



Escuela Superior de Informática – Universidad de Castilla-La Mancha

Apuntes de Ingeniería del Software II

(temario de teoría)

Versión 1.5

Macario Polo Usaola

macario.polo@uclm.es

<http://www.inf-cr.uclm.es/www/mpolo>

Nota importante: Estos apuntes constituyen un complemento a la materia impartida en la asignatura Ingeniería del Software II, pero no sustituyen ni a las clases presenciales ni a la bibliografía recomendada en la Guía Docente.

Índice

Capítulo 1. Presentación de los contenidos principales de la asignatura	1
1. Conceptos básicos del paradigma orientado a objetos	1
1.01 Objeto.....	1
1.02 Clase.....	1
1.03 Ocultamiento y encapsulación.....	2
1.04 Instancia.....	2
1.05 Mensaje.....	3
1.06 Sistema orientado a objetos	4
1.07 Polimorfismo.....	4
1.08 Herencia.....	4
1.08.1 Herencia múltiple	6
1.08.2 Clases abstractas e interfaces	7
1.09 Vinculación dinámica.....	9
1.10 Sobrecarga de operadores	12
2. Representación de la estructura de problemas mediante diagramas de clases	12
2.01 Relaciones de asociación	13
2.02 Relaciones de agregación.....	13
2.03 Relaciones de dependencia.....	14
3. Introducción a la arquitectura multicapa.....	16
4. Representación de estos conceptos en lenguajes de programación.....	21
4.01 Figura 3 (página 6).....	21
4.02 Figura 6 (página 9).....	22
4.03 Figura 8 (página 10).....	22
4.04 Figura 9 (página 11) y Figura 10.....	23
4.05 Figura 11 (página 12).....	24
4.06 Figura 17 (página 16).....	24
4.07 Figura 21 (página 21).....	24
5. Ejercicios	25
5.01 Biblioteca	25
5.02 Préstamo de un libro.....	25
5.03 Traducción a código.....	25

6.	Procesos de desarrollo de software orientado a objetos.....	26
6.01	Un vistazo al Proceso Unificado.....	26
6.01.1	Fases del Proceso Unificado.....	27
6.02	Programación Externa (<i>eXtreme Programming</i> o <i>XP</i>)	28
7.	La importancia de la verificación y validación	29
	Capítulo 2. Diagramas de clases	31
1.	Introducción	31
2.	Clases.....	31
2.01	Notación para atributos	33
2.02	Notación para operaciones	34
3.	Interfaces.....	35
4.	Relaciones entre clasificadores.....	36
4.01	Asociaciones	36
4.01.1	Traducción a código de clases asociativas.....	37
4.02	Agregaciones.....	38
4.03	Asociaciones n-arias.....	38
4.04	Generalización.....	39
4.05	Dependencias	40
5.	Otros elementos de modelado.....	40
5.01	Utilidades (<i>utilities</i>).....	40
5.02	Enumeraciones.....	41
5.03	Clases anidadas	41
5.04	Tipos e implementaciones.....	41
5.05	Clases parametrizadas (plantillas).....	41
6.	Construcción de diagramas de clases en OMT	42
7.	Verificación y validación de diagramas de clases.....	43
8.	Ejercicios	44
8.01	Gestión de PFC.....	44
8.02	Gestión de solicitudes de ayuda por asistencia a eventos.....	44
8.03	Integración	45

Capítulo 3. Arquitectura multicapa (I)	47
1. Introducción	47
2. Arquitectura de tres capas	47
3. Construcción de la base de datos	50
3.01 Patrón “una clase, una tabla” (1C1T)	51
3.02 Patrón “un árbol de herencia, una tabla” (1AH1T)	51
3.03 Patrón “un camino de herencia, una tabla” (1CH1T)	52
4. Operaciones de persistencia	53
4.01 Acceso a la base de datos: patrón Agente (<i>Broker</i>)	54
4.01.1 El patrón “Singleton”	56
4.01.2 Detalles de implementación	56
4.02 Asignación de responsabilidades de persistencia	57
4.02.1 Patrón “Experto”	57
4.02.2 Patrón “Fabricación pura”	58
4.02.3 Patrón RCRUD (<i>Reflective CRUD</i>)	60
5. Ejercicios	62
5.01 Tolerancia a fallos	62
5.02 Diferentes modelos	62
5.03 Gestión de PFC	62
5.04 Ayudas por asistencia a eventos	62
5.05 Integración (I)	63
5.06 Integración (II)	63
5.07 Construcción de bases de datos	63
5.08 Escritura de código	63
Capítulo 4. Diagramas de interacción	65
1. Introducción	65
2. Diagramas de secuencia	65
2.01 Condiciones y bucles	67
3. Diagramas de colaboración	68
4. Nivel de detalle	69

5.	Verificación y validación de diagramas de interacción.....	70
6.	Ejercicios	70
6.01	Transacciones	70
6.02	Gestión de PFC	70
Capítulo 5. Análisis y especificación de requisitos		71
1.	Diagramas de casos de uso.....	71
1.01	Verificación y validación de diagramas de casos de uso	73
2.	Casos de uso y diagramas de interacción	74
3.	Derivación de una estructura de clases a partir de los diagramas de interacción...	74
4.	Los casos de uso en el Proceso Unificado.....	74
5.	Contratos a nivel de análisis	75
5.01	Cómo hacer contratos	76
5.02	Especificación de precondiciones y postcondiciones	76
5.03	Verificación y validación de contratos.....	77
6.	Contratos a nivel de diseño y codificación	77
6.01	Contratos y aserciones en Java	78
7.	Ejercicios	81
7.01	Gestión de PFC	81
7.02	Ayudas por asistencia a eventos.....	81
7.03	Contratos y aserciones	82
Capítulo 5 bis. Recapitulación.....		83
8.	Introducción	83
9.	El Proceso Unificado, de un vistazo	83
10.	El modelo de análisis.....	85
11.	Modelo de diseño.....	88
Capítulo 6. Arquitectura multicapa (II)		91
1.	Introducción	91
2.	El patrón Modelo-Vista-Controlador	91
2.01	El patrón Observador.....	95

2.02	Otros usos de los observadores	95
3.	Ejercicios	95
3.01	Escritura de código para conectar el observador y el observado (I)	95
3.02	Escritura de código para conectar el observador y el observado (II)	95
3.03	La propia pantalla actúa de observador	95
4.	Cachés de objetos	96
4.01	Compartición de instancias	98
4.02	Tipos de cachés	100
5.	Ejercicios	101
5.01	Algoritmos de gestión de cachés (I)	101
5.02	Algoritmos de gestión de cachés (II)	101
Capítulo 7. Acceso a instancias remotas		103
1.	Introducción	103
2.	Compartición de instancias remotas	103
2.01	Representación real del diseño	105
2.02	Adición de observadores remotos	106
2.03	Detalles de implementación	108
2.04	Alternativas de implementación	109
3.	Gestión de la persistencia en el servidor	111
4.	Ejemplo	112
4.01	Aplicación servidora	113
4.02	Aplicación cliente	114
Capítulo 8. Máquinas de estados		115
1.	Introducción	115
2.	Elementos de las máquinas de estados	117
3.	Ejemplos y notación	120
4.	Ejercicios	127
4.01	PFC	127
4.02	Ascensor	127
Capítulo 9. Diagramas de actividad		131

1. Introducción.....	131
2. Elementos de los diagramas de actividad.....	132
3. Ejemplos y notación.....	133
Capítulo 10. Patrones de diseño (I).....	137
1. Introducción.....	137
2. Patrones de creación.....	137
2.01 <i>Abstract factory</i> (fábrica abstracta).....	137
2.01.1 Ejercicio: parchís.....	140
2.02 <i>Builder</i> (constructor).....	140
2.02.1 Ejercicio: parchís.....	142
2.02.2 Ejercicio: alternativas de diseño e implementación.....	142
3. Patrones estructurales.....	143
3.01 Patrón <i>Adapter</i> (Adaptador).....	143
3.02 Patrón <i>Composite</i> (Compuesto).....	145
3.03 Patrón <i>Facade</i> (Fachada).....	146
3.04 Patrón <i>Flyweight</i> (Peso mosca).....	147
3.04.1 Ejemplo: las fichas del ajedrez.....	148
3.05 Patrón <i>Proxy</i>	150
4. Patrones de comportamiento.....	151
4.01 Patrón <i>Chain of responsibility</i> (cadena de responsabilidad).....	151
4.02 Patrón <i>Mediator</i> (Mediador).....	151
4.03 Patrón <i>Interpreter</i> (Intérprete).....	152
4.04 Patrón <i>State</i> (Estado).....	153
Capítulo 11. Pruebas de sistemas orientados a objeto.....	155
1. Introducción.....	155
2. Pruebas mediante mutación.....	155
2.01 Terminología.....	155
2.02 Generación de mutantes.....	156
2.03 Pruebas de caja negra mediante mutación de interfaces.....	158
3. La estrategia de pruebas de Programación Extrema.....	159

