Interfaz* y *Lógica* que teníamos en el proyecto Eclipse. Se creará además otros paquetes que incluirán los import que aparecen en el proyecto como *awt*, *swing* u *org*. Para mostrar los diagramas basta con seleccionar el paquete y hacer doble clic en el modelo que aparece en cada paquete. En la Figura 5.70 se muestra el diagrama generado dentro del paquete *lógica*, se muestran todas las clases, y las clases *Departamento* y *Empleado* aparecen relacionadas.

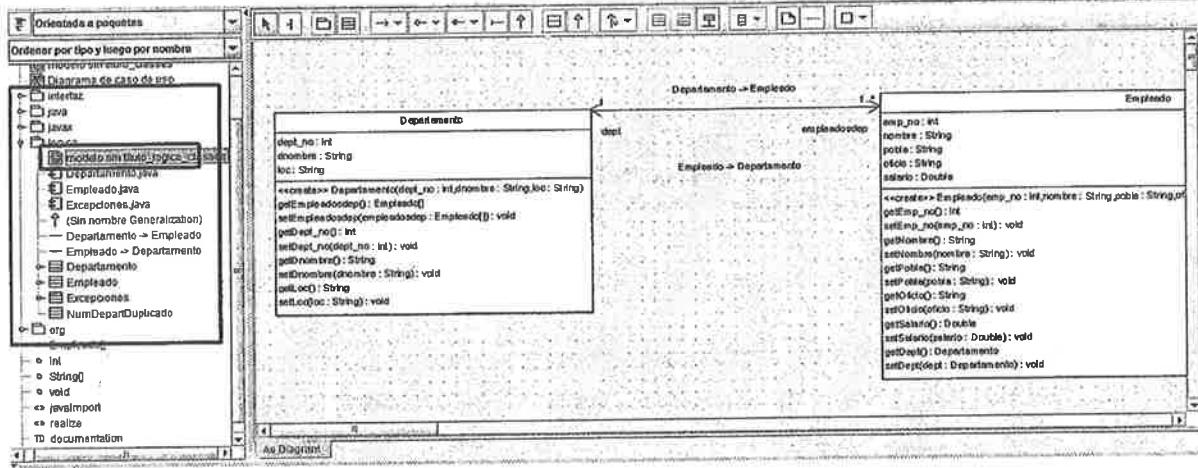


Figura 5.70. Diagrama de clases del paquete *lógica*.

4. Observa en el paquete *interfaz* las relaciones de dependencia y de generalización. Si se seleccionan una relación de generalización y se observan las propiedades podemos ver la jerarquía de las ventanas, así *JFrame* es el padre de la clase *Principal*, o *JDialog* es el padre de la clase *Consultas*. Véase la Figura 5.71.

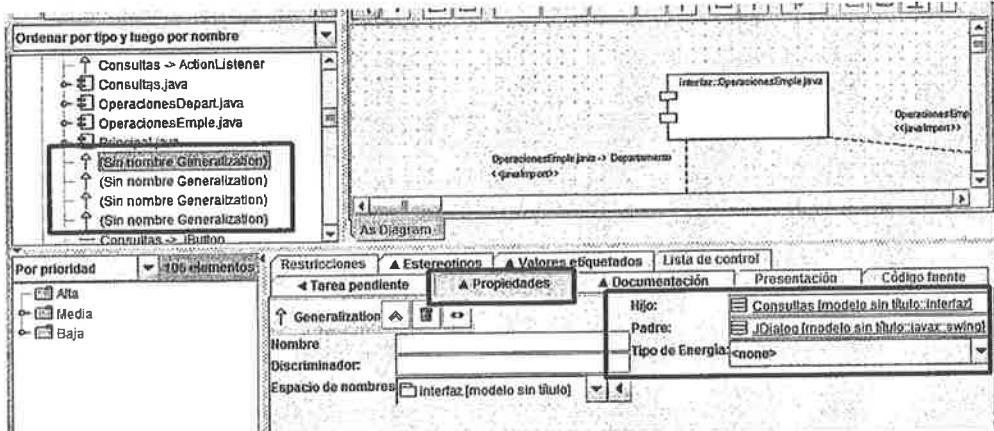


Figura 5.71. Propiedades de una generalización del paquete *interfaz*.

5. Dentro del paquete *interfaz* observa que se ha creado para cada clase del tipo *JFrame* o *JDialog* (*Principal.java*, *Consultas.java*, *OperacionesDepart.java* y *OperacionesEmple.java* que son las ventanas de la aplicación) un componente. Si se accede a los componentes se pueden ver las relaciones de dependencia de la ventana

con los elementos de la ventana, es decir con *JButton*, *JLabel*, *JPanel*, o *JTextField*. Esto es así porque las ventanas necesitan a esos elementos para su funcionamiento. Observa también que tienen una relación de dependencia con las clases *Departamento* y *Empleado*, pues crean esos objetos y trabajan con esos objetos. Si se añaden esos componentes al diagrama de clases del paquete *lógica*, aparecerá una relación de dependencia entre ellos. Véase la Figura 5.72.

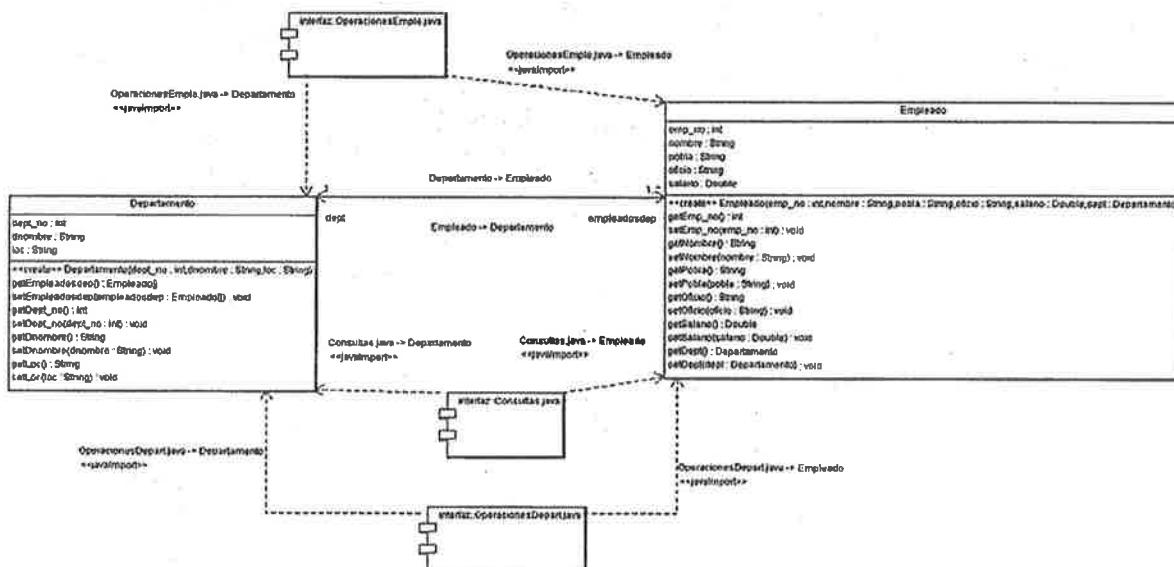


Figura 5.72 Clases relacionadas con los componentes ventanas.

INGENIERIA INVERSA CON WHITESTARUML

Para crear en *WhiteStarUML* el diagrama de clases del proyecto seguimos los siguientes pasos:

1. Abrir *WhiteStarUML*, crear un proyecto vacío, seleccionar los perfiles *Java 1.5 Profile* y *UML Standard Profile* y crear el modelo. Pulsar en la opción de menú *Tools->Java 1.5 ->Reverse Engineer*, Figura 5.73.

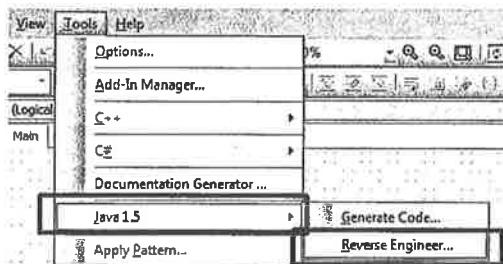


Figura 5.73. Ingeniería inversa en *WhiteStarUML*.

2. Se muestra una ventana desde la que tenemos que seleccionar la carpeta donde tenemos el código fuente de nuestras clases, si no tenemos el perfil de java incluido pedirá incluirlo al proyecto.

Localizamos el proyecto eclipse en nuestro disco duro y pulsamos en el botón *Add All* (Figura 5.74), se mostrarán en la parte inferior todas las clases java del proyecto eclipse. Se pulsa el botón *Next* y en la siguiente ventana se selecciona el paquete que almacenará el resultado, en nuestro caso seleccionamos *Logical View*, véase la Figura 5.74.

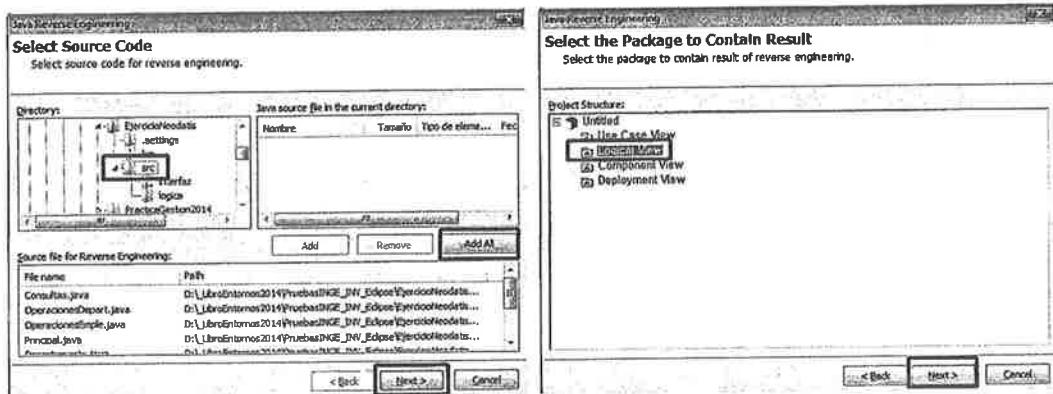


Figura 5.74. Seleccionar código fuente en WhiteStarUML.

3. En la siguiente pantalla se configuran las opciones para realizar la ingeniería inversa, véase la Figura 5.75. Se marcan las 4 casillas que aparecen dentro del grupo *Generate the following visibility only* y marcamos la opción *Create the field to the association*, para que cree las asociaciones. Se marca *Create overview diagram* para que cree un diagrama nuevo y se pone nombre al diagrama. Se pulsa el botón *Run(R)* para dar comienzo al proceso.

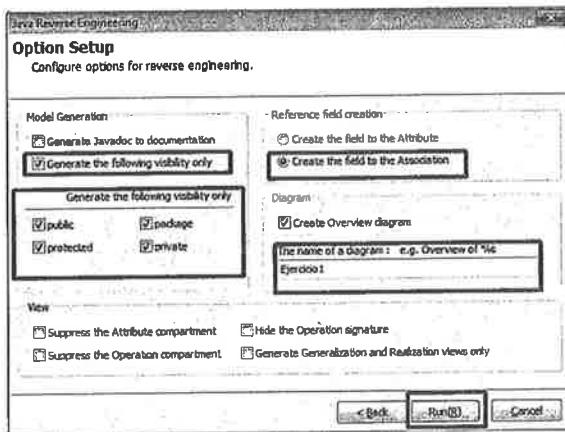


Figura 5.75. Opciones para la ingeniería inversa.

4. Si todo va bien, se muestra un mensaje indicando que el proceso se ha realizado con éxito, se pulsa el botón *Aceptar*. Se muestra una ventana informando de las clases y diagramas generados y se pulsa al botón *Finish(F)* para finalizar.

A continuación observa los diagramas de clases que se han creado, uno para cada paquete. Observa también la asociación entre *Empleado* y *Departamento*. Véase la Figura 5.76.

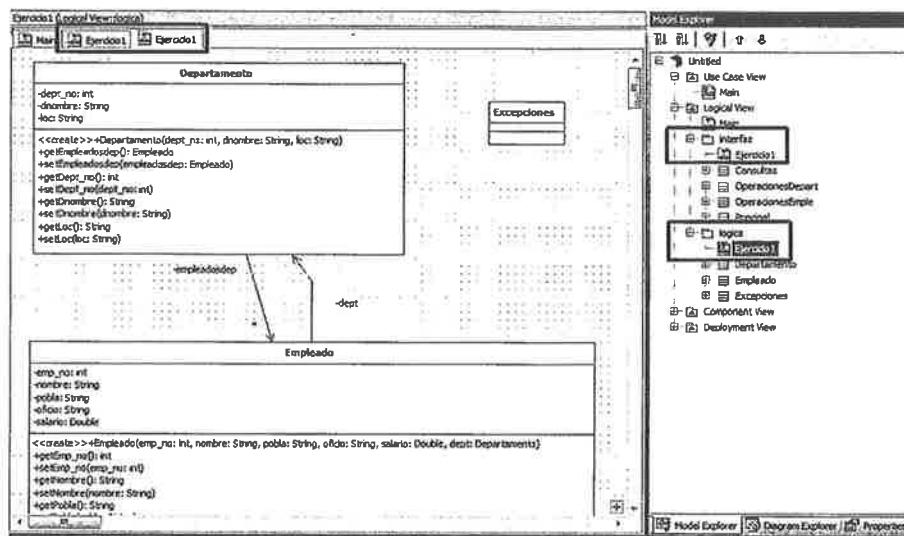


Figura 5.76. Diagramas de clase generados con WhiteStarUML.

ACTIVIDAD 5.5

Realiza la ingeniería inversa utilizando las dos herramientas estudiadas del proyecto *Gestion* que se encuentra en la carpeta de recursos del capítulo. La estructura del proyecto se muestra en la Figura 5.77.

El proyecto está formado por 4 paquetes.

- **Conexión:** contiene una única clase, que se utiliza para conectarse a la BD.
- **Datos:** contiene las clases persistentes asociadas a la BD. Estas clases deben aparecer relacionadas.
- **Operaciones:** contiene las clases que operan con los datos de la BD.
- **Ventanas:** contiene todas las ventanas que se utilizan en la aplicación. Estas clases aparecerán también relacionadas. Todas las clases heredan *barramenu* excepto la clase *Inicio*.

NOTA: Observa que las relaciones que se crean entre las clases persistentes la multiplicidad no es la correcta, pues la versión de java no traduce bien los tipos set (por ejemplo una zona tiene un set de proyectos, en esta relación no se muestra la multiplicidad *).



Figura 5.77. Estructura del proyecto de la actividad.

COMPRUEBA TU APRENDIZAJE

1. ¿Cuáles son los principios del modelado Orientado a Objetos?
2. En UML ¿Cuáles son Diagramas de estructura y qué representan?
3. En UML ¿Cuáles son Diagramas de comportamiento?
4. En UML ¿Cuáles son Diagramas de interacción y qué representan?
5. ¿Qué son los diagramas de clase y para qué se utilizan?
6. ¿Qué son los diagramas de casos de uso y para qué se utilizan?
7. ¿Qué son los diagramas de secuencia y para qué se utilizan?
8. ¿Qué es un diagrama de paquetes y para qué se utilizan?
9. Componentes de una clase.
10. ¿Qué representan las asociaciones y la multiplicidad en un diagrama de clases?
11. ¿Qué es la navegabilidad entre clases?
12. ¿Qué es una Clase Asociación?
13. ¿Qué es una Asociación de generalización y cómo se representa?
14. ¿Qué es una Asociación de composición y cómo se representa?
15. ¿Qué es una Asociación de agregación y cómo se representa?
16. ¿Qué son los estereotipos?
17. ¿Qué representa el estereotipo Boundary?
18. ¿Qué representa el estereotipo Control?
19. ¿Qué representa el estereotipo Entity?

ACTIVIDADES DE AMPLIACIÓN

1. Se desea realizar el análisis de un sistema de gestión informática de una pequeña red de hoteles rurales. El sistema debe proporcionar una ventana para la gestión de las reservas que se realizan en estos hoteles, las operaciones que se deben realizar desde esta ventana son: altas, bajas y modificaciones de reservas, consultas de reservas, generación de facturas y generación de listado de clientes.

Se dispone de una base de datos donde se encuentra almacenada la información de los hoteles, las habitaciones, los clientes, las reservas y las facturas. La información es la siguiente:

- Datos de hoteles son: código, nombre, dirección y teléfono.
- Datos de habitaciones son: número de habitación, estado y número de camas.
- Datos de clientes son: el número de cliente, el nombre, y el teléfono.

- Datos de la reserva son: el número de reserva, la fecha de entrada, la fecha de salida, numero de habitación y número de personas.
- Datos de la factura son: el número de factura, la fecha, el número de cliente, el importe y el IVA.
- Un hotel se compone (utiliza una asociación de composición) de muchas habitaciones. De las habitaciones se pueden realizar muchas reservas, y una reserva será solo de una habitación.
- Igualmente los clientes pueden realizar muchas reservas, y una reserva pertenecerá solo a un cliente.
- Las facturas se crean a partir de las reservas. Una reserva generará 0 o 1 factura, y una factura será de una reserva.

SE PIDE:

Identificar las clases de diseño y sus relaciones: clases del tipo Entidad, de Control e Interfaz.

Identificar los atributos y las operaciones.

Identificar los paquetes: un paquete para las clases Entidad, otro para las clases de Control y otro para las clases Interfaz.

Realizar el diagrama de clases y de paquetes utilizando las herramientas estudiadas.

Generar el código java de las clases del modelo.

2. Guarda el modelo anterior con otro nombre y realiza los siguientes cambios en el diagrama de clases del tipo *Entidad*:
 - Considera ahora que el hotel se compone de 3 tipos de habitaciones, las suites, individuales y dobles. Estos tipos heredan la clase *Habitaciones* creada anteriormente.
 - Las reservas se realizan de estas habitaciones.
 - Datos de la suite son: metros, servicios y calculogastos (operación).
 - Datos de las individuales son: categoría y calculorebaja (operación).
 - Datos de las normales son: minibar y extra.
 - Generar el código java de las clases del modelo.
3. Carga el proyecto ArgoUML de la Actividad 5.5 y realiza un diagrama de clases en el que se relacionen los componentes creados para cada ventana (*VGestionSuministros*, *VGestionPiezas*, *VGestionProveedores*, *VGestionProyectos* y *barrademenu*), el paquete *operaciones*, el paquete *datos*, y las clases persistentes. Debe aparecer una relación de dependencia que surge por los *import*. Por ejemplo, en la Figura 5.78 se muestra la relación entre el componente de la ventana *VGestionSuministros* con los paquetes *datos* y *operaciones*. La relación es de dependencia.

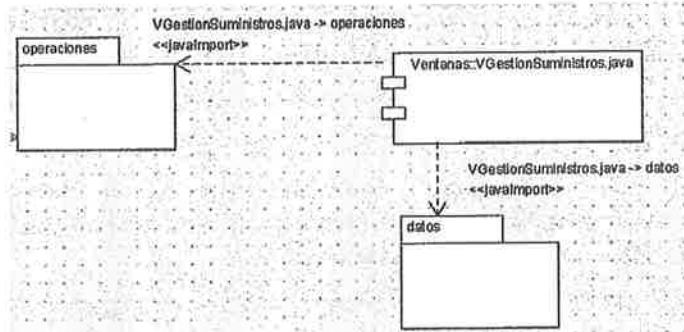


Figura 5.78. Asociaciones del componente *VGestionSuministros*.

4. Realiza con las herramientas estudiadas el diagrama de clases que cumpla estas especificaciones.

Un centro de instalaciones deportivas quiere hacer una aplicación de reservas.

En el centro existen varias instalaciones deportivas, (piscinas, gimnasios, frontones, etc.), datos de las instalaciones son su nombre y el precio hora, entre otros.

Las instalaciones deportivas pueden ser de 2 tipos, interiores y exteriores. De las interiores nos interesa saber el consumo de luz, y de las exteriores el gasto de mantenimiento de jardín.

El centro en cuestión cuenta con muchos socios, de los cuales se almacenan su dirección, ciudad, provincia, teléfono, nombre y cuota.

Existen una serie de artículos que se pueden alquilar junto con las reservas, (balones, redes, raquetas, etc.)

Cada instalación es reservada por un socio en una fecha dada desde una hora de inicio hasta una hora de fin. Un socio podrá hacer varias reservas.

Cada reserva puede tener asociada uno o varios artículos deportivos que se alquilan aparte. Por ejemplo, si yo quiero hacer una reserva para jugar a voleibol tengo que reservar una instalación polideportiva más un artículo red, más un artículo balón.

5. Realiza cambios al ejercicio anterior para que la aplicación de reservas permita hacer lo siguiente:

La aplicación debe proporcionar una ventana inicial que dará paso a las ventanas para gestionar los datos de las reservas y de los socios:

- La gestión de las reservas incluye las operaciones de altas, bajas, modificaciones, consultas y listado de reservas.
- La gestión de los socios de las instalaciones incluirá altas, bajas, modificaciones, consultas y listados de socios.

Identifica las clases de diseño del tipo Control y tipo Interfaz y sus relaciones. Identifica los atributos y las operaciones. Identifica los paquetes (incluye las clases del tipo Entidad del ejercicio anterior).

Realiza el diagrama de clases y de paquetes utilizando las herramientas estudiadas.

CAPÍTULO 6

ELABORACIÓN DE DIAGRAMAS DE COMPORTAMIENTO

CONTENIDOS

- Diagramas de casos de uso.
- Diagrama de secuencia
- Diagrama de comunicación
- Diagrama de estado
- Diagrama de actividades

OBJETIVOS

- Identificar los tipos de diagramas de comportamiento.
- Interpretar diagramas de interacción.
- Interpretar el significado de los diagramas.
- Elaborar diagramas de comportamiento e interacción sencillos.

RESUMEN DEL CAPÍTULO

El modelado es una parte esencial en los grandes proyectos software y útil incluso en pequeños proyectos. UML nos permite modelar casi cualquier tipo de aplicación que se ejecuta en cualquier tipo y combinación de hardware, sistema operativo, lenguaje de programación, red, etc. Este capítulo hace una introducción a los diagramas de comportamiento centrados en modelar lo que sucede en un sistema software.

6.1. INTRODUCCIÓN

En el capítulo anterior vimos cómo modelar diagramas de clases, estos diagramas nos dan información sobre la estructura estática del sistema, no nos dan información sobre el comportamiento dinámico del mismo. Para modelar esta información utilizamos otros tipos de diagramas UML, los diagramas de comportamiento.

Dentro del grupo de diagramas de comportamiento se incluyen los diagramas de casos de uso, actividad, estado e interacción. Los diagramas de interacción incluyen el diagrama de secuencia, diagrama de comunicación, diagrama de tiempos y diagrama de vista de interacción.

Los diagramas de comportamiento muestran las características de comportamiento de un sistema o proceso de negocio. Incluyen los siguientes tipos:

DIAGRAMA	RESUMEN
Diagrama de casos de uso	Describe el comportamiento del sistema desde el punto de vista de un usuario/sistema que interactúa con él
Diagrama de actividad	Parecido a los diagramas de flujo, muestra los pasos, puntos de decisión y bifurcaciones. Son útiles para modelar el flujo en un caso de uso o entre casos de uso
Diagrama de estado	Muestra el conjunto de estados por los cuales pasa un objeto y cómo se producen las transiciones de un estado a otro
Diagrama de secuencia	Muestra cómo interactúan unos objetos con otros
Diagrama de comunicación	Muestra las interacciones entre los elementos en tiempo de ejecución
Diagrama de tiempos	Definen el comportamiento de diferentes objetos dentro de una escala de tiempo
Diagrama de vista de interacción	Muestra la cooperación entre otros diagramas de interacción

En este capítulo se estudiarán algunos de estos diagramas a través de ejemplos sencillos y se utilizarán herramientas para elaborarlos.

6.2. DIAGRAMA DE CASOS DE USO

Los **casos de uso** modelan el sistema desde el punto de vista del usuario. Constituyen una excelente herramienta para la obtención de los requisitos software durante la fase de análisis de un proyecto. Deben cumplir los siguientes objetivos¹:

- Definir los requisitos funcionales y operativos del sistema, diseñando un conjunto de escenarios que faciliten una descripción de cómo se usará el sistema.
- Proporcionar una descripción clara de cómo el usuario interactúa con el sistema y viceversa.
- Proporcionar una base para la validación de las pruebas.

¹ Ingeniería del software. Un enfoque práctico. Roger S. Pressman.

Un caso de uso se suele escribir en un lenguaje sencillo y sin tecnicismos para que pueda ser entendido y comprendido por todos los participantes en el proceso de análisis. Un conjunto completo de casos de uso especifican todas las diferentes formas de utilizar el sistema y por lo tanto, definen todo el comportamiento requerido del mismo.

Utilizando UML se puede crear una representación visual de los casos de uso llamada **diagrama de casos de uso**.

6.2.1. Elementos del diagrama de casos de uso

Los elementos de un diagrama de casos de uso son los siguientes:

- **Los actores.** Un actor puede ser cualquier cosa que interactúa con el sistema y es externo a él. No necesariamente es una persona, puede ser un dispositivo u otro sistema. Se representan mediante un monigote con un nombre debajo.
- **Los casos de uso.** Representan una unidad funcional del sistema que se realiza tras una orden de algún agente externo que puede ser un actor u otro caso de uso. Un caso de uso es iniciado por un actor, a partir de ese momento, ese actor junto con otros interactúan con el sistema participando de ese caso de uso. Se representan mediante un óvalo o elipse y dentro se escribe una descripción textual.
- **Las relaciones.** Existen varios tipos de relaciones que se verán más adelante. La más típica es la asociación entre actores y casos de uso que se representan mediante una línea continua, significa la participación del actor en el caso de uso.
- Puede aparecer un rectángulo que muestre los límites del sistema.

La Figura 6.1 representa un modelo de caso de uso con dos actores y dos casos de uso. El rectángulo muestra los límites del sistema, dentro de él se pone el nombre. Las líneas representan la comunicación entre el actor y el caso de uso.

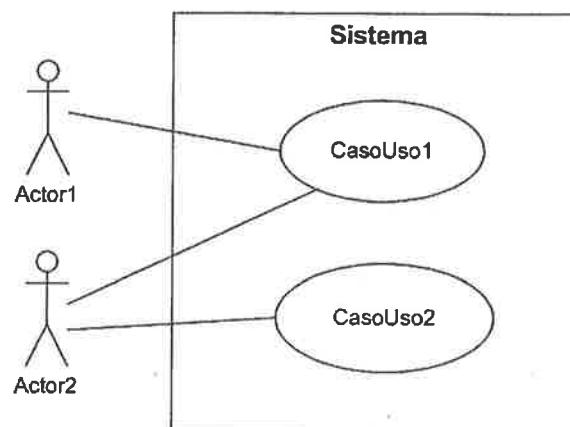


Figura 6.1. Diagrama de caso de uso.

Ejemplo 1. Caso de uso que representa a dos actores interactuando con una máquina de refrescos. El cliente que compra una bebida y el reponedor que abastece de bebidas a la máquina. Los casos de uso serán *ComprarRefresco* y *Reabastecer*. Véase Figura 6.2.

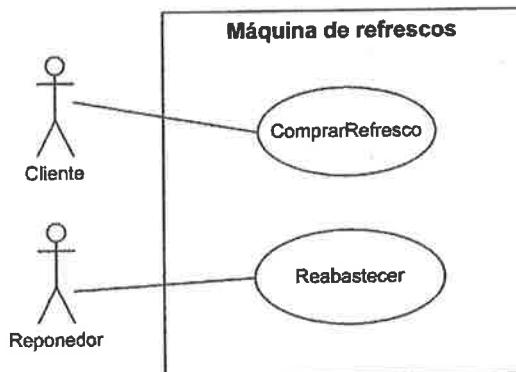


Figura 6.2. Ejemplo 1.

Ejemplo 2. Caso de uso que representa a dos actores interactuando con un sistema de venta de productos. El cliente interactúa con el sistema para buscar y comprar productos. El administrativo interactúa con el sistema para insertar nuevos productos. Se definen dos casos de uso para representar la interacción del cliente con el sistema: *BuscarProductos* y *ComprarProductos*; y un caso de uso para representar la interacción del administrativo: *InsertarProductos*.

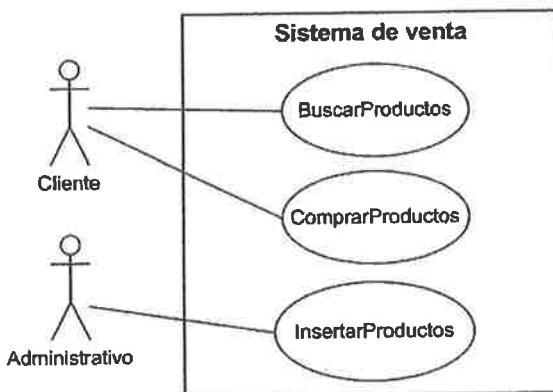


Figura 6.3. Ejemplo 2.

Los casos de uso siempre lo inician los actores para solicitar o modificar información del sistema, o para informar al sistema de que ha ocurrido algo en su entorno que le incumbe. El nombre del caso de uso debe coincidir con el objetivo del actor principal, que es normalmente el que comienza el caso de uso.

6.2.2. Identificar actores

Los actores son entidades externas al sistema que proporcionan los estímulos al mismo. Normalmente son seres humanos pero pueden ser también otros sistemas o incluso dispositivos. Para identificar los actores se necesita saber qué y quién interactúa con el sistema y qué roles desempeñan cuando interactúan con él. Hay que tener en cuenta los siguientes puntos a la hora de definir actores:

- Los actores son siempre externos al sistema.
- Interactúan directamente con el sistema.
- Representan roles que personas y elementos desempeñan en relación al sistema.

- Necesitan un nombre que describa el papel que desempeñan.
- La misma persona o elemento puede interpretar varios roles como actores distintos.

Por ejemplo, en una tienda online pueden interactuar diferentes actores: un usuario normal que visita la tienda, un usuario registrado que compra productos, un administrativo que comprueba el estado del stock de los productos, realiza pedidos a proveedores y da de alta nuevos productos; el empleado de almacén encargado de tramitar los pedidos de los clientes, etc.

6.2.3. Identificar casos de uso

Para identificar los casos de uso es necesario entender lo que el sistema debe hacer, y la mejor forma de empezar es identificar los actores que usarán el sistema y cómo lo usarán. En base a esto, nos planteamos una serie de preguntas:

- ¿Qué tareas realizan los actores involucrados en el sistema?
- ¿Qué información crea, almacena, modifica, destruye o lee el actor?
- ¿Debe el actor notificar al sistema los cambios externos ocurridos?
- ¿Debe el sistema informar a algún actor de cambios internos?
- ¿Interactúa el sistema con algún sistema externo?

Los casos de uso se documentan con texto informal. Describen tanto lo que hace el actor como lo que hace el sistema cuando interactúa con él. Una plantilla para una especificación sencilla de casos de uso contiene la siguiente información:

- **Nombre** del caso de uso.
- **ID** del caso de uso.
- **Breve descripción** de lo que se espera que haga el caso de uso.
- **Actores implicados en el caso de uso.** Existen dos tipos de actores: principales y secundarios. Los primeros son los que activan el caso de uso, los segundos interactúan con el caso de uso después de haberse activado; suelen ser otros sistemas, componentes externos o dispositivos con los cuales interactúa nuestro sistema.
- **Precondiciones.** Condiciones que se deben de cumplir antes de que el caso de uso pueda empezar.
- **Curso normal.** Pasos en el caso de uso que llevan a la finalización con éxito del mismo. Los pasos se escriben ordenados cronológicamente.
- **Postcondiciones.** Condiciones que se deben cumplir al final del caso de uso.
- **Alternativas.** Durante la ejecución de un caso de uso, suelen aparecer errores o excepciones. Por ejemplo, un usuario accede a un sistema escribiendo el nombre de usuario y la clave, y los escribe mal. El sistema deberá informar esta situación al usuario que introduce los datos. Esas desviaciones del curso normal del caso de uso se llaman alternativas.

Ejemplo 3. Partimos del diagrama de casos de uso representado en la Figura 6.3. La siguiente tabla muestra un ejemplo de caso de uso que describe los pasos que sigue un cliente para buscar productos en el sistema de venta según los criterios solicitados. Como precondición se indica que el cliente que interactúa debe estar registrado en el sistema, la postcondición será que el sistema muestra los productos que coincidan con el criterio de búsqueda:

Nombre: BuscarProductos
ID: CU-1
Descripción:
El cliente solicita consultar productos que cumplan una serie de criterios de búsqueda. El sistema muestra los datos de los productos solicitados.
Actores: Cliente.
Precondiciones:
Se requiere que el cliente esté registrado en el sistema.
Curso normal del caso de uso:
<ol style="list-style-type: none"> 1. El cliente selecciona buscar productos. 2. El sistema solicita los criterios de búsqueda. 3. El cliente introduce los criterios solicitados. 4. El sistema busca productos que cumplen los criterios introducidos por el cliente. 5. El sistema encuentra productos y se los muestra al cliente.
Postcondiciones:
El sistema muestra los productos encontrados.
Alternativa 1:
<ol style="list-style-type: none"> 5. El sistema no encuentra productos. 6. El sistema informa al cliente de que no hay productos que cumplan el criterio de búsqueda. 7. Fin del caso de uso.

En el ejemplo se ha incluido una alternativa en el paso 5 para el caso que el sistema no encuentre productos que cumplan los criterios de búsqueda. En este caso el sistema informa al cliente de que no ha encontrado productos. Normalmente los flujos alternativos no regresan al flujo principal ya que a menudo tratan con errores y excepciones. En el paso 7 se indica el final del caso de uso.

Ejemplo 4. Partimos del ejemplo anterior. En este caso se incluye un paso condicional para comprobar si el sistema encuentra o no productos. También se incluye un paso repetitivo para que el cliente realice todas las búsquedas que desee. No se indica ninguna postcondición, ya que el curso normal del caso de uso puede o no encontrar productos:

Nombre: BuscarProductos
ID: CU-1
Descripción:
El cliente solicita consultar productos que cumplan una serie de criterios de búsqueda. El sistema muestra los datos de los productos solicitados.
Actores: Cliente.
Precondiciones:
Se requiere que el cliente esté registrado en el sistema.

Caso normal del uso:

1. El cliente selecciona buscar productos.
2. El sistema solicita los criterios de búsqueda.
3. El cliente introduce los criterios solicitados.
4. El sistema busca productos que cumplan los criterios introducidos por el cliente.
5. *Si* el sistema encuentra algún producto
 - 5.1 *Para* cada producto encontrado
 - 5.1.1. El sistema muestra sus datos.
6. *Si/No*
 - 6.1 El sistema comunica al cliente que no ha encontrado productos.
7. El cliente repite los pasos 2 a 6 hasta finalizar las búsquedas.

Postcondiciones:

Ninguna.

Alternativas:

Ninguna

En los pasos 5 y 6 se ha definido un paso condicional. Para ello se utilizan las palabras clave *SI* y *SI NO*, para indicar una ramificación en el flujo. Detrás de la palabra *SI* se indica una expresión booleana sencilla que devuelva verdadero o falso. En este caso la expresión es *SI el sistema encuentra algún producto*, que puede devolver verdadero o falso. Debajo del *SI* y del *SI NO* se ha escrito un texto sangrado que indica el paso a realizar si la expresión es verdadera o no. En este caso no se ha indicado ninguna postcondición y es que las ramificaciones reducen el número de postcondiciones. Recuerda que las postcondiciones son condiciones que se deben cumplir al final del caso de uso, y en este caso, en el curso normal no siempre el sistema encuentra los productos.

En el Paso 5.1 se ha utilizado la palabra clave *Para*. Esta palabra la podemos utilizar para modelar la repetición. En el ejemplo se ha escrito: *Para cada producto encontrado* y debajo *El sistema muestra sus datos*, es decir, para cada producto que cumpla los criterios de búsqueda el sistema lo que hace es mostrar los datos. En el Paso 7 se ha definido un paso repetitivo indicando que se repitan los Pasos 2 a 6 hasta que el cliente no desee realizar más búsquedas.

Ejemplo 5. A continuación se describe el caso de uso *InsertarProductos*. El administrativo solicita dar de alta un nuevo producto; se requiere que esté registrado en el sistema. El sistema pedirá los datos del nuevo producto y validará si son correctos; si son correctos se inserta el producto, en caso contrario se comunica al cliente que ha introducido datos erróneos. No se indica ninguna postcondición por el mismo motivo que en el ejemplo anterior:

Nombre: InsertarProductos**ID:** CU-2**Descripción:**

El administrativo solicita insertar un producto en el sistema. El sistema inserta los datos del producto introducido por el administrativo.

Actores: Administrativo.**Precondiciones:**

Se requiere que el administrativo esté registrado en el sistema.

Curso normal del caso de uso:

1. El administrativo solicita insertar un producto.
2. El sistema solicita los datos del producto.
3. El administrativo introduce los datos.
4. El sistema valida los datos introducidos por el administrativo.
5. Si los datos son correctos
 - 5.1 El sistema inserta el producto.
6. Si NO
 - 6.1 El sistema comunica al cliente que ha introducido datos erróneos.
7. El administrativo repite los pasos 2 a 6 hasta finalizar la inserción de productos.

Postcondiciones:

Ninguna.

Alternativas:

Ninguna

En este ejemplo se podría haber definido una alternativa para los datos introducidos de manera errónea como se hizo en el *Ejemplo 3* en lugar de utilizar la estructura SI-SI NO. En el paso 7 se ha definido un paso repetitivo que indica la repetición de los Pasos 2 a 6 hasta que el administrativo finalice la inserción de productos.

Ejemplo 6. En este ejemplo se describe el caso de uso *ComprarProductos*. El cliente puede comprar uno o varios productos siempre y cuando haya existencias. Cuando finaliza la compra, el sistema le muestra el total de la misma y solicita al cliente los datos para realizar el pago. El cliente introduce los datos y el sistema muestra la confirmación del pago:

Nombre: ComprarProductos

ID: CU-3

Descripción:

El cliente selecciona los productos a comprar y escribe la cantidad deseada. El sistema comprueba si hay existencias y muestra el importe por cada producto seleccionado. Cuando el cliente no desea comprar más productos, el sistema muestra el importe total de la compra y solicita al cliente el pago de la misma.