3.01	Pruebas utilizando <i>junit</i>	159
3.02	Ejemplo.....	161
3.03	Detalles de implementación de <i>junit</i>	164
Capítulo 12. Notaciones formales. Lenguaje OCL.		167
1.	Introducción.....	167
2.	Tipos predefinidos de OCL.....	167
2.01	Tipos básicos no escalares.....	167
2.02	Tipos básicos escalares.....	169
2.03	Colecciones.....	169
2.03.1	<i>Set</i> (conjunto).....	170
2.03.2	<i>Bag</i> (bolsa).....	170
2.03.3	<i>Sequence</i> (secuencia).....	171
2.03.4	Significado de algunas operaciones de las colecciones.....	171
3.	Ejemplos.....	172
4.	Tipos del modelo.....	176
5.	Precondiciones y postcondiciones.....	176
6.	Representación de diagramas de estados.....	177
7.	Enumeraciones.....	178
8.	Mensajes.....	179
Capítulo 13. Programación extrema.....		181
1.	Introducción.....	181
2.	Modelo de proceso.....	181
2.01	Planificación.....	181
2.01.1	Recepción de las historias del usuario.....	181
2.01.2	Plan de entregas (<i>release plan</i>).....	182
2.02	Diseño.....	182
2.03	Codificación.....	183
2.04	Integración continua.....	183
2.05	40 horas semanales.....	183
3.	Las cuatro variables de los proyectos de desarrollo de software.....	183

4.	Programación dirigida por las pruebas (<i>test-driven development</i>).....	184
5.	Patrones para pruebas	192
5.01	Objetos Mock (patrón <i>Mock Objects</i>).....	192
5.02	Cadena de <i>log</i> (patrón <i>Log String</i>).....	193
5.03	Pruebas de excepciones (patrones <i>Crash Test Dummy</i> y <i>Exception Test</i>) 193	
5.04	Uso del método <i>setUp</i> (patrón <i>Fixture</i>).....	193
6.	Ejercicio.....	194
7.	Pruebas de aceptación.....	195
	Capítulo 14. Desarrollo de sistemas específicos	196
1.	Introducción	196
2.	Introducción al desarrollo de aplicaciones web.....	196
2.01	Envío de parámetros a través de una URL.....	197
2.02	<i>http</i> es un protocolo sin estado	200
2.03	Arquitectura de una aplicación web.....	202
2.04	Ejercicio.....	204
2.05	Ejercicio.....	204
3.	Introducción a los Servicios web.....	204
3.01	WSDL	205
3.02	Escritura de un cliente que acceda a un servicio web	206
3.03	Ejercicio práctico.....	208
3.04	Ejercicio.....	209
3.05	Ejercicio.....	209
4.	Introducción a las aplicaciones basadas en componentes distribuidos	209
4.01	DCOM.....	210
4.02	EJB.....	210
4.02.1	Tipos de EJBs.....	211
4.02.2	Ciclo de vida de los EJB	213
4.03	Acceso al EJB desde el cliente	215
4.04	Algunas ideas sobre Ingeniería del Software Basada en Componentes	215
5.	Nociones sobre Sistemas de Información Geográfica.....	217

5.01	Tipos de datos en SIG	217
Capítulo 15. Otras arquitecturas		223
6.	Introducción	223
7.	Arquitectura <i>pipeline</i> , de “tubería y filtro” o de “flujo de datos”	223
7.01	Ejemplo	223
8.	Arquitectura de componentes	224
Índice de términos		227

CAPÍTULO 1. PRESENTACIÓN DE LOS CONTENIDOS PRINCIPALES DE LA ASIGNATURA

1. Conceptos básicos del paradigma orientado a objetos

1.01 Objeto

Entidad que es capaz de almacenar su estado (información) y que posee un conjunto de operaciones (que definen su comportamiento) que permiten examinar o modificar su estado.

1.02 Clase

Plantilla que describe la forma en que están estructurados internamente una familia de objetos. Los objetos de la misma clase tienen la misma definición de sus operaciones y de su estructura de información.

La estructura de una clase viene determinada por un conjunto de atributos o campos. En general, se denomina métodos a las operaciones que ya tienen implementación (si bien hay quien utiliza indistintamente los términos *método*, *operación*, *función*, *función miembro*, *servicio*...). Podemos referirnos al conjunto de atributos y operaciones de una clase mediante el término *miembros*.

Ejemplo. La Figura 1 muestra la representación de la clase *Tarjeta* utilizando notación UML. Podríamos suponer que se trata de la descripción de una tarjeta bancaria. Se observan tres *compartimentos* que contienen: el nombre de la clase, la lista de atributos (con su tipo y visibilidad) y la lista de operaciones (con su visibilidad, tipo de retorno y tipos y nombres de los parámetros) a la que responderán los objetos de esta clase.

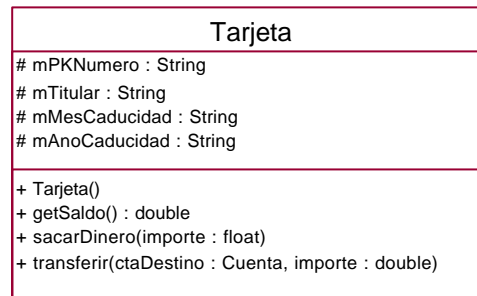


Figura 1. Una clase

1.03 Ocultamiento y encapsulación

El *ocultamiento* es la capacidad que tienen los objetos para ocultar su estructura interna a otros objetos. Es deseable que el conjunto de atributos de un objeto permanezca inaccesible al resto de objetos del sistema, de manera que, cuando un objeto A quiere modificar el estado de otro objeto B, lo haga a través del conjunto de operaciones *públicas* de B. De este modo, se garantiza que el propio objeto es el único capaz de modificar su estado. Se dice entonces que la estructura y el comportamiento del objeto están *encapsulados* en el propio objeto.

La visibilidad indica el grado en que los atributos y operaciones de una clase son accesibles a otras clases. El símbolo “#” denota que el miembro es *protegido* (accesible sólo al propio objeto y a sus descendientes); el símbolo “+” denota que el miembro es *público* (accesible a todos los objetos del sistema); el símbolo “-” denota que el miembro es *privado* (accesible sólo al propio objeto); el símbolo “~” denota que el miembro es visible a nivel de *paquete*. Además, hay lenguajes de programación que permiten formas adicionales de visibilidad: en C++ existen las clases *amigas*, que pueden acceder a los miembros privados de la clase con la cual tienen una relación de “amistad”. Estos casos especiales se resuelven con estereotipos, que son un medio que nos da UML para añadir información adicional de clasificación a cualquier elemento (Stevens, 2000). Los estereotipos se representan encerrando un texto entre comillas españolas («»).

1.04 Instancia

Objeto creado a partir de una clase, la cual describe la estructura de la instancia. El estado de la instancia en cada momento se va definiendo según se van ejecutando operaciones sobre ella.

Las instancias se crean utilizando un tipo especial de métodos llamados *constructores*, que contienen el código de inicialización de los atributos, etc. El código de un constructor puede ser bastante complejo, acceder a dispositivos externos, incluir llamadas a otros métodos o constructores de otras clases, etc., por lo que la simple creación de una instancia de una clase podría requerir una cantidad de tiempo considerable.

En muchas ocasiones, utilizaremos indistintamente los términos instancia y objeto.

1.05 Mensaje

Estímulo enviado desde una instancia a otra. En otras palabras, es la ejecución de una operación de un objeto realizada por otro objeto. Al hecho de enviar el mensaje se le llama *paso de mensajes* o *flujo de eventos*.

Ejemplo. La siguiente figura muestra con notación UML un *diagrama de secuencia* en el que se muestra el paso de mensajes que se produce cuando Paco Pil, actor que interactúa con el sistema, saca con éxito dinero de su tarjeta (a los elementos externos al sistema pero que interactúan con él se les denomina *actores*). Paco Pil le dice a su tarjeta que quiere sacar dinero, para lo cual le envía el mensaje *sacarDinero(float)*. La tarjeta comprueba que Paco Pil tiene saldo en la cuenta asociada a la tarjeta y, como estamos describiendo el *escenario* en que sí que lo tiene, se resta la cantidad solicitada de la cuenta.

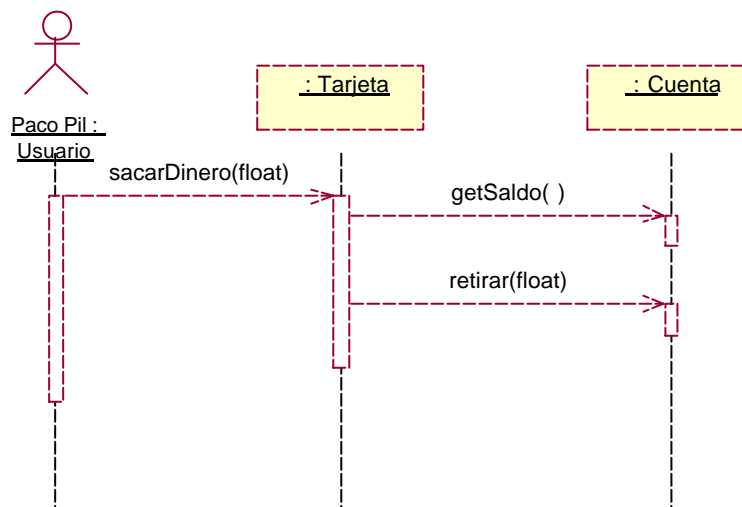


Figura 2. Un *diagrama de secuencia* que muestra un pequeño paso de mensajes.

El término *escenario* denota una situación muy concreta que se da en la ejecución de alguna funcionalidad del sistema.

1.06 Sistema orientado a objetos

Comunidad de objetos que sirve ciertas funcionalidades utilizando intensamente paso de mensajes.

1.07 Polimorfismo

El emisor de un mensaje no necesita conocer la instancia de la clase receptora; lo único que el emisor sabe es que el receptor es capaz de responder a ese mensaje. De este modo, es el receptor quien conoce la manera en que debe interpretarse el mensaje recibido.

1.08 Herencia

Capacidad que tiene una clase de heredar estructura y comportamiento de una o más clases.

La clase de la que se hereda (*clase base*, *superclase* o *supertipo*) define la estructura y el comportamiento de las clases que heredan (*clase derivada*, *subclase* o *subtipo*). Las subclases pueden añadir nuevos atributos, nuevas operaciones y redefinir operaciones heredadas. Es decir, que las subclases especializan la estructura o el comportamiento de la superclase, razón por la cual se las llama también *especializaciones*. Puesto que la superclase representa una especificación general de la estructura y comportamiento de las subclases, se dice que la superclase *generaliza* las subclases.

Para estar seguros de que una jerarquía de herencia está correctamente elegida, deberemos comprobar que sigue la *regla del 100%* y la *regla “es un”*:

- Regla del 100%: el 100% de la definición del supertipo debe ser aplicable al subtipo; además, el subtipo debe requerir el 100% de la definición del supertipo.
- Regla “es un”: todos los subtipos de un supertipo deben ser un supertipo (es decir, el subtipo es una forma del supertipo).

Junto con el polimorfismo, la herencia es uno de los mecanismos más potentes del paradigma orientado a objetos.

Ejemplo. En nuestro sistema bancario, podemos hacer una distinción entre tarjetas de crédito y débito utilizando herencia. En este caso, los dos tipos de tarjeta tienen ca-

racterísticas similares (un número de tarjeta con 16 dígitos, un titular, una fecha de caducidad, una cuenta asociada), pero la de crédito tiene una propiedad (el propio *crédito*) que no posee la de débito. Además, el comportamiento de ambas es diferente (cuando se paga con una tarjeta de débito, el importe de la compra se carga inmediatamente a la cuenta asociada; al pagar con una de crédito, la compra se resta del crédito de la tarjeta y se paga un tiempo después).

La Figura 3 representa la situación descrita con un diagrama de clases en notación UML.

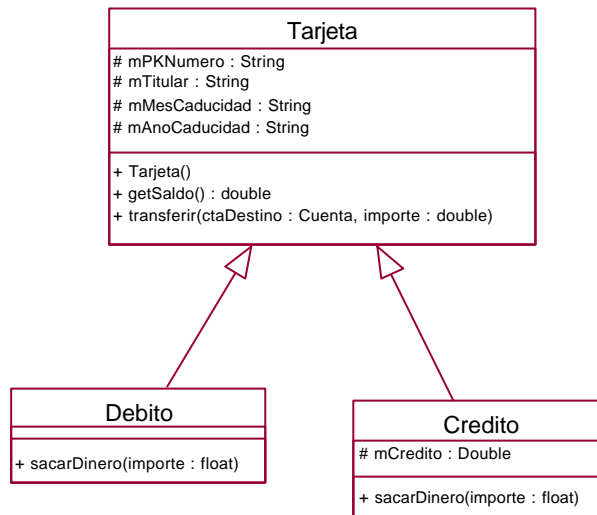


Figura 3. Un pequeño árbol de herencia.

Ejemplo. Un error frecuente consiste en crear jerarquías de herencia en las que no se cumple alguna o ninguna de las dos reglas arriba citadas. Supongamos que existen unas “tarjetas monedero” cuya única función es realizar pequeños pagos: la tarjeta posee una memoria EPROM que se puede cargar en cajeros automáticos con un máximo de 120 €. Cuando se realiza un pago en un comercio, el comerciante introduce la tarjeta en un lector no conectado a ningún tipo de línea; el lector resta el importe de la tarjeta de la EPROM y almacena en su propia memoria el importe cobrado junto a los datos de la tarjeta. Más adelante, el comerciante lleva el lector a su banco, que le abona las cantidades percibidas.

Es tentador aprovechar la estructura que tenemos de la clase *Tarjeta* para crear la especialización *Monedero*; como *Monedero* no admite la operación *transferir(Cuenta*,

double), decidimos redefinirla en *Monedero* dejando su código vacío para garantizar que no haga nada:

```
public void transferir(Cuenta ctaDestino, double importe) {
    return;
}
```

Esto podemos representarlo de este modo:

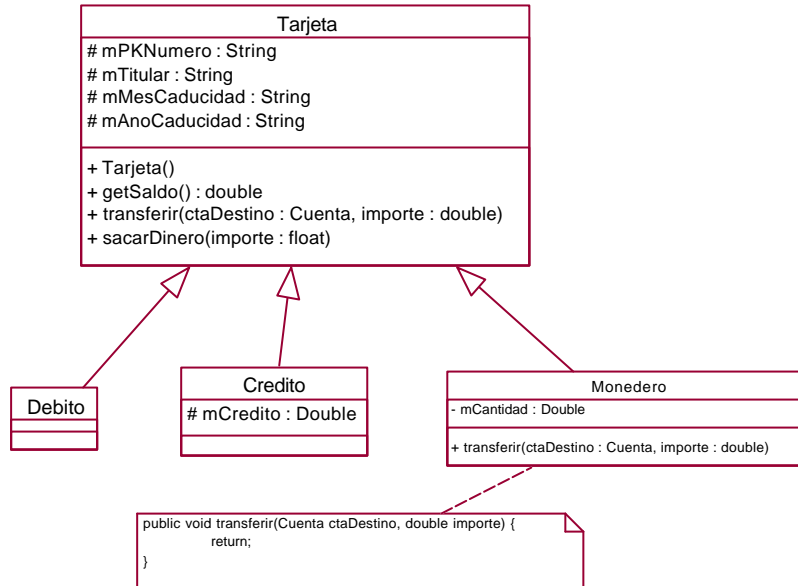


Figura 4. Una jerarquía de herencia incorrecta

Resulta bastante claro que esta jerarquía no cumple la regla del 100%, y es muy dudoso que cumpla la regla “es un”.