Actores: Cliente.

Precondiciones:

Se requiere que el cliente esté registrado en el sistema.

Curso normal del caso de uso:

1. El cliente selecciona comprar productos.
2. El cliente selecciona un producto y escribe la cantidad.
3. El sistema valida los datos introducidos.
4. El sistema muestra el importe y el total acumulado.
5. El cliente repite los pasos 2 a 4 hasta indicar el fin de la compra.
6. El sistema calcula el importe total y lo muestra al cliente.
7. El sistema finaliza la compra y solicita la forma de pago al cliente.

8. El cliente selecciona la forma de pago.
9. El sistema solicita los datos al cliente según la forma de pago elegida.
10. El cliente introduce los datos.
11. El cliente realiza el pago.
12. El sistema muestra la confirmación del pago y registra la compra del cliente.

Postcondiciones:

El cliente realiza la compra de productos.

Alternativa 1:

- 3.1 El producto seleccionado no tiene existencias.
- 3.2 El sistema informa al cliente de que no hay existencias.
- 3.3 Continuar en el paso 5 del curso normal.

Alternativa 2:

- 6.1 Si el importe total es 0.
- 6.1.1 Fin del caso de uso.

Alternativa 3:

- 10.1 Los datos introducidos por el cliente no son correctos.
- 10.2 El sistema informa al cliente.
- 10.3 Fin del caso de uso.

En este ejemplo no se ha utilizado ningún paso condicional, se han utilizado varias alternativas. En el Paso 5 se ha definido un paso repetitivo indicando que se repitan los Pasos 2 a 4 hasta que el cliente finalice la compra. La *Alternativa 1* muestra la situación en la que al validar los datos del producto, se comprueba que no tiene existencias; esta situación parte del paso 3, por ello se han numerado los pasos como 3.1, 3.2 y 3.3. En este caso se informa al cliente y se continúa con el Paso 5, donde el cliente puede seguir comprando si así lo desea.

En la *Alternativa 2* se parte del Paso 6, situación en la que se calcula el importe total. Si este importe es 0 es que el cliente no ha seleccionado productos con existencias disponibles, en esta situación finaliza el caso de uso. Por último, la *Alternativa 3* muestra la situación en que los datos introducidos por el cliente para el pago de la compra (paso 10) no son correctos, en este caso finaliza el caso de uso.

A la hora de describir un caso de uso se deben utilizar frases sencillas, no se deben hacer referencias a elementos de la interfaz de usuario, tales como página principal, pantalla de ingreso, hacer clic en un botón, etc. Igualmente se deben evitar detalles arquitectónicos. Por ejemplo, para indicar que el sistema debe almacenar un pedido, hemos de evitar expresiones de este tipo: *el sistema almacena el pedido en la base de datos MySQL*. Sería más correcto indicar: *el sistema almacena el pedido*.

A menudo se utiliza el término **escenario** en el contexto del UML. Según Martin Fowler la palabra escenario se refiere a una sola ruta a través de un caso de uso, una ruta que muestra una particular combinación de condiciones dentro de dicho caso de uso. Cada caso de uso es una colección de escenarios, y cada escenario es una secuencia de pasos que ilustran el comportamiento. Por ejemplo, en el caso de uso de comprar productos tenemos varios escenarios, uno en el que todo va bien, otro donde no hay suficientes existencias en el producto seleccionado, otro en que los datos introducidos por el cliente para el pago no son correctos y otro en el que el importe total es cero (porque no se han seleccionado productos o los que se han seleccionado no tienen existencias) y no se puede realizar la compra.

ACTIVIDAD 6.1

Realiza un caso de uso describiendo los pasos que sigue un usuario para identificarse en un sistema. El usuario debe estar registrado en el sistema. El sistema le pedirá su nombre y su contraseña, el sistema le dará acceso si esos datos son correctos. El sistema bloquea al usuario si escribe 3 veces mal la contraseña.

6.2.4. Relaciones en un diagrama de casos de uso

En un diagrama de casos de uso nos podemos encontrar con varios tipos de relaciones:

RELACIÓN	FUNCIÓN	NOTACIÓN
Asociación	Es la línea de comunicación entre un actor y un caso de uso en el que participa	—
Extensión «extend»	Permite que un caso de uso extienda su comportamiento con uno o más fragmentos de comportamiento de otro. Se utiliza para especificar que el comportamiento de un caso de uso es diferente dependiendo de ciertas circunstancias. Se representa mediante una flecha discontinua con la palabra «extend». La relación apunta al caso de uso que se extenderá	— — →
Generalización de casos de uso	Es como la generalización entre clases. El caso de uso hijo hereda el comportamiento y el significado del caso de uso padre. Se representa mediante una flecha continua con una cabeza triangular en el caso de uso padre	→
Inclusión «include» o «uses»	Permite que un caso de uso base incluya el comportamiento de otro caso de uso. Se representa mediante una flecha discontinua con la palabra «include». La relación apunta al caso de uso a ser incluido	— — →

Las relaciones de extensión, generalización e inclusión de la tabla anterior representan las relaciones entre los casos de uso. A menudo las relaciones «extend» e «include» son fuente de confusión para quienes mezclan los significados de ambos conceptos, por lo tanto deben usarse con discreción y solo donde mejoren la claridad del modelo de caso de uso. Veamos cómo diferenciarlas:

- La relación «include» ocurre cuando se tiene una porción de comportamiento que es similar en más de un caso de uso. Si por ejemplo tenemos dos casos de uso A y B que tienen una serie de pasos en común, se ponen esos pasos en un tercer caso de uso C; y A y B lo incluyen para usarlo, véase Figura 6.4. Los casos de uso incluidos son como subrutinas. En esta relación es necesario que ocurra el caso incluido para satisfacer el objetivo del caso de uso base.

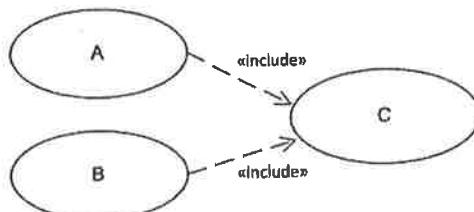


Figura 6.4. Relación «include».

- Se usa la relación «extend» cuando se tiene un caso de uso que extiende o amplía la funcionalidad de otro caso de uso (caso de uso base). Por ejemplo, el caso de uso B extiende la funcionalidad del caso de uso A añadiendo algunos pasos, véase Figura 6.4. Una diferencia con respecto al anterior es que el caso de uso que es extendido no sabe nada del caso de uso que lo extiende. En la relación «extend» el caso de uso de extensión (el B) no es indispensable que ocurra, y cuando lo hace ofrece un valor extra al objetivo original del caso de uso base (el A). El caso de uso base se describe como un caso de uso normal, no contiene ninguna referencia al caso de uso extendido. No obstante debe contener una descripción textual que marque los puntos de extensión en los que puede ser ampliado. Más adelante se verá un ejemplo de cómo dibujar un caso de uso extendido.



Figura 6.5. Relación «extend».

Ejemplo 7. En el modelo de un sistema de viajes aéreos se tienen dos casos de uso que tienen unos pasos comunes que se incluyen en otro caso de uso. Los casos de uso *RealizarReserva* y *ModificarReserva* incluyen el caso de uso *ComprobarAsientos* disponibles. Véase Figura 6.6.

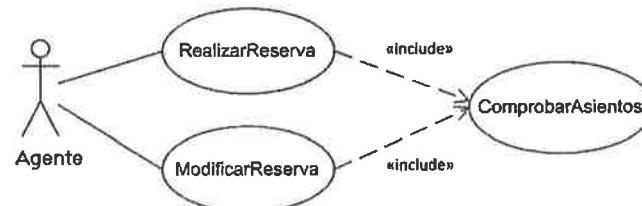


Figura 6.6. Un caso de uso incluido.

ACTIVIDAD 6.2

Se dispone de un sistema de personal donde el director puede realizar dos operaciones: consultar y modificar datos de los empleados. Para realizar esas operaciones se necesita localizar los detalles del empleado a modificar o consultar. ¿Cuántos casos de uso se podrían definir? Realiza la representación gráfica.

Ejemplo 8. En el siguiente ejemplo se amplía el caso de uso *RealizarReserva* para incluir la situación de que se puedan reservar asientos en primera clase. El caso de uso de extensión se llama *RealizarReserva 1^a clase*. El punto de extensión se llama *Establecer clase de asiento*. La circunstancia que causa que se active la extensión es la condición de que el pasajero solicite 1^a clase. La Figura 6.7 muestra como dibujar un caso de uso extendido.

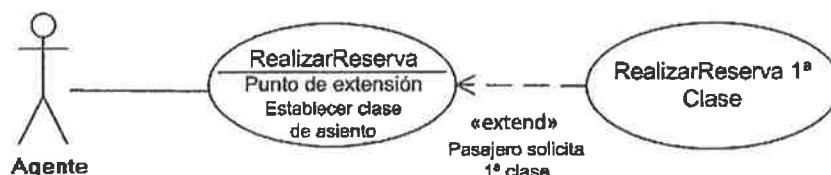


Figura 6.7. Un caso de uso extendido.

ACTIVIDAD 6.3

Representa gráficamente los siguientes casos de uso: un sistema de procesamiento de textos tiene un caso de uso básico llamado *EditarDocumento*, este es ampliado para que el usuario pueda comprobar la ortografía del documento, el caso de uso de extensión se llama *ComprobarOrtografia*, la condición que lo activa es que el usuario selecciona la opción de comprobar la ortografía del documento.

Ejemplo 9. En el ejemplo del sistema de venta a la hora de comprar productos se requería que el cliente estuviese registrado en el sistema. Supongamos ahora que cuando un cliente quiere realizar una compra, si no está registrado se le ofrece la posibilidad de registrarse en ese momento. Para ello, se añade un caso de uso para que el cliente se registre, que extienda el caso de uso de comprar productos. También se pueden comprar productos sin pasar por el caso de uso de registrar cliente. Véase Figura 6.8.

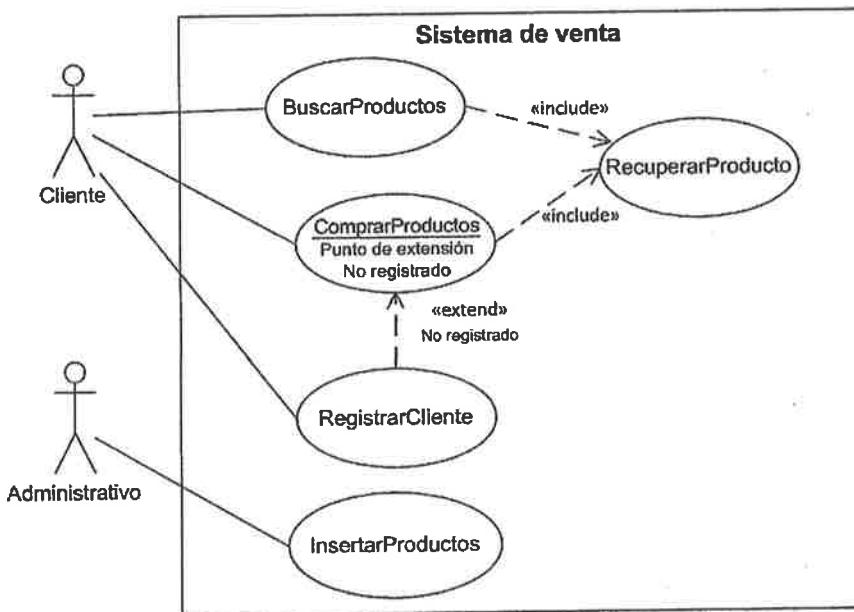


Figura 6.8. Sistema de venta con relaciones «include» y «extend».

Ejemplo 10. En el ejemplo del sistema de venta cuando se van comprando los productos se necesita recuperar su información para obtener el precio. También cuando se realiza la búsqueda de productos se necesita recuperar dicha información para mostrarla al cliente. Por tanto, podemos añadir un nuevo caso de uso en el que se recupere la información de un producto, este caso de uso se incluirá mediante una relación «include» en los otros dos casos de uso. Véase Figura 6.8.

Vamos a ver cómo quedaría la descripción de los casos de uso con las relaciones «include» y «extend». El caso de uso *RecuperarProducto* se expone a continuación:

Nombre: RecuperarProducto
ID: CU-4
Descripción:
El cliente recupera los datos de los productos solicitados.

Actores: Cliente.
Precondiciones:
Se requiere que el cliente esté registrado en el sistema.
Curso normal del caso de uso:
<ol style="list-style-type: none"> 1. El sistema solicita los criterios de búsqueda. 2. El cliente introduce los criterios solicitados. 3. El sistema encuentra el producto o productos solicitados según los criterios especificados.
Postcondiciones:
El sistema encuentra los datos del producto o productos solicitados
Alternativas:
<ol style="list-style-type: none"> 3. El sistema no encuentra los productos solicitados. 4. Fin del caso de uso.

Nombre: BuscarProductos
ID: CU-1
Descripción:
El cliente solicita consultar productos que cumplan una serie de criterios de búsqueda. El sistema muestra los datos de los productos solicitados.
Actores: Cliente.
Precondiciones:
Se requiere que el cliente esté registrado en el sistema.
Curso normal del caso de uso:
<ol style="list-style-type: none"> 1. El cliente selecciona buscar productos. 2. Include (RecuperarProducto) 3. Para cada producto encontrado. <ol style="list-style-type: none"> 3.1 El sistema muestra sus datos. 4. El cliente repite los Pasos 2 a 3 hasta finalizar las búsquedas.
Postcondiciones:
El sistema muestra al cliente los productos solicitados
Alternativas:
<ol style="list-style-type: none"> 3. El sistema no encuentra los productos solicitados <ol style="list-style-type: none"> 3.1 El sistema informa al cliente. 4. Continuar en el Paso 4.

El caso de uso *ComprarProductos* incluye el punto de extensión *No registrado* cuando el sistema solicita al cliente la forma de pago de la compra. Se requiere que el cliente esté registrado para poder realizar el pago. No se numera el paso porque no forma parte del curso normal del caso de uso, el resto de pasos del caso de uso son similares a los vistos inicialmente:

Nombre: ComprarProductos
ID: CU-3
Descripción:
El cliente selecciona los productos a comprar escribiendo la cantidad deseada. El sistema comprueba si hay existencias y muestra el importe por cada producto seleccionado. Cuando el cliente no desea comprar más productos, el sistema muestra el importe total de la compra y solicita al cliente el pago de la misma.
Actores: Cliente.
Precondiciones:
Cuentas registradas y no registradas en el sistema.
Curso normal del caso de uso:
<ol style="list-style-type: none"> 1. El cliente selecciona comprar productos. 2. Include (RecuperarProducto). 3. El cliente introduce la cantidad. 4. El sistema valida los datos introducidos. 5. El sistema muestra el importe y el total acumulado. 6. El cliente repite los Pasos 2 a 4 hasta indicar el fin de la compra. 7. El sistema calcula el importe total y lo muestra al cliente. 8. El sistema finaliza la compra y solicita la forma de pago al cliente. <p>Punto de extensión: No registrado.</p> <ol style="list-style-type: none"> 9. El cliente selecciona la forma de pago. 10. El sistema solicita los datos al cliente según la forma de pago elegida. 11. El cliente introduce los datos. 12. El cliente realiza el pago. 13. El sistema muestra la confirmación del pago y registra la compra del cliente. <p>.....</p> <p>.....</p>

Nombre: RegistrarCliente
ID: CU-5
Descripción:
El cliente se da de alta en el sistema para poder comprar productos.
Actores: Cliente.
Precondiciones:
El cliente no esté registrado en el sistema.
Curso normal del caso de uso:
<ol style="list-style-type: none"> 1. El sistema solicita los datos del cliente. 2. El cliente introduce los datos solicitados.
Postcondiciones:
El sistema da de alta al cliente.

En general, la inclusión de estas relaciones hace que los diagramas de casos de uso sean más complicados y difíciles de leer. Las relaciones «include» y «extend» generan mucha polémica entre la gente que modela casos de uso. Es preferible diseñar un modelo de casos de uso simple que facilite la comunicación entre todas las personas que forman parte del proyecto, aunque ello signifique utilizar menos elementos gráficos de UML.

La relación de generalización en los casos de uso se utiliza cuando se tiene uno o más casos de uso que son especificaciones de un caso de uso más general. Por ejemplo, supongamos que en nuestro sistema de ventas se pueden buscar dos tipos de productos, libros y CDs de música, entonces podemos representar el diagrama de casos de uso con un caso de uso padre *BuscarProductos* y dos especializaciones, el caso de uso *BuscarLibros* y el caso de uso *BuscarCD*, véase Figura 6.9.

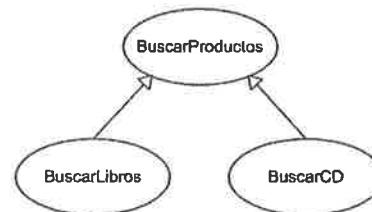


Figura 6.9. Relación de generalización en buscar productos.

Ejemplo 11. En este ejemplo se muestra un caso de uso que se encarga de verificar la identidad de un usuario, el caso de uso se llama *ValidarUsuario*. Esta se puede verificar de varias formas, introduciendo una palabra clave, escaneando la retina del ojo o mediante la huella dactilar del usuario. Podemos representar el caso de uso padre con tres especializaciones: *ComprobarClave*, *EscanearRetina* y *ObtenerHuella*; cada una añade su propio comportamiento. Véase Figura 6.10.

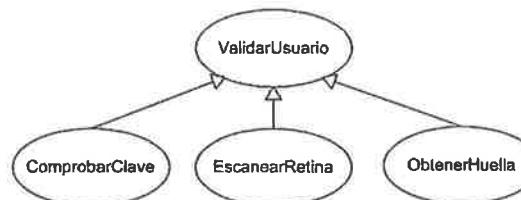


Figura 6.10. Relación de generalización al validar usuario.

El caso de uso *ComprobarClave* comprueba y verifica la clave introducida por teclado, *EscanearRetina* busca entre los modelos de retina de ojo el correspondiente al usuario y *ObtenerHuella* busca la huella dactilar del usuario entre los modelos de huellas dactilares que tiene registrado el sistema. La generalización entre casos de uso se representa por una línea continua dirigida hacia el caso de uso padre, en este caso es *ValidarUsuario*, con la punta de una flecha en forma triangular.

Ejemplo 12. A partir de la siguiente descripción vamos a desarrollar un sistema de gestión de notas donde los profesores pueden registrar y actualizar las notas de los alumnos y los alumnos pueden consultarla. La lista de requerimientos para este sistema es:

- El profesor puede introducir y modificar las notas de los alumnos. Para ello seleccionará el curso, la asignatura y el alumno, y dará el valor a la nota. Las notas se almacenan en una base de datos. Para que un profesor pueda introducir o modificar notas debe estar autenticado en el sistema, es decir, debe introducir el nombre de usuario y la contraseña.

- Los alumnos pueden consultar la nota. Las notas se pueden consultar por asignatura (el alumno elige la asignatura) o se pueden consultar todas las asignaturas del alumno con su nota. Para que un alumno pueda consultar las notas debe estar autenticado en el sistema, es decir, debe haber introducido su nombre de usuario y su contraseña.
- El sistema debe generar un listado de notas por asignatura. Para ello el profesor seleccionará el curso y la asignatura, el sistema generará un informe PDF con dicha información.
- Cuando los usuarios son autenticados el sistema muestra la funcionalidad disponible dependiendo del tipo de perfil del usuario.

La Figura 6.11 muestra los actores y casos de uso del sistema.

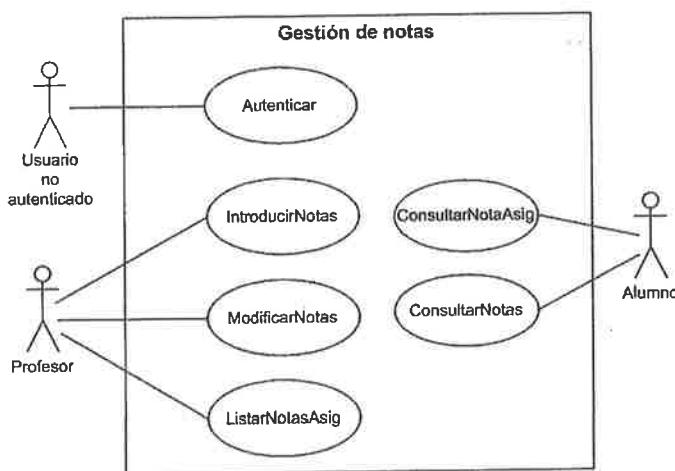


Figura 6.11. Sistema de gestión de notas, casos de uso y actores.

La siguiente tabla muestra los actores y los casos de uso con los que interactúan:

ACTORES	CASOS DE USO
Usuario no autenticado (representa a un usuario del sistema, que una vez autenticado será un profesor o un alumno)	<i>Autenticar</i> : permite a un usuario acceder al sistema, introduciendo su nombre de usuario y su contraseña. Dependiendo del tipo de usuario accederá a unas funciones u otras del sistema. El actor principal es el usuario no autenticado
Profesor	<i>IntroducirNotas</i> : permite al profesor introducir las notas de los alumnos <i>ModificarNotas</i> : permite al profesor modificar las notas de los alumnos <i>ListarNotasAsig</i> : permite al profesor obtener un listado de notas por asignatura
Alumno	<i>ConsultarNotaAsig</i> : permite a un alumno consultar la nota de una asignatura <i>ConsultarNotas</i> : permite a un alumno consultar todas sus notas en todas sus asignaturas

La descripción del caso de uso *Autenticar* es la siguiente:

Nombre: Autenticar
ID: CU-GN1
Descripción:
El caso de uso permite a un usuario acceder al sistema introduciendo su nombre de usuario y contraseña. Dependiendo del tipo de usuario accederá a unas funciones u otras del sistema. Si los datos introducidos no son correctos el sistema solicitará de nuevo los datos al usuario.
Actores: Usuario del sistema no autenticado.
Precondiciones:
Se requiere que el usuario no haya sido autenticado por el sistema.
Curso normal del caso de uso:
<ol style="list-style-type: none"> 1. El usuario solicita autenticarse. 2. El sistema solicita los datos al usuario. 3. El usuario introduce el nombre de usuario y la contraseña. 4. El sistema valida los datos introducidos.
Postcondiciones:
El usuario accede al sistema.
Alternativas:
<ol style="list-style-type: none"> 4.1 El usuario o la contraseña no son correctos. 4.2 El sistema muestra un mensaje indicando el error. 4.3 El sistema solicita que se introduzcan de nuevo los datos. 4.4 Continuar en el paso 2 del curso normal.

Los casos de uso *IntroducirNotas* y *ModificarNotas* son similares. La descripción del caso de uso *IntroducirNotas* es la siguiente:

Nombre: IntroducirNotas
ID: CU-GN2
Descripción:
El profesor introduce las notas de los alumnos. Debe seleccionar el curso, la asignatura y el alumno al que asignará la nota.
Actores: Profesor.
Precondiciones:
Se requiere que el usuario haya sido autenticado en el sistema.
Curso normal del caso de uso:
<ol style="list-style-type: none"> 1. El profesor selecciona introducir notas. 2. El sistema solicita el curso y la asignatura. 3. El profesor introduce el curso y la asignatura. 4. El sistema muestra la lista de alumnos. 5. El profesor selecciona el alumno e introduce la nota. 6. Repetir el Paso 5 hasta que el profesor termine de introducir notas.

7. El sistema valida las notas introducidas.

Postcondiciones:

El sistema registra las notas de los alumnos.

Alternativas:

- 7.1 Las notas introducidas no son correctas.
- 7.2 Fin del caso de uso.

La descripción del caso de uso *ListarNotasAsig* es la siguiente:

Nombre: ListarNotasAsig

ID: CU-GN3

Descripción:

Se desea obtener un listado de notas por asignatura. Para ello el profesor seleccionará el curso y la asignatura. El profesor solicitará al sistema un listado con la nota de los alumnos en la asignatura seleccionada, el sistema generará un informe PDF con dicha información.

Actores: Profesor.

Precondiciones:

Se requiere que el usuario haya sido autenticado en el sistema.

Curso normal del caso de uso:

1. El profesor selecciona listar notas por asignatura.
2. *El sistema solicita el curso y la asignatura.*
3. *El profesor introduce el curso y la asignatura.*
4. *El sistema muestra la lista de alumnos.*
5. El sistema solicita confirmación para generar el listado.
6. El profesor confirma la generación del listado.

Postcondiciones:

El sistema obtiene un informe en PDF con las notas de los alumnos en la asignatura seleccionada.

Alternativas:

- 6. El profesor no confirma la generación del listado.
- 7. Fin del caso de uso.

En la descripción de los dos casos de uso anteriores se puede observar que se repiten los pasos 2, 3 y 4. En este sentido podemos ponerlos en un caso de uso aparte y utilizar una relación «include». En la Figura 6.12 se muestra el caso de uso *SolicitarCurAsig* que contiene pasos comunes a los casos de uso *IntroducirNotas*, *ModificarNotas* y *ListarNotasAsig*. En estos casos de uso, será necesario sustituir los Pasos 2, 3 y 4 por la inclusión de este nuevo caso de uso: *Include(SolicitarCurAsig)*; y renumerar los pasos del flujo normal y de las alternativas.

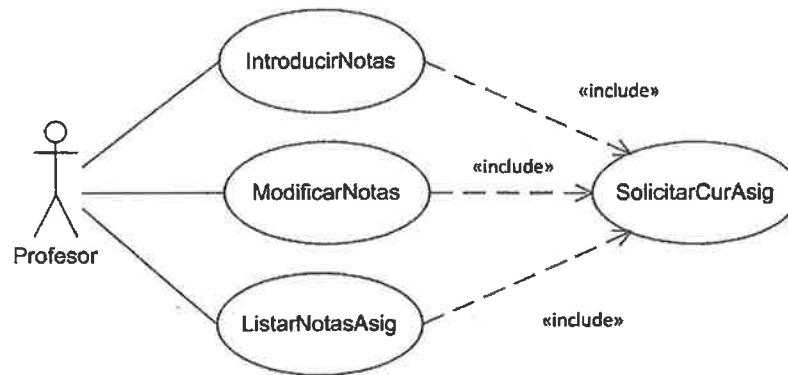


Figura 6.12. Sistema de gestión de notas, relación «include».

La descripción del caso de uso *SolicitarCurAsig* es la siguiente:

Nombre: <i>SolicitarCurAsig</i>
ID: CU-GN4
Descripción:
El sistema recupera los alumnos de un curso y una asignatura seleccionada por el profesor.
Actores: Profesor.
Precondiciones:
Se requiere que el usuario haya sido autenticado en el sistema.
Curso normal del caso de uso:
<ol style="list-style-type: none"> 1. El sistema solicita el curso y la asignatura. 2. El profesor introduce el curso y la asignatura. 3. El sistema muestra la lista de alumnos.
Postcondiciones:
El sistema obtiene los alumnos solicitados.

La descripción de los casos de uso *ConsultarNotaAsig* y *ConsultarNotas* es la siguiente:

Nombre: <i>ConsultarNotaAsig</i>
ID: CU-GN5
Descripción:
Este caso de uso permite al alumno consultar la nota de una asignatura. El alumno seleccionará la asignatura y el sistema mostrará la nota.
Actores: Alumno.
Precondiciones:
Se requiere que el usuario haya sido autenticado en el sistema.

Curso normal del caso de uso:

1. El alumno selecciona consultar nota de una asignatura.
2. El sistema solicita la asignatura.
3. El alumno introduce la asignatura.
4. El sistema muestra la nota.

Postcondiciones:

El sistema obtiene la nota de la asignatura solicitada por el alumno.

Nombre: ConsultarNotas

ID: CU-GN6

Descripción:

Este caso de uso permite al alumno consultar las notas que tiene en sus asignaturas.

Actores: Alumno.

Precondiciones:

Se requiere que el usuario haya sido autenticado en el sistema.

Curso normal del caso de uso:

1. El alumno selecciona consultar notas.
2. El sistema solicita confirmación para obtener las notas.
3. El alumno confirma la obtención de notas.

Postcondiciones:

El sistema muestra al alumno las notas de sus asignaturas.

ACTIVIDAD 6.4

Añade un nuevo caso de uso al sistema de gestión de notas que permita al profesor consultar las notas de un alumno. El sistema solicitará del profesor los datos del alumno, y le devolverá las notas en sus asignaturas. Considera alternativas si los datos introducidos no son correctos.

6.3. HERRAMIENTAS PARA LA ELABORACIÓN DE DIAGRAMAS

A continuación vamos a ver una serie de herramientas software para el desarrollo de diagramas UML. Empezaremos con el diagrama de casos de uso. En los siguientes apartados se utilizarán las herramientas para el diagrama correspondiente.

6.3.1. Eclipse

Para desarrollar diagramas UML con Eclipse se necesita tener el plugin *UML Designer* instalado en el entorno. Para ello pulsamos en la opción de menú *Help-> Eclipse Marketplace*. En el campo *Find* escribimos *uml* y pulsamos en el botón *Go* para que busque el software disponible que contenga la palabra *uml*. Seleccionamos *UML Designer* para nuestra versión de Eclipse y pulsamos el botón *Install*, véase Figura 6.13.

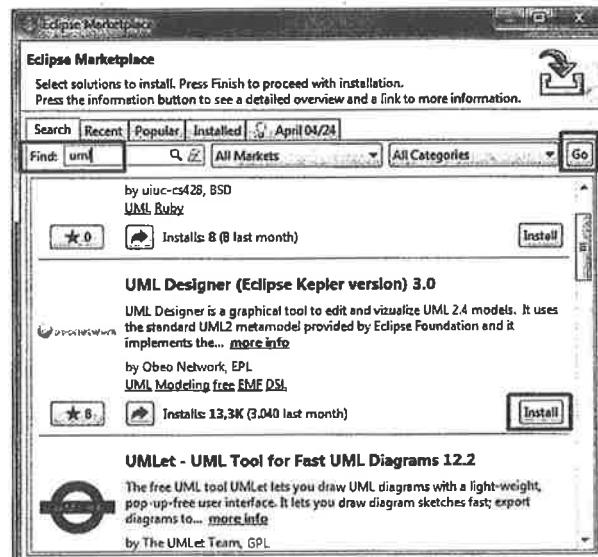


Figura 6.13. Instalación de UML Designer.

A continuación nos pide confirmar las características que se van a instalar, se pulsa el botón *Confirm >*. En la siguiente pantalla se aceptan los términos de la licencia y se pulsa el botón *Finish*. Al terminar nos pedirá reiniciar Eclipse.

Para crear un caso de uso hemos de crear un proyecto UML. Pulsamos en la opción de menú *File->New-> Other-> UML Designer -> UML Project* (véase Figura 6.14), escribimos el nombre del proyecto y pulsamos el botón *Next*, seleccionamos *Model* de la lista *Model Object* y luego pulsamos el botón *Finish*. Nos pregunta si queremos abrir la perspectiva *Modeling*, le indicamos que sí.

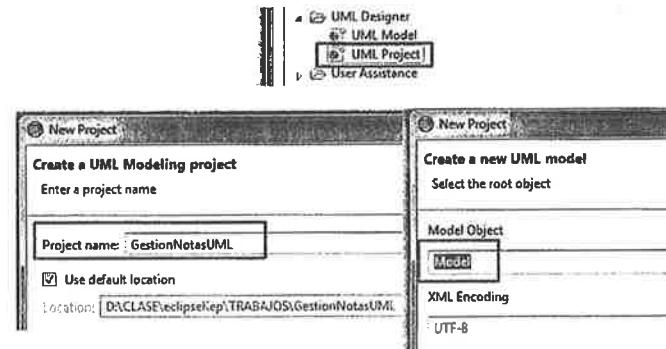


Figura 6.14. Creación de un proyecto UML en Eclipse.

En el proyecto creado se deben mostrar 3 elementos: *Project dependencies* (se utiliza para importar modelos externos para ciertas representaciones del proyecto, puede contener modelos semánticos y modelos gráficos), *model.uml* (contiene el tipo de modelo semántico, en este caso es UML) y el archivo *representations.aird* que contiene los datos necesarios para visualizar los diagramas (u otros tipos de representaciones). Estos dos últimos elementos se pueden desplegar para navegar por su contenido.

A continuación pulsamos con el botón derecho del ratón en <Model> *NewModel* y seleccionamos *New Representation-> NewModel Use Case Diagram* (Figura 6.15). Escribimos un nombre para el caso de uso y pulsamos *OK*.

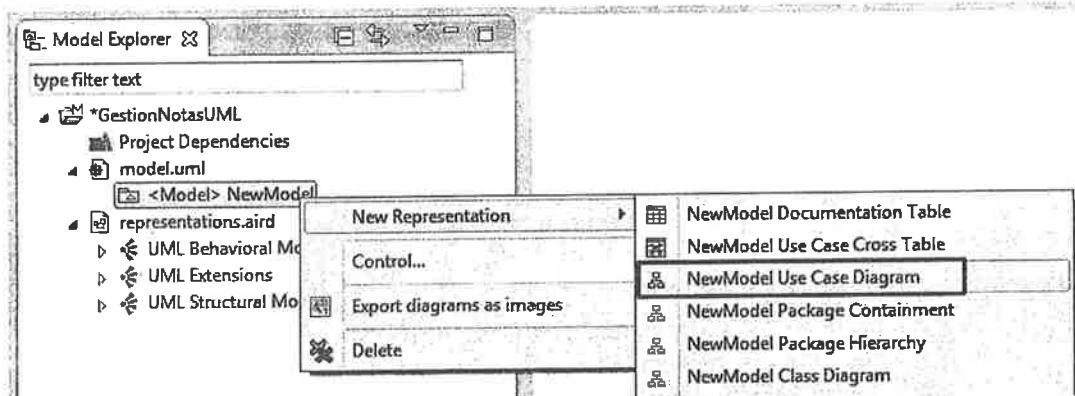


Figura 6.15. Creación de un caso de uso en Eclipse.

Se muestra la pantalla de la perspectiva *Modeling* con varias vistas (véase Figura 6.16):

- A la izquierda, se muestra el explorador de modelos, *Model Explorer*, que es la interfaz de un usuario para interactuar con sus modelos. Se observa el proyecto que se ha creado y los elementos que forman parte del mismo.
- La vista *Outline* ofrece una visión estructural del documento o modelo actualmente abierto. Para los diagramas, se muestra una vista en miniatura de todo el diagrama. En diagramas grandes se utiliza para navegar con facilidad a otras partes del diagrama.
- La vista *Properties* proporciona información detallada sobre el elemento seleccionado en ese momento. Dependiendo de la naturaleza del elemento seleccionado, algunas de estas propiedades se pueden editar directamente en la ventana de propiedades.
- La vista *Problems* contiene marcadores de información, advertencias o errores. Aquí es donde veremos los errores de validación de los modelos que hagamos.
- En la parte central se muestra el área de edición donde podemos empezar a añadir los elementos del diagrama de casos de uso. Se observan una serie de herramientas que nos permitirán seleccionar, organizar y alinear los elementos del diagrama. A la derecha se muestra una paleta con los elementos que se pueden añadir al diagrama, se pueden observar los símbolos utilizados para los casos de uso. Se selecciona uno de ellos y a continuación se hace clic en el área de edición para colocarlo.

La Figura 6.17 muestra el modelo de casos de uso del sistema de gestión de notas. Para marcar el límite del sistema se ha utilizado el elemento *Component*. Los actores se han colocado fuera del componente y los casos de uso dentro. Para validar el diagrama se pulsa con el botón derecho del ratón en una zona del área de edición donde no haya ningún elemento y se selecciona la opción *Validate diagram* del menú contextual. Si ocurre algún error se mostrará en la vista *Problems*. La opción *Export diagram as image* nos permite exportar el diagrama a una imagen JPG. Este entorno no incluye modelar los casos de uso con extensiones.



Figura 6.16. Perspectiva Modeling.

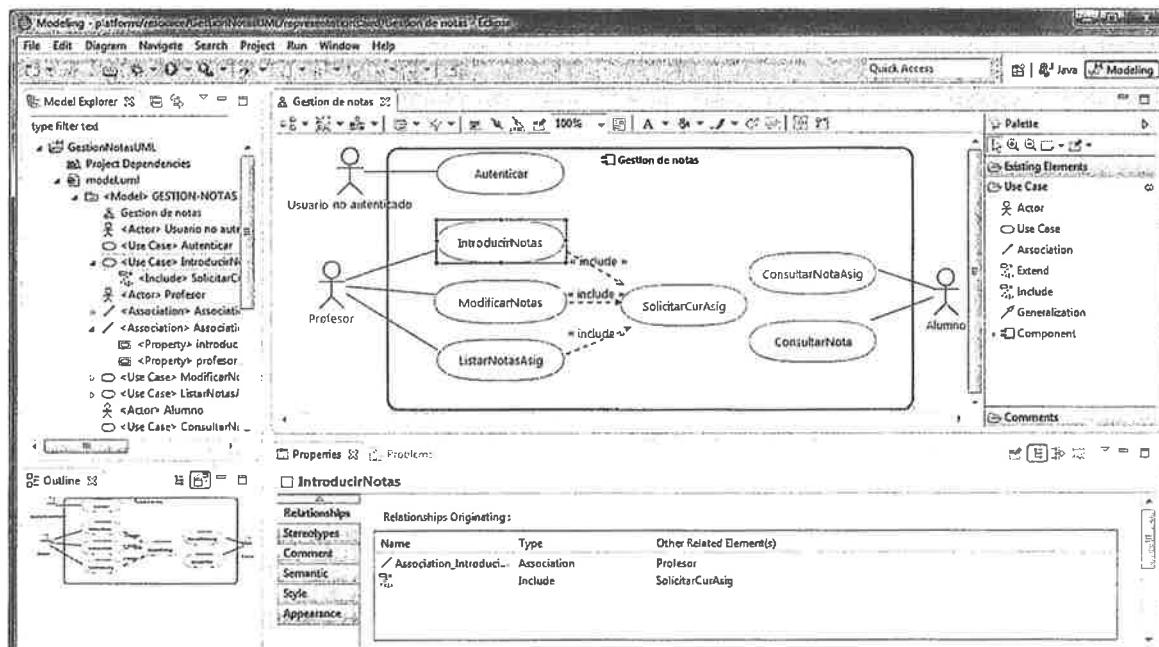


Figura 6.17. Sistema de gestión de notas en Eclipse.