1.08.1 Herencia múltiple

Algunos lenguajes de programación permiten que una subclase herede de más de una superclase. Esto se conoce como *herencia múltiple*.

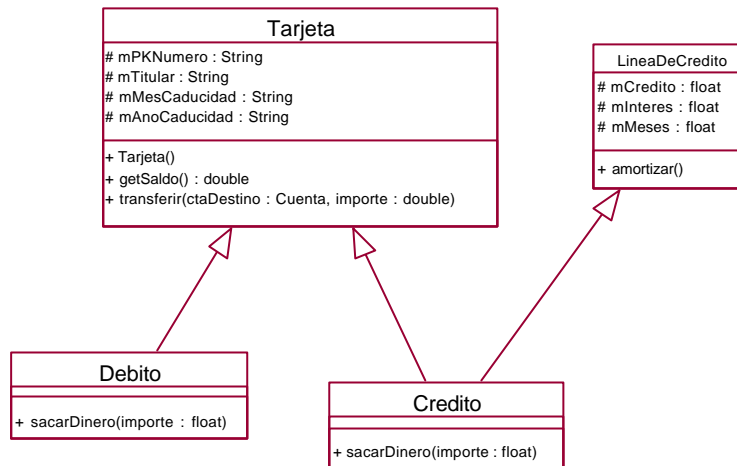


Figura 5. Una tarjeta de crédito es una tarjeta y una línea de crédito.

1.08.2 Clases abstractas e interfaces

Una clase es *abstracta* cuando la propia definición de la clase impide que existan instancias de esa clase. Las clases abstractas se utilizan para definir subclases cuando concurren las siguientes circunstancias:

- (1) existe un conjunto de clases (que serán las subclases) con características estructurales o de comportamiento comunes
- (2) existen diferencias de comportamiento o estructura entre las subclases
- (3) en el sistema orientado a objetos no tiene sentido la existencia de instancias de la superclase

...o bien cuando:

- (4) se desea definir un protocolo de comunicación para enviar mensajes a una familia de clases (es decir, un conjunto de clases que respondan al mismo estímulo)

Una clase es *abstracta pura* cuando posee una *operación abstracta*, que es una operación de la cual se ofrece su signatura en la clase abstracta pura pero no su implementación. Ésta debe ser dada en las subclases.

Una *interfaz* es una colección de operaciones abstractas que son implementadas por una o más clases. Las interfaces representan un protocolo de comunicación o de envío de mensajes con las clases que implementan las operaciones (de hecho, en Smalltalk las interfaces se llaman *protocolos*). Puesta que una interfaz no implementa ninguna operación, podemos entender que una interfaz es una clase abstracta pura. Algunos lenguajes

de programación incorporan directamente el concepto de *interfaz*, como es el caso de lenguaje Java. Este lenguaje no permite herencia múltiple (es decir, una clase no puede heredar de más de una superclase), pero sí que una clase implemente más de una interfaz. Por ello, se dice que las interfaces permiten la simulación de herencia múltiple.

Ejemplo. En nuestro banco, todas las tarjetas que poseen los clientes son o bien de crédito o bien de débito, pero no existen “tarjetas a secas”, por lo que la clase *Tarjeta* es claramente abstracta. En UML, una clase abstracta se representa escribiendo en cursiva el nombre de la clase, como en la siguiente figura. Además, puesto que sobre ambos tipos de tarjeta puede ejecutarse la operación *sacarDinero(float)*, pero con una implementación diferente en cada una, podemos declarar esta operación como abstracta en *Tarjeta* (véase cómo las operaciones abstractas también se representan escribiéndolas en cursiva). Las especializaciones de *Tarjeta* (*Crédito* y *Débito*) implementarán las operaciones abstractas que hereden; si no las implementan, serán también abstractas.

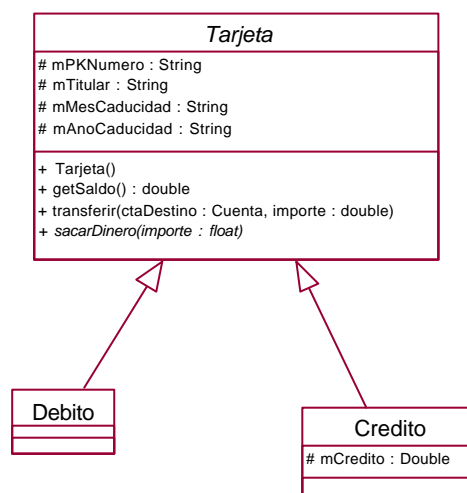


Figura 6. La clase Tarjeta es abstracta.

Ejemplo. Supongamos que, a la hora de realizar un pago con una tarjeta cualquiera, debe comprobarse en un sistema remoto que la tarjeta no ha sido anulada. Lo que realmente nos importa de tal sistema es que responda a la operación *esValida(String):boolean*, sin importarnos cómo funcione por dentro la operación, en qué tipo de máquina se ejecuta, en qué base de datos se busca la información, etc.

Es decir, que lo que nosotros necesitamos del sistema remoto es una interfaz de comunicación adecuada y nada más. La Figura 7 representa esta circunstancia con notación UML: el icono que se usa habitualmente para representar una interfaz es el círculo

(como en la parte izquierda), aunque también puede usarse un icono igual que el de una clase pero con el *estereotipo* «Interface» (como en lado derecho).

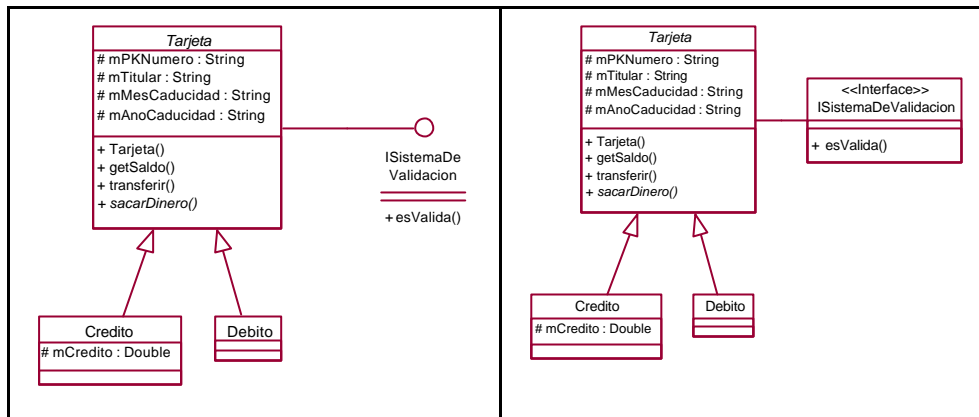


Figura 7. La clase Tarjeta, conectada a una interfaz.

El sistema remoto debe implementar las operaciones contenidas en la interfaz *ISistemaDeValidacion* (la “I” que aparece delante no es obligatoria, pero es una buena manera de saber que estamos refiriéndonos a una interfaz); esto es, debe *implementar la interfaz*. Esto se muestra en la siguiente figura:

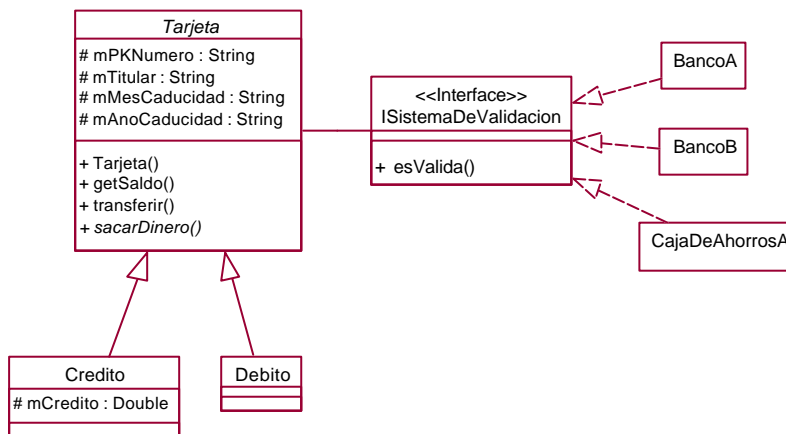


Figura 8. Diferentes entidades de crédito implementan la interfaz.

1.09 Vinculación dinámica

Existe *vinculación dinámica* o *postergada* cuando, en tiempo de compilación, el compilador no es capaz de determinar qué versión de una operación polimórfica debe ejecutarse, cosa que se resuelve en tiempo de ejecución.