Cuando en un proyecto se diseñan varios componentes y en cada componente se incluyen varios casos de uso, resulta útil obtener una tabla de referencias de casos de uso. Para ello se pulsa con el botón derecho del ratón en <Model> NewModel (o el nombre que le hayamos dado)

y se selecciona *New Representation-> NewModel Use Case Cross Table*, le damos un nombre y pulsamos *OK*. Se generará una tabla que relaciona los casos de uso con los componentes y diagramas que forman el proyecto.

Para obtener una imagen del diagrama se pulsa sobre él con el botón derecho del ratón y se selecciona la opción *Export diagrams as images*, véase Figura 6.18

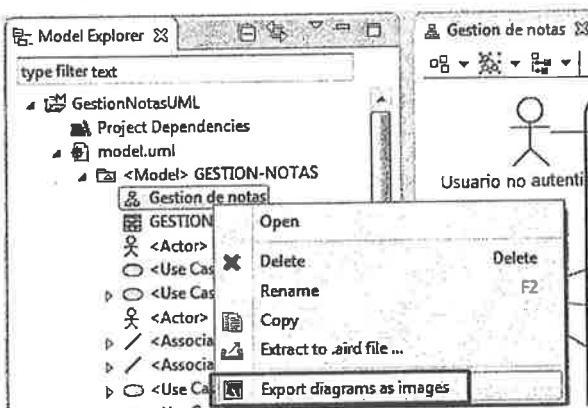


Figura 6.18. Obtener imagen del diagrama en Eclipse.

ACTIVIDAD 6.5

Se trata de abrir un proyecto UML completo para ver diagramas de componentes, de casos de uso, de secuencia, de clases, etc. Asegúrate que el entorno Eclipse está en la perspectiva *Modeling*. Si no está activada selecciona la opción de menú *Window -> Open Perspective-> Other-> Modeling*. Pulta en la opción de menú *File-> New-> Example-> UML -> Travel Agency UML*. Este proyecto contiene un modelo UML para un negocio de agencia de viajes. El proyecto está formado por dos archivos: *agency.uml* (que contiene la definición no gráfica de los elementos del modelo) y *representations.aird* (contiene la representación gráfica de los elementos del modelo). Navega por los diagramas seleccionando los elementos bajo el archivo *representations.aird*.

6.3.2. ArgoUML

ArgoUML es una herramienta de modelado UML de código abierto basada en Java que incluye soporte para todos los diagramas UML 1.4 estándar. Se ejecuta en cualquier plataforma Java y está disponible en diez idiomas. Su instalación es sencilla, solo hay que descargar el archivo y seguir los pasos del instalador. Se distribuye bajo la *Licencia Pública Eclipse (EPL) 1.0*.

La ventana principal de ArgoUML tiene una barra de menús, barra de herramientas, y cuatro paneles principales (véase Figura 6.19):

- A la izquierda el panel de navegación donde tenemos una vista en forma de árbol de los diagramas del modelo.
- A la derecha el panel de edición donde se muestra el diagrama actual. Es en esta zona donde se arrastran y colocan los elementos del diagrama.
- Debajo a la izquierda el panel qué hacer, muestra una lista de tareas pendientes. Ayuda a los diseñadores a resolver problemas que vayan surgiendo.

- Por último, debajo y a la derecha se muestra el panel con los detalles del elemento seleccionado. Desde aquí se definen las propiedades del mismo, como por ejemplo su nombre.

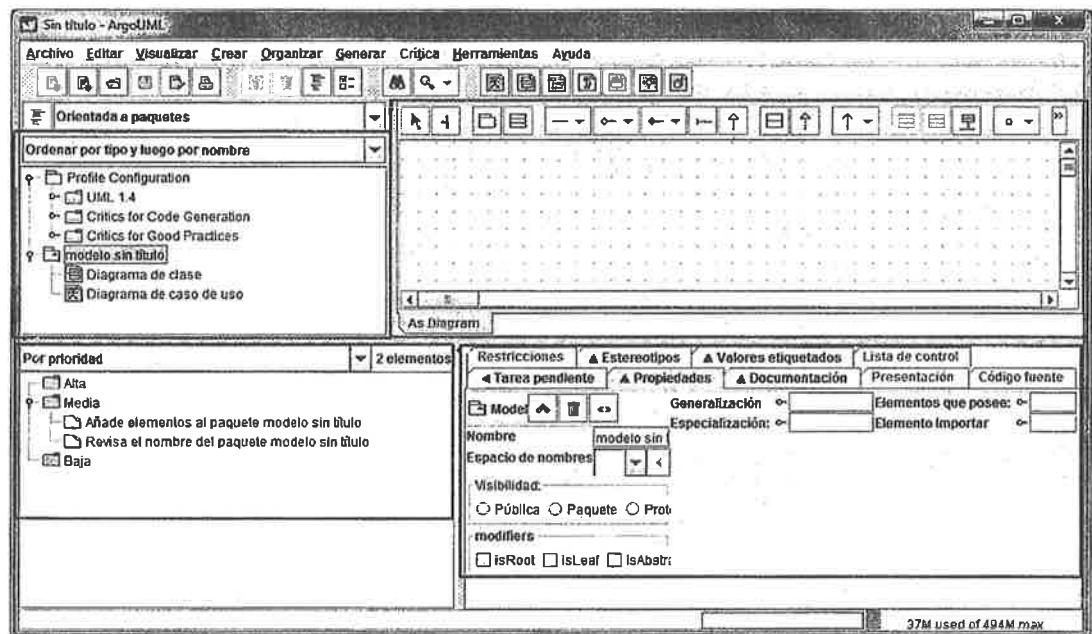


Figura 6.19. Pantalla principal de ArgoUML.

Por defecto al abrir ArgoUML en el modelo aparece un diagrama de clase y un diagrama de caso de uso. Al hacer clic en cualquiera de ellos, se muestra en el panel de edición los iconos para elaborar el diagrama de clases o de casos de uso (según el elegido). También podemos crear un caso de uso pulsando con el botón derecho del ratón sobre el modelo, a continuación elegimos *Crear diagrama -> Diagrama de casos de uso*. Véase Figura 6.20.

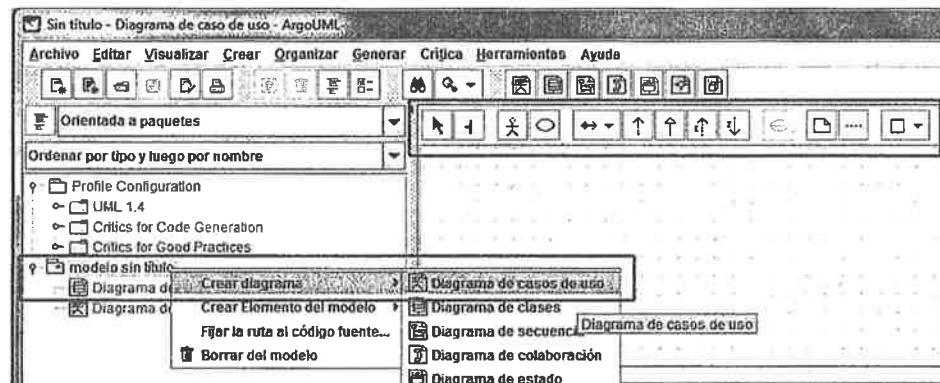


Figura 6.20. Crear diagrama de casos de uso en ArgoUML.

La Figura 6.21 muestra la pantalla de ArgoUML con el diagrama de casos de uso para el sistema de venta. Se muestra marcado el caso de uso *ComprarProductos* que añadía un punto de extensión. En el panel de navegación (a la izquierda) se muestran los elementos del diagrama. Algunos, como por ejemplo las relaciones de inclusión y asociación, no tienen nombre; este se puede asignar seleccionando el elemento en el diagrama y accediendo a la pestaña de propiedades.

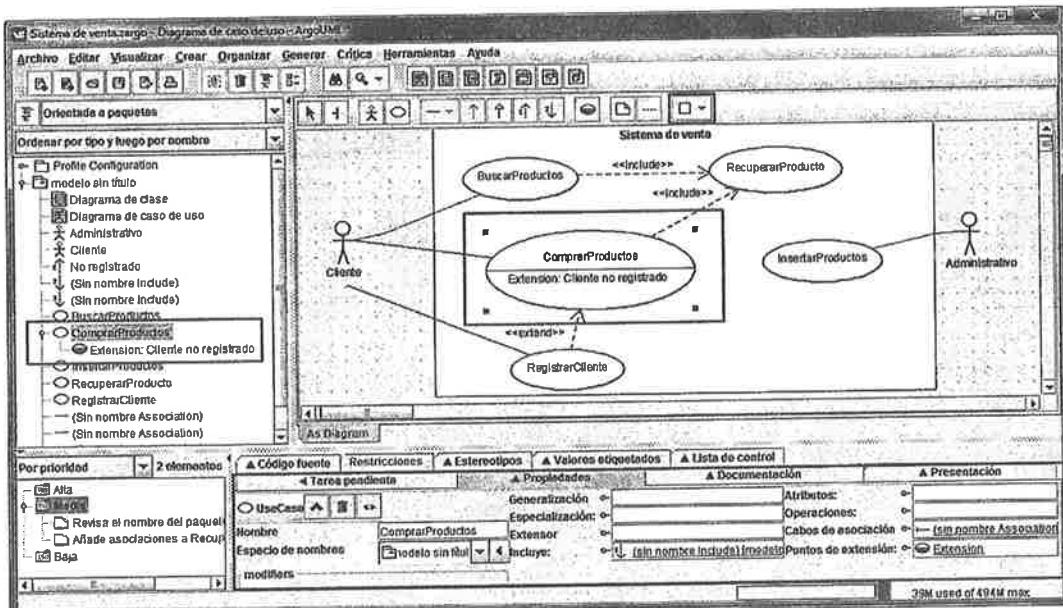


Figura 6.21. Sistema de venta en ArgoUML.

Para crear el límite del sistema y escribir el nombre usamos la herramienta que se muestra en la barra de botones donde están los elementos que se arrastran al diagrama.

Para crear la relación de asociación pulsamos en el botón *Asociación nueva* de la barra de botones y arrastramos del actor al caso de uso o viceversa. También se puede hacer teniendo seleccionado el actor y arrastrando desde el ícono (que se muestra a su derecha o izquierda) al caso de uso, véase Figura 6.22.

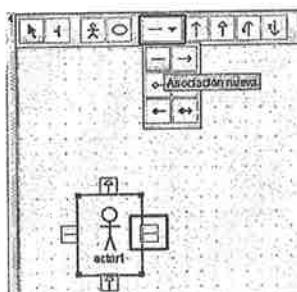


Figura 6.22. Relación de asociación en ArgoUML.

Para borrar un elemento del modelo, hemos de pulsar en él con el botón derecho del ratón desde el panel de navegación y seleccionar la opción *Borrar del modelo* **Borrar del modelo**. En un mismo modelo podemos crear varios diagramas.

Para crear un punto de extensión primero creamos la relación extiende sobre el caso de uso a ser extendido. Por ejemplo, en el sistema de venta pulsamos en el botón *Crear extensión* y arrastramos desde el caso de uso *RegistrarCliente* al caso de uso *ComprarProductos*. En el panel de navegación se añade dentro del caso de uso que se extiende un nuevo elemento con el ícono del punto de extensión y el nombre *newEP: loc*. Pulsando en él se accede a las propiedades donde podemos asignarle el nombre.

Para mostrar el punto de extensión se pulsa en el caso de uso con el botón derecho del ratón y se selecciona la opción *Mostrar-> Mostrar el comportamiento de los puntos de extensión*. Desde la opción *Agregar* se puede añadir un nuevo punto de extensión al caso de uso.

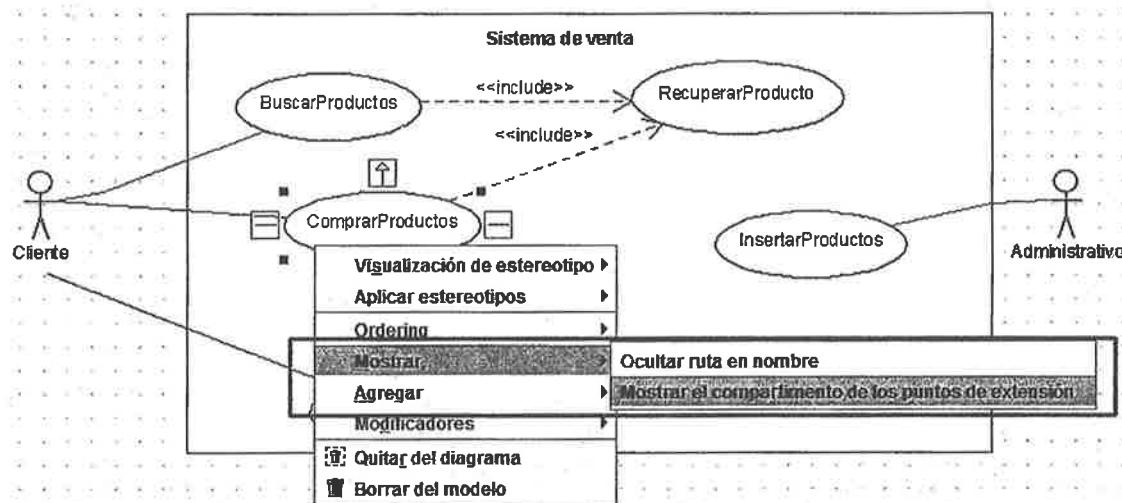


Figura 6.23. Visualizar los puntos de extensión en ArgoUML.

Se puede obtener una imagen del diagrama pulsando sobre él con el botón derecho del ratón y seleccionando la opción *Copiar diagrama al portapapeles como imagen*, véase Figura 6.24.

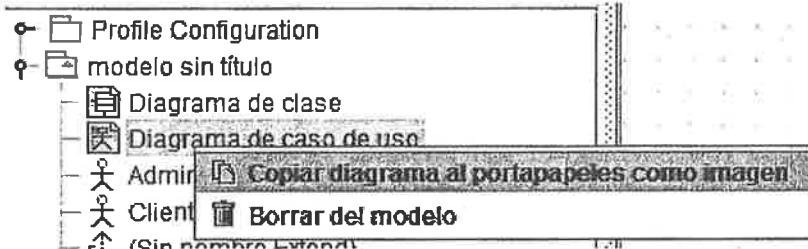


Figura 6.24. Copiar diagrama al portapapeles en ArgoUML.

ACTIVIDAD 6.6

Realiza en ArgoUML el diagrama de casos de uso para el sistema de venta y el de gestión de notas.

6.3.3. WhiteStarUML

WhiteStarUML es una herramienta de modelado de software que soporta UML. Ofrece once tipos diferentes de diagramas ya que acepta la notación UML 2.0. Es un proyecto de código abierto escrito en Delphi que se ejecuta en plataformas Win32. La instalación es fácil, solo hay que descargar el instalador desde la URL <http://sourceforge.net/projects/whitestaraml/> ejecutarlo y seguir los pasos. Deriva de StarUML.

Una vez finalizada la instalación se puede iniciar el programa. Al iniciar lo hemos de seleccionar la plantilla para nuestro proyecto, por defecto viene seleccionada la plantilla *Default Approach*, para empezar con un proyecto vacío seleccionamos *Empty Project* y pulsamos *OK*, véase Figura 6.25.

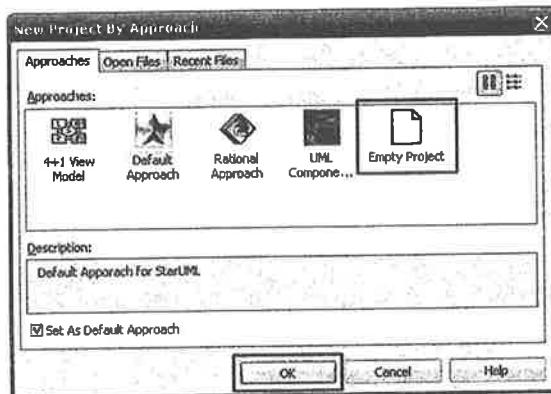


Figura 6.25. Crear un proyecto vacío en WhiteStarUML.

Lo primero que hay que hacer es añadir un modelo. Esto se puede hacer pulsando en el menú *Model-> Add-> Model*, o bien desde el *Model Explorer* se pulsa con el botón derecho del ratón en *Untitled* y se selecciona *Add -> Model*, véase Figura 6.26.

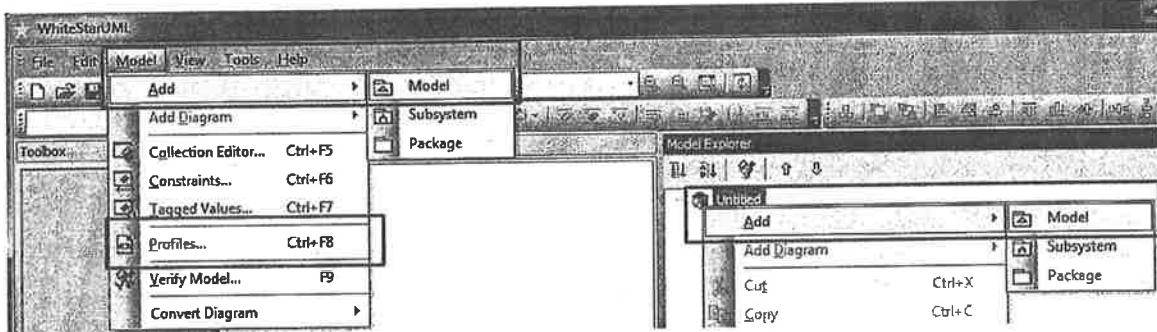


Figura 6.26. Crear un modelo en WhiteStarUML.

Desde la opción de menú *Model->Profiles* se establece el perfil que se utilizará en el proyecto, esto determina qué símbolos y convenciones se usarán en los diagramas. Hemos de incluir *Java 1.5 Profile* y *UML Standard Profile*. A continuación, para hacer un diagrama de casos de uso, se pulsa con el botón derecho del ratón sobre el modelo y se selecciona *Add Diagram -> Use Case Diagram*, véase Figura 6.27.

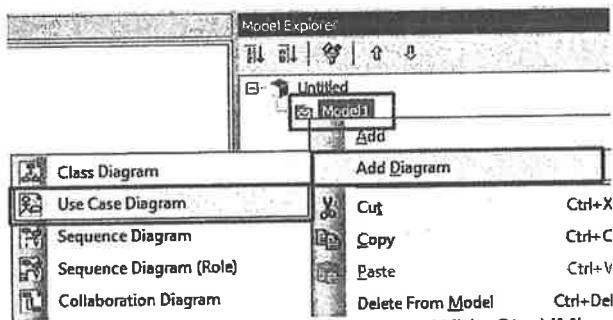


Figura 6.27. Crear un caso de uso en WhiteStarUML.

Automáticamente se muestra a la izquierda la barra de herramientas con los iconos para el diagrama de casos de uso y en la parte central la zona de edición donde incluiremos los elementos del diagrama. Se pulsa en el ícono de la barra de herramientas y seguidamente se pulsa

en la zona de edición, esto crearía el elemento con un nombre genérico, se cambia el nombre haciendo clic sobre él, véase Figura 6.28. A la derecha se muestra el *Model Explorer* con los elementos que se van añadiendo al diagrama y la ventana de propiedades del elemento seleccionado. Desde la propiedad *Name* se puede asignar nombre a los elementos del modelo.

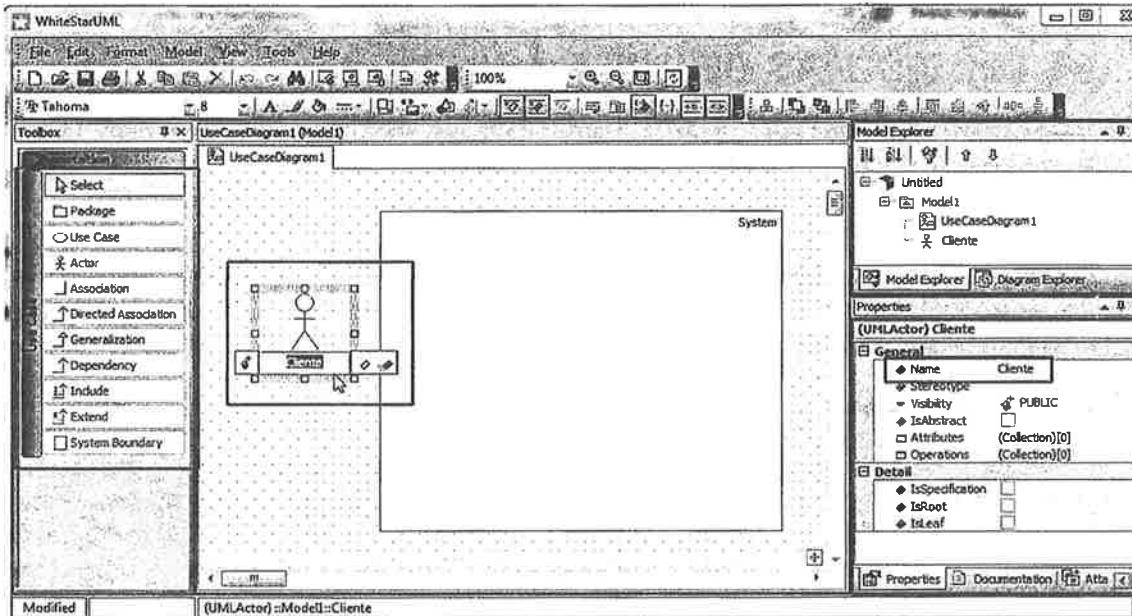


Figura 6.28. Barra de herramientas y edición del caso de uso en WhiteStarUML.

El botón *System Boundary* se utiliza para crear el límite del sistema. Para crear un punto de extensión, se selecciona el caso de uso y se pulsa con el botón derecho del ratón sobre él. Del menú contextual que se muestra se selecciona *Add-> Extension Point* (véase Figura 6.29). Desde el *Model Explorer* damos nombre al punto de extensión, por ejemplo, podemos poner *Cliente no registrado*.

Para eliminar elementos del modelo se pulsa sobre él con el botón derecho del ratón desde el *Model Explorer* y se selecciona la opción *Delete From Model* del menú contextual. Para guardar el proyecto pulsamos en la opción de menú *File->Save* o *File Save As* y seleccionamos la ubicación para guardar el proyecto.

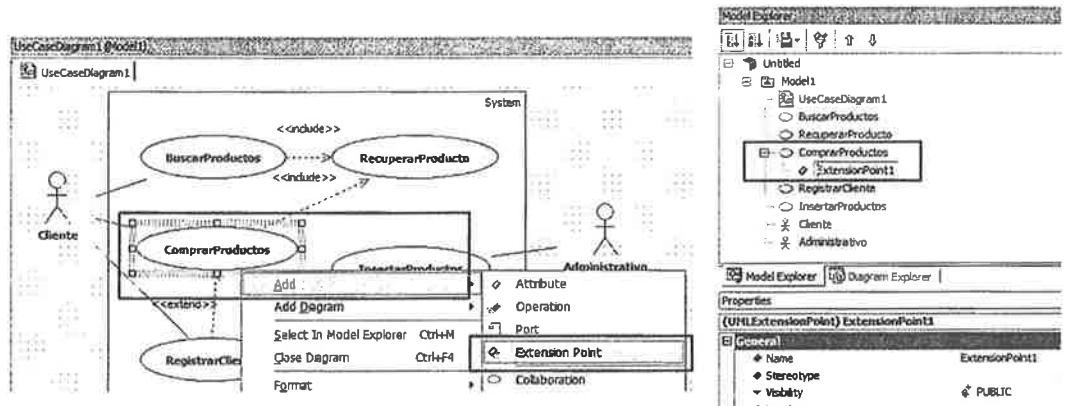


Figura 6.29. Punto de extensión en WhiteStarUML.

ACTIVIDAD 6.7

Realiza en WhiteStarUML el diagrama de casos de uso para el sistema de venta y el de gestión de notas.

6.4. DIAGRAMAS DE INTERACCIÓN

El diagrama de interacción describe el intercambio de secuencias de mensajes entre las partes de un sistema. Proporciona una visión integral del comportamiento de un sistema, es decir, muestra el flujo de control a través de varios objetos.

Dentro de la categoría de diagramas de interacción tenemos dos diagramas que se centran en aspectos distintos: el diagrama de secuencia (centrado en la ordenación temporal de los mensajes) y el diagrama de comunicación o colaboración (centrado en la organización estructural de los objetos que envían y reciben mensajes).

6.4.1. Diagrama de secuencia

El **diagrama de secuencia de sistema** muestra gráficamente los eventos que fluyen de los actores al sistema. Para su elaboración se parte de los casos de uso elaborados durante la etapa de análisis. Un diagrama de secuencia tiene dos dimensiones: la dimensión vertical que representa el tiempo y la dimensión horizontal que representa los roles que participan en la interacción. La Figura 6.30 muestra el diagrama de secuencia de sistema para el caso de uso *BuscarProductos*.

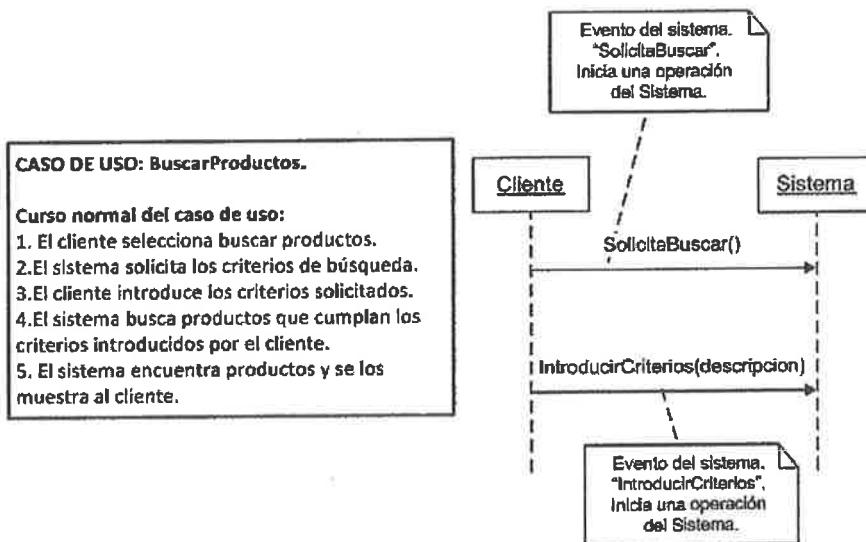


Figura 6.30. Diagrama de secuencia de sistema para el caso de uso *BuscarProductos*.

Cada rol, ya sea actor u objeto, se representa mediante un rectángulo distribuido horizontalmente en la zona superior del diagrama. A cada uno se le asocia una línea vertical llamada **línea de vida**, donde se describe la interacción a lo largo del tiempo. De cada línea vertical salen diferentes mensajes representados mediante flechas que muestran la interacción, encima de la flecha se muestra el mensaje. Es importante la secuencia de los mensajes en esta línea ya que indican el orden en que ocurren los eventos. A menudo el diagrama de secuencia va acompañado de la descripción del curso normal de eventos del caso de uso.

En la Figura 6.30 se muestran horizontalmente los roles que participan en la interacción: el cliente y el sistema, con sus líneas de vida (algunas herramientas gráficas, como WhiteStarUML, utilizan para representar el actor, el símbolo del monigote en lugar del rectángulo). Se muestran dos mensajes, que parten del cliente (están señalizados con una nota). El primer mensaje (también le podemos llamar evento), *SolicitaBuscar()*, inicia la operación de búsqueda de productos en el sistema. El segundo, *IntroducirCriterios(descripción)*, inicia la operación de búsqueda de productos según los criterios especificados, en este caso se ha considerado una búsqueda de productos por descripción.

Los principales elementos en un diagrama de secuencia son los siguientes:

SÍMBOLO	FUNCIÓN	NOTACIÓN
Marco	Se utiliza para dar un borde visual al diagrama de secuencia. A la izquierda del marco se escribe la etiqueta sd seguida de un nombre	sd Ejemplo
Línea de vida	Representa a un participante durante la interacción. Normalmente contiene un rectángulo con el nombre del objeto y una línea punteada. Algunas veces un diagrama de secuencia tendrá una línea de vida con un símbolo de actor en la parte superior	Objeto1 Cliente
Actor	Representa el papel desempeñado por un usuario	
Mensaje	Mensaje síncrono	
	Mensaje asíncrono	
	Mensaje de retorno	
Activación	Son opcionales. Representan el tiempo durante el que se ejecuta una función. Se suelen poner cuando está activo un método, ya sea porque está efectuando operaciones o porque se encuentra esperando la devolución de otro método	

Los mensajes representan la comunicación entre los participantes. Se dibujan como flechas dirigidas desde el participante que lo envía hasta el que lo ejecuta. Se etiquetan con un nombre acompañado o no de parámetros. Pueden ser de varios tipos:

- **Mensaje síncrono:** su comportamiento es el siguiente: cuando se envía un mensaje a un objeto, no se recibe el control hasta que el objeto receptor ha finalizado la ejecución.

- **Mensaje asíncrono:** representa flujo de control asíncrono. En este caso, quien envía un mensaje asíncrono continúa con su trabajo después de enviado, es decir, no necesita esperar la finalización del mismo en el objeto receptor. Este tipo de mensaje se representa mediante una flecha con la punta abierta o con media punta de flecha abierta. Se utiliza en sistemas multihilo donde se producen procesos concurrentes.
- **Mensaje de retorno:** representa un mensaje de confirmación. Su uso es opcional. Se representa mediante una flecha punteada.

La Figura 6.31 muestra el diagrama de secuencia de sistema del caso de uso *BuscarProductos* con activaciones y mensajes de retorno. Tanto las activaciones como los mensajes de retorno son opcionales, conviene no abusar de ellos y utilizarlos solo para aumentar la claridad del diagrama.

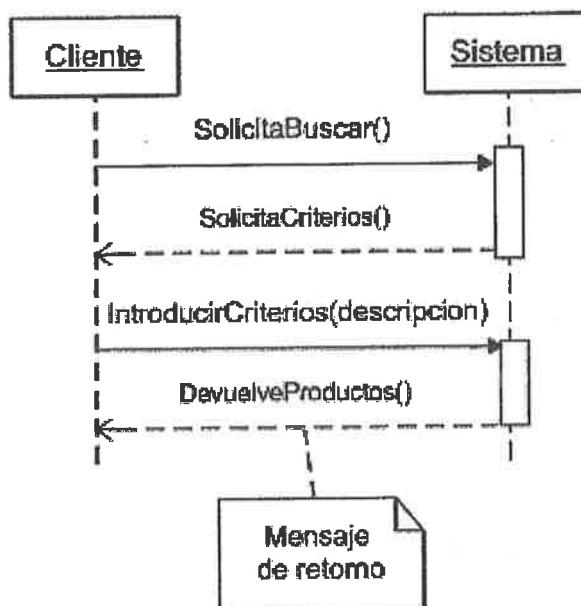


Figura 6.31. Diagrama de secuencia con activaciones y mensajes de retorno.

En un diagrama de secuencia, los mensajes que se repiten se pueden incluir dentro de un marco. Por ejemplo, el curso normal del diagrama de casos de uso *ComprarProductos* se puede representar como se muestra en la Figura 6.32. Se encierra en una caja, etiquetada con **loop**, los eventos de recuperar el producto y escribir la cantidad del producto que el cliente desea comprar. Se escribe debajo la condición del bucle entre corchetes, para el ejemplo se ha etiquetado como *[Más productos]*, indicando que se repita mientras el cliente quiera comprar más productos.

Más adelante se mostrará como representar bucles y alternativas en los diagramas de secuencia utilizando una extensión denominada **fragmento combinado**; estos utilizarán una palabra clave o etiqueta (**loop**, **opt**, **alt**) para indicar el tipo.

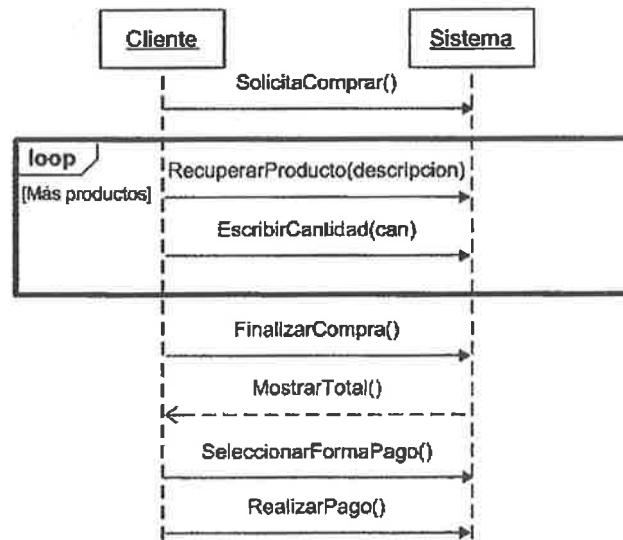


Figura 6.32. Diagrama de secuencia de sistema del caso de uso *ComprarProductos*.

En los ejemplos vistos hasta ahora se ha utilizado el **diagrama de secuencia de sistema** para describir dentro de un caso de uso los **eventos que fluyen desde el cliente hacia el sistema**. Solo se incluía en la dimensión horizontal esos dos roles. En un sistema orientado a objetos, donde tenemos varias clases y varios objetos que se comunican unos con otros, las clases y los objetos se representan en la dimensión horizontal.

Existe una nomenclatura a la hora de nombrar los rectángulos que representan los objetos: **nombre:tipo**. Tanto el nombre como el tipo pueden omitirse, pero no ambos a la vez. El nombre representa el objeto dentro de la interacción, el tipo es el tipo de objeto (es decir, la clase). Ejemplos:

Una clase <i>Empleado</i>	<i>Empleado</i>
<i>emp</i> es un objeto de la clase <i>Empleado</i>	<i>emp:Empleado</i>
Un objeto cualquiera de la clase <i>Empleado</i>	<i>:Empleado</i>

Ejemplo 13. En el análisis del sistema de venta se han diseñado varias clases, una clase para datos de cliente, otra clase para datos de un producto, una clase para representar una venta y otra clase para representar una línea de venta. Un cliente puede tener muchas ventas, una venta puede tener muchas líneas de venta y cada línea de venta tiene un producto. Para representar que una venta puede tener muchas líneas de venta se define un atributo *ArrayList* en la clase *Venta*: *ArrayList<LineaVenta> lineas*. Estas clases se agrupan en el paquete *Datos*. El diagrama y sus relaciones se muestran en la Figura 6.33.

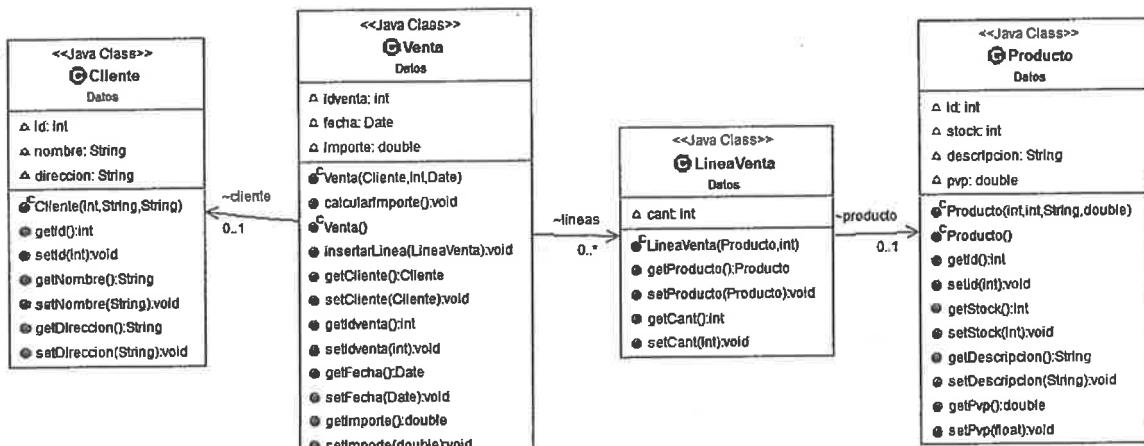


Figura 6.33. Diagrama de clases del paquete Datos del sistema de venta.

Todas ellas definen métodos get y set para acceder a los datos. Para la gestión de productos se ha diseñado otra clase llamada *GestorProductos* (incluida en el paquete *Gestion*) con las operaciones a realizar sobre los productos: recuperar, insertar, eliminar y modificar un producto, actualizar las existencias de un producto y obtener una lista con los identificadores de todos los productos, véase Figura 6.34.

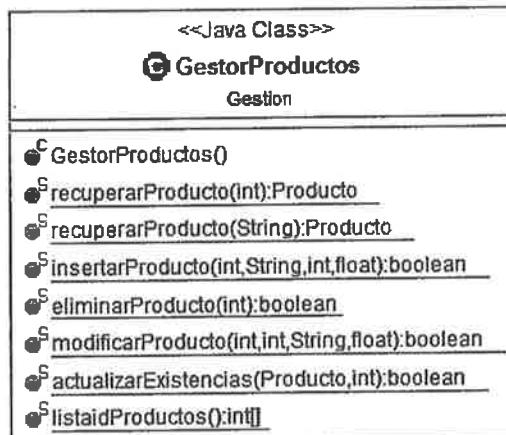


Figura 6.34. Clase GestorProductos.

El aspecto de esta clase es el siguiente:

```

package Gestion;
import Datos.Producto;
public class GestorProductos {
    public static Producto recuperarProducto(int id) {
        .....
        return producto;
    }

    public static Producto recuperarProducto(String descripcion) {
  
```

```
    .....
    return producto;
}

public static boolean insertarProducto (int id, String descripcion,
        int stock, float pvp) {
    .....
    return valor;
}

public static boolean eliminarProducto(int id) {
    .....
    return valor;
}

public static boolean modificarProducto(int id, int stock,
        String descripcion, float pvp) {
    .....
    return valor;
}

public static boolean actualizarExistencias(Producto p,
        int cantidad) {
    int cant = p.getStock() - cantidad;
    p.setStock(cant);
    return true;
}

public static int[] listaIdProductos () {
    .....
    return valor;
}
```

Se define también una clase *Pantalla* que constituye la interfaz gráfica desde la que el cliente se comunica con el sistema. El diagrama de secuencia correspondiente al caso de uso *BuscarProductos* es el mostrado en la Figura 6.35. Muestra lo siguiente:

- El cliente escribe la descripción del producto que desea buscar en la pantalla gráfica (objeto de la clase *Pantalla*)
 - El objeto pantalla envía el mensaje *recuperarProducto(desc)* a la clase *GestorProductos* pidiéndola obtener los datos del producto cuya descripción se envía. Esta operación devuelve un objeto de la clase *Producto*, esto se indica en el rectángulo etiquetado como *p.Producto*. En este caso *GestorProductos* es una clase, no un objeto, entonces *recuperarProducto()* es un método estático.
 - El obteto pantalla envía los mensajes *getId()*, *getPvp()* y *getStock()* al objeto *p.Producto* solicitando los datos del producto, el identificador, el pvp y el stock.

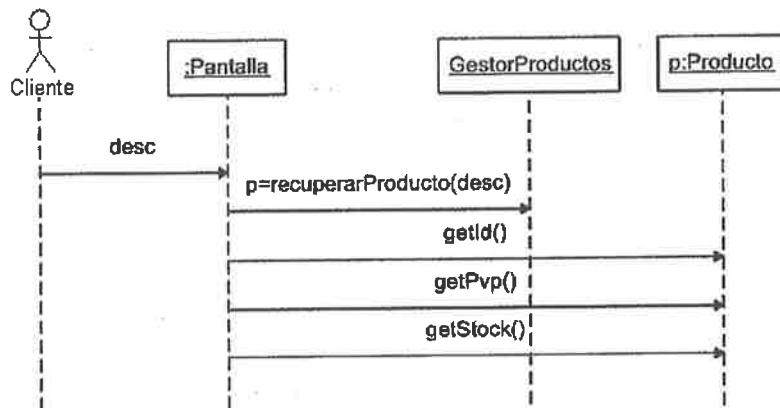


Figura 6.35. Diagrama de secuencia del Ejemplo 13.

CREACIÓN Y DESTRUCCIÓN DE OBJETOS

En un diagrama de secuencia se puede mostrar la creación y destrucción de objetos. La creación se representa mediante un mensaje que termina en el objeto que va a ser creado. El mensaje puede llevar la identificación «create». La destrucción del objeto da lugar a la finalización de su línea de vida, se denota mediante una X grande en su línea de vida. El mensaje puede llevar la identificación «destroy». Las Figuras 6.36 y 6.37 muestran un ejemplo de creación y destrucción de objetos.

En la Figura 6.36 se muestra el diagrama de secuencia correspondiente al caso de uso de *InsertarProductos*. El administrativo escribe los datos del producto en la pantalla gráfica (objeto de la clase *Pantalla*). El objeto pantalla envía el mensaje *insertarProducto(id, desc, pvp, stock)* al gestor de productos (que se encarga de gestionar todas las operaciones para la gestión de productos) con los datos del producto a insertar. Desde el gestor de productos se creará el objeto *Producto* con los datos enviados por el objeto *Pantalla*.

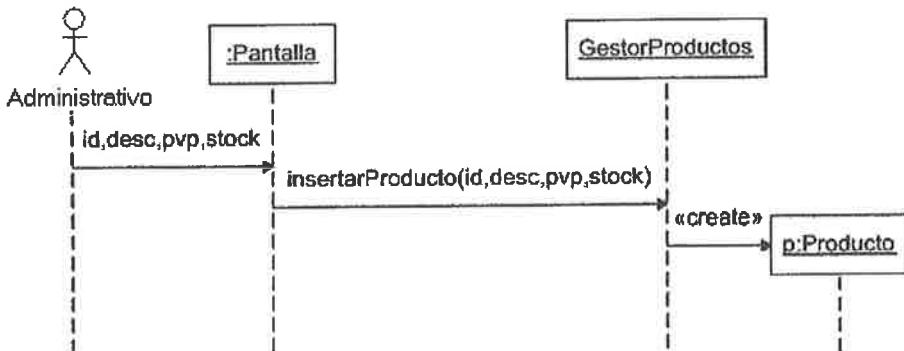


Figura 6.36. Creación de un objeto.

En la Figura 6.37 se muestra el diagrama de secuencia correspondiente al caso de uso de eliminar un producto. El administrativo escribe el identificador del producto a eliminar en la pantalla gráfica. El objeto pantalla envía el mensaje *eliminarProducto(id)* al gestor de productos con el identificador del producto a eliminar. Desde el gestor de productos se recupera el objeto *Producto* y se elimina incluyendo la X en su línea de vida.

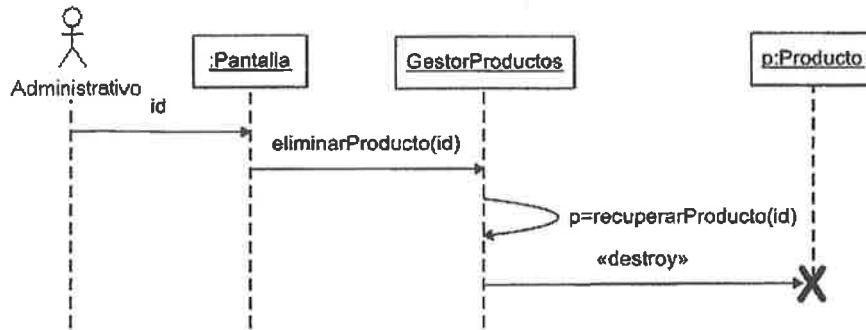


Figura 6.37. Destrucción de un objeto.

En la línea de vida de la clase *GestorProductos* se muestra un mensaje enviado a sí mismo (es decir, dentro de la clase *GestorProductos* se hace una llamada a la operación *recuperarProducto(id)*), a este mensaje se le llama **autodelegación** o **mensaje reflexivo**. La **autodelegación** es un mensaje que un objeto se envía a sí mismo regresando la flecha del mensaje de vuelta a la misma línea de vida.

Cuando se programa en Java normalmente no se destruye explícitamente un objeto, ya que el recolector de basura se encarga de esta acción. Sin embargo, hay veces que queremos resaltar lo que hacemos con el objeto.

ALTERNATIVAS Y BUCLES

En los diagramas de secuencia podemos introducir extensiones para dar soporte a los bucles y a las alternativas. A estas extensiones se les llama **fragmentos combinados**, hay varios tipos, entre ellos están las alternativas y los bucles.

Ya vimos en el caso de uso *ComprarProductos* como se podían representar, encerrados en un marco o caja, los eventos que se repiten. En UML podemos modelar varios tipos de alternativas:

- Utilizando el operador **opt** seguido de una condición, si la condición se cumple, el contenido del marco se ejecuta. Véase Figura 6.38.
- Utilizando el operador **alt**, este va seguido de varias condiciones y al final la palabra clave **else**. El marco se divide en varias zonas, dependiendo de las condiciones que haya, cuyo contenido se ejecuta si se cumple la condición asociada. La parte **else** se ejecuta si no se cumplen las condiciones. Se utiliza para modelar estructuras *if...then...else*.

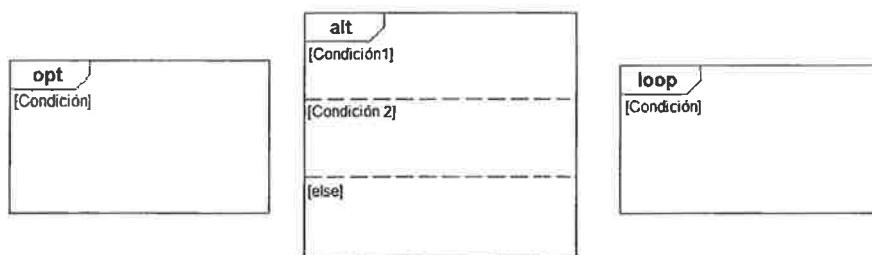


Figura 6.38. Marcos que representan alternativas y bucles.

Para representar un bucle se utiliza el operador **loop** seguido de una condición. Lo encerrado en el marco se ejecutará mientras se cumpla la condición. En la Figura 6.39 se muestra un diagrama de secuencia con un bucle y una alternativa. En este caso, el objeto pantalla manda el

mensaje `listaidProductos()` a la clase `GestorProductos` para obtener una lista con todos los identificadores de productos. En el bucle, para cada identificador de producto que se encuentre en la lista se obtiene un objeto `Producto`, y si el stock del producto es mayor que 100 se muestran sus datos.

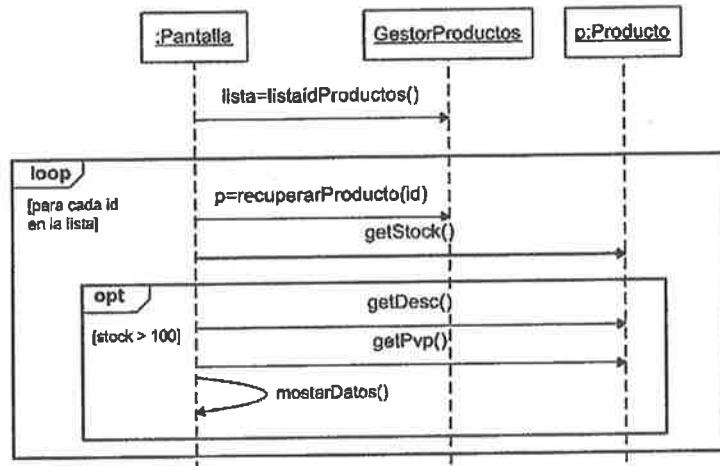


Figura 6.39. Diagrama de secuencia con bucle y alternativa.

Ejemplo 14. El siguiente ejemplo muestra el diagrama de secuencia considerando las clases y objetos que intervienen en el caso de uso *ComprarProductos*, véase Figura 6.41. Para la gestión de compras de productos se ha diseñado la clase llamada `GestorVentas` (incluida en el paquete `Gestion`) con las operaciones a realizar sobre las compras de productos: iniciar el proceso de venta, obtener el identificador de la venta, insertar los datos de la línea de venta en la venta, finalizar la venta y realizar el pago, véase Figura 6.40.

En Java la clase `GestorVentas` tiene el siguiente aspecto:

```

package Gestion;
import Datos.*;
public class GestorVentas {
    static Venta v;
    public static void iniciarVenta(Cliente c) {
    }

    private static int obtenerIdventa() {
        .....
        return valor;
    }

    public static void datosLinea(int id, int cantidad) {.....}

    public static void finalizarVenta()
    {
        v.calcularImporte();
        realizarPago();
    }

    private static void realizarPago() {.....}
}
  
```

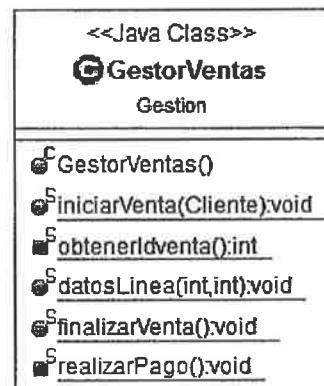


Figura 6.40. Clase GestorVentas.

El diagrama de secuencia se muestra en la Figura 6.41. Muestra lo siguiente:

- Desde el objeto pantalla se envía el mensaje *iniciarVenta(Cliente)* a la clase *GestorVentas*.
- Desde esta clase se envía un mensaje a sí misma, *obteneridVenta()*, para obtener el identificador de la venta.
- Una vez obtenido el identificador se crea un objeto *Venta*.
- Se encierra en un marco etiquetado con **loop** el conjunto de operaciones que se van a repetir por cada producto que deseé comprar el cliente.
- El objeto pantalla envía el mensaje *datosLinea(id, cantidad)* al gestor de ventas, con el identificador de producto y la cantidad deseada.
- Desde la clase *GestorVentas* se envía el mensaje *recuperarProducto(id)* a la clase *GestorProductos* solicitando los datos del producto cuyo identificador se envía. La operación devuelve un objeto de la clase *Producto*.
- Desde la clase *GestorVentas* se envía el mensaje *getStock()* al objeto producto recuperado para obtener el stock actual del producto.
- Se encierra en un marco etiquetado con **opt** el conjunto de operaciones que se realizan si hay stock en el producto solicitado.
- Si hay stock el gestor de ventas envía el mensaje *actualizarExistencias(id, cant)* al gestor de productos para que actualice el stock. Después crea un objeto *LíneaVenta* con los datos del producto y la cantidad. Por último envía el mensaje *insertarLinea(lin)* al objeto venta creado inicialmente para añadir una nueva línea a la venta.
- Al finalizar el proceso repetitivo, cuando el cliente no desea comprar más productos, desde el objeto pantalla se envía el mensaje *finalizarVenta()* al gestor de ventas.
- Entonces el gestor de ventas envía el mensaje *calcularImporte()* al objeto venta creado.
- Por último, desde la clase *GestorVentas* se envía el mensaje de autodelegación *realizarPago()*.

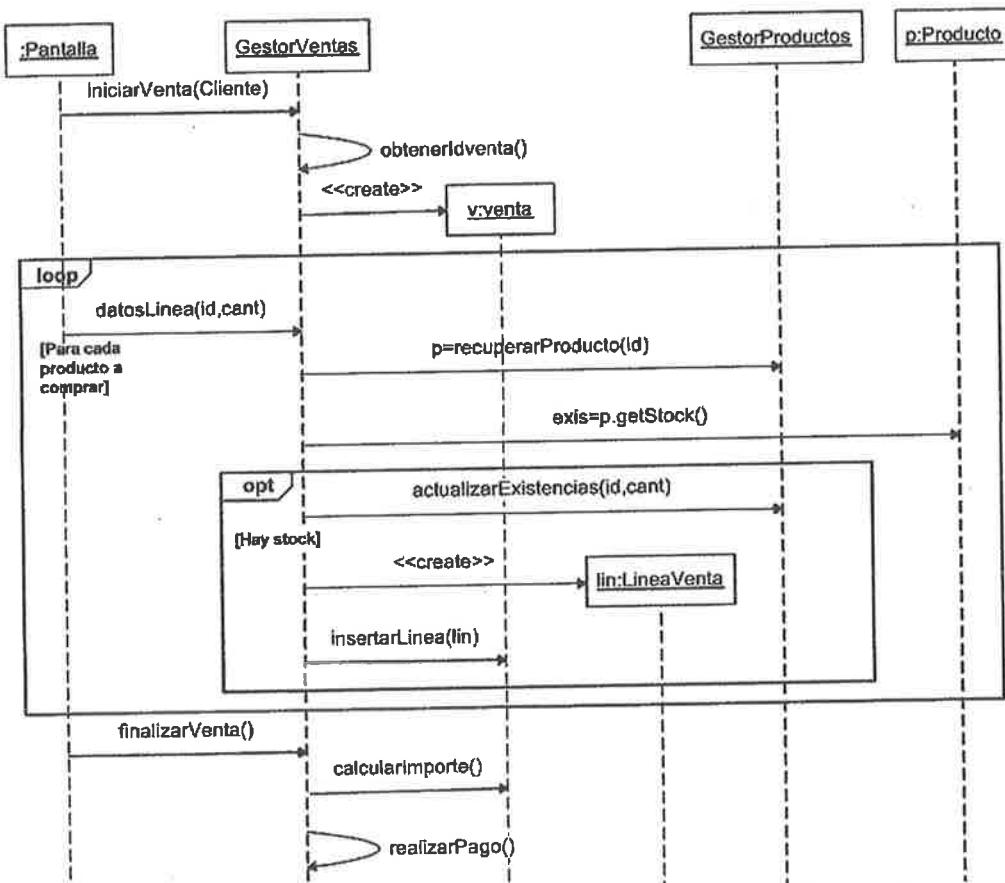


Figura 6.41. Diagrama de secuencia con clases y objetos, caso de uso ComprarProductos.

Podemos imaginar que desde el objeto :Pantalla se realizan las siguientes llamadas a las operaciones de la clase GestorVentas:

```

GestorVentas.iniciarVenta(Cliente);
while (el cliente quiera comprar productos) {
    //introducir datos del producto a comprar
    GestorVentas.datosLinea(id de producto, cantidad);
}
GestorVentas.finalizarVenta();

```

6.4.2. Diagrama de comunicación o colaboración

El diagrama de colaboración muestra los objetos junto con los mensajes que se envían entre ellos. Representa la misma información que el diagrama de secuencia, de hecho muchas de las herramientas UML permiten convertir el diagrama de secuencia en un diagrama de colaboración y viceversa.

A diferencia del diagrama de secuencia, centrado en el orden a través del tiempo en que ocurren los mensajes; el diagrama de colaboración se centra en el contexto y la organización general de los objetos que envían y reciben mensajes. El diagrama de objetos muestra los objetos y sus relaciones, el de colaboración es una extensión a este, además de la relación entre los objetos muestra los mensajes que se envían entre sí.

Un diagrama de colaboración es como un grafo con los siguientes elementos:

SÍMBOLO	FUNCIÓN	NOTACIÓN
Objetos o roles	Se representan mediante un rectángulo que contiene el nombre y la clase del objeto en el siguiente formato: <i>objeto:Clase</i> . Se puede omitir el nombre del objeto o la clase pero no ambos a la vez	<code>objeto:Clase</code>
Enlaces	Son los arcos del grafo que conectan los objetos. En un mismo enlace se pueden mostrar muchos mensajes, pero cada uno de ellos con un número de secuencia único	_____
Mensajes	Se representan mediante una flecha dirigida con un nombre y un número de secuencia	1: Mensaje() →
Número de secuencia	Indica el orden de un mensaje dentro de la interacción. Comienza en 1 y se incrementa en una unidad por cada nuevo mensaje en el flujo de control. El mensaje que inicia el diagrama no lleva número de secuencia	
Iteración	Se representa colocando un * después del número de secuencia y una condición encerrada entre corchetes. Ejemplo: 2*[para cada identificador de la lista]:prepara()	
Alternativa	Las alternativas se indican con condiciones entre corchetes, por ejemplo: 2 [condición]: operación(). Los caminos alternativos tendrán el mismo número de secuencia, seguido del número de subsecuencia	
Anidamiento	Se puede mostrar el anidamiento de mensajes con números de secuencia y subsecuencia. Por ejemplo 2.1, significa que el mensaje con número de secuencia 2 no acaba de ejecutarse hasta que no se han ejecutado todos los 2. X	

Para la numeración de los mensajes se pueden usar varios esquemas:

- Numeración simple: empieza en 1, se va incrementando en 1 y no hay ningún nivel de anidamiento; véanse Figuras 6.42 y 6.44.
- Numeración decimal: se muestran varios niveles de subíndices para indicar anidamiento de operaciones. Por ejemplo, 1 es el primer mensaje; 1.1 es el primer mensaje anidado en el mensaje 1, 1.2 es el segundo mensaje anidado en el mensaje 1; y así sucesivamente; véanse Figuras 6.43 y 6.45.

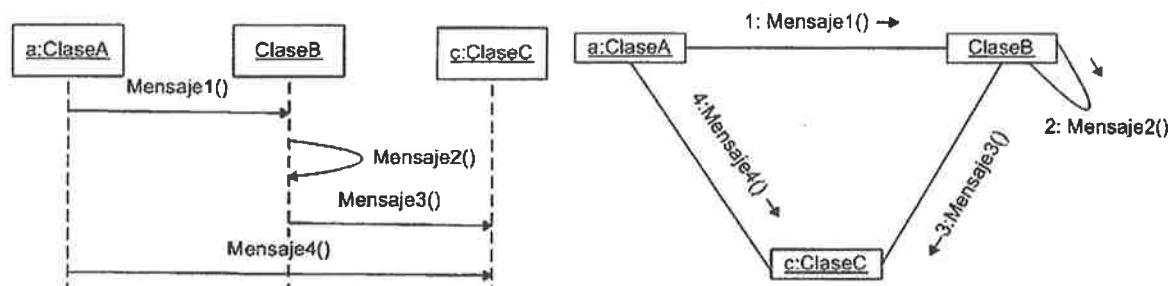


Figura 6.42. Diagrama de secuencia y diagrama de colaboración con numeración simple.

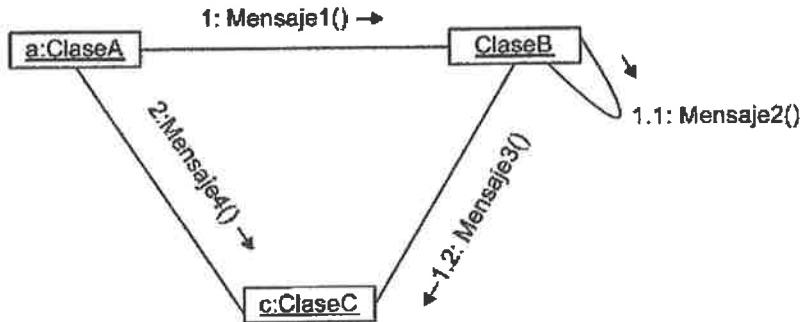


Figura 6.43. Diagrama de colaboración con numeración decimal.

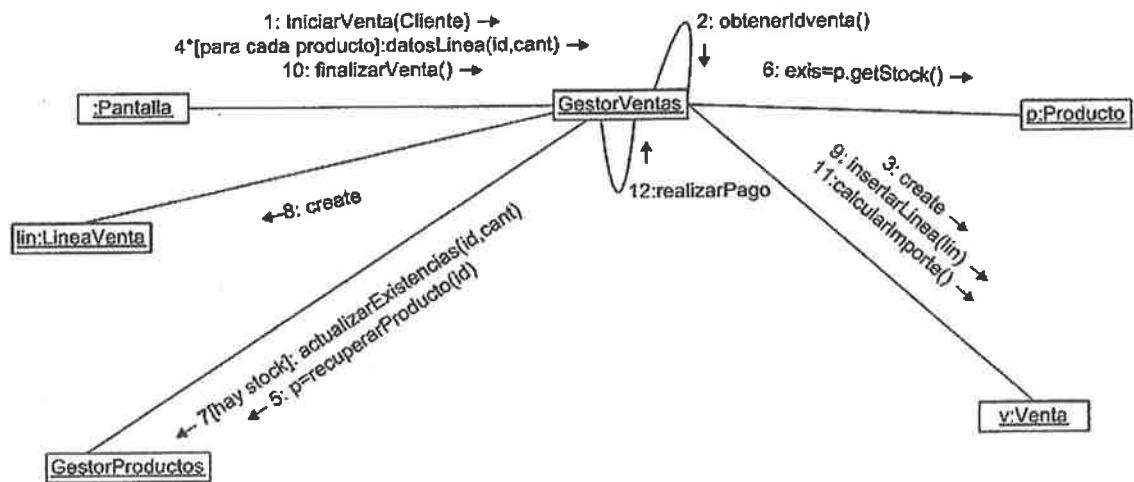


Figura 6.44. Diagrama de colaboración del caso de uso ComprarProductos con numeración simple.

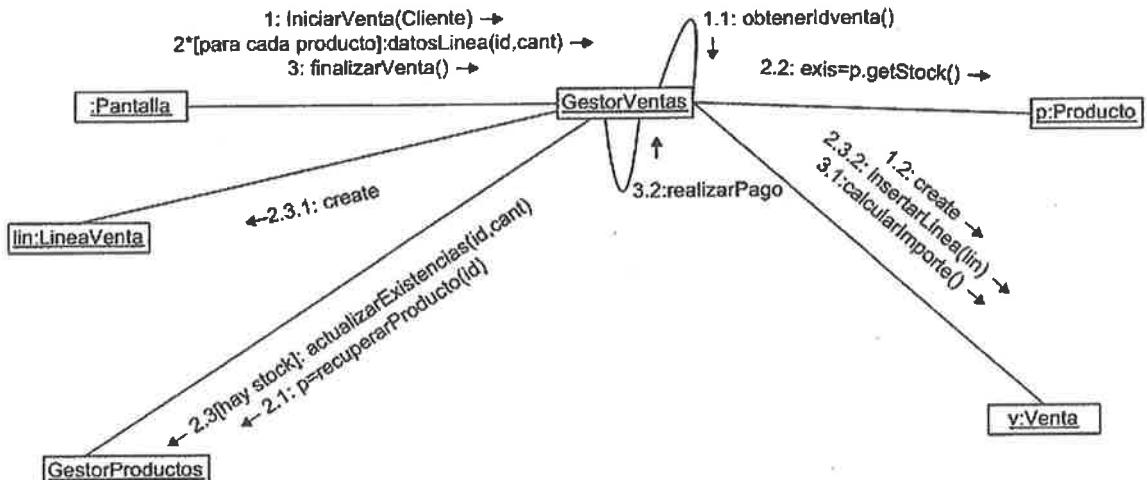


Figura 6.45. Diagrama de colaboración del caso de uso ComprarProductos con numeración decimal.

En las Figuras 6.44 y 6.45 se muestran diferentes esquemas para la numeración de los mensajes, quizás en la figura con numeración decimal resulte más difícil apreciar la secuencia general de los mensajes que en la figura con numeración simple.

6.4.3. Elaboración de diagramas de interacción

A continuación vamos a ver cómo construir los diagramas de interacción usando las herramientas vistas anteriormente.

ECLIPSE

Para crear un diagrama de secuencia en Eclipse, pulsamos con el botón derecho del ratón en *<Model>* y seleccionamos *New Representation -> Sequence Diagram*, véase Figura 6.46. Escribimos un nombre o dejamos el nombre por defecto y pulsamos el botón *OK*.

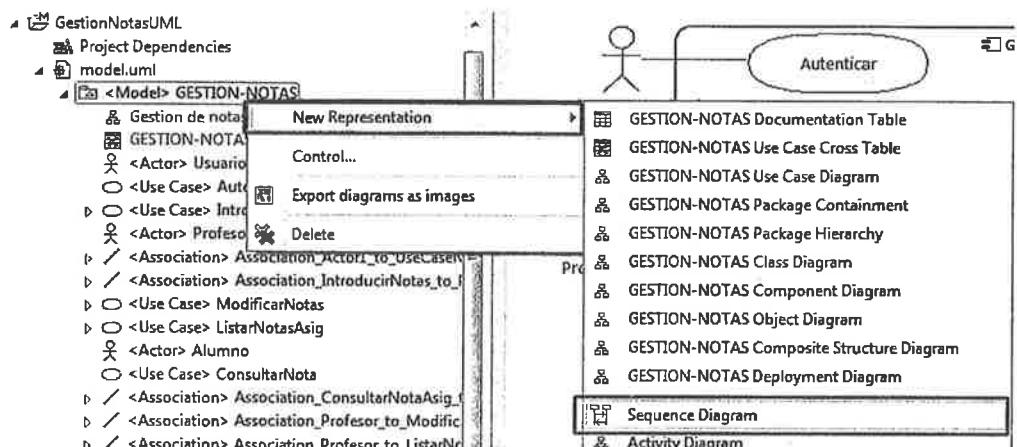


Figura 6.46. Creación de un diagrama de secuencia en Eclipse.

Se observa a la derecha la paleta con los elementos que se pueden añadir al diagrama. Desde la carpeta *Existing Elements* se pueden elegir los actores o clases que tengamos definidos en el modelo. Se pulsa en *Actor*, a continuación se hace clic en la pantalla de edición, entonces se muestran los elementos del modelo; se elige un actor o una clase y se pulsa *Finish*. Automáticamente se muestra en el área de edición el actor o la clase con su línea de vida, véase Figura 6.47.

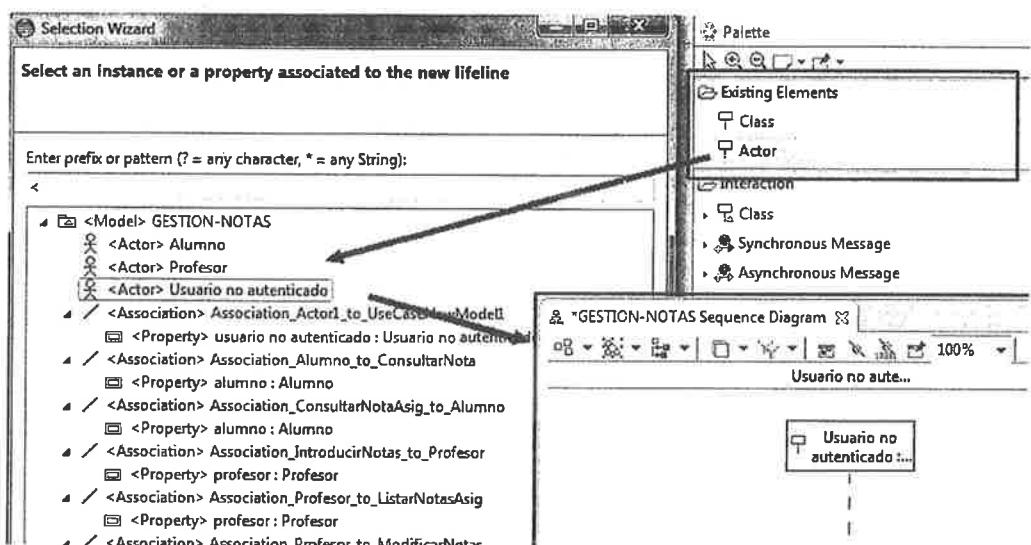


Figura 6.47. Creación de un diagrama de secuencia en Eclipse.

Para añadir un elemento (clase o actor) que no esté en el modelo pulsamos en *Class* o *Actor* desde el bloque de *Interaction*, y a continuación pulsamos en el área de edición, véase Figura 6.48.

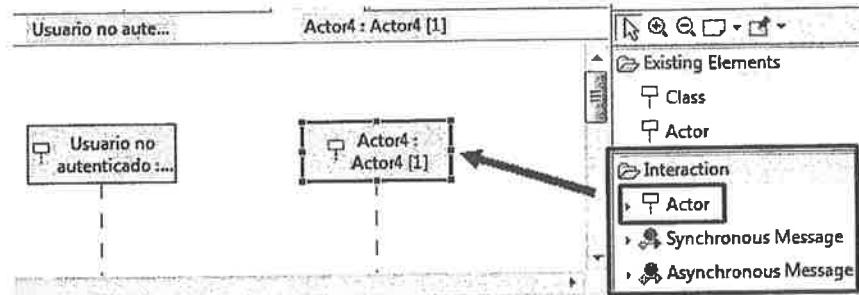


Figura 6.48. Añadir en Eclipse un elemento al diagrama que no forma parte del modelo.

Para añadir los mensajes entre los elementos del diagrama pulsamos en *Synchronous Message* y a continuación en *Synchronous Message and Operation Creation*; y arrastramos desde la línea de vida del emisor al receptor, véase Figura 6.49. Se creará un nuevo mensaje (Figura 6.50) o una nueva operación. Si en el receptor hay definidas operaciones (por ejemplo, cuando se trata de un objeto de una clase), pulsamos en *Synchronous Message*, al arrastrar desde la línea de vida del emisor al receptor se muestra la lista de operaciones del objeto receptor para elegir el que nos interese (más adelante se mostrará un ejemplo); en este caso la opción *Synchronous Message and Operation Creation* crearía una nueva operación en el receptor.

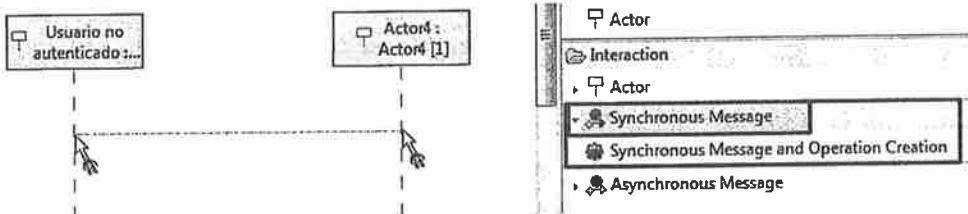


Figura 6.49. Añadir mensajes en Eclipse.

Se crea el mensaje entre el emisor y el receptor (*Message_0*). El mensaje se compone de tres elementos: el mensaje de envío (que es la ocurrencia de envío del mensaje), el mensaje en sí mismo y el mensaje recibido (ocurrencia de recepción del mensaje). También se crea la activación y el mensaje de retorno (*Message_0_reply*). Desde el *Model Explorer* y desde las propiedades del mensaje se pueden apreciar los elementos que se generan por cada mensaje. Véase Figura 6.50.

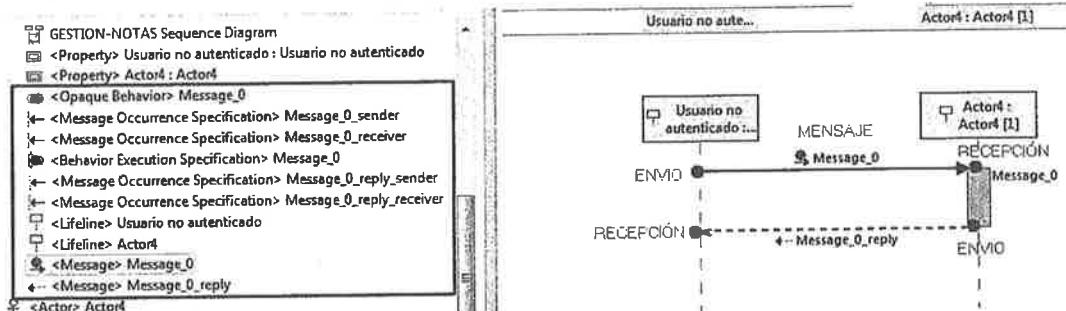


Figura 6.50. Mensajes generados en Eclipse entre emisor y receptor.

Por ejemplo para el mensaje *Message_0* se generan los siguientes elementos:

- El mensaje propiamente dicho: <Message> *Message_0*.
- El mensaje de recepción: <Message Occurrence Specification> *Message_0_receiver*.
- El mensaje de envío: <Message Occurrence Specification> *Message_0_sender*.

Para el mensaje *Message_0_reply* se generan los siguientes elementos:

- El mensaje propiamente dicho: <Message> *Message_0_reply*.
- El mensaje de recepción: <Message Occurrence Specification> *Message_0_reply_receiver*.
- El mensaje de envío: <Message Occurrence Specification> *Message_0_reply_sender*.

Los elementos <Opaque Behavior> *Message_0* y <Behavior Execution Specification> *Message_0* se refieren a la operación que se ejecuta en el objeto receptor.

Ahora desde el diagrama cambiamos los nombres de los mensajes para que se correspondan, por ejemplo con el caso de uso de *Autenticar* del sistema de gestión de notas. También, accedemos a las propiedades del *Actor 4*, en *Name* escribimos el nombre, por ejemplo *Sistema*, y de la lista *Represents* no elegimos ningún elemento, lo dejamos en blanco. El color rectángulo cambiará a gris. El resultado final del diagrama de secuencia de sistema con activaciones y mensajes de retorno para el caso de uso se muestra en la Figura 6.51.

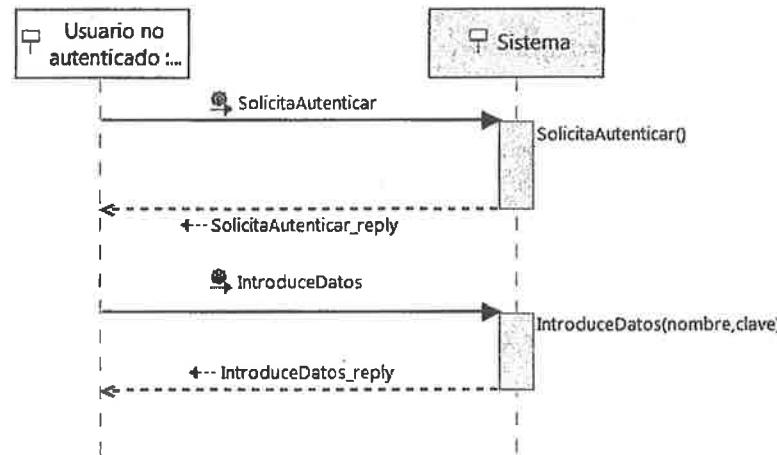


Figura 6.51. Diagrama de secuencia de sistema del caso de uso *Autenticar*.

Ejemplo 15. Se va a desarrollar el diagrama de secuencia partiendo de un diagrama de clases diseñado para resolver el siguiente problema. Se trata de llevar a cabo la gestión de los datos de los empleados de una empresa, las clases que se han diseñado son las siguientes:

- Del tipo entidad: *Empleado*, con la información de los empleados, código, nombre y dirección, las operaciones get y set y la operación *crearEmpleado()*.
- Del tipo control: *OperacionesEmple*, con las operaciones a realizar sobre los empleados: insertar, modificar, borrar, consultar y validar los datos de empleados.
- Del tipo interfaz: *VentanaEmple*, esta clase representa la ventana con la que interactúa el operador. Desde aquí se elegirá la operación que se desea hacer.

El diagrama de clases se muestra en la Figura 6.52. Cada clase se ha agrupado en un paquete, la clase *Empleado* en el paquete *DatosEmple*, la clase *OperacionesEmple* en el paquete *GestorEmple* y la clase *VentanaEmple* en el paquete *VentanasEmple*. En la Figura 6.53 se muestra el diagrama de paquetes con los import.