Cuando un puntero definido de una clase base apunta a una clase derivada, y hay una llamada una operación que se redefine en la clase derivada, la versión de la operación que debe ejecutarse se decide en tiempo de ejecución y hay, por tanto, vinculación dinámica. El principal inconveniente de la vinculación dinámica es que ralentiza la ejecución del programa.

Ejemplo. Supongamos que nuestro banco ofrece a sus clientes dos tipos de préstamos: hipotecarios y de consumo. Los primeros suelen destinarse a la compra de una vivienda, y de ellos interesa conocer el porcentaje del valor de la vivienda que se ha concedido; de los segundos debe almacenarse el destino del dinero que se ha concedido. Además, ambos tipos de préstamos pueden tener interés fijo o interés variable. Éste se calcula sumando un diferencial (por ejemplo, 0,5) a un tipo de interés que determina una entidad externa (el Banco Central Europeo, por ejemplo). Sea cual sea el tipo del préstamo, todos los meses debe calcularse la cantidad que debe cobrarse por el préstamo, operación que depende del tipo de interés del préstamo.

La siguiente figura ilustra esta situación: la clase *Préstamo* es abstracta porque o bien hay préstamos hipotecarios o préstamos de consumo, pero no préstamos como tales. Cuando se desea calcular la cantidad mensual, alguien (algún objeto) ejecutará la operación pública *calcularPagoMensual():float* sobre todas las instancias de clase *Préstamo*. Esta operación debe conocer el tipo de interés de cada préstamo, que el propio préstamo conoce ejecutando su operación protegida *getInteres():float*. Esta operación llama al método abstracto *getInteres():float* de la clase abstracta *Interés*. Evidentemente, el interés de cada préstamo (aunque sea de clase *Interés*), está instanciado o bien a un tipo *Fijo* o *Variable*, siendo en estas dos clases en las que se encuentra la implementación de la operación *getInteres():float*.

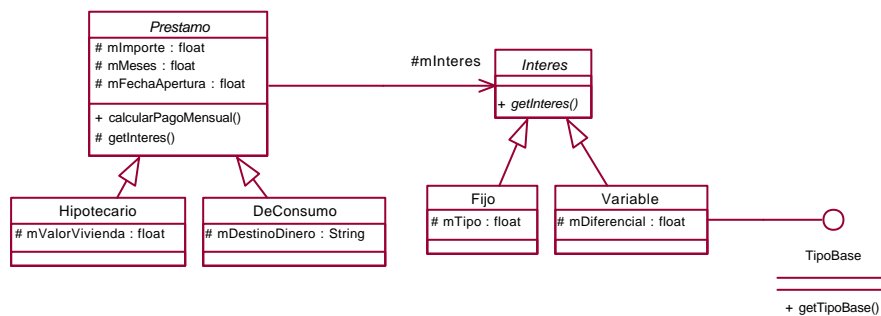


Figura 9. Diferentes tipos de préstamos con diferentes tipos de interés.

En la figura anterior, la flecha que conecta *Préstamo* con *Interés* se llama *asociación*, y significa que el cada objeto de clase *Préstamo* conoce un objeto de clase *Interés*.

La Figura 10 muestra la forma en que se le dice a un *Préstamo* (independientemente de que sea *Hipotecario* o *DeConsumo*) que calcule el importe de su recibo mensual. El préstamo conoce su interés preguntando al tipo de interés al cual conoce, sin importarle que se trate de un tipo fijo o variable.

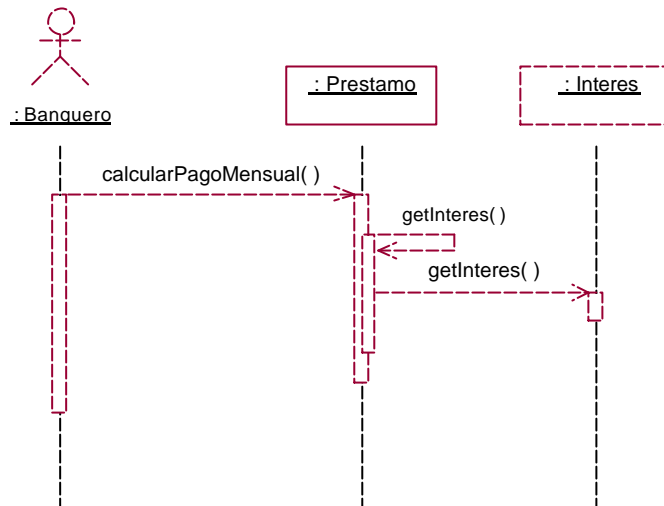


Figura 10.

Si queremos extender el diagrama de la Figura 10 para que se muestre que se calculan los pagos de todos los préstamos, podríamos utilizar el siguiente diagrama:

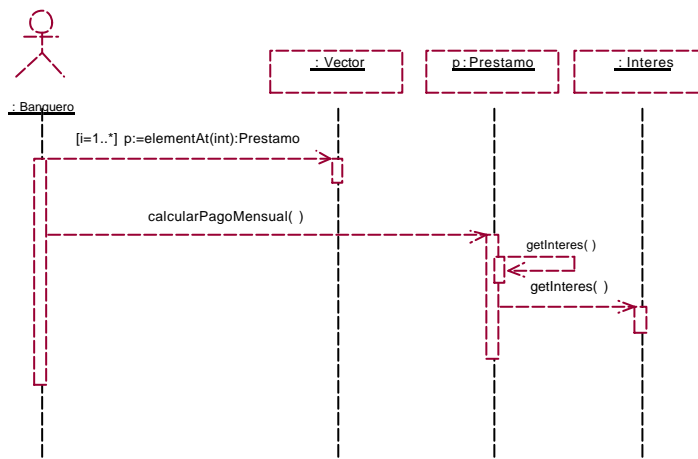


Figura 11. Mensaje enviado a varias instancias contenidas en un vector.

El diagrama anterior es un diagrama de secuencia; para cada diagrama de secuencia existe un único *diagrama de colaboración*. La siguiente figura muestra el diagrama de colaboración obtenido a partir del de secuencia anterior. A ambos tipos de diagramas también se les conoce como *diagramas de interacción*.

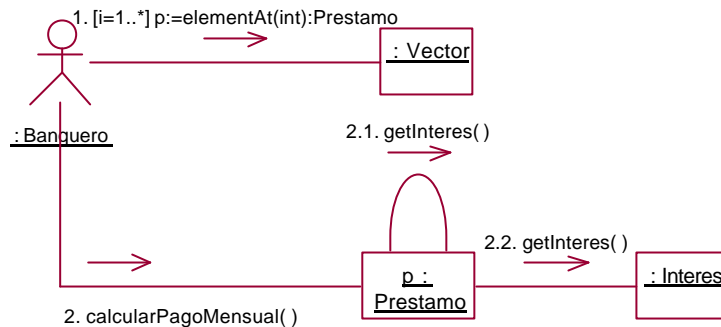


Figura 12. Diagrama de colaboración equivalente al diagrama de secuencia de la Figura 11.

1.10 Sobrecarga de operadores

Algunos lenguajes de programación (como C++) permiten la redefinición de un operador para que actúe sobre clases para las que está predefinido. Realmente, todos los lenguajes de programación vienen “de fábrica” con sus operadores sobrecargados (podemos aplicar el operador “+” para que sume enteros con enteros, reales con reales, enteros con reales, etc.), pero algunos lenguajes nos dan la posibilidad de especificar nuevos tipos de datos sobre los que actúan los operadores. Así, por ejemplo, podríamos sobrecargar el operador unario “-” para que, por ejemplo, diese la vuelta a una cadena de caracteres, de manera que el siguiente fragmento de código escribiría “alocarac aloH”::

```
CString s="Hola caracola";
CString reves=-s;
cout<<reves;
```

En realidad, un operador sobrecargado no es más que un método cuyo nombre coincide con el de un operador.

2. Representación de la estructura de problemas mediante diagramas de clases

Los diagramas de clases representan la estructura de sistemas orientados a objeto (aunque pueden utilizarse para otros propósitos). Están formados por clases (incluyendo interfaces) y por relaciones entre clases.

Las posibles relaciones entre clases son relaciones de *asociación*, de *agregación*, de *dependencia* y de *herencia*. A continuación comentamos las tres primeras.