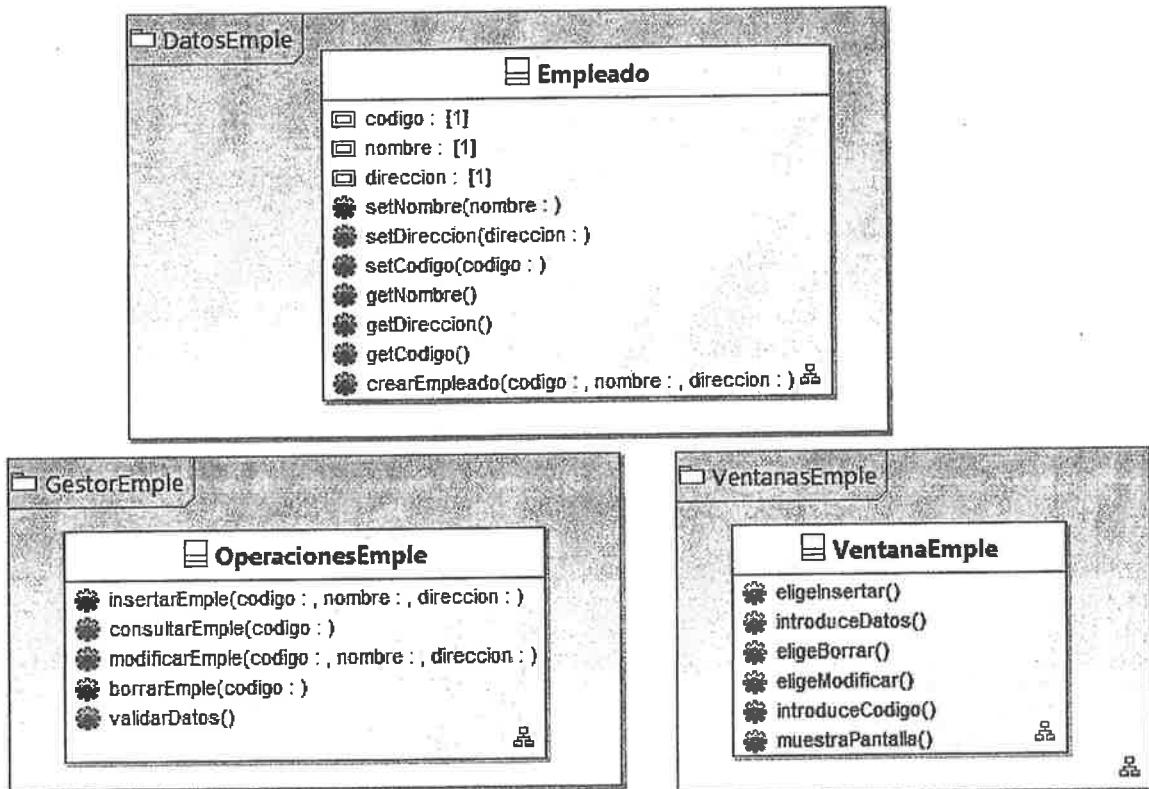


Figura 6.52. Diagrama de clases Ejemplo 15.

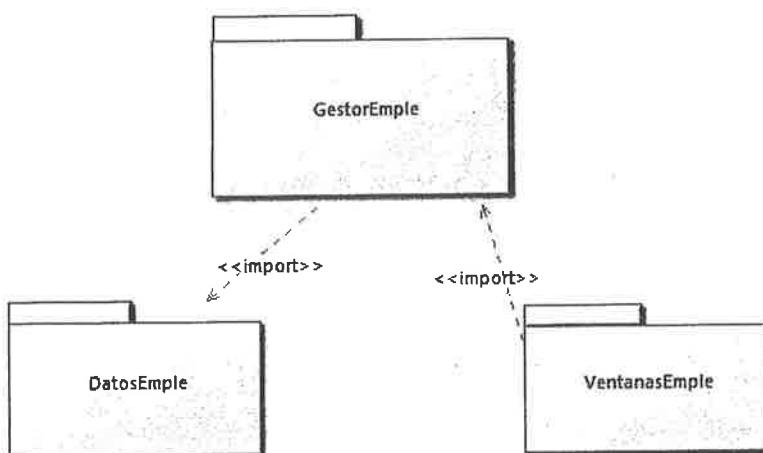


Figura 6.53. Jerarquía de paquetes Ejemplo 15.

En este sistema solo el operador manejará la aplicación y las operaciones a realizar son altas, bajas, modificaciones y consultas sobre los datos de empleados. El diagrama de casos de uso se creará asociado al paquete *VentanasEmple*, ya que el operador interactuará sólo con las clases de tipo ventanas.

Para crear el diagrama asociado al paquete, se selecciona el paquete (desde el diagrama de paquetes), se pulsa el botón derecho del ratón y se elige la opción *Navigate-> New detail: Use Case Diagram*, véase Figura 6.54. Creamos así el caso de uso para que luego, al crear el diagrama de secuencia aparezcan los actores que intervienen en los casos de uso.

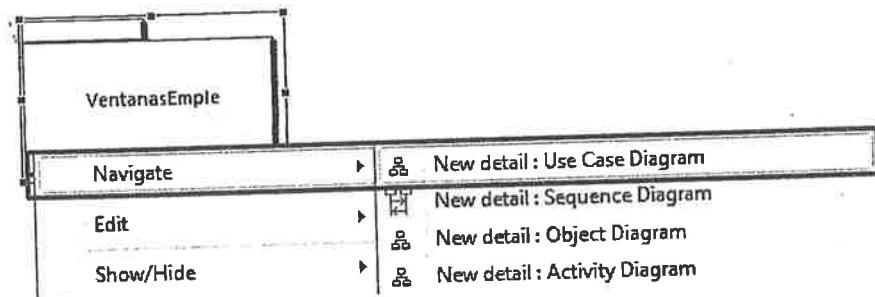


Figura 6.54. Crear caso de uso asociado al paquete *VentanasEmple*.

El diagrama de casos de uso se muestra en la Figura 6.55. En el *Model Explorer* se puede observar que el caso de uso está dentro del paquete *VentanasEmple*.

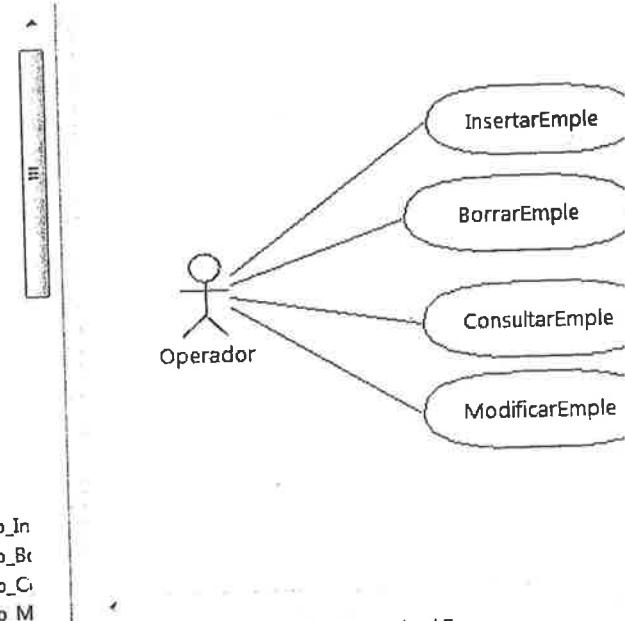
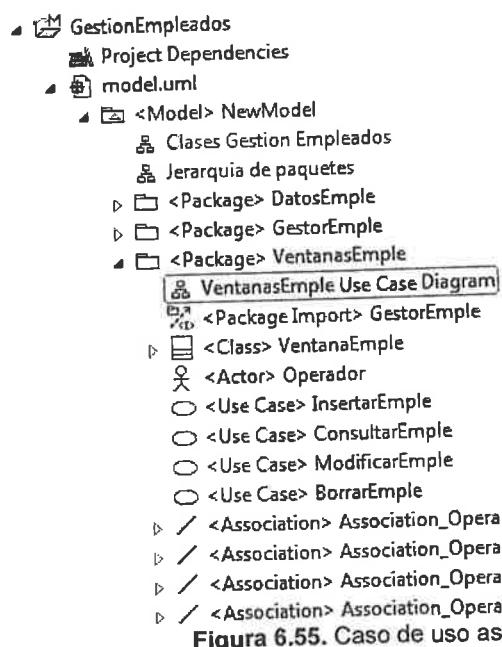


Figura 6.55. Caso de uso asociado al paquete *VentanasEmple*, Ejemplo 15.

La descripción del caso de uso *InsertarEmple* es la siguiente:

Nombre: InsertarEmple
ID: 1
Descripción:
El operador del sistema introduce los datos de los empleados de la empresa.
Actores: Operador.
Precondiciones:
Se requiere que el operador esté conectado al sistema.
Curso normal del caso de uso:
<ol style="list-style-type: none"> 1. El operador elige la opción de insertar empleado. 2. El sistema solicita los datos del empleado. 3. El operador introduce el código, nombre y la dirección. 4. El sistema valida los datos introducidos.
Postcondiciones:
El sistema da de alta al empleado.
Alternativas:
<ol style="list-style-type: none"> 4.1 Los datos introducidos por el operador no son válidos. 4.2 El sistema informa al operador. 4.3 Fin del caso de uso.

Antes de crear el diagrama de secuencia es necesario asociar a la clase *VentanaEmple* los casos de uso, ya que el operador se comunica con el sistema a través de las operaciones definidas en esta clase. Para ello desde el diagrama de clases hacemos doble clic en la clase *VentanaEmple*. Pulsamos en el botón que aparece a la derecha de *Use Case*, elegimos los cuatro casos de uso, pulsamos el botón *Add* para añadirlos, después pulsamos *OK*, véase Figura 6.56, y por último se pulsa el botón *Finish*.

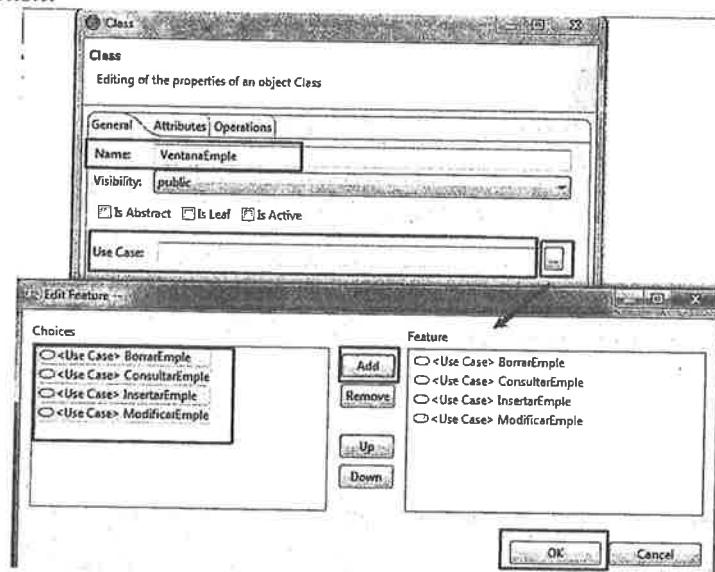


Figura 6.56. Casos de uso asociados a la clase *VentanaEmple*, Ejemplo 15.

Para crear el diagrama de secuencia asociado al paquete *VentanasEmple* se selecciona el paquete (desde el diagrama de paquetes), se pulsa el botón derecho del ratón y se elige la opción *Navigate-> New detail: Sequence Diagram*. A continuación desde la carpeta *Existing Elements* de la paleta del diagrama, se buscan las clases y/o los actores del modelo y se llevan al área de edición. En el ejemplo se muestra el operador y las clases *VentanaEmple*, *OperacionesEmple* y *Empleado*, véase Figura 6.57.

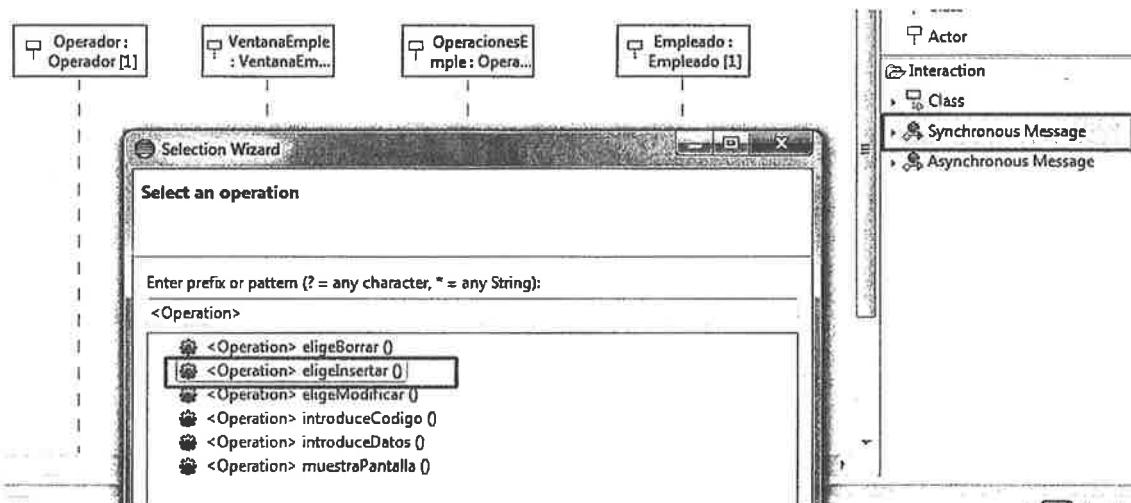


Figura 6.57. Mensajes en la clase *VentanaEmple*, Ejemplo 15.

Para añadir los mensajes se pulsa en *Synchronous Message* y a continuación arrastramos desde la línea de vida del emisor del mensaje a la línea de vida del receptor. Se mostrarán las operaciones que hay en el objeto receptor. Por ejemplo, al crear un mensaje entre el *Operador* y *VentanaEmple* se muestran todas las operaciones definidas en la clase *VentanaEmple*, véase Figura 6.57. La opción *Synchronous Message and Operation Creation* permitirá crear nuevas operaciones que se insertarán en la clase receptora.

En primer lugar el operador envía el mensaje *eligeInsertar()* al objeto de la clase *VentanaEmple*. A continuación el objeto de la clase *VentanaEmple* se envía a sí mismo el mensaje *muestraPantalla*, seguidamente el operador envía el mensaje *introduceDatos* al objeto de la clase *VentanaEmple*. Desde *VentanaEmple* se envía el mensaje *insertarEmple* al objeto de la clase *OperacionesEmple*, que se envía a sí mismo el mensaje *validarDatos*. Por último, se envía el mensaje *crearEmpleado* al objeto de la clase *Empleado*. El resultado final se muestra en la Figura 6.58, por claridad se han eliminado los mensajes *reply_receiver* y no se han añadido los parámetros al escribir las operaciones entre objetos.

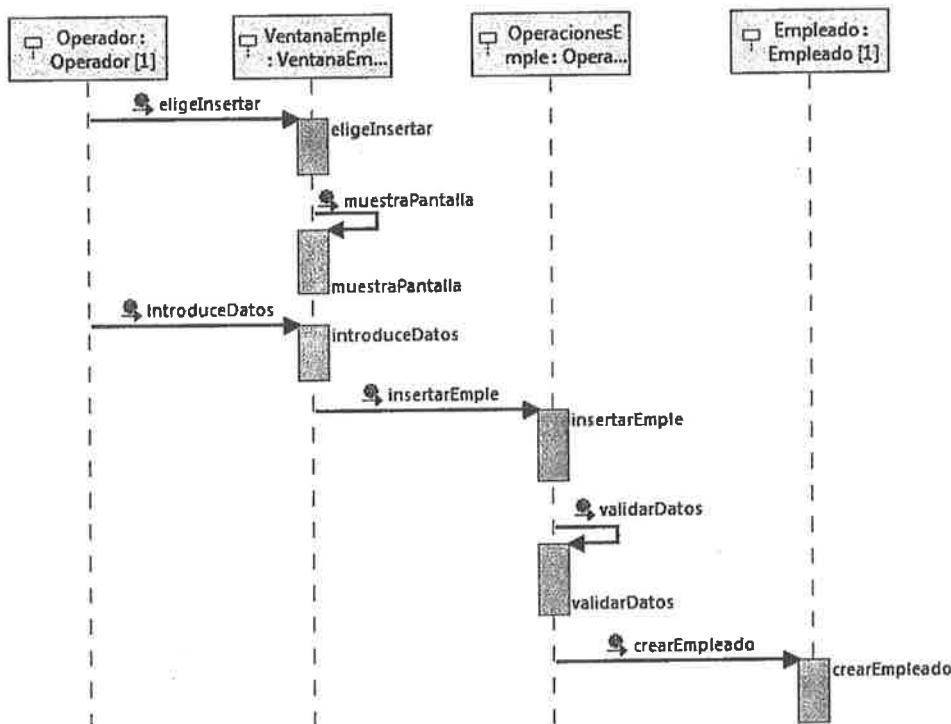


Figura 6.58. Diagrama de secuencia del ejemplo 15.

En esta versión de UML Designer no existe la opción de crear un diagrama de colaboración a partir del diagrama de secuencia y no se dispone de las opciones para añadir alternativas y bucles a los diagramas.

ACTIVIDAD 6.8

Realiza los diagramas de secuencia para el resto de casos de uso de la gestión de empleados.

WHITESTARUML

Para crear un diagrama de secuencia y colaboración en WhiteStarUML empezamos creando un proyecto vacío. Desde la opción de menú *Model->Profiles* establecemos el perfil que se utilizará en el proyecto, incluimos *Java 1.5 Profile* y *UML Standard Profile*. En primer lugar creamos un diagrama de casos de uso (repasar el Epígrafe 6.3.3 para ver cómo crear el diagrama).

Una vez que tenemos el diagrama de casos de uso, creamos el diagrama de secuencia de sistema para uno de los casos de uso, por ejemplo, para el caso de uso de *InsertarEmpleado*. Seleccionamos el caso de uso, pulsamos sobre el botón derecho del ratón y seleccionamos *Add Diagram-> Sequence Diagram (Role)*, véase Figura 6.59. (La opción *Add Diagram-> Sequence Diagram* también crea un diagrama de secuencia).

Se muestra a la izquierda la barra de herramientas con los elementos a añadir en el diagrama. En primer lugar y dentro del *Model Explorer*, pulsamos en el monigote del operador y después pulsamos sobre la zona de edición. Para añadir el rol de sistema, pulsamos en el botón *ClassifierRole* y después en la zona de edición, véase Figura 6.60.

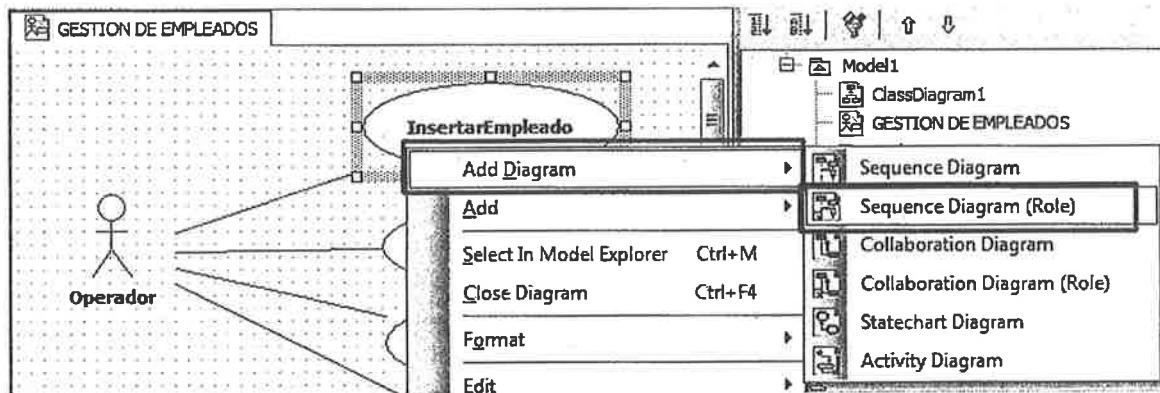


Figura 6.59. Crear un diagrama de secuencia en WhiteStarUML.

Para añadir los mensajes pulsamos en el botón *Message* → arrastramos desde el emisor al receptor y le asignamos un nombre. Desde la propiedad *ActionKind* se define el tipo de mensaje, por defecto es *CALL*. A la izquierda del mensaje se muestra el número de secuencia. Añadimos el resto de mensajes y observamos que a cada uno se le asigna su número de secuencia, véase Figura 6.60.

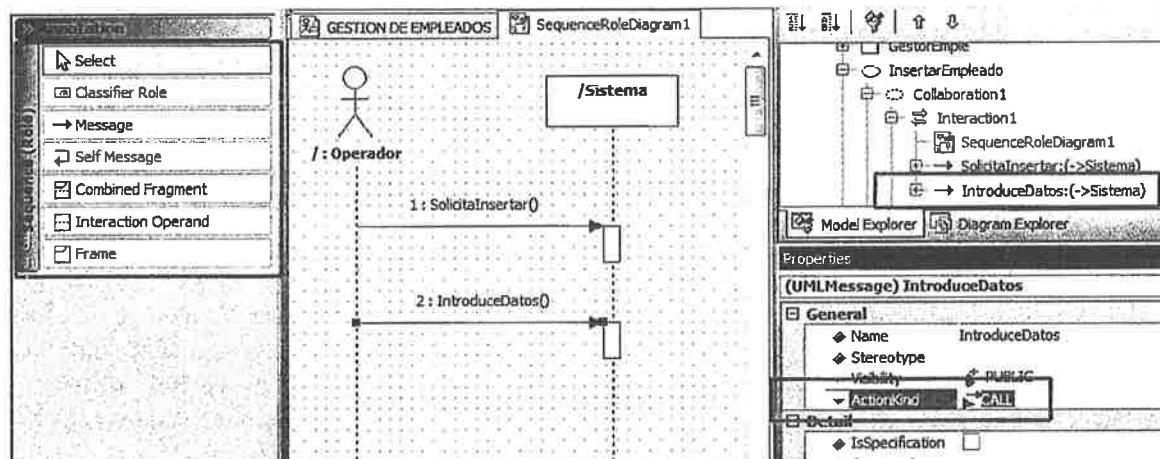


Figura 6.60. Diagrama de secuencia de sistema en WhiteStarUML.

En este entorno podemos representar alternativas y bucles pulsando en el botón *Combined Fragment* ; desde la propiedad *InteractionOperator* se puede elegir la operación que representa: *alt*, *opt*, *loop*, etc; en la propiedad *Name* se escribe la condición.

WhiteStarUML permite convertir un diagrama de secuencia en un diagrama de colaboración (y viceversa). Para ello se selecciona el diagrama de secuencia desde el *Model Explorer*, a continuación se pulsa en el menú *Model* y se selecciona *Convert Diagram -> ConvertSequence (Role) to Collaboration (Role)*, véase Figura 6.61.

Si todo va bien se muestra un mensaje indicando que la conversión se ha realizado correctamente y se muestra el diagrama con todos los elementos amontonados, pulsamos en *Aceptar* para cerrar la ventanita con el mensaje. A continuación hemos de colocar los elementos para que las flechas se vean en orden, pulsamos sobre las flechitas y las vamos colocando; véase Figura 6.62.

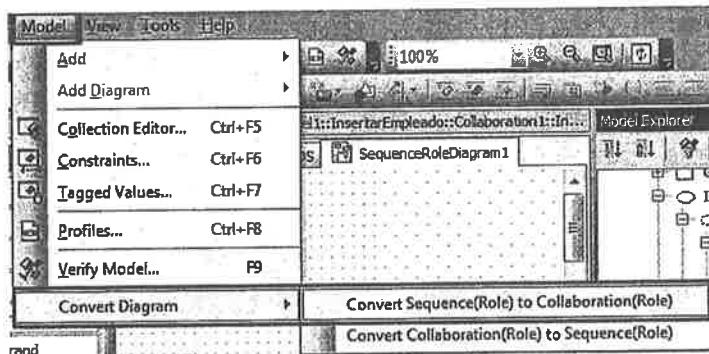


Figura 6.61. Convertir diagrama de secuencia en diagrama de colaboración en WhiteStarUML.

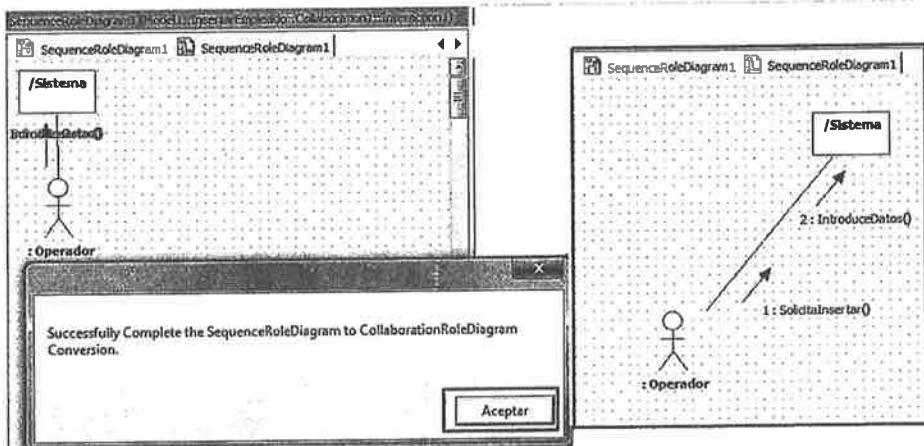


Figura 6.62. Diagrama de colaboración en WhiteStarUML.

A continuación construimos el diagrama de secuencia a partir de las clases, como se hizo con el ejemplo en Eclipse. En primer lugar creamos las clases dentro de sus paquetes. Una vez creadas, desde el *Model Explorer* pulsamos sobre el modelo con el botón derecho del ratón y seleccionamos *Add Diagram-> Sequence Diagram (Role)*. Añadimos los elementos que formarán parte del diagrama: el operador y las tres clases, véase Figura 6.63.

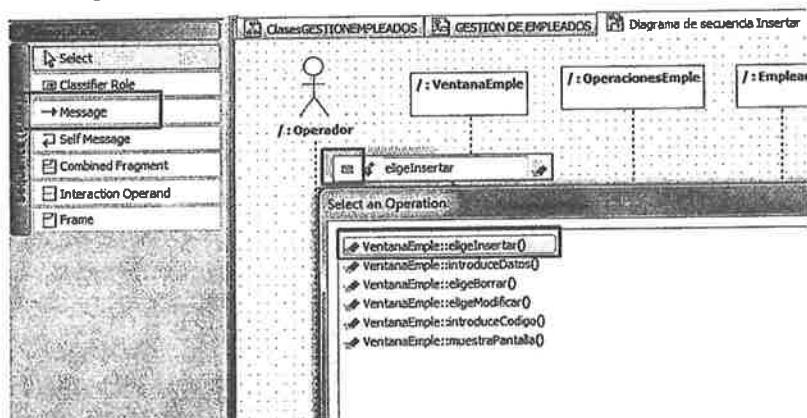


Figura 6.63. Añadir mensajes en WhiteStarUML.

Para añadir los mensajes se pulsa en *Message* → y se arrastra desde la línea de vida del objeto emisor al receptor, véase Figura 6.63. Se muestra un campo de texto desde el que

podemos escribir la operación o seleccionarla del objeto receptor. El botón *SelfMessage* crea un mensaje de un rol al mismo rol en el diagrama actual.

El botón *Select an Operation* muestra las operaciones del objeto receptor, seleccionamos la operación y pulsamos *OK*. El botón *Add operation* añade una nueva operación a la clase del objeto receptor. Para el ejemplo, pulsamos el botón *Select an Operation* y elegimos la operación que nos interese del objeto receptor. Repetimos los pasos para generar el diagrama de secuencia completo. El diagrama completo se muestra en la Figura 6.64.

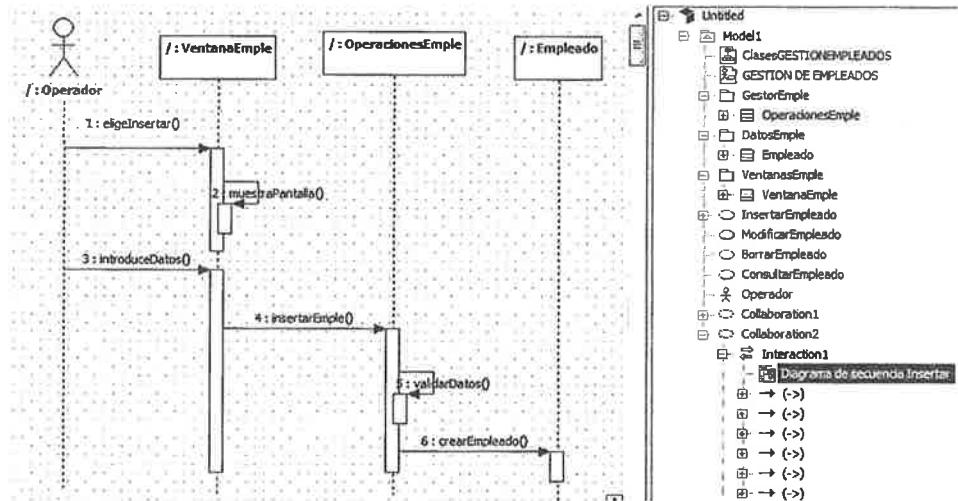


Figura 6.64. Diagrama de secuencia del caso de uso *InsertarEmpleado* en WhiteStarUML.

Una vez creado el diagrama de secuencia, generamos el de colaboración. Para ello, seleccionamos el diagrama de secuencia desde el *Model Explorer* y pulsamos en la opción de menú *Model-> Convert Diagram -> ConvertSequence (Role) to Collaboration (Role)*. El resultado final se muestra en la Figura 6.65.

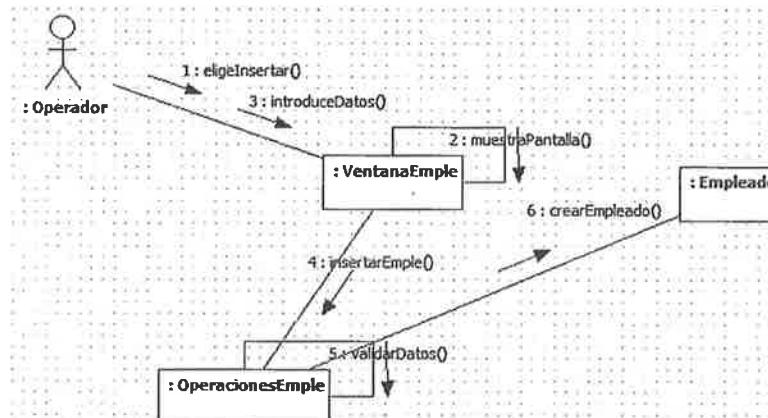


Figura 6.65. Diagrama de colaboración del caso de uso *InsertarEmpleado* en WhiteStarUML.

ACTIVIDAD 6.9

Realiza el diagrama de secuencia del caso de uso *ComprarProductos*. Crea un diagrama de clases para definir las clases, no hace falta que crees atributos y operaciones. A la hora de crear el

diagrama de secuencia y escribir los mensajes entre emisor y receptor genera las operaciones para las clases. Para indicar la creación de un objeto se selecciona la propiedad *ActionKind* del mensaje y se elige la opción *CREATE*. Genera después del diagrama de colaboración. Desde las propiedades del mensaje en el diagrama de colaboración se puede documentar el bucle (*Detail->Iteration*) y la alternativa (*Detail->Branch*), véase Figura 6.66.

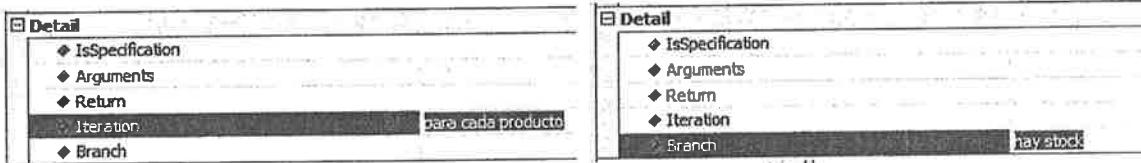


Figura 6.66. Bucle y alternativa en las propiedades del mensaje

ARGOUML

Al abrir ArgoUML en el modelo aparece un diagrama de clase y un diagrama de casos de uso. Creamos el diagrama de casos de uso y el diagrama de clases para la gestión de empleados. Para crear el diagrama pulsamos con el botón derecho del ratón sobre el modelo, a continuación elegimos *Crear diagrama -> Diagrama de secuencia*. Véase Figura 6.67. Desde las propiedades escribimos el nombre.

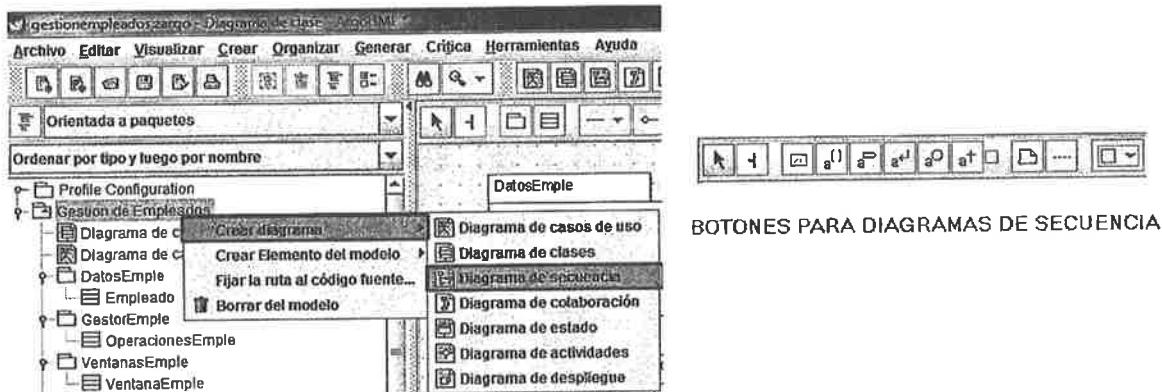


Figura 6.67. Crear diagrama de secuencia en ArgoUML.

En la parte superior de la pantalla de edición se muestra la barra de herramientas con varios botones:

- Rol clasificador nuevo.** Crea un nuevo rol en el diagrama con su línea de vida.
- a⁽¹** *Acción de llamada.* Crea un mensaje de llamada entre un emisor y un receptor, la creación del mensaje lleva incluido un mensaje de retorno.
- a⁽²** *Crear acción de envío.* Crea un mensaje de envío entre emisor y receptor, el mensaje no incluye mensaje de retorno.
- a⁽³** *Crear acción de retorno.* Crea un mensaje de retorno entre emisor y receptor.
- a⁽⁴** *Crear acción.* Se utiliza para crear un objeto.
- a⁽⁵** *Crear acción de destrucción.* Se utiliza para destruir un objeto.

 **Barrido.** Se pulsa y se arrastra por la pantalla, se realiza un barrido de todos los elementos.

 Se utiliza para añadir comentarios.

Partiendo del diagrama de casos de uso y del diagrama de clases, realizamos el diagrama de secuencia. Arrastramos el operador y las clases *VentanaEmple*, *OperacionesEmple* y *Empleado* a la zona de edición, y damos nombre a los elementos. Pulsamos en el botón  **Acción de llamada** para añadir los mensajes entre emisor y receptor. Pulsamos con el botón derecho del ratón sobre el mensaje, elegimos *Operation* y la operación que nos interese para que se muestre en el mensaje. Este tipo de mensaje crea dos mensajes uno de llamada y el de retorno, véase Figura 6.68.

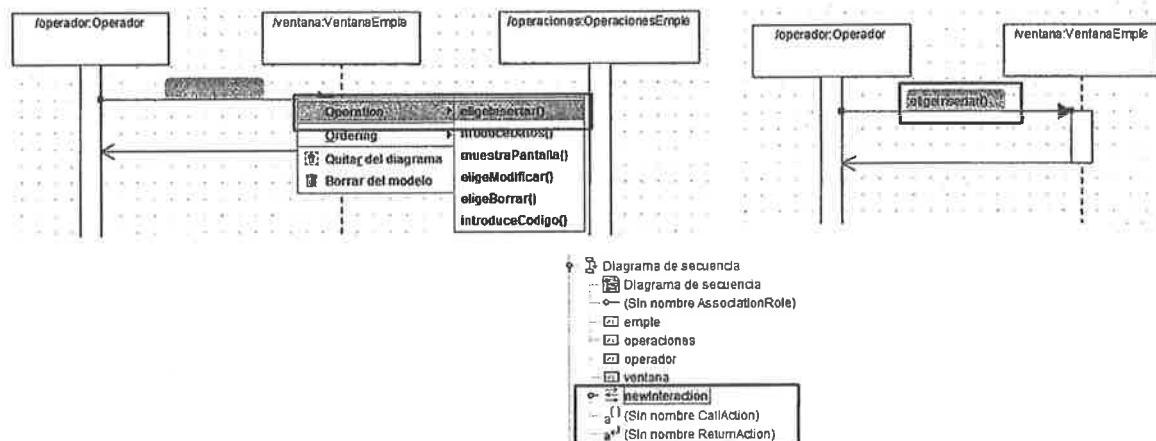


Figura 6.68. Añadir mensajes en ArgoUML.

Conviene dar un nombre a los mensajes que se van generando. Para añadir un mensaje de llamada a sí mismo pulsamos en el botón  **Acción de llamada**, y en la línea de vida donde queremos aplicarlo, dibujamos una especie de rectángulo haciendo clic en las esquinas y terminando en la línea de vida, véase Figura 6.69.

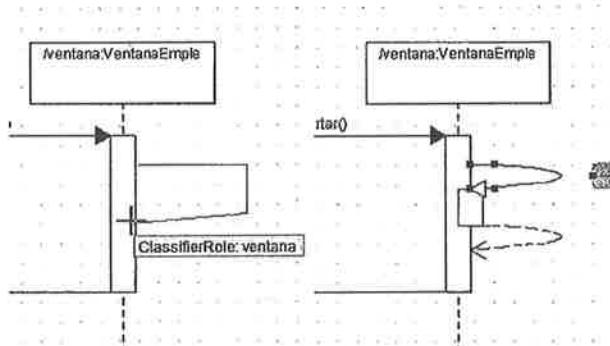


Figura 6.69. Mensaje de llamada así mismo en ArgoUML.

A la hora de crear el empleado podemos utilizar el botón  **Crear acción** que nos permite crear un objeto. El diagrama de secuencia con ArgoUML se muestra en la Figura 6.70.

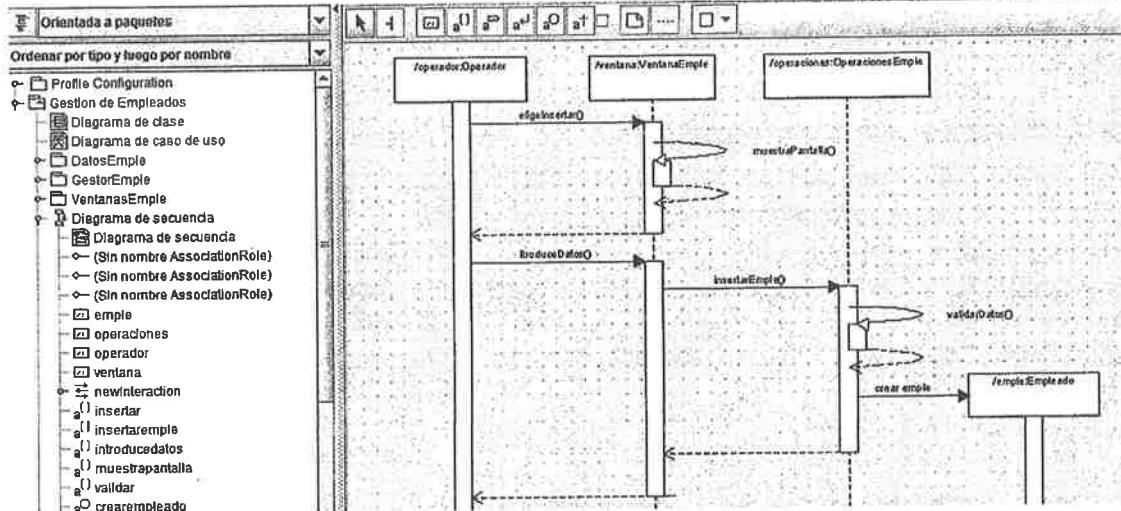


Figura 6.70. Diagrama de secuencia en ArgoUML.

La versión de ArgoUML utilizada soporta los diagramas UML 1.4 estándar, por ello no hay opciones para representar los bucles y las alternativas, ya que estas se definieron en el estándar UML 2.

El diagrama de colaboración se crea manualmente. Se puede crear dentro de un paquete, dentro del modelo o dentro de una colaboración (*unattachedCollaboration*). Por ejemplo, para crearlo dentro del modelo pulsamos sobre el modelo con el botón derecho del ratón y elegimos *Crear diagrama->Diagrama de colaboración*, véase Figura 6.71.

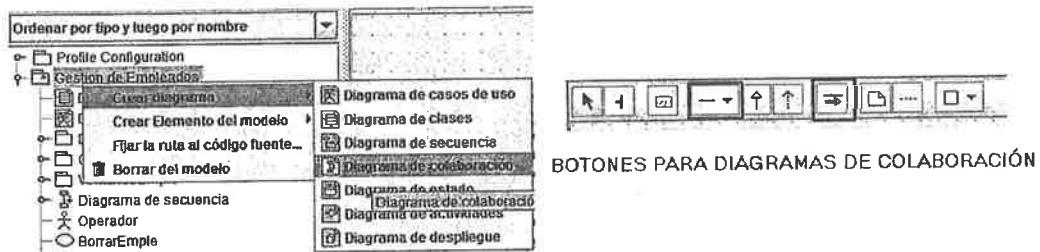


Figura 6.71. Crear diagrama de colaboración en ArgoUML.

Para crear el diagrama seleccionamos los elementos del explorador de la izquierda, el *Operador* y las clases *OperacionesEmple*, *VentanaEmpleo* y *Empleado*; y los arrastramos al área de edición del diagrama. Les asignamos un nombre.

Para crear la asociación entre dos elementos, pulsamos en el botón *Role de asociación nuevo* y arrastramos desde el emisor al receptor. A continuación y teniendo la asociación seleccionada, pulsamos en el botón *Añadir mensaje* , vemos que aparece el número de secuencia. Hacemos doble clic en él para crear la acción de llamada. Podemos escribir el nombre de la operación (si no existe en la clase, entonces se creará), véase Figura 6.72.

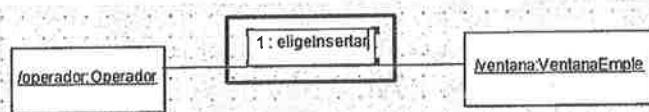


Figura 6.72. Añadir mensaje en el diagrama de colaboración en ArgoUML.

O bien, desde el explorador seleccionamos la acción de llamada y elegimos dentro de la pestaña de propiedades la operación *eligeInsertar()* de la lista *Operación*, véase Figura 6.73.

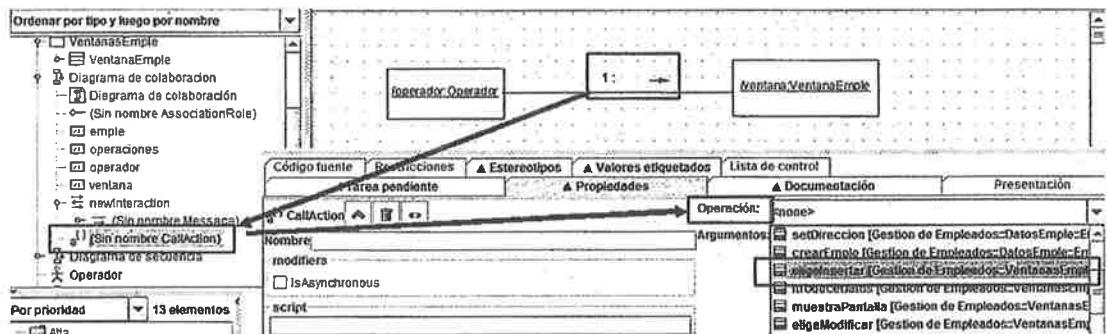


Figura 6.73. Selección de la acción de llamada en el diagrama de colaboración en ArgoUML.

Repetimos los pasos para crear el resto de asociaciones y mensajes entre los elementos del diagrama, en el orden en que ocurren para que los números de secuencia aparezcan ordenados. El resultado final se muestra en la Figura 6.74. Para crear varios mensajes en una misma asociación, se selecciona la asociación y después se pulsa el botón *Añadir mensaje*. Los mensajes a sí mismo se realizan como en el diagrama de secuencia.

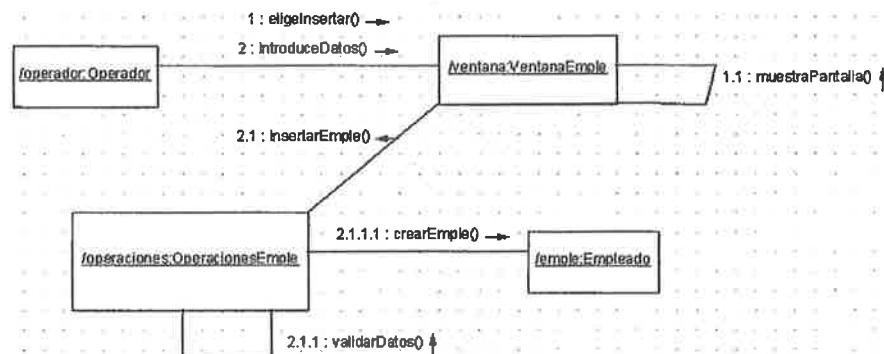


Figura 6.74. Diagrama de colaboración en ArgoUML.

6.5. DIAGRAMA DE ESTADO

Los diagramas de estados se utilizan para describir el comportamiento de un sistema. Describen todos los estados posibles en los que puede entrar un objeto particular y la manera en que cambia su estado como consecuencia de los eventos que le llegan. Son útiles para los objetos con un comportamiento significativo.

En la mayor parte de las técnicas orientadas a objetos, los diagramas de estados se dibujan para una sola clase mostrando el comportamiento de un solo objeto durante todo su ciclo de vida.

Un diagrama de estados se compone de los siguientes elementos:

- **Estado.** Es una condición o situación durante la vida de un objeto, durante la cual se satisface alguna condición, ejecuta alguna actividad a realizar o espera por algún evento².

² El lenguaje unificado de modelado. Manual de referencia. James Rumbaugh , Ed: Addison-Wesley, 2007.

- **Evento.** Es algo que ocurre durante la ejecución de un sistema. La ocurrencia de un evento en el objeto es la aparición de un estímulo que puede producir la transición de un estado a otro. Esta ocurrencia puede ser de varios tipos: una condición que se cumple, la recepción de una llamada a una operación, el paso de cierto período de tiempo o la recepción de una señal de otro objeto del modelo.
- **Transición.** Relación entre dos estados que indica que un objeto que se encuentra en un primer estado (estado inicial), cuando se produce un evento específico y se satisface una condición especificada, realizará ciertas acciones y entrará en el segundo estado (estado final). Véase Figura 6.75.



Figura 6.75. Representación gráfica de una transición.

Los elementos básicos de un diagrama de estados son los siguientes:

SÍMBOLO	FUNCIÓN	NOTACIÓN
Estado inicial del objeto	Es el punto de partida del diagrama de estados. Se corresponde con el estado inicial del objeto a la salida de su fase de creación. También se le conoce como pseudoestado inicial	●
Estado final del objeto	Es el último estado del objeto que cuando se entra en él indica que se ha finalizado la ejecución. No todos los objetos tienen estado final	●
Transición	Se representa como una flecha continua desde el estado origen al estado destino etiquetada con el nombre de evento que dispara la transición. Una transición reflexiva tiene el mismo estado inicial y final	Evento →
Estado	Se dibuja como un rectángulo con los bordes redondeados con su nombre en el interior	Estado

DEFINICIONES: MÁQUINA DE ESTADOS Y DIAGRAMA DE ESTADOS³

Máquina de estados: es una especificación de las secuencias de los estados por los que un objeto o una interacción pasa en respuesta a eventos durante su vida, junto con sus efectos correspondientes. Una máquina de estados se representa mediante un diagrama de estados.

Diagrama de estados: nombre usado por David Harel para su extensión a la notación básica de la máquina de estados que incluye estados anidados y estados concurrentes. Esta notación sirvió como base para la notación de la máquina de estados de UML.

Diagrama de máquina de estados: es un diagrama que demuestra una máquina de estados, incluyendo estados simples, transiciones y estados compuestos anidados. El concepto original fue inventado por David Harel, que los llamó los diagramas de estados.

³ El lenguaje unificado de modelado. Manual de referencia. James Rumbaugh , Ed: Addison-Wesley, 2007.

Los diagramas de estado no solo se aplican a las clases, también se pueden vincular a los casos de uso para describir su ejecución, para estos, un estado representa un paso en su ejecución.

Ejemplo 16. En la Figura 6.76 se muestra el diagrama de estados del caso de uso de *ComprarProductos*. Se inicia el proceso de venta pasando a un estado de espera hasta que se seleccionen los productos que se van a comprar. Cuando se produce el evento *seleccionarProducto* se pasa al estado *Introduciendo Productos*. En este estado permanecerá hasta que se produzca el evento de finalizar la venta, entonces se pasa al estado en el que el sistema espera el pago de la venta, estado *Esperando Pago*, de este estado se sale cuando se efectúa el pago. Una vez realizado el pago, la venta finaliza.

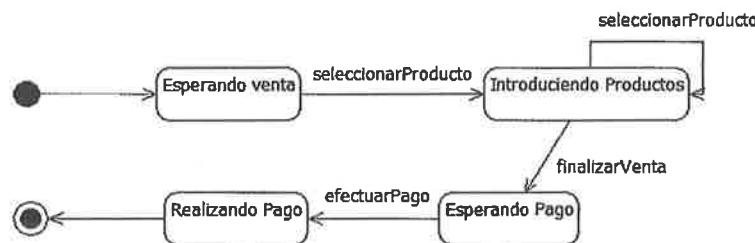


Figura 6.76. Estados para el caso de uso de *ComprarProductos*.

Ejemplo 17. En la Figura 6.77 se muestra la clase *Libro* con tres atributos y tres operaciones. El diagrama de estado muestra los estados por los que puede pasar un objeto *Libro* en una biblioteca. Del estado inicial del objeto se pasa al estado *Disponible*, en este estado el libro puede ser prestado o dado de baja. Si ocurre el evento *prestar()* el objeto pasa al estado *Prestado*, si ocurre el evento *darDeBaja()* el objeto pasa al estado *Baja*, y desde este estado se llega al estado final del objeto. Desde el estado *Prestado*, se pasa al estado *Disponible* si ocurre el evento *devolver()*.

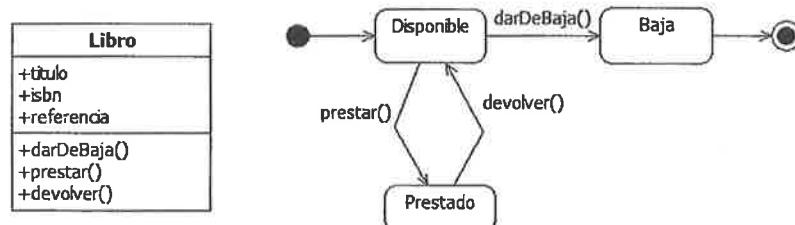


Figura 6.77. Estados de un objeto Libro.

Ejemplo 18. En la Figura 6.78 se muestra los estados por los que pasa la vida laboral de una persona. Inicialmente está desempleada, si se la contrata cambia de estado y empieza a trabajar, en este estado la pueden despedir, entonces vuelve a estar desempleada. Por último, cuando se jubila cambia de estado y finaliza la vida laboral. Se muestra la clase *Persona* con las operaciones (los eventos) que producen un cambio de estado en el objeto: *contratar()*, *despedir()* y *jubilar()*.

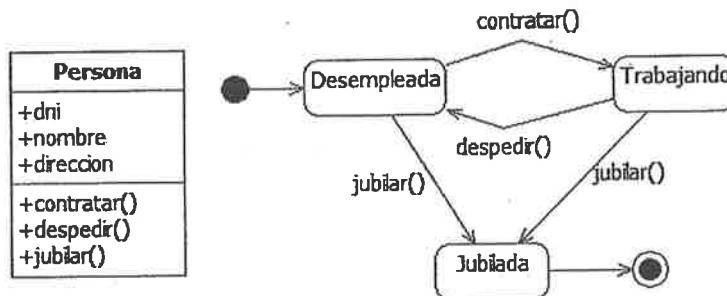


Figura 6.78. Estados de un objeto Persona.

ACTIVIDAD 6.10

Realiza un diagrama de estados que muestre los distintos estados por los que pasa una persona desde que nace hasta que muere. Considera los siguientes estados: niño, adulto, soltero y casado. El niño nace, del estado niño al estado adulto se pasa cuando la edad es > 18, del estado de adulto se pasa a soltero, el soltero cambia de estado si se casa y el casado cambia de estado a soltero si se divorcia. Etiqueta las transiciones entre un estado y otro que consideres necesarias.

Las transiciones se etiquetan con el nombre de evento que dispara la transición. Una transición sin evento asociado se dispara al terminar la actividad que se realiza en el estado correspondiente. La sintaxis de una etiqueta de transición tiene tres partes (véase Figura 6.79), las cuales son opcionales: *Evento [Condición de guarda] / Acción*:

- *Evento*: es el evento que da origen a una transición.
- *Condición de guarda*: es una expresión lógica que se evalúa cuando ocurre el evento. Si la condición de guarda es verdadera, entonces se produce la transición; si se evalúa a falso, la transición no ocurre. Se escribe entre corchetes.
- *Acción*: es una expresión procedimental que se ejecuta si se produce la transición. Suelen ser procesos que suceden con rapidez y se corresponden con algún método declarado en la clase. También en la acción se puede solicitar a otro objeto un servicio (*OtroObjeto.Operación*).

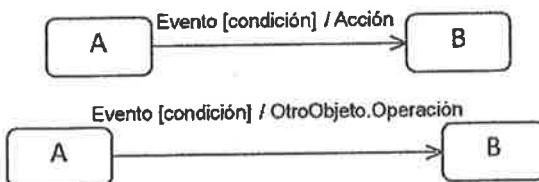


Figura 6.79. Sintaxis de una transición.

En la Figura 6.80 se muestra el diagrama de estado de la clase *VentanaEmple* del ejemplo de la gestión de empleados. Las transiciones se han etiquetado con *Evento/Acción*.

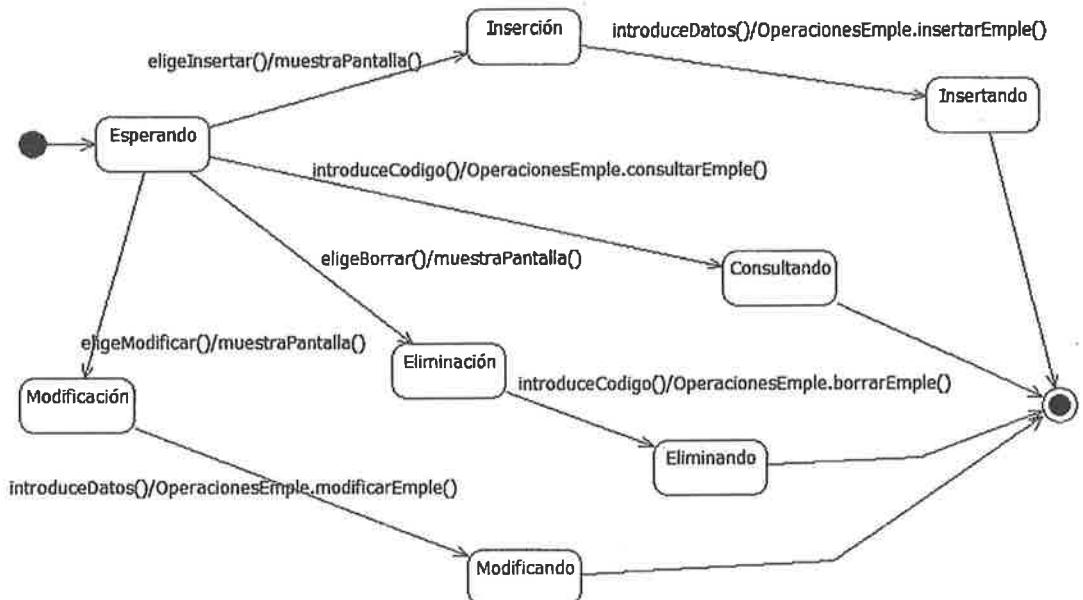


Figura 6.80. Diagrama de estado de la clase *VentanaEmple*.

Inicialmente un objeto de la clase *VentanaEmple* se encuentra esperando que ocurra un evento. Cuando se produce el evento *eligeInsertar()*, el sistema dispara una transición, se produce la acción *muestraPantalla()*. Después de que se haya realizado esta acción el sistema entra en el estado *Inserción*. Permanecerá en este estado hasta que se produzca el evento *introduceDatos()*. Cuando se produce dicho evento, el sistema dispara una transición provocando la acción de *InsertarEmple()* sobre un objeto *OperacionesEmple*. Después de que se haya realizado esta acción el sistema entra en el estado *Insertando*. Finalizada la inserción se origina la transición al estado final. El proceso es similar para el resto de eventos *introduceCodigo()*, *eligeBorrar()*, y *eligeModificar()*.

El estado inicial no tiene evento, sí tiene acción (véase la Figura 6.81); el evento es la creación de la máquina de estados o diagrama de estados; la acción sería la primera acción invocada después de la creación de la máquina de estados. En la transición hacia el estado final sí puede haber un par *evento/acción* (véase la Figura 6.81), la acción será la última invocada por la máquina de estados.

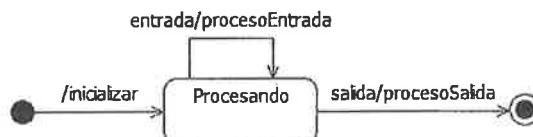


Figura 6.81. Eventos y acciones en transiciones de estados inicial y final.

Los estados se pueden especificar con más detalle. Se pueden especificar con dos compartimentos (Figura 6.82), el nombre del estado se indica en el compartimento superior. En el inferior se escriben pares *evento/ acción* con acciones especiales que nos dicen qué hacer cuando ocurre un evento; estas acciones o actividades se realizan mientras los objetos permanecen en un estado, pueden tardar más (que las acciones definidas en la transición) y suelen corresponderse con métodos de la clase. Cada nombre de evento puede aparecer más de una vez en un único estado. Se pueden especificar diferentes tipos de eventos:

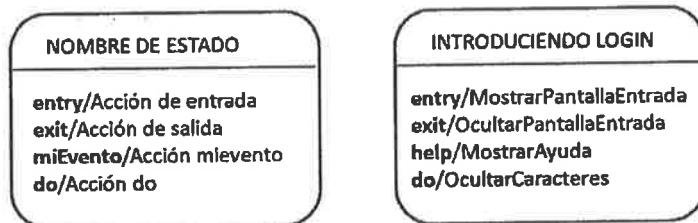


Figura 6.82. Compartimentos de un estado y diferentes actividades.

- **entry/acción de entrada.** Cualquier transición que entra en un estado invoca a la acción indicada a la derecha de *entry* si la hay. También se puede etiquetar como *entrada/acción de entrada*.
- **exit/ acción de salida.** La acción a la derecha de *exit* se ejecuta cuando se sale de un estado. También se puede etiquetar como *salida/acción de salida*.
- **do/acción interna.** Una acción interna empieza después de que la actividad de entrada se complete y puede que termine por sí misma o por la salida a otro estado. Se ejecuta mientras se está en un estado.
- **evento/acción de evento.** Son transiciones internas que no causan un cambio de estado. Si el evento ocurre se ejecuta la acción indicada.

Una acción consiste en asignar un valor a un atributo, crear o destruir un objeto, efectuar una operación, enviar una señal a otro objeto o a sí mismo, etc. En la Figura 6.82 se muestra un estado con compartimentos y diferentes acciones: al entrar en el estado *INTRODUCIENDO LOGIN* se realiza la acción de *MostrarPantallaEntrada*, al salir del estado se realiza la acción *OcultarPantallaEntrada*, cuando se produce el evento *help* se ejecuta la acción *MostrarAyuda* y por último, mientras se esté en el estado se ejecuta la actividad *OcultarCaracteres*.

En la Figura 6.83 se muestra el diagrama de estado para la clase *GestorVentas*. Se ha seguido el diagrama de secuencia de la Figura 6.41 para construirle. En el diagrama de secuencia se muestran tres llamadas desde la clase *Pantalla* a métodos de la clase *GestorVentas()*, estos métodos se han definido como los eventos que producen cambios de estado en el objeto. Son los siguientes: *iniciarVenta()*, *datosLinea()* y *finalizarVenta()*.

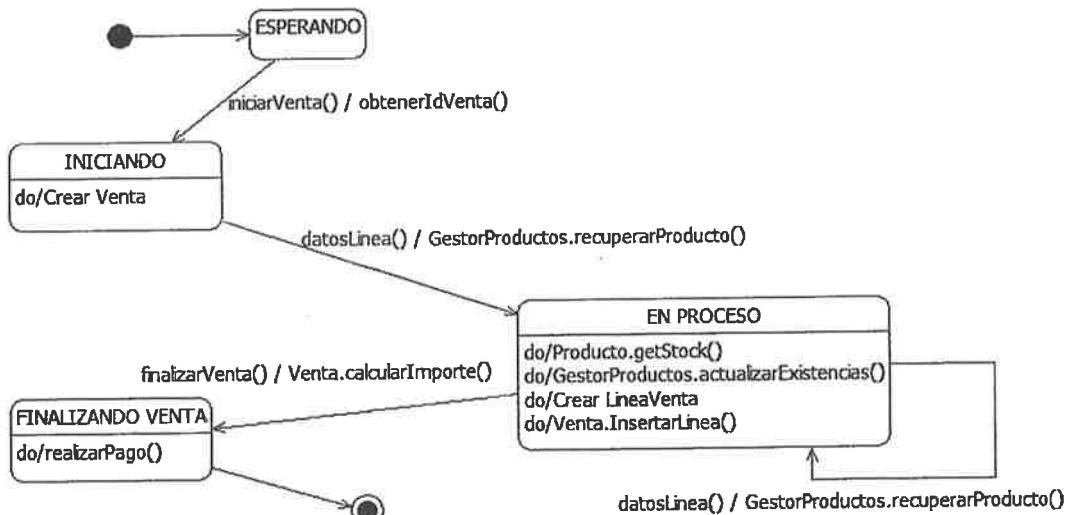


Figura 6.83. Diagrama de estado de la clase *GestorVentas*.

Inicialmente el sistema está en un estado de espera hasta que le llega el evento *iniciarVenta()* que hace que se ejecute la acción *obtenerIdVenta()*, una vez ejecutada esta acción el objeto entra en el estado *INICIANDO*, dentro de este estado se crea una venta.

Cuando se produce el evento *datosLinea()*, se dispara una transición produciendo la acción *GestorProductos.recuperarProductos()*; finalizada la acción el objeto entra en estado *EN PROCESO* donde se realizan las acciones indicadas en *do!*. En este estado permanecerá hasta que se produzca el evento *finalizarVenta()*. Si se vuelve a producir el evento *datosLinea()* se volverán a ejecutar las acciones *do!*.

6.5.1. Elaboración de diagramas de estado

A continuación vamos a ver cómo construir los diagramas de estado en diferentes entornos.

ECLIPSE

Para crear un diagrama de estado, pulsamos con el botón derecho del ratón en *<Model>* y seleccionamos *New Representation -> State Machine Diagram*, o también si lo queremos crear dentro de un paquete, pulsamos sobre él con el botón derecho del ratón y seleccionamos las opciones anteriores, véase Figura 6.84. Escribimos un nombre o dejamos el nombre por defecto y pulsamos el botón *OK*.

En la ventana de edición se muestran dos marcos y la barra de herramientas a la derecha, véase Figura 6.85. El marco más externo representa la máquina de estados, el interno nombrado como *new Region 1* representa una región. Una región es un fragmento de una máquina de estados que contiene estados, pseudoestados y transiciones. Es dentro de la región donde hemos de incluir los estados y las transiciones.

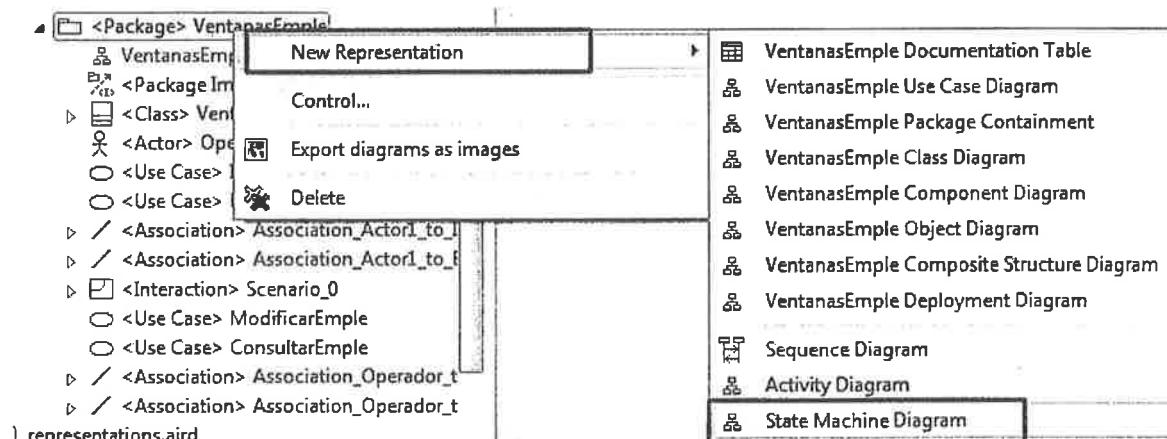


Figura 6.84. Creación de un diagrama de estado en Eclipse.

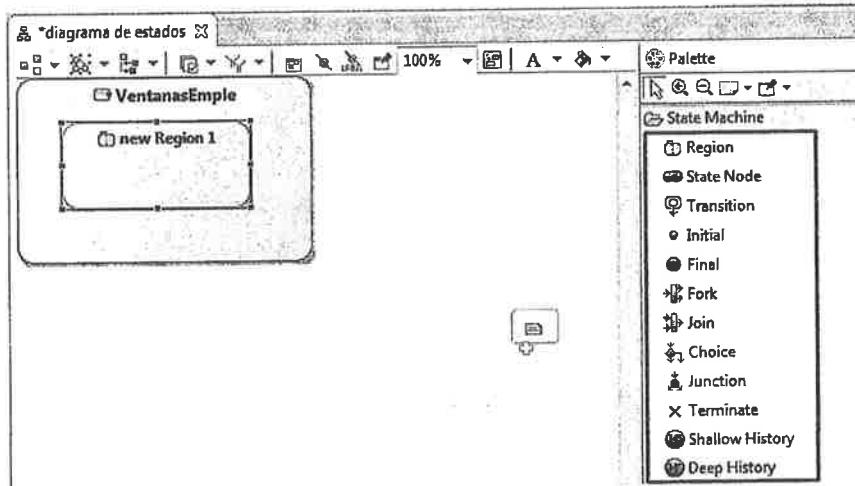


Figura 6.85. Región y barra de herramientas para el diagrama de estados en Eclipse.

Los elementos básicos que utilizaremos para crear un diagrama de estados sencillo son los siguientes:

- **State Node**: se utiliza para crear un estado.
- **Transition**: para crear una transición entre estados.
- **Initial**: para crear el estado inicial.
- **Final**: para crear el estado final.

Se pulsa sobre uno de estos iconos y a continuación se pulsa en la región. Para esta versión en las transiciones se indica la condición de guarda, esta se encierran entre corchetes. Una vez creada, desde las propiedades se indica la acción para la transición. Para crear la etiqueta en la transición, se pulsa sobre ella hasta que aparezca el cuadrado desde el que podremos escribir la condición de guarda, véase Figura 6.86.

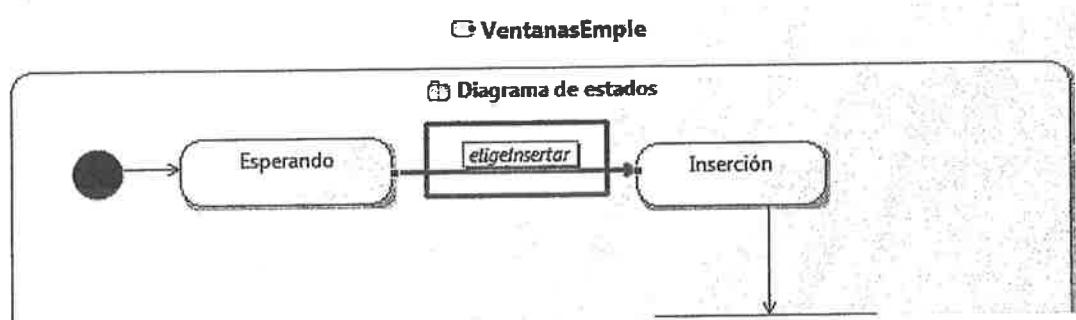


Figura 6.86. Etiquetar una transición en Eclipse.

Para añadir la acción a la transición desde el *Model Explorer* y teniendo seleccionada la transición, pulsamos sobre la flechita a la izquierda de *<Constraint>* para crear la acción asociada a la condición de guarda, entonces se crea el elemento *<Opaque Expression>*. Desde las propiedades de este elemento seleccionamos de la lista *Behavoir* la operación correspondiente, en este caso *muestraPantalla*, véase Figura 6.87. Estas operaciones de la lista *Behavoir* se definieron en el diagrama de secuencia.

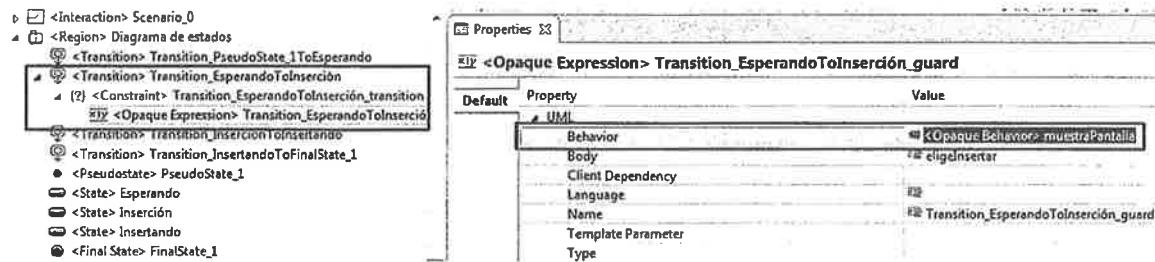


Figura 6.87. Añadir acción a una transición en Eclipse.

WHITESTARUML

Para crear un diagrama de estado desde el modelo pulsamos con el botón derecho del ratón y seleccionamos *Add Diagram-> Statechart Diagram*, véase Figura 6.88. Se muestra a la izquierda la barra de herramientas.

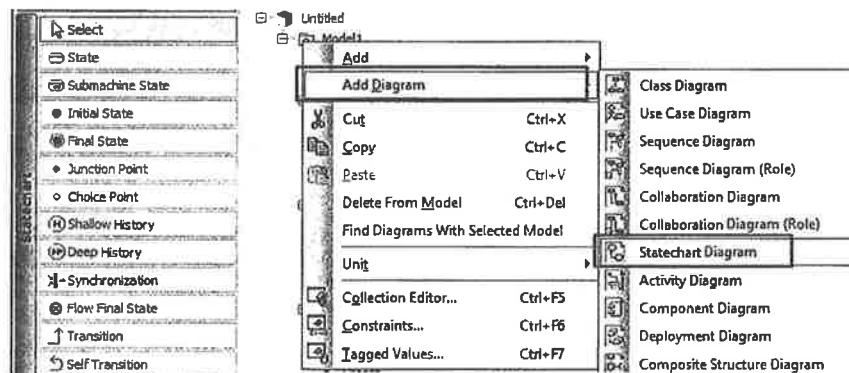


Figura 6.88. Crear diagrama de estado en WhiteStarUML.

Los elementos que utilizaremos para crear un diagrama de estados sencillo son los siguientes:

- *State*: se utiliza para crear un estado.
- *Initial State*: se utiliza para crear el estado inicial.
- *Final State*: se utiliza para crear el estado final.
- *Transition*: se utiliza para crear una transición.
- *Self Transition*: se utiliza para crear una transición reflexiva.

Para crear un estado pulsamos en el ícono de estado y después pulsamos sobre el área de edición. Escribimos el nombre, a su derecha aparecen 3 botones que permiten definir los pares *evento/acción* dentro del estado (Figura 6.89):

- *Add EntryAction*: permite definir un evento del tipo *entry/acción de entrada*.
- *Add DoAction*: permite definir un evento del tipo *do/acción interna*.
- *Add ExitAction*: permite definir un evento del tipo *exit/acción de salida*.

Estos eventos también se pueden crear pulsando con el botón derecho del ratón sobre el estado y seleccionando *Add -> Entry Action, Do Action o Exit Action* del menú popup (Figura 6.89).

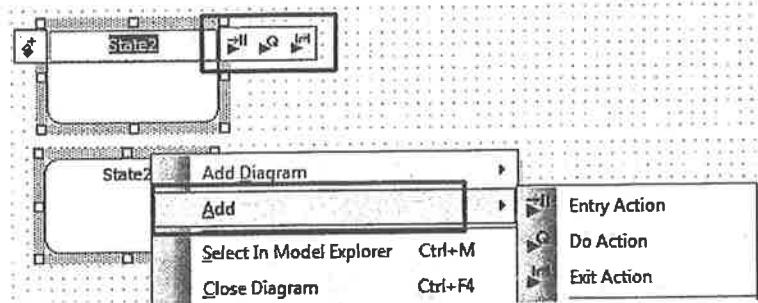


Figura 6.89. Añadir pares evento/acción en un estado en SarUML.

La creación de los pares *evento/acción* de un estado se puede llevar a cabo desde la zona de edición pulsando en el botón cuya acción nos interesa añadir, o bien accediendo a las propiedades *EntryActions*, *DoActions* y *Exit Actions* del estado correspondiente, véase Figura 6.90. Se pulsa en el botón y se abre el editor de colecciones con varias pestañas; desde cada una de las pestañas y usando los botones de la izquierda añadimos las acciones. El botón + y el botón – nos permiten añadir y eliminar acciones.

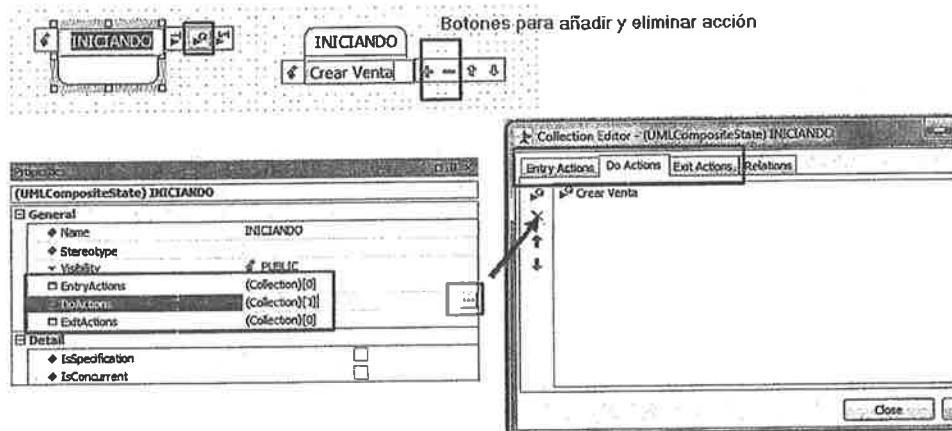


Figura 6.90. Añadir pares evento/acción en un estado en WhiteStarUML.

Para crear una transición pulsamos en el botón de transición \nearrow y arrastramos el puntero del ratón desde el estado inicial al estado final dentro de la zona de edición. Desde las propiedades de la transición podemos asignarla un nombre mediante la propiedad *Name*, definir el evento o eventos mediante la propiedad *Triggers*, la condición de guarda mediante la propiedad *GuardCondition* y la acción o acciones mediante la propiedad *Effects*, Figura 6.91.