2.01 Relaciones de asociación

Una relación de asociación entre dos clases A y B denota que los objetos de clase A conocen permanentemente instancias de clase B.

Ejemplos.




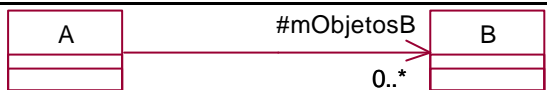
	Cada instancia de A conoce a una de B, y cada instancia de B conoce a una de A
	Cada instancia de A conoce a 0 o 1 instancias de B, y cada instancia de B conoce a una de A
	Cada instancia de A conoce a 0 o más instancias de B; las instancias de B no conocen a ninguna instancia de A
	Igual que el anterior, pero además especificamos que la lista de objetos de tipo B que conoce A están representados por un atributo llamado <i>mObjetos</i> que es protegido

Figura 13. Ejemplos de relaciones de asociación.

En sentido estricto, las relaciones asociación las representaríamos en C++ mediante punteros:

```
class A {
    B[] *mObjetos;
    ...
}
```

2.02 Relaciones de agregación

Si hay una agregación desde A hacia B se denota que dentro de los objetos de clase A hay instancias de clase B.

Ejemplos.

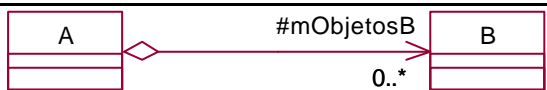

	Cada instancia de A tiene en su interior un conjunto de instancias de clase B.
	Cada instancia de A tiene en su interior una instancia de clase B; además, la instancia de B sabe dentro de qué instancia de A está.

Figura 14. Ejemplos de relaciones de asociación.

2.03 Relaciones de dependencia

Las relaciones de asociación y agregación denotan *relaciones permanentes*; es decir, durante toda la vida de las instancias al menos una conoce a la otra. Puede ocurrir que una clase necesite de otra clase sólo en determinados momentos de su vida, de tal manera que, por ejemplo, la clase A tenga una operación que tome como parámetro un objeto de clase B, o que la clase B tenga una operación en cuyo interior se crea una instancia de clase C. En este caso hablamos de relaciones temporales o *dependencias*.

Ejemplo. En la siguiente figura, representamos que *Tarjeta* depende de *Cuenta* porque *Tarjeta* tiene una operación que toma una *Cuenta* como parámetro.

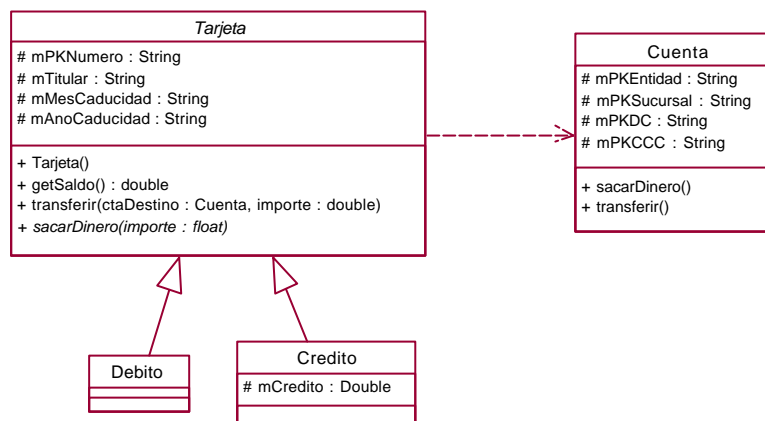


Figura 15. *Tarjeta* depende de *Cuenta*.

Ejemplo de uso incorrecto. Podemos suponer que cada tarjeta tiene una cuenta asociada, sobre la que se van cargando los pagos (si es de débito) o de la que se cobra cada mes el crédito consumido (si es de crédito). Es decir, que la clase *Tarjeta* tiene un conocimiento permanente de su *Cuenta* asociada. En este caso, representaríamos únicamente la relación de asociación desde *Tarjeta* hacia *Cuenta*, pero no la de dependencia. Sería incorrecto, por tanto, utilizar la siguiente figura:

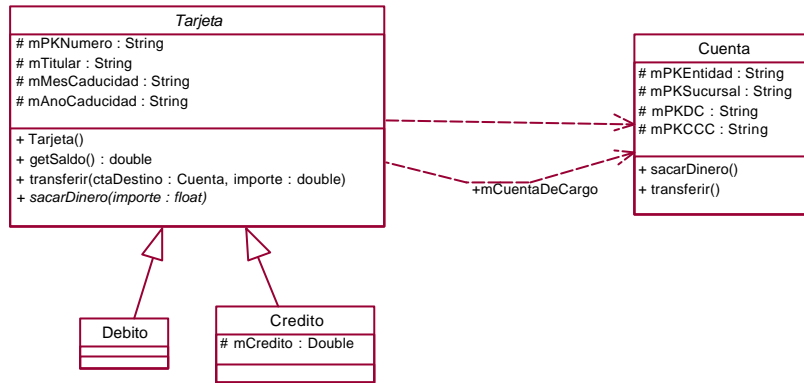


Figura 16. No debe representarse la relación de dependencia porque ya hay una de asociación, que tiene un carácter “más fuerte”.

Ejemplo. Si ocurriera que la tarjeta no sabe en qué cuenta está, pero que la cuenta sí que conoce a las tarjetas que tiene domiciliadas, sí sería correcto representar ambas relaciones:

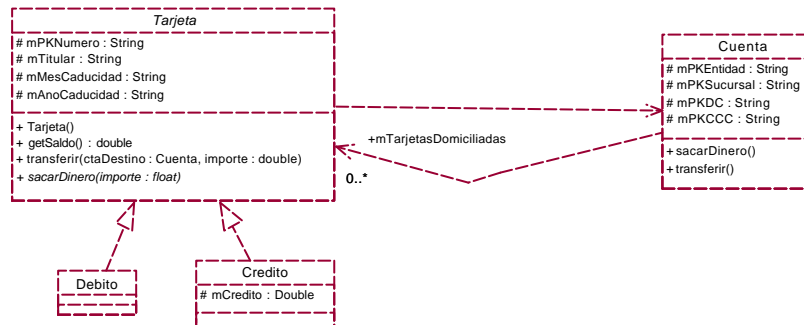


Figura 17

Ejemplo de uso incorrecto. Si ocurre que la tarjeta sabe su cuenta asociada, y que además la cuenta sabe qué tarjetas tiene, sólo debería representarse la relación de asociación, navegable en ambos sentidos. En la siguiente figura, por tanto, sobraría la relación de dependencia.

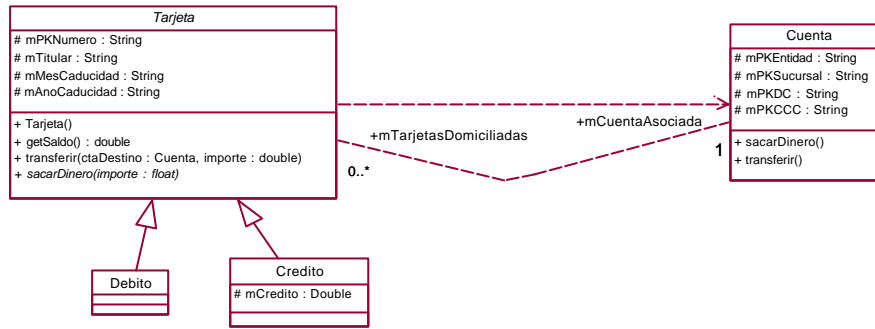


Figura 18. Sobre la relación de dependencia, porque la asociación (sin puntas de flecha) denota que ambas clases ya se conocen de forma permanente.

3. Introducción a la arquitectura multicapa

Uno de los supuestos beneficios del paradigma orientado a objetos es que fomenta la *reutilización del software*: en efecto, conceptos como encapsulación y herencia facilitan la construcción de clases que podemos desarrollar solamente una vez para incorporarlas en varios sistemas.

En nuestro caso, sería deseable que la clase *Cuenta* pudiera realizar transferencias tanto si se lo solicita un empleado del banco desde su terminal, como si se lo pide un cliente desde su casa empleando su navegador. Para que esto sea posible, debemos separar la lógica de la aplicación de la forma en que se presenta la información a los usuarios. Esto se consigue estructurando el sistema en capas.