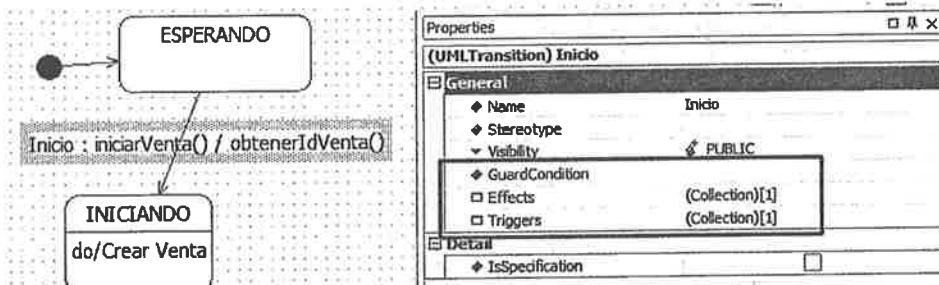


Figura 6.91. Propiedades de una transición en WhiteStarUML.

ARGOUML

Para crear el diagrama de estado pulsamos con el botón derecho del ratón sobre el modelo, a continuación elegimos *Crear diagrama -> Diagrama de estado*. Véase Figura 6.92. Se muestra la pantalla de edición y encima la barra de herramientas con los botones que se pueden usar para la elaboración del diagrama. También se puede crear el diagrama de estado dentro de un paquete, en este caso las operaciones que se usarán para las transiciones y los pares *evento/acción* serán los métodos de las clases que estén en dicho paquete. Para el siguiente ejemplo se crea el diagrama de estado dentro del modelo.

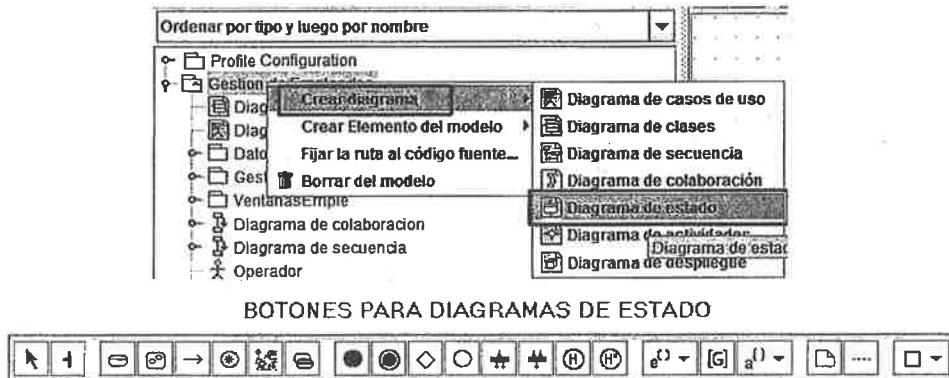


Figura 6.92. Crear diagrama de estado en ArgoUML.

Para crear un diagrama de estados sencillo utilizaremos los siguientes botones:

- *Crear estado simple*: se utiliza para crear un estado.
- *Crear estado inicial*: se utiliza para crear el estado inicial.
- *Crear estado final*: se utiliza para crear el estado final.
- *Crear transición*: se utiliza para crear una transición.
- botón para crear diferentes tipos de eventos.
- *Crear guarda*: se utiliza para crear la condición de guarda.
- *Llevar a cabo actividad*: botón para crear acciones.

Para crear un estado pulsamos en el botón *Crear estado simple* y después pulsamos sobre el área de edición. Escribimos el nombre, desde las propiedades podemos definir las acciones de entrada (*entry/*) desde la lista *Acción Entrada*, de salida (*exit/*) desde la lista *Acción de salida* o de llevar a cabo una actividad (*do/*) desde la lista *Llevar a cabo actividad*, véase Figura 6.93.

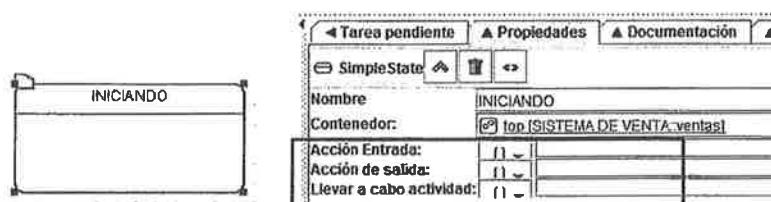


Figura 6.93. Propiedades de un estado en ArgoUML.

Por ejemplo, para añadir una actividad (*do/*) pulsamos sobre la lista que aparece a la derecha de *Llevar a cabo actividad* y elegimos una de las acciones, en este caso, una acción de llamada; se muestra una nueva ventana de propiedades, en el campo *Nombre* escribimos el nombre, escribimos el texto que aparecerá en el diagrama de estado (debajo de *script*) y elegimos la operación de la lista *Operación*, en esta lista se muestran los métodos de las clases definidas, véase Figura 6.94. En el estado se mostrará la acción *do/* creada.

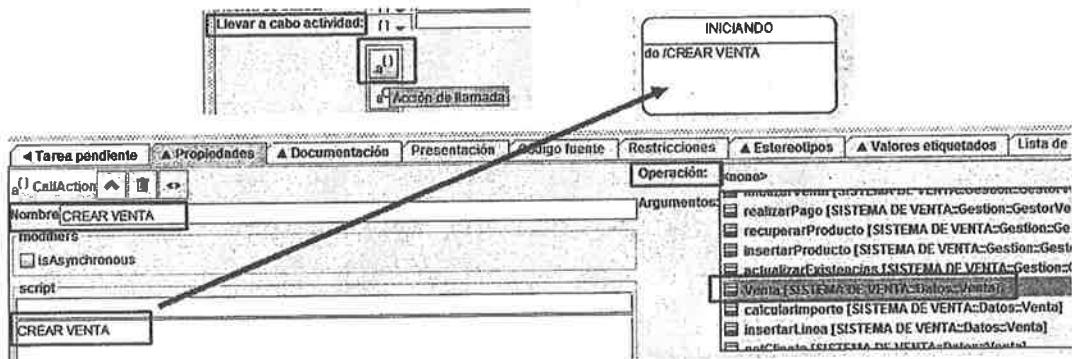


Figura 6.94. Añadir una acción *do/* en ArgoUML.

Para crear una transición pulsamos en el botón de *Crear transición* → y arrastramos el puntero del ratón desde el estado inicial al estado final dentro de la zona de edición. Desde las propiedades asignamos nombre a la transición. Para crear el evento de llamada, pulsamos en el botón a la derecha de *Disparador* ↗, véase Figura 6.95.

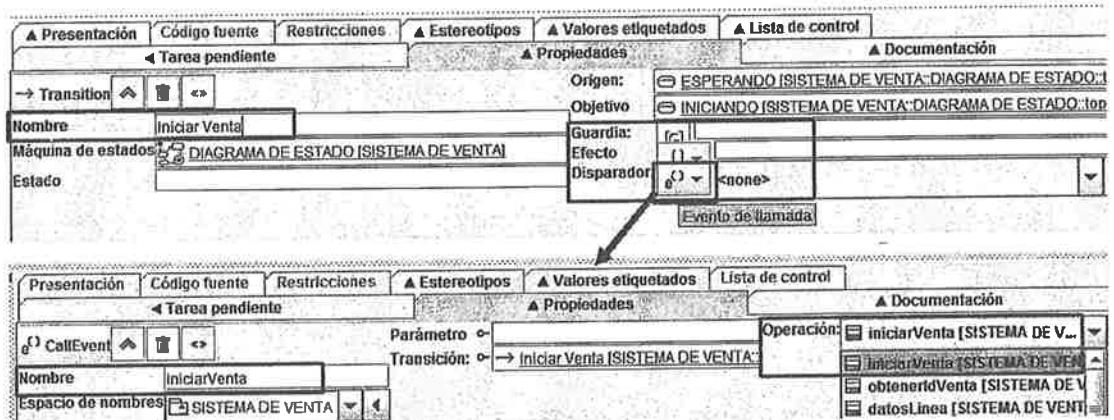


Figura 6.95. Añadir evento a una transición en ArgoUML.

Se abre una nueva ventana de propiedades para el evento de llamada, el nombre que escribimos aquí es el que se mostrará en la transición. De la lista *Operación* seleccionamos la operación *iniciarVenta*, véase Figura 6.95.

Para crear la acción pulsamos en el botón de acción de llamada que se muestra a la derecha de *Efecto* ↗. Se abre una nueva ventana de propiedades para la acción de llamada. Escribimos un nombre y debajo de *script* escribimos el nombre que se mostrará en la transición. De la lista *Operación* seleccionamos la operación *obtenerIdVenta*, véase Figura 6.96.

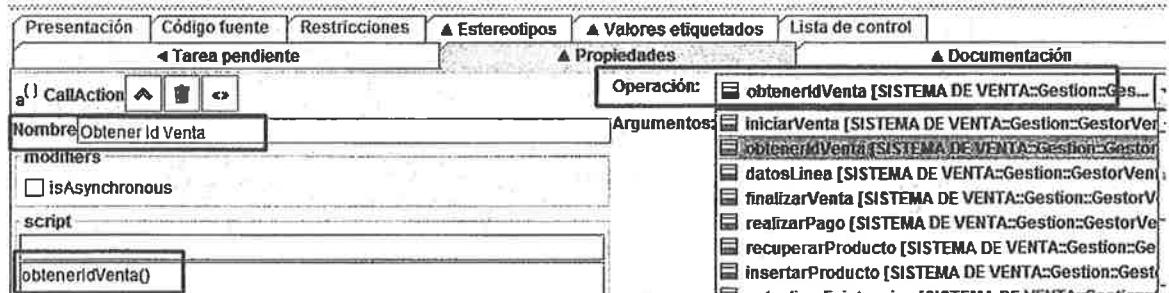


Figura 6.96. Añadir acción a una transición en ArgoUML.

La Figura 6.97 muestra los elementos generados hasta el momento en el diagrama de estado, se muestran dos estados y dos transiciones. En el navegador de elementos de la izquierda se muestra una de las transiciones tanto en el estado inicial como en el final. En el estado INICIANDO se muestra la acción *do/* creada. Se observa también que el evento *iniciarVenta* aparece definido fuera del diagrama de estados.

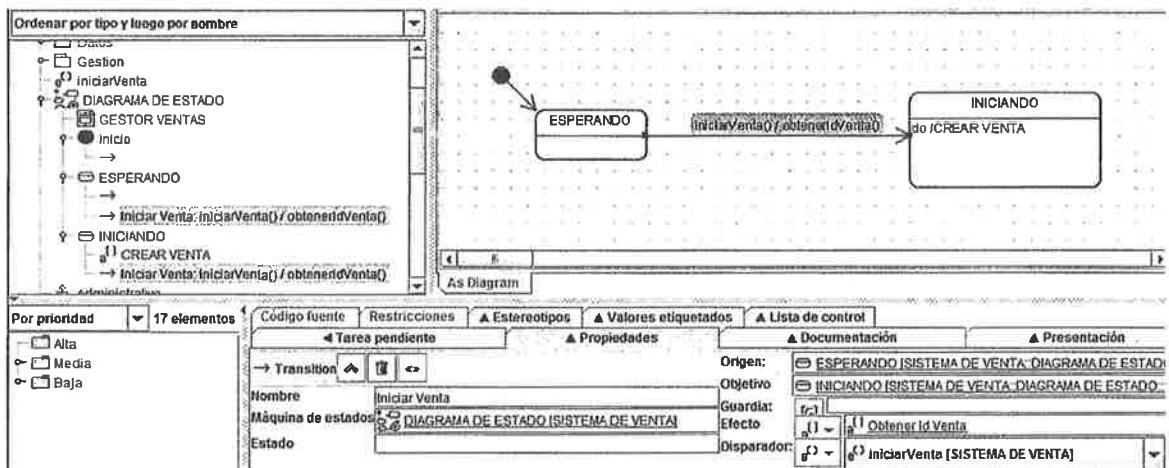


Figura 6.97. Propiedades de una transición en ArgoUML.

Para definir varias acciones de entrada, de salida o de actividad; dentro de las propiedades del estado, pulsamos sobre el botón *Nueva secuencia de acciones* desde la lista *Acción Entrada*, *Acción de salida* o *Llevar a cabo actividad*, dependiendo del lugar donde queramos añadir varias acciones. Se abre una nueva ventana de propiedades, escribimos un nombre para el conjunto de acciones y a continuación pulsamos en el botón de acción que nos interese, véase Figura 6.98.

A continuación se van añadiendo las acciones y asignando la operación como se vio en la Figura 6.94. Según se vayan incluyendo nuevas acciones se irán mostrando en las propiedades y en el estado.

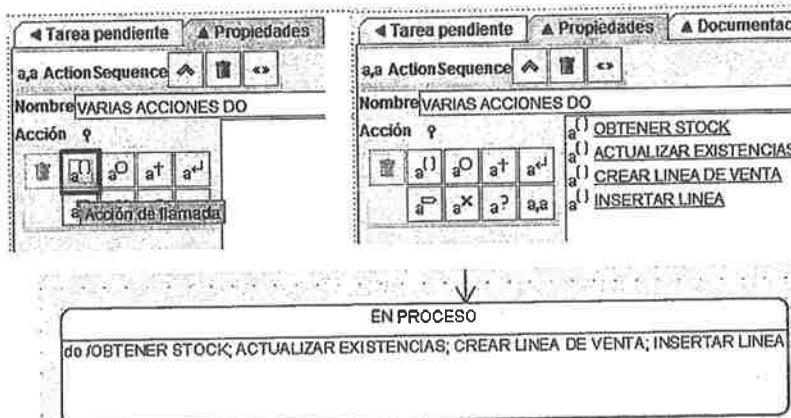


Figura 6.98. Añadir varias acciones a un estado en ArgoUML.

6.6. DIAGRAMA DE ACTIVIDAD

El diagrama de actividad es una variante del diagrama de estado, muestra el flujo de control entre actividades. Sirven para modelar: un proceso de negocio o un flujo de trabajo (workflow), el flujo de control de una operación o un caso de uso y la dinámica de un conjunto de objetos.

Los elementos básicos de un diagrama de actividad son los siguientes:

SÍMBOLO	FUNCIÓN	NOTACIÓN
Nodo inicial	Es el punto de partida del diagrama de actividad	●
Nodo final de actividad	Representa el término de la ejecución de las actividades de un diagrama. Puede haber 0 o más estados finales	●
Flujo de control	Se muestra como una flecha, determina la actividad que va a continuación de otra	→
Actividad o acción	Se dibuja como una caja con las esquinas redondeadas con el nombre en su interior	Nombre
Bifurcación o fusión	Se muestra como un rombo de decisión. La bifurcación puede tener un flujo de entrada y dos o más de salida. En cada flujo de salida se coloca una condición de guarda En la fusión los caminos antes separados se pueden volver a juntar en un rombo con varias entradas y una salida. En este caso no hay condición de guarda en la entrada	
División o unión	Se muestra mediante múltiples flechas saliendo o entrando en una barra gruesa de sincronización La división representa la separación de un flujo de control sencillo en dos o más flujos de control concurrentes. Tienen una transición de entrada y dos o más de salida La unión marca el fin del flujo de actividades concurrentes. Tienen dos o más transiciones de entrada y una de salida	

Una **actividad** es una especificación de comportamiento que describe los pasos secuenciales y concurrentes de un procedimiento de cómputo. Los flujos de trabajo, algoritmos y el código de computadora son ejemplos de procedimiento que a menudo se modelan como actividades⁴. Un diagrama de actividad muestra la descomposición de una actividad en sus componentes.

Una **acción** es un nodo de actividad atómica, es decir, el cómputo más pequeño que se puede expresar en UML. Una actividad es un conjunto de acciones que modelan un proceso; por ejemplo, enjabonar, enjuagar o secar un coche son acciones de la actividad “Lavar un coche”.

La Figura 6.99 muestra un diagrama de actividad con una bifurcación y una fusión. Se observa que en la bifurcación se indica la condición y en la fusión no es necesario. A la derecha se muestra el correspondiente diagrama de actividad sin bifurcaciones ni fusiones.

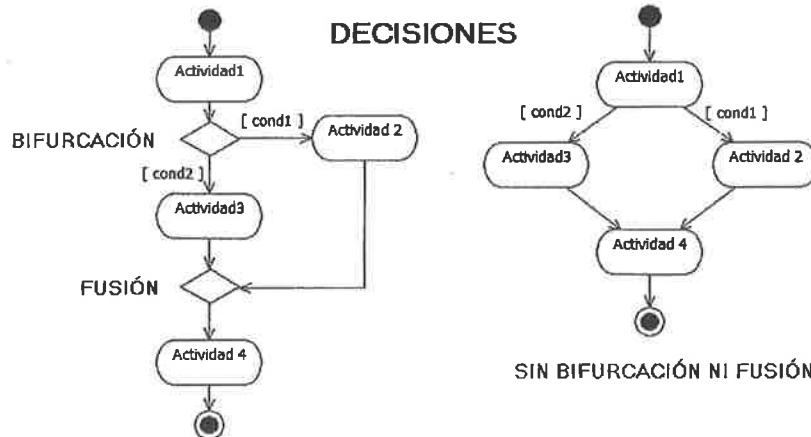


Figura 6.99. Bifurcación y fusión en diagramas de actividad.

La Figura 6.100 muestra un diagrama de actividad con una división y una unión. En la división se ve como un flujo de control sencillo, se divide en dos flujos de salida que van a dos actividades diferentes. Las actividades 2 y 3 después de la división continúan en paralelo. La unión representa la sincronización de dos o más flujos de control concurrentes, el flujo de salida de la unión se dispara cuando han finalizado todos los flujos de entrada en la unión, es decir, cuando las actividades 2 y 3 han finalizado.

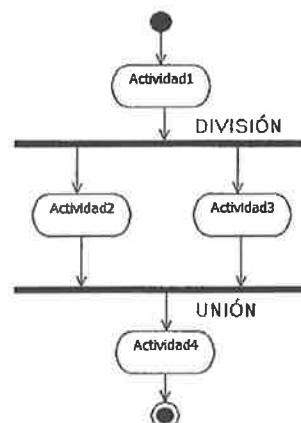


Figura 6.100. División y unión en diagramas de actividad.

⁴ El lenguaje unificado de modelado. Manual de referencia. James Rumbaugh , Ed: Addison-Wesley, 2007.

Los diagramas de actividad son los que menos se utilizan, son útiles para describir un comportamiento paralelo (por ejemplo en aplicaciones multitarea), o para mostrar qué comportamientos interactúan entre varios casos de uso. Para mostrar cómo colaboran los objetos usamos el diagrama de interacción y para mostrar cómo cambia un objeto a lo largo de su vida se utiliza el diagrama de estado.

A continuación se muestra el diagrama de actividad para el caso de uso de *ComprarProductos*, Figura 6.101, se observa un gran parecido con los diagramas de flujo.

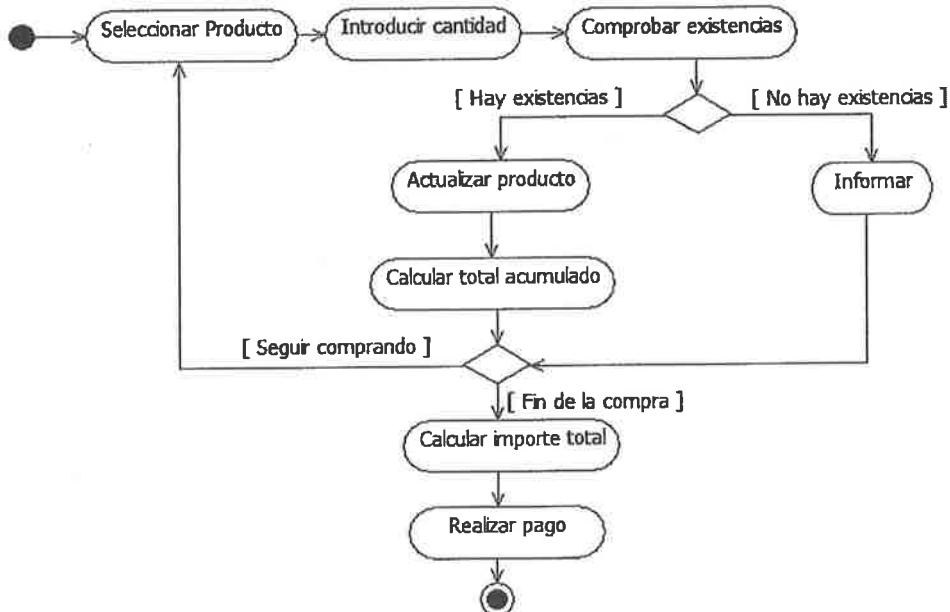


Figura 6.101. Diagrama de actividad para el caso de uso de *ComprarProductos*.

6.6.1. Elaboración de diagramas de actividad

A continuación vamos a ver cómo construir diagramas de actividad sencillos en diferentes entornos.

ECLIPSE

Para crear un diagrama de actividad, pulsamos con el botón derecho del ratón en *<Model>* y seleccionamos *New Representation -> Activity Diagram*, también se puede crear dentro de un paquete. Escribimos un nombre o dejamos el nombre por defecto y pulsamos el botón *OK*.

En la ventana de edición se muestra un marco y la barra de herramientas a la derecha, Es dentro de este marco donde hemos de incluir los elementos del diagrama, véase Figura 6.102. Los elementos básicos para crear un diagrama de actividad son los siguientes:

- *Initial Node*: para crear el nodo inicial del diagrama.
- *Activity Final Node*: para crear el nodo final del diagrama de actividad.
- *Control Flow*: se utiliza para crear el flujo de control entre las actividades.
- *Activity State*: se utiliza para crear un nodo de actividad o acción.

- **Decision Node:** Se utiliza en las decisiones, para crear una bifurcación. Se muestra con color azul.
- **Merge Node:** se utiliza para crear la fusión de caminos separados. Se muestra con color morado.
- **Fork Node:** se utiliza para la separación de un flujo de control sencillo en dos o más flujos de control concurrentes.
- **Join Node:** se utiliza para marcar el fin del flujo de actividades concurrentes.

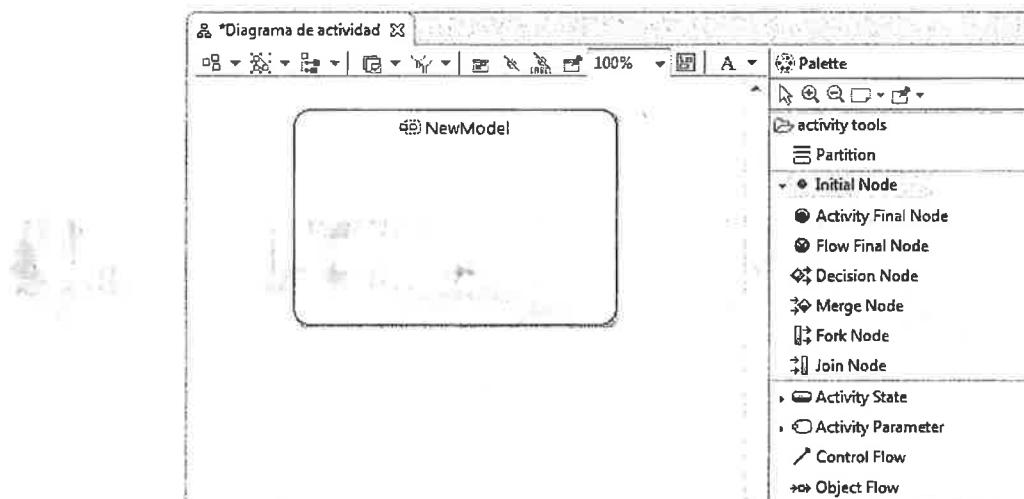


Figura 6.102. Creación de diagrama de actividad y barra de herramientas en Eclipse.

La forma de añadir los elementos es como en otros diagramas, se pulsa sobre el botón y seguidamente en el área de edición. Para añadir la condición en una bifurcación pulsamos sobre el flujo hasta que aparezca un cuadrito en blanco donde escribiremos la condición, véase Figura 6.103.

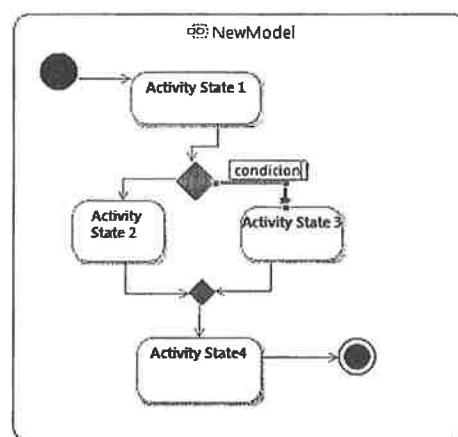


Figura 6.103. Creación de la condición en una bifurcación en Eclipse.

WHITESTARUML

Para crear un diagrama de actividad desde el modelo, pulsamos con el botón derecho del ratón y seleccionamos *Add Diagram-> Activity Diagram*. Se muestra a la izquierda la barra de herramientas, se observa que los elementos se parecen mucho a los elementos del diagrama de estado, véase Figura 6.104. Los elementos que utilizaremos para crear un diagrama de actividad sencillo son los siguientes:

- ● *Initial State*: se utiliza para crear el nodo inicial del diagrama.
- ● *Final State*: se utiliza para crear el nodo final.
- ↗ *Transition*: se utiliza para crear el flujo de control entre las actividades.
- ○ *ActionState*: para crear un nodo de actividad en el diagrama.
- ◊ *Decision*: se utiliza para representar la bifurcación y fusión.
- ▷ *Synchronization*: se utiliza para representar la división y la unión.

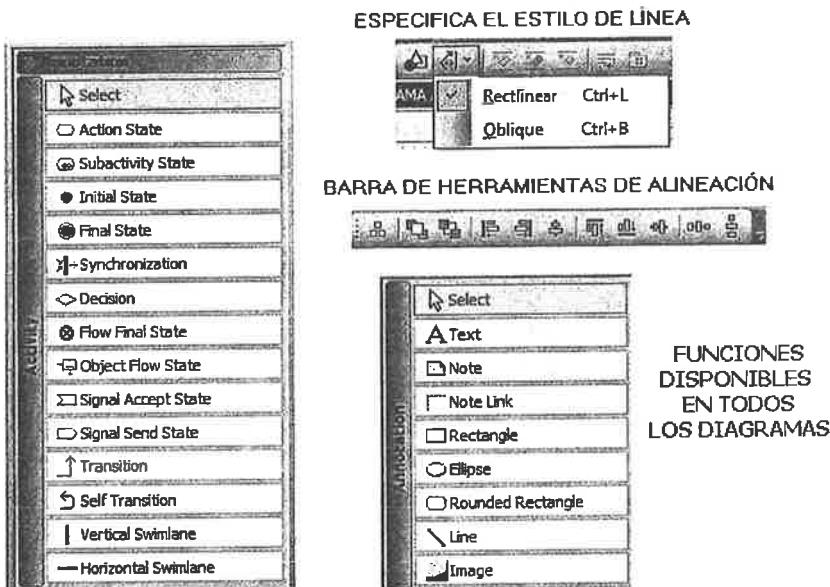


Figura 6.104. Barra de herramientas para diagramas de actividad en WhiteStarUML y otras herramientas.

En la Figura 6.104 se muestran diferentes herramientas que se utilizan para dar estilo a las líneas de los diagramas o para alinear los elementos. Las funciones de la barra de *Annotation* están siempre disponibles en la paleta de herramientas, independientemente del tipo de diagrama a desarrollar.

Para añadir los elementos pulsamos en el ícono y después sobre el área de edición. A las actividades le asignamos un nombre. Para añadir la condición en una bifurcación, pulsamos sobre el flujo y desde la ventana de propiedades añadimos la condición en la propiedad *GuardCondition*, véase Figura 6.105. También se pueden añadir condiciones en los flujos aunque no exista un símbolo de decisión, en la Figura 6.99 se muestra un ejemplo donde se define una condición en un flujo que no sale de una decisión.

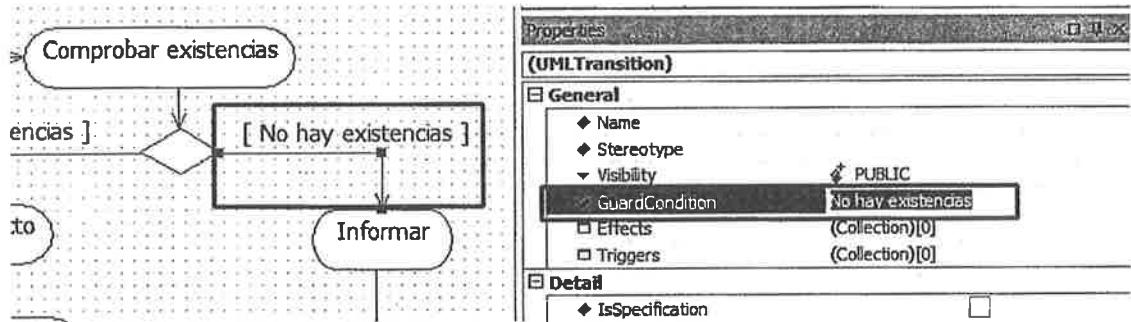


Figura 6.105. Especificar condición de una decisión en WhiteStarUML.

ARGOUML

Para crear el diagrama de actividad, pulsamos con el botón derecho del ratón sobre el modelo, a continuación elegimos *Crear diagrama -> Diagrama de actividades*. Se muestra la pantalla de edición y encima la barra de herramientas con los botones que se pueden usar para la elaboración del diagrama, véase Figura 6.106. También se puede crear el diagrama de actividad dentro de un paquete:

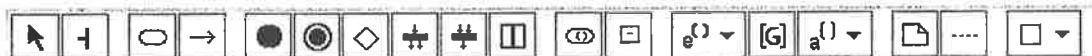


Figura 6.106. Barra de herramientas para el diagrama de actividad en ArgoUML.

Para crear un diagrama de actividad sencillo utilizaremos los siguientes botones:

- *Crear estado inicial*: se utiliza para crear el nodo inicial de la actividad.
- *Crear estado final*: se utiliza para crear el nodo final.
- *Crear transición*: se utiliza para crear un flujo de control entre actividades.
- *Crear salida*: se utiliza para representar la bifurcación y fusión.
- *Acción de estado nuevo*: se utiliza para crear un nodo de actividad.
- *Crear bifurcación*: se utiliza para representar la división de flujo.
- *Crear unión*: se utiliza para representar la unión de flujo.

Para añadir los elementos, pulsamos en el ícono y después sobre el área de edición. Al nodo de actividad le asignamos el nombre haciendo clic sobre él (este nombre es distinto al definido en la propiedad *Nombre*, aunque podemos poner el mismo para identificar mejor a los elementos en el panel de la izquierda). Para añadir la condición en una bifurcación, pulsamos sobre el flujo y desde la ventana de propiedades pulsamos en el botón a la derecha de *Guardia*, se abre una nueva pestaña de propiedades, debajo de expresión escribimos la condición. Véase Figura 6.107.

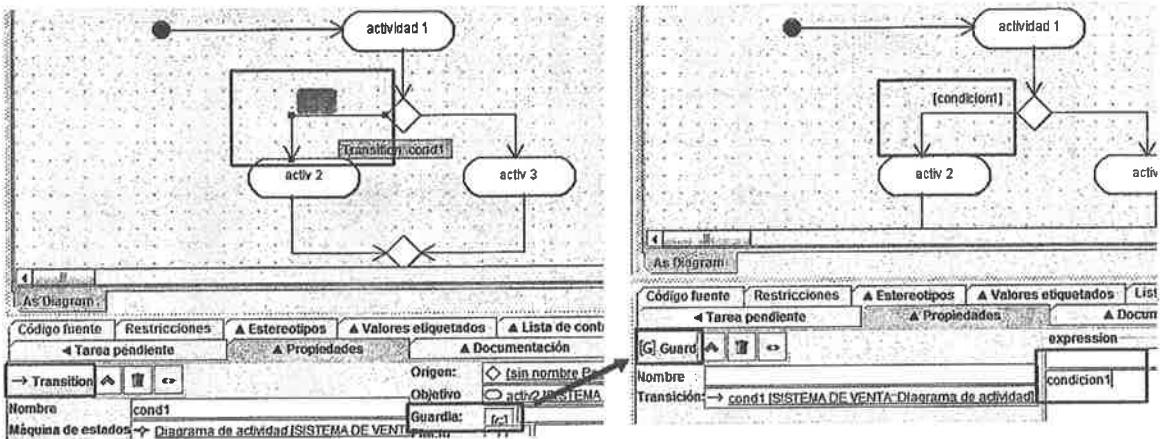


Figura 6.107. Especificar condición de una decisión en ArgoUML.

6.7. INGENIERÍA INVERSA EN ECLIPSE

En el capítulo anterior aprendimos a generar diagramas de clases con las herramientas ArgoUML y WhiteStarUML. En este capítulo aprenderemos a generar los diagramas de clases y de secuencia utilizando el entorno Eclipse.

Partimos de un proyecto Java de Eclipse, la estructura se muestra en la Figura 6.108. Está formado por los paquetes *Datos* y *Proceso*. En el primero tenemos las clases *Alumno* y *Notas* y en el segundo la clase *Principal* con el método *main()*.

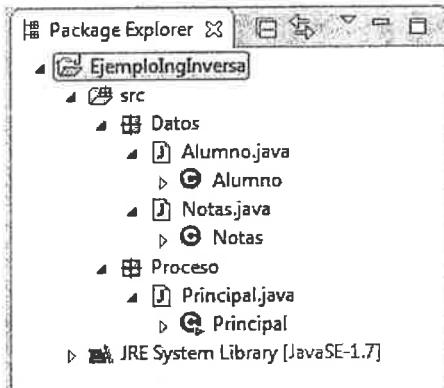


Figura 6.108. Proyecto de ejemplo en Eclipse.

La clase *Alumno* tiene 4 atributos (*dni*, *nombre*, *direccion* y *curso*), un constructor con parámetros y los métodos get y set para los atributos. La clase *Notas* tiene 3 atributos (*dni*, *asignatura* y *nota*) un constructor con parámetros y los métodos get y set para los atributos. El código de la clase *Principal* es el siguiente:

```
package Proceso;
import Datos.Alumno;
import Datos.Notas;

public class Principal {
    public static void main(String[] args) {
        Alumno alum = new Alumno{}
```

```

        ("12345678A", "JUAN", "BERROCALEJO", "1DAM");
Notas nota = new Notas
        (alum.getDni(), "ENTORNOS DE DESARROLLO", 7);
System.out.println(nota.getDni() + "*" +
        nota.getAsignatura() + "*" + nota.getNota());
}
}
}

```

En Eclipse hay varios plugins que nos permiten generar diagramas de clases con sus relaciones y diagramas de secuencia. Algunos de estos plugins son *ObjectAid UML* y *eUML2*.

Para instalar *ObjectAid UML* seguimos los siguientes pasos:

1. Pulsar en la opción de menú *Help ->Install New Software*.
2. Pulsar en el botón *Add*, en *Name* escribir un nombre, por ejemplo *OBJECT AID* y en *Location* escribimos la URL <http://www.objectaid.net/update>, pulsamos el botón *OK*. Véase Figura 6.109.

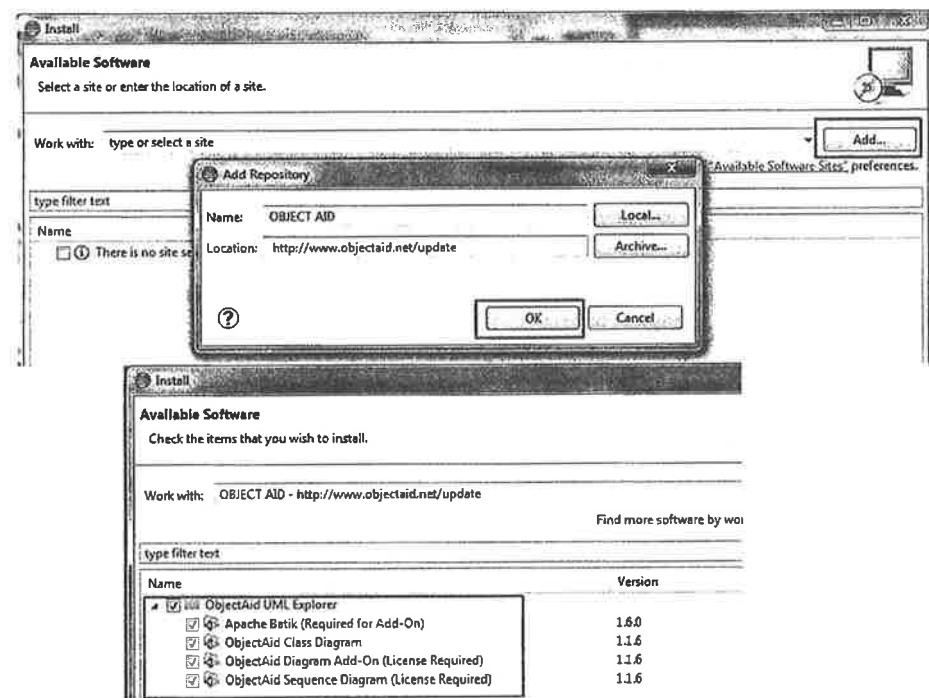


Figura 6.109. Instalación del plugin Object AID en Eclipse.

3. A continuación seleccionar *ObjectAid UML Explorer* y pulsar el botón *Next..*, se muestran los detalles de la instalación, pulsar de nuevo en el botón *Next..* A continuación aceptar los términos de licencia y se pulsa el botón *Finish*.
4. Se muestra una ventana avisando de que estamos instalando software que contiene contenido sin firmar, pulsamos el botón *OK* para continuar. Finalmente el proceso de instalación solicita que reiniciemos Eclipse.

Para hacer uso del plugin y crear el diagrama de clases de nuestro proyecto Eclipse seguimos estos pasos:

- Pulsamos con el botón derecho del ratón sobre el proyecto (o sobre una carpeta o paquete) y seleccionamos *New->Other->ObjectAid UML Diagram->Class Diagram* y pulsamos el botón *Next* (véase Figura 6.110).

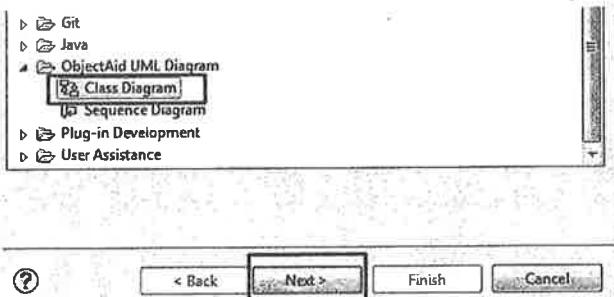


Figura 6.110. Crear diagrama de clase con *Object AID*, paso 1.

- Escribimos el nombre, marcamos las casillas para que se muestren todas las relaciones y pulsamos el botón *Finish*. Véase Figura 6.111 En el proyecto (o donde hayamos creado el diagrama) se muestra un fichero con la extensión *.ucls*.

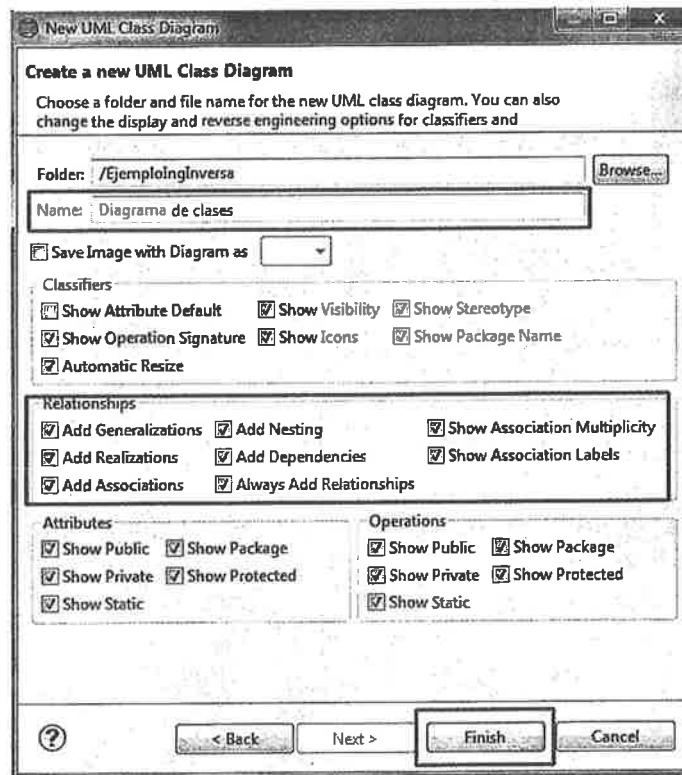


Figura 6.111. Crear diagrama de clase con *Object AID*, paso 2.

- En el ejemplo se ha creado el fichero *Diagrama de clases.ucls*. A continuación hacemos doble clic sobre el fichero creado para que se muestre el área de edición y arrastramos las clases del proyecto. El resultado final se muestra en la Figura 6.112, en el diagrama se muestran de forma automática las relaciones entre las clases.

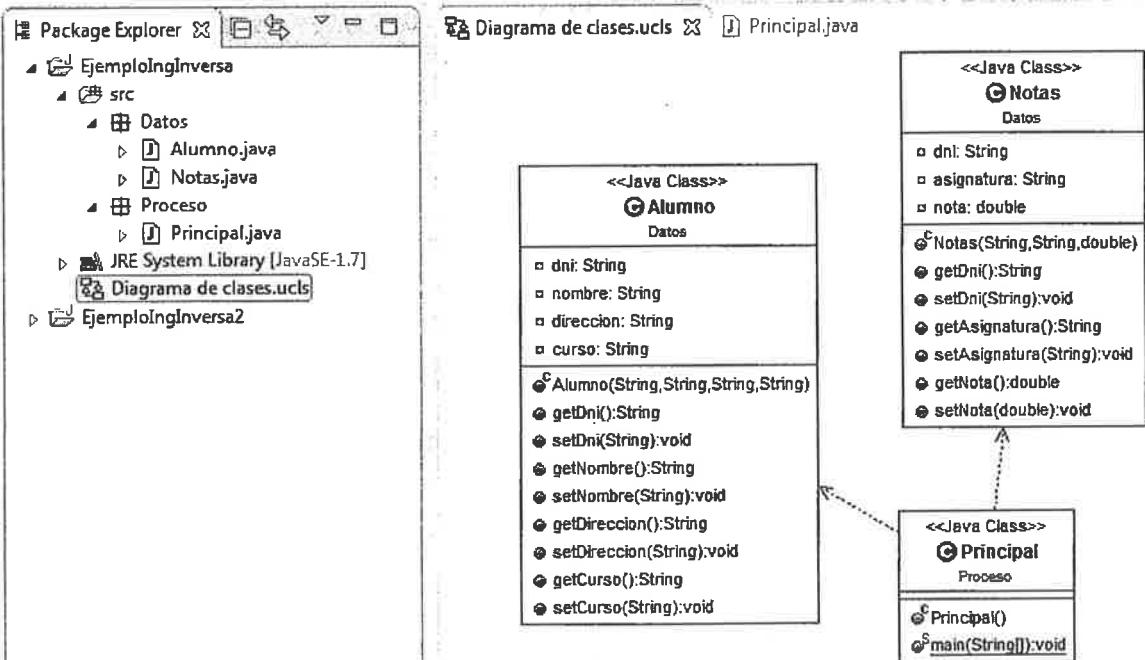


Figura 6.112. Crear diagrama de clase con Object A/D, paso 3.

ACTIVIDAD 6.11

Añade el siguiente código a la clase *Alumno*, donde se crea un nuevo atributo de tipo *ArrayList* de objetos *Notas* y los métodos *get* y *set*:

```
ArrayList<Notas> notas;
public ArrayList<Notas> getNotas() {
    return notas;
}
public void setNotas(ArrayList<Notas> notas) {
    this.notas = notas;
}
```

Genera después el diagrama de clases. ¿Hay alguna diferencia con respecto al mostrado en la Figura 6.112?

Para poder generar el diagrama de secuencia necesitamos obtener una licencia. Desde la URL de *Object AID* <http://www.objectaid.com/install-license>, se muestran los pasos a seguir para obtener una gratuita y configurarla en el entorno Eclipse. La licencia gratuita tiene una duración de un mes.

Una vez que tenemos configurada la licencia en Eclipse, pulsamos con el botón derecho del ratón sobre el proyecto y seleccionamos *New->Other->ObjectAid UML Diagram->Sequence Diagram* y pulsamos el botón *Next*. Escribimos el nombre y pulsamos el botón *Finish*. Se habrá creado un nuevo fichero con la extensión *useq*, véase Figura 6.113.

A continuación arrastramos la clase *Principal* a la zona de edición del diagrama de secuencia, véase Figura 6.113. Después arrastramos el método *main()* a la línea de vida de la clase *Principal*, véase Figura 6.114.

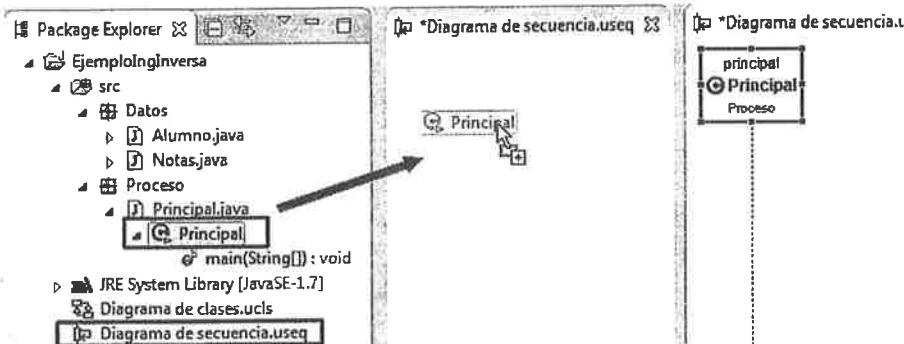


Figura 6.113. Diagrama de secuencia en Object AID, paso 1.

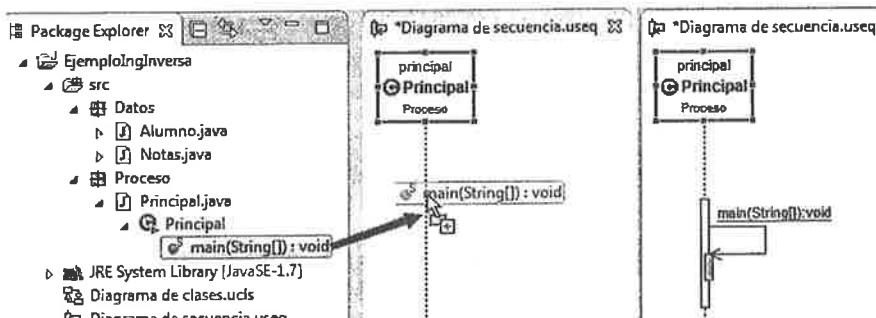


Figura 6.114. Diagrama de secuencia en Object AID, paso 2.

Por último, pulsamos con el botón derecho del ratón sobre el método *main()* y seleccionamos *Add Called Operations*, se generarán las llamadas a los métodos, véase Figura 6.115.

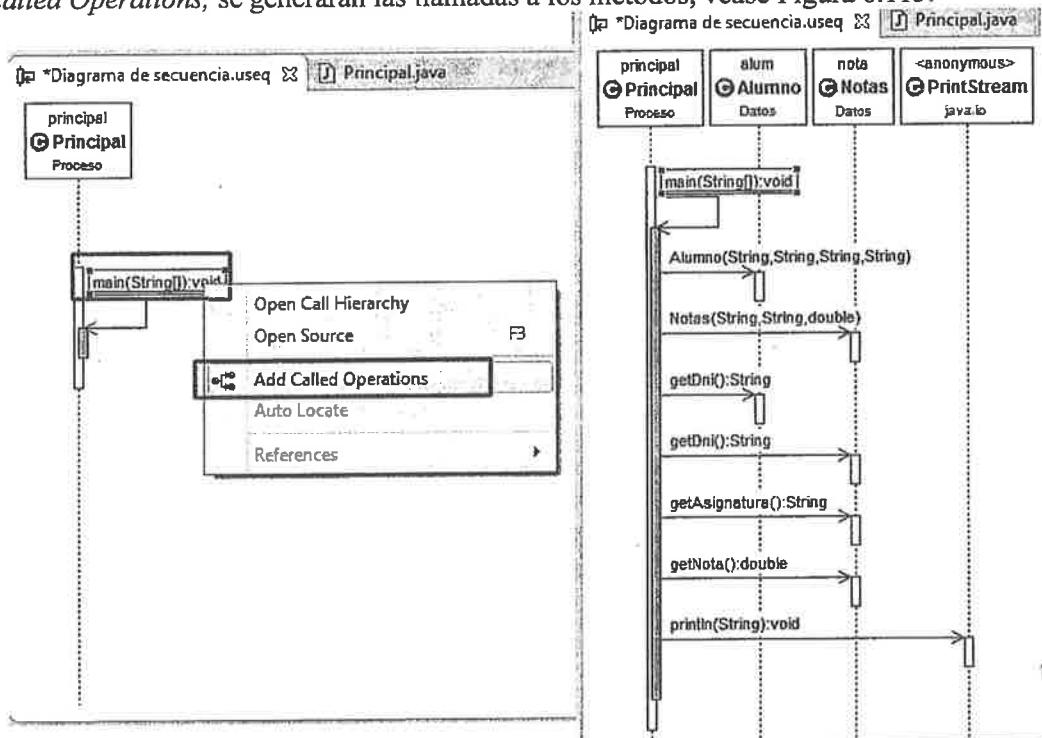


Figura 6.115. Diagrama de secuencia en Object AID, paso 3.

Para instalar el plugin eUML2 seguimos los siguientes pasos:

1. Desde la URL <http://www.soyatec.com/euml2/installation/> descargamos una licencia de evaluación para probar eUML2 (véase Figura 6.116). Para Kepler descargamos el fichero *eUML2-Studio-Edition-4.0.1.20131210+dependencies_for_eclipse4.3.zip*, y lo guardamos en la carpeta C:\PLUGIN. (La licencia tiene una duración de un mes).

	Free Edition	Studio Edition (including eUML2 Modeler, eDepend, eMF Modeler, and eDatabase)
Kepler (eclipse 4.3.x)	---	4.0.1.20131210 An evaluation license is bundled inside
Juno (eclipse 4.2.x)	---	4.0.0.20120830 An evaluation license is bundled inside
Indigo (eclipse 3.7.x)	3.7.1.20110624	3.7.1.20110624 An evaluation license is bundled inside

Figura 6.116. Descarga del plugin eUML2 para Eclipse Kepler.

2. Pulsar en la opción de menú *Help ->Install New Software*. A continuación se pulsa el botón *Add*, seguidamente se pulsa en *Archive* para localizar el fichero que se ha descargado. Se pulsa el botón *OK*, véase Figura 6.117.

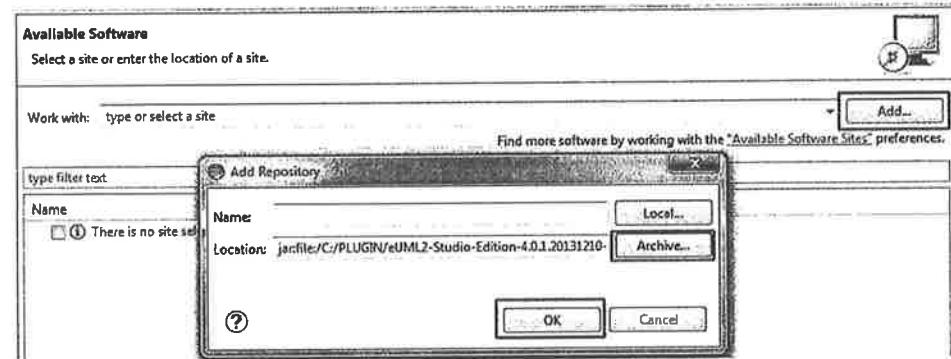


Figura 6.117. Instalación del plugin eUML2, paso 2.

3. Seleccionar todos los elementos y pulsar el botón *Next* (Figura 6.118), se muestran los detalles de la instalación, pulsar de nuevo el botón *Next*. Aceptar los términos de la licencia y pulsar el botón *Finish*. Comienza el proceso de instalación (se necesita conexión a Internet). Al finalizar nos pide reiniciar Eclipse para que los cambios tengan efecto.

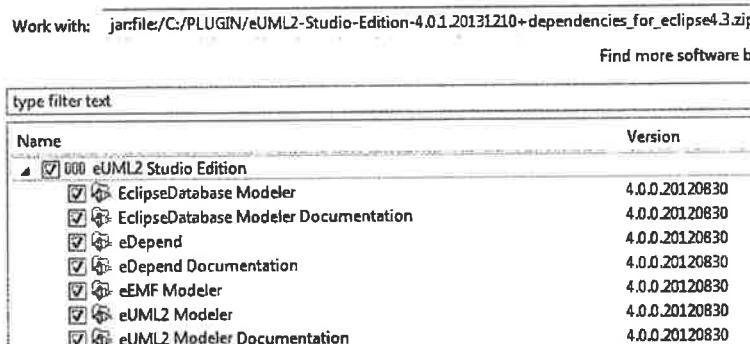


Figura 6.118. Instalación del plugin eUML2, paso3.

2. En la siguiente ventana seleccionamos los mensajes que queremos mostrar en él y pulsamos *OK*, véase Figura 6.124.
3. Automáticamente se genera el diagrama, véase Figura 6.125. Se crea un nuevo fichero, en este caso con la extensión *usd* que contiene el diagrama, lo guardamos.

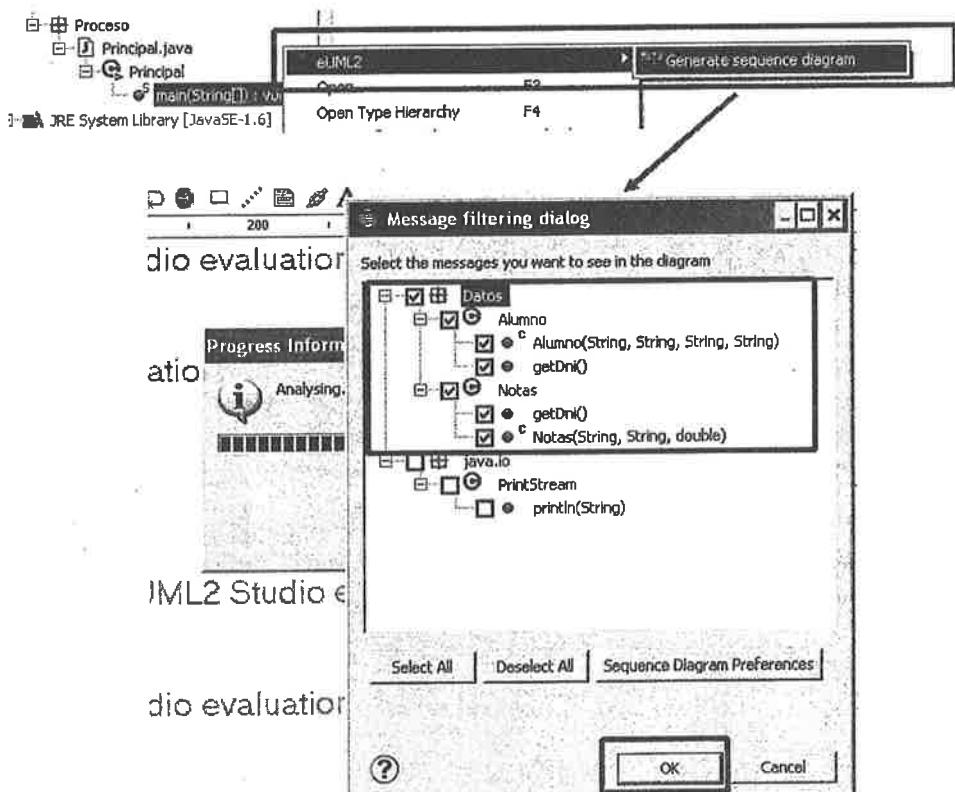


Figura 6.124. Crear diagrama de secuencia con eUML2.

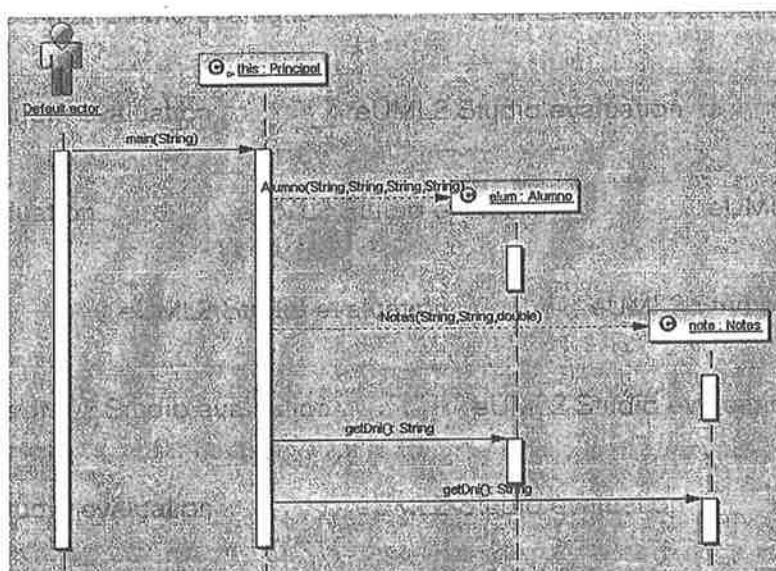


Figura 6.125. Diagrama de secuencia generado con eUML2.

COMPRUEBA TU APRENDIZAJE

1. Construye el diagrama de casos de uso para el siguiente enunciado: en una oficina se lleva a cabo la gestión de proyectos. La única persona que controla los proyectos es el administrador, cuyas funciones son las siguientes:
 - Puede agregar, eliminar y actualizar un proyecto, pero para eliminar y actualizar es necesario encontrar el proyecto en cuestión.
 - A la hora de actualizar un proyecto se pueden dar dos situaciones: cambiar la información sobre las tareas del proyecto o cambiar los recursos asociados al proyecto.
 - Para informar a todos los miembros del equipo sobre los avances en el proyecto se envía un documento por e-mail.

Utiliza las relaciones «include», «extend» y de generalización que consideres.

2. Escribe un diagrama de casos de uso para modelar la interacción de un cliente y un empleado de un banco con un cajero automático. No utilices relaciones «extend» e «include». Las especificaciones son las siguientes:
 - El cajero automático lo puede utilizar el cliente del banco y el empleado de la sucursal.
 - El cliente debe identificarse en el cajero antes de realizar cualquier operación.
 - El cliente puede cambiar el pin, sacar dinero, consultar el saldo y consultar los últimos movimientos.
 - El empleado utiliza el cajero únicamente para reponer dinero.
3. Realiza una segunda aproximación del diagrama del ejercicio 2 considerando la relación «extend».
4. Realiza una tercera aproximación del diagrama del ejercicio 2 considerando la relación «include».
5. Realiza la descripción del caso de uso de sacar dinero del ejercicio 2. La situación que vive el cliente al retirar el dinero es la siguiente: el cliente se acerca al cajero automático de su banco, introduce la tarjeta, escribe el pin, solicita retirar dinero y escribe la cantidad a retirar. El cajero le entrega el dinero solicitado siempre y cuando la operación se pueda realizar. Por último, el cliente retira la tarjeta y se va. No incluyas alternativas.
6. Completa la descripción del caso de uso anterior incluyendo las siguientes alternativas: PIN incorrecto para la tarjeta, PIN introducido de forma incorrecta durante 3 veces consecutivas y la cantidad solicitada supera el saldo.
7. Realiza un diagrama de actividad para el caso de uso de sacar dinero del cajero automático.
8. Disponemos de las siguientes clases Java incluidas en el paquete con nombre *Datos*:

<pre> package Datos; public class Puerta { private double alto; private double ancho; public Puerta(double alto, double ancho) { this.alto = alto; this.ancho = ancho; } public double getAlto() { return alto; } public void setAlto(double alto) { this.alto = alto; } public double getAncho() { return ancho; } public void setAncho(double ancho) { this.ancho = ancho; } } </pre>	<pre> package Datos; public class Ventana { private double alto; private double ancho; public Ventana(double alto, double ancho) { this.alto = alto; this.ancho = ancho; } public double getAlto() { return alto; } public void setAlto(double alto) { this.alto = alto; } public double getAncho() { return ancho; } public void setAncho(double ancho) { this.ancho = ancho; } } </pre>
<pre> package Datos; public class Habitacion { Puerta puerta; Ventana ventana; double metros; public Habitacion(double metros) { this.metros = metros; } public void setPuerta(Puerta puerta) { this.puerta = puerta; } public Ventana getVentana() { return ventana; } public void setVentana(Ventana ventana) { this.ventana = ventana; } public double getMetros() { return metros; } public void setMetros(double metros) { this.metros = metros; } } </pre>	<pre> package Datos; public class Casa { int numhabit;//num de habitaciones Habitacion [] habitaciones; int n; //para llenar el array public Casa(int numhabit) { this.numhabit=numhabit; habitaciones = new Habitacion[numhabit]; n=0; } public void addHabitacion (Habitacion h){ habitaciones[n] = h; n++; } public Habitacion[] getHabitaciones() { return habitaciones; } } </pre>

Se pide realizar el diagrama de secuencia para la operación *main()* de la clase *ConstruirCasas*:

```

package Proceso;
import Datos.*;

public class ConstruirCasas {
    public static void main(String[] args) {
        Casa casa = new Casa(1); //casa con una habitación
        Habitacion h = new Habitacion(15); //15 metros
    }
}

```

```

Puerta p = new Puerta(2.10, 1); // alto 2.10, ancho 1

h.setPuerta(p);
Ventana v = new Ventana(1.60, 1.20); // alto 1.60 ancho 1,20
h.setVentana(v);

casa.addHabitacion(h); // añadir habitación a la casa

Habitacion hab[] = casa.getHabitaciones();
System.out.println("Número de Habitaciones:" + hab.length);
}
}

```

9. Construye el diagrama de secuencia para la operación *main()* de la clase *ConstruirCasa2*:

```

package Proceso;
import Datos.*;

public class ConstruirCasa2 {
    public static void main(String[] args) {
        Casa casa = new Casa(1); // casa con una habitación
        boolean conventana = true;
        Habitacion h = new Habitacion(15);
        Puerta p = new Puerta(2.10, 1); // alto 2.10, ancho 1
        h.setPuerta(p);

        if (conventana) {
            Ventana v = new Ventana(1.60, 1.20); // alto 1.60 ancho 1.20
            h.setVentana(v);
        }

        casa.addHabitacion(h); // añadir habitación a la casa

        Habitacion hab[] = casa.getHabitaciones();
        System.out.println("Número de Habitaciones:" + hab.length);
    }
}

```

10. Construye el diagrama de secuencia para la operación *main()* de la clase *ConstruirCasa3*:

```

package Proceso;
import Datos.*;

public class ConstruirCasa3 {
    public static void main(String[] args) {
        // Crear una casa con 4 habitaciones
        Casa casa = new Casa(4);
        boolean conventana = true;

        for (int i = 0; i < 4; i++) {
            Habitacion h = new Habitacion(15);
            Puerta p = new Puerta(2.10, 1); // alto 2.10, ancho 1
            h.setPuerta(p);

            if (conventana) {
                Ventana v = new Ventana(1.60, 1.20);
                h.setVentana(v);
            }
            casa.addHabitacion(h);
        }
    }
}

```

```

        Habitacion hab[] = casa.getHabitaciones();
        System.out.println("Número de habitaciones:" + hab.length);
    }
}

```

11. Crea el diagrama de colaboración de los diagramas de secuencia creados anteriormente. Utiliza para ello la herramienta WhiteStarUML.
12. La Figura 6.126 muestra un diagrama de secuencia de una aplicación java en la que un usuario se identifica ante el sistema. Construye a partir de este diagrama las clases y métodos que intervienen en la aplicación:

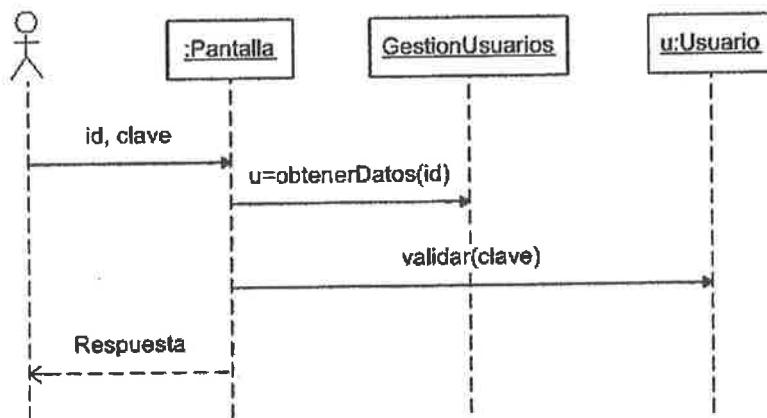


Figura 6.126. Diagrama de secuencia del Ejercicio 12.

13. Realiza el diagrama de casos de uso para una empresa de Radiotaxis que ha solicitado el desarrollo de un sistema que le apoye en sus procesos clave. Hay tres tipos de usuarios: administrativos, choferes, y el gerente. Los requerimientos son los siguientes:

Los administrativos de la empresa de Radiotaxis podrán:

- Dar de alta nuevos clientes.
- Dar de alta reservas de viajes indicando el cliente, el chofer solicitado, la dirección de origen, de destino y la fecha y hora de salida. Será necesario consultar los datos del chofer solicitado para comprobar si está disponible. Si al dar de alta una reserva, el cliente no existe en el sistema se podrá dar de alta directamente. Desde aquí también se podrá confirmar la reserva que se está dando de alta.
- Confirmar y cancelar las reservas ya dadas de alta.

Los choferes podrán consultar las reservas que tienen asignadas para el día de la fecha.

El gerente podrá dar de alta nuevos choferes al sistema y liquidar las comisiones de los choferes mensualmente, para ello se necesita consultar la información del chofer.

14. Realiza la descripción del caso de uso de dar de alta reservas de viajes. Considera los pasos que lleva a cabo el administrativo para dar de alta las reservas. Ten en cuenta varias alternativas: si el cliente no existe se puede dar de alta, chofer no disponible o inexistente, datos incorrectos, etc.

15. Realiza un diagrama de estado para un torno de metro. La situación es la siguiente: inicialmente el torno está bloqueado. Si el usuario introduce el ticket el torno se desbloquea para que pase, una vez que el usuario ha pasado el torno vuelve al estado de bloqueado. Si el usuario intenta pasar con el torno bloqueado se emite una alarma.
16. Realiza un diagrama de estado para un reproductor de mp3. La situación es la siguiente: inicialmente el reproductor está parado. Si se pulsa el botón play empieza a reproducir música, el estado pasa a parado si terminan las canciones o si se pulsa el botón de stop. Si se pulsa el botón pausa, y el reproductor está sonando entonces cambia de estado; de este estado se sale cuando se pulsa el botón stop o el botón play.
17. Se desea cambiar la hora y los minutos de un reloj digital. Para ello el reloj dispone de los botones A y B. Inicialmente el reloj muestra la hora y los minutos. Si pulso el botón A la hora parpadea, en esta situación si pulso el botón B se cambia la hora. Si pulso A de nuevo los minutos parpadean, en esta situación si pulso el botón B se cambian los minutos. Por último, si pulso el botón A la hora se muestra sin parpadear. Realiza el diagrama de estados para este reloj.
18. ¿Cuál de las siguientes afirmaciones sobre los diagramas de estados es correcta?
 - a) Son adecuados para describir el comportamiento de varios objetos en un mismo caso de uso.
 - b) Son adecuados para describir el comportamiento de un objeto.
 - c) Son adecuados para mostrar la secuencia general de acciones de varios objetos y casos de uso.
19. ¿Cuál de las siguientes afirmaciones sobre los diagramas de interacción es correcta?
 - a) Son adecuados para describir el comportamiento de varios objetos en un mismo caso de uso.
 - b) Son adecuados para describir el comportamiento de un objeto.
 - c) Son adecuados para mostrar la secuencia general de acciones de varios objetos y casos de uso.
20. ¿Cuál de las siguientes afirmaciones sobre los diagramas de actividad es correcta?
 - a) Son adecuados para describir el comportamiento de varios objetos en un mismo caso de uso.
 - b) Son adecuados para describir el comportamiento de un objeto.
 - c) Son adecuados para mostrar la secuencia general de acciones de varios objetos y casos de uso.

ACTIVIDADES DE AMPLIACIÓN

1. Se desea representar el diagrama de casos de uso del funcionamiento de una empresa agrícola dedicada a la siembra de cereales. En la empresa las funciones del personal son las siguientes:
 - El obrero se encarga de preparar el terreno para la siembra, sembrar, controlar la cosecha, recolectar la cosecha y enviarla al almacén.
 - El encargado se encarga de solicitar material para la siembra. Puede haber tres tipos de solicitudes de material: solicitar semillas, solicitar abonos y solicitar maquinaria agrícola.
 - Preparar el terreno para la siembra incluye también solicitar material para la siembra, siempre que se necesite.
 - El encargado también se encarga de enviar el informe de plagas, para ello es necesario controlar la cosecha.
2. En una gestión de almacén el Jefe de Marketing es el encargado de actualizar el catálogo de los productos que se venden en el almacén. La actualización consistirá en mantener los productos borrando o añadiendo nuevos. También puede para un producto determinado modificar sus características como el nombre, precio, etc. El proceso que hace es el siguiente:
 - La aplicación mostrará una pantalla en la que se puede ver la lista del catálogo actual. Desde esa pantalla se podrá seleccionar la operación a realizar: añadir, borrar o modificar.
 - Si se pulsa añadir se mostrará la ventana de inserción de datos, para teclear la descripción, el precio, el código, también el proveedor. El proveedor se puede elegir de una lista o se puede añadir un nuevo proveedor. En este caso se mostrará una nueva ventana para teclear sus datos y añadirlo a la base de datos. Finalmente el producto se añadirá a la base de datos.
 - Si se ha elegido borrar el producto, se selecciona del catálogo, se consulta si tiene pedidos y si no está en ningún pedido se borra del catálogo.
 - Si se ha elegido modificar el producto, se mostrará la ventana con los datos del producto para modificarlos, se teclean los datos y se modifica.
3. Realiza la descripción del caso de uso ACTUALIZAR CATÁLOGO.

BIBLIOGRAFÍA

Ingeniería del Software. Un enfoque práctico. Roger S Pressman, 2001; Mc Graw Hill, España. ISBN: 84-481-3214-9.

Análisis y diseño detallado de Aplicaciones Informáticas de Gestión. Mario G. Piattini, Jose A. Calvo-Manzano, Joaquin Cervera Bravo, y Luis Fernández Sanz. Rama. ISBN 84-7897-233-1.

Análisis y diseño detallado de Aplicaciones Informáticas de Gestión. Sara Mª García Blanco, Enrique Morales Ramos. Paraninfo. ISBN: 8497322304.

Compiladores, principios, técnicas y herramientas. Alfred V. Aho, Ravi Sethi, Murray Hill, Jeffrey D. Ullman. Pearson Addison Wesley. ISBN: 9789702611332.

Montaje y mantenimiento de equipos. Alicia Ramos, Mª Jesús Ramos, Santiago Viñas. McGraw-Hill. ISBN: 978-84-481-8036-2. Edición 2009.

El lenguaje unificado de modelado. Manual de referencia (2ª ED). Rumbaugh, Jacobson y Booch. Addison-Wesley, 2007. ISBN 9788478290871

UML gota a gota. Martin Fowler; Kendall Scott, Alhambra mexicana, S.A., 1999. ISBN 9789684443648.

UML para programadores Java. Martin, Rober C. Alhambra Longman S.A. ISBN: 978-84-205-4109-9

UML 2: Iniciación, ejemplos y ejercicios corregidos. Laurent Debrauwer, Fien Van Der Heyde. Ediciones ENI.

Programación UML. Howard Podeswa, Anaya Multimedia. ISBN 9788441527195.

UML 2 (Programación). Jim Arlow, Lla Neustadt. Anaya Multimedia. ISBN 9788441520332.

Programación orientada a objetos. 2ª Ed. Prentice-Hall, 1998. Joyanes, L. ISBN: 8448105907

VisualSVN Server: <http://www.visualsvn.com/server/>

TortoiseSVN: <http://tortoisessvn.net/>

Oracle SQLDeveloper:

<http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html>

The Unified Modeling Language, <http://www.uml-diagrams.org/>
<http://www.literateprogramming.com/documentation.pdf>
http://www.math.unipd.it/~tullio/IS-1/2009/Approfondimenti/ISO_12207-2008.pdf
<http://alarcos.esi.uclm.es/per/fruiz/curs/mso/comple/IEEE1219.pdf>
<http://www.bvindecopi.gob.pe/normas/isoiec12207.pdf>
<http://temariotic.wikidot.com/lenguajes-de-programacion>
http://zarza.usal.es/~fgarcia/doc/tuto2/I_1.htm
<http://www.revista.unam.mx/vol.1/num2/art4/index.html>
<http://ocw.uc3m.es/ingenieria-informatica/metodologia-de-desarrollo-visual>
Object Aid: <http://www.objectaid.com/>
ArgoUML: <http://argouml.tigris.org/>
eUML2: <http://www.soyatec.com/euml2/>
Eclipse marketplace: <https://marketplace.eclipse.org/>
UML Designer: <http://marketplace.obeonetwork.com/module/uml>
WhiteStarUML: <http://sourceforge.net/projects/whitestaruml/>



Entornos de Desarrollo

El contenido de este libro tiene una orientación práctica. En el Capítulo 1 se estudian los procesos, actividades y tareas involucradas en el desarrollo, explotación y mantenimiento de un producto de software. Se estudia la relación entre un programa informático y los distintos componentes del ordenador, se analizan diferentes lenguajes de programación, se estudian los diferentes estados por los que pasa un programa desde que se escribe hasta que se ejecuta en el ordenador, se introduce el concepto de máquina virtual.

En el siguiente capítulo se estudian los entornos de desarrollo, sus funciones y características. Se estudian dos entornos de desarrollo diferentes, uno orientado al uso de bases de datos que es el **SQL Developer**, y el segundo más orientado a la programación de aplicaciones el **Eclipse**. Además se aprenderá la instalación y el manejo de los plugins **WindowBuilder** para crear ventanas para las aplicaciones Java y el plugin **ERMaster** para crear modelos de datos en distintas bases de datos como **Oracle** y **MySQL**.

En el Capítulo 3 se estudian dos enfoques para el diseño de casos de prueba y diferentes técnicas en cada uno para probar el código de los programas. Se estudia el proceso de depuración en el entorno Eclipse definiendo puntos de ruptura y examinando variables durante la ejecución de un programa. Para elaborar pruebas unitarias se utiliza la herramienta **JUNIT** en el entorno Eclipse, las pruebas se realizan sobre clases Java.

En el Capítulo 4 se estudian herramientas que permiten a un programador mejorar y optimizar el proceso de desarrollo de software; así como herramientas de control de versiones como **Subversion en Eclipse** y **SQL Developer**, además el cliente **Tortoise** y el servidor **VisualSVN**. También se aprenderá a documentar clases java utilizando **Javadoc** de **ECLIPSE**. Y por último se aprenderá a **refactorizar** proyectos en **ECLIPSE**.

En los capítulos 5 y 6 se aprenderá el lenguaje **UML** y sus diagramas. Se estudiarán a través de múltiples ejemplos los diagramas de clases y de comportamiento: casos de uso, de actividad, estado e interacción; de estos últimos se estudia el diagrama de secuencia y el de comunicación o colaboración. Se estudiarán herramientas como **ArgoUML** y **WhiteStarUML** y en Eclipse el plugin **UML2**. Al final del capítulo 6 se muestra como generar los diagramas de clases y de secuencia utilizando el entorno Eclipse a través de los plugins **Object AID** y **eUML2**.

El material disponible para la realización de los ejercicios y actividades se puede descargar del site: <https://sites.google.com/site/libroentornosdedesarrollo>.