Una aplicación de multicapa consta al menos de tres *capas*, que habitualmente son las siguientes:

- *Presentación*, en la que residen las ventanas que se encargan de mostrar la información y con las que interactúa el usuario.
- *Dominio* (también llamada de Negocio o de Procesamiento), en la que reside toda la lógica que describe los mecanismos para resolver el problema.
- *Almacenamiento*, en la que reside la posible base de datos gestionada por la aplicación, más un conjunto de clases adicionales que veremos más adelante.

Pueden existir capas adicionales que se encarguen de otras funcionalidades, como la gestión de las comunicaciones con otros sistemas.

Ejemplo. En la Figura 19 se representa un fragmento de la posible estructura de tres capas del sistema de gestión bancario. Aunque su diseño no es muy bueno, cumple con nuestros propósitos ilustrativos:

- La capa del centro es la capa de Dominio, en la que residen las clases *Cuenta*, *Tarjeta*, *Préstamo* o *Cliente*, que son las que podemos ir identificando en el enunciado del problema.
- La capa de la izquierda es la capa de Presentación, en la que residen las clases con las que interactúa el usuario. Nótese que las clases situadas en esta capa conocen a las ubicadas en Dominio (nótese la visibilidad de la relación, denotada por la punta de flecha), pero no al revés.
- La capa de la derecha es la capa de Almacenamiento. La base de datos es un sistema software comprado a un proveedor externo, con el cual nos comunicamos a través de un protocolo de comunicación definido, en el caso de lenguaje Java, en la interfaz *Connection*. De alguna manera, el gestor de base de datos implementará las operaciones contenidas en dicha interfaz.

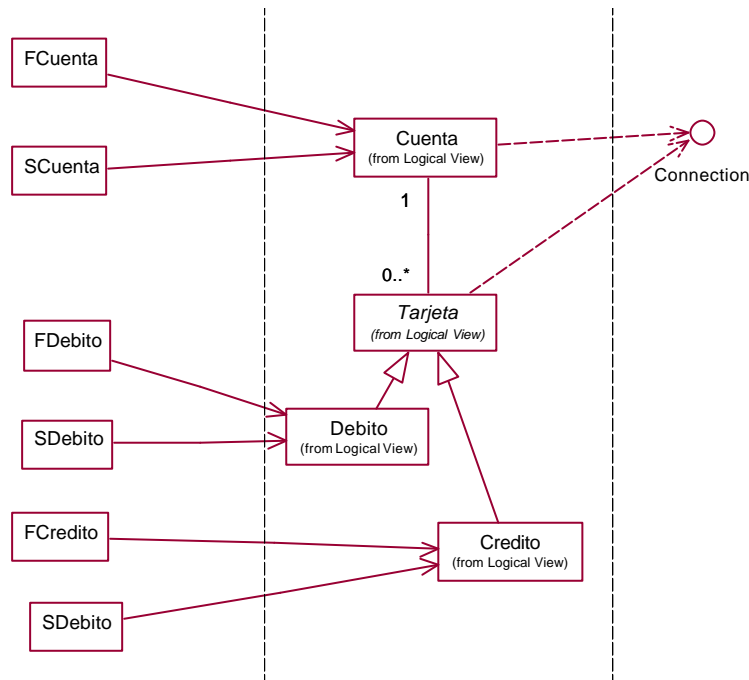


Figura 19. Tres capas para el sistema de gestión bancario.

Las clases cuyas instancias se guardan en algún sistema de almacenamiento persistente se denominan *clases persistentes*. Al conjunto mínimo de operaciones necesario

para gestionar la persistencia de las instancias de una clase se le conoce como *operaciones CRUD*:

- *Create* se utiliza para guardar una instancia de una clase en la base de datos (a este proceso se le llama *desmaterializar*). Si la base de datos es relacional, corresponderá a una operación *Insert*.
- *Read* se utiliza para crear instancias de una clase a partir de la información contenida en la base de datos (a este proceso se le llama *materializar*). Si la base de datos es relacional, la operación corresponderá habitualmente a una operación *Select*.
- *Update* se utiliza para actualizar el estado de la instancia en la base de datos. Si ésta es relacional, se corresponderá con una instrucción *Update* del lenguaje SQL.
- *Delete* sirve para eliminar de la base de datos registros correspondientes a ciertas instancias. Si la base de datos es relacional, corresponderá a una operación *Delete*.

Pueden proponerse muchos diseños alternativos, y todos buenos, de un sistema orientado a objetos. Por ejemplo, si todas nuestras clases persistentes van a implementar las cuatro operaciones *CRUD*, podríamos hacerlas a todas especializaciones de una superclase denominada *Persistente*, como en la Figura 20. En ésta, hemos sustituido las relaciones de dependencia del diseño anterior por una única relación de asociación entre la superclase *Persistente* y la interfaz que representa la conexión con la base de datos. En el código de la clase *Persistente*, esta asociación está representada mediante el atributo protegido *mBD*, por lo que es accesible a todas sus especializaciones.

Las operaciones *CRUD* estarán probablemente implementadas en las tres clases concretas que hay en la capa de Dominio. Esto, sin embargo, hace que estas clases, que son las que tienen la mayor parte de la complejidad de la aplicación, tengan un excesivo *acoplamiento* con la base de datos, de manera que si cambiamos de sistema gestor de base de datos, tal vez tendríamos que reescribir parte del código de estas clases, recompilarlas, etc. (piénsese en qué pasaría si cambiamos de una base de datos relacional, con la que trabajamos con SQL, a una orientada a objetos, que trabaje con OQL; o si, dentro del modelo relacional, cambiamos de Access, en la que los valores booleanos se insertan con las palabras *true* y *false*, a SQL Server, en la que se representan con *0* y *1*).

Existen diferentes tipos de soluciones para desacoplar la capa de Dominio de la capa de Almacenamiento, que estudiaremos a su debido tiempo a lo largo del curso.

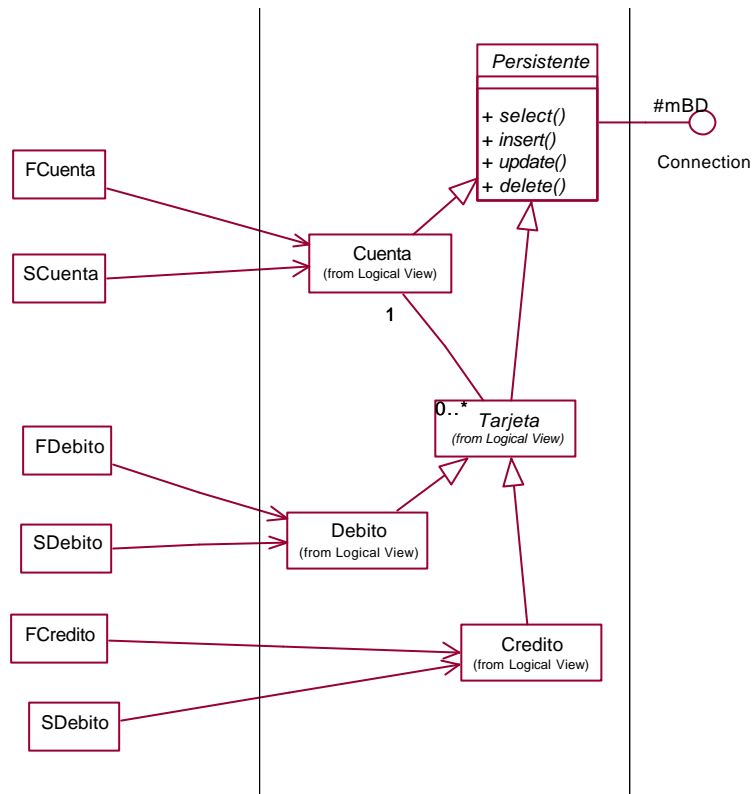


Figura 20. Un diseño alternativo al de la Figura 19.

Hemos indicado que las clases de la capa de Dominio deben estar desacopladas de la capa de Presentación; pero, ¿qué ocurre si un objeto de la capa de Dominio modifica su estado y quiere comunicarlo a una ventana (que es una instancia de una clase de la capa de Presentación)? Veamos qué pasa en estas dos situaciones:

- Si el cambio de estado se produce por un mensaje transmitido por un usuario a través de una ventana, el nuevo estado puede ser devuelto a la ventana como el resultado de la operación. Esto se ilustra en la Figura 21, que muestra el posible paso de mensajes que produce cuando un empleado del banco realiza una transferencia de una cuenta a otra. Si el resultado de la operación es *true*, se muestra un mensaje.

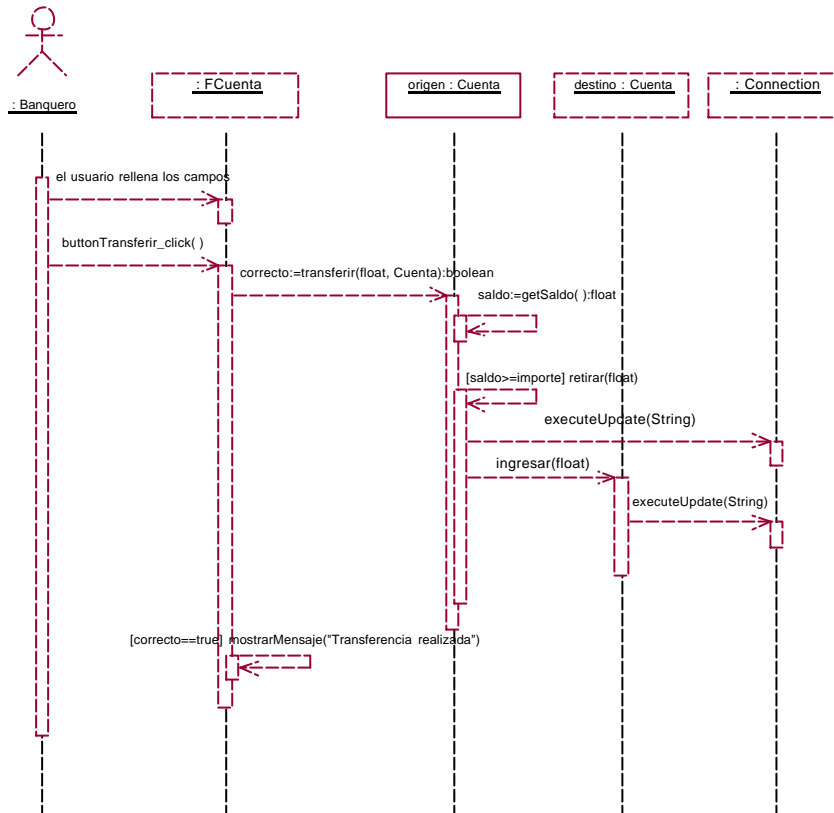


Figura 21. La ventana se actualiza porque conoce el resultado de la ejecución de un método.

- Puede ocurrir que un objeto de Dominio *A* cambie de estado porque se lo diga otro objeto de Dominio *B* (tal vez de la misma clase), ocurriendo además que *A* deba modificar una ventana que “le está mirando”. Supongamos que, en la figura anterior, hay otro banquero viendo el estado de la cuenta destino justo en el momento en que su saldo está siendo actualizado. Si la clase *Cuenta* no conoce a las ventanas que lo observan, ¿cómo puede una cuenta comunicarse con ellas? El diagrama de secuencia de la Figura 22 es incorrecto, ya que, según nuestros diagramas de clase, las instancias de *Cuenta* no conocen instancias de *FCuenta*.

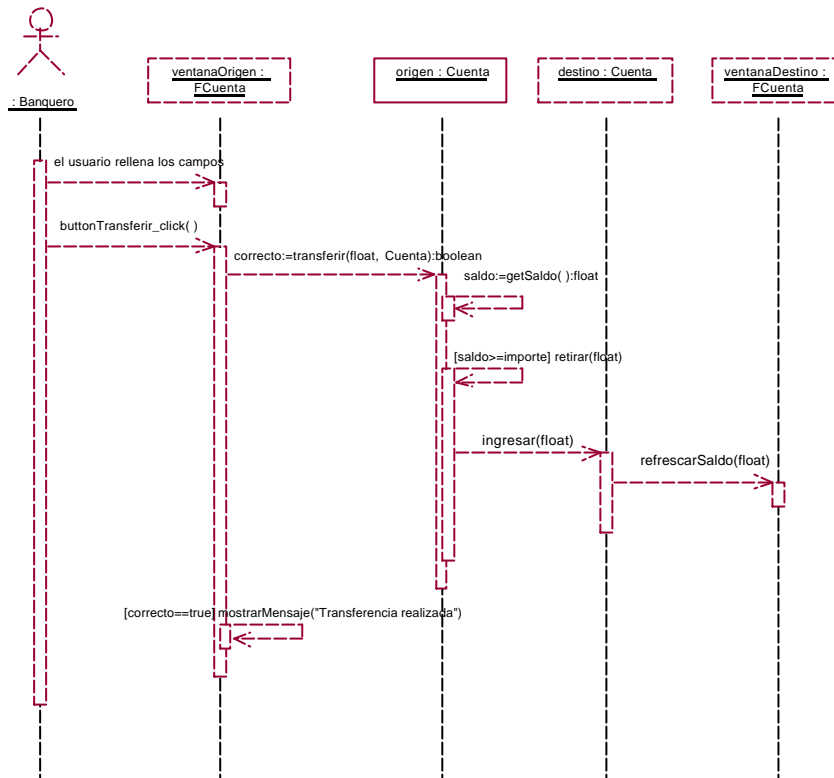


Figura 22. El mensaje *refrescarSaldo(float)* desde el objeto *destino* a *ventanaDestino* es incorrecto porque no hay asociaciones ni agregaciones desde *Cuenta* hacia *FCuenta*.

¿Cuál es la solución? ¿Usar dependencias? No, porque eso significaría que la cuenta origen pasaría como parámetro a la cuenta destino la ventana en la que se está mostrando su estado, y es evidente que la cuenta origen no conoce de absolutamente nada a la ventana de la cuenta de destino. La solución pasa por el uso de observadores, que estudiaremos más adelante.

4. Representación de estos conceptos en lenguajes de programación

A continuación mostramos fragmentos de código Java que muestran la “traducción” de algunos de las figuras vistas en las páginas anteriores.

Salvo que se indique lo contrario, los ejemplos no han sido compilados, por lo que es posible que contengan algún error.

4.01 Figura 3 (página 5)

```
public class Tarjeta {
```

```

protected String mPKNumero, mTitular,
    mMesCaducidad, mAnoCaducidad;

public Tarjeta() {
}

public double getSaldo() {
}

public void transferir(Cuenta ctaDestino, double importe) {
}

public class Debito extends Tarjeta {
    public void sacarDinero(float importe) {
    }
}

public class Credito extends Tarjeta {
    public void sacarDinero(float importe) {
    }
}

```

4.02 Figura 6 (página 8)

```

public abstract class Tarjeta {
    protected String mPKNumero, mTitular,
        mMesCaducidad, mAnoCaducidad;

    public Tarjeta() {
    }

    public double getSaldo() {
    }

    public void transferir(Cuenta ctaDestino, double importe) {
    }

    public abstract void sacarDinero(importe float);
}

public class Debito extends Tarjeta {
    /** Aunque esta operación no aparece en el diagrama de clases, podemos
        suponer que se encuentra implementada ya que esta clase
        no es abstracta */
    public void sacarDinero(float importe) {
    }
}

public class Credito extends Tarjeta {
    protected Double mCredito;

    /** Aunque esta operación no aparece en el diagrama de clases, podemos
        suponer que se encuentra implementada ya que esta clase
        no es abstracta */
    public void sacarDinero(float importe) {
    }
}

```

4.03 Figura 8 (página 9)

Ofrecemos sólo el código de *Tarjeta*, de *ISistemaDeValidacion* y de *BancoA*.

```

public abstract class Tarjeta {
    protected String mPKNumero, mTitular,
        mMesCaducidad, mAnoCaducidad;
    /** En el diagrama no le hemos dado nombre al atributo que representa

```

```

    la interfaz en Tarjeta; decidimos llamarlo mSistema **/
protected ISistemaDeValidacion mSistema;

public Tarjeta() {
}

public double getSaldo() {
}

public void transferir(Cuenta ctaDestino, double importe) {
}

public abstract void sacarDinero(importe float);
}

public interface ISistemaDeValidacion {
    public boolean esValida();
}

public class BancoA implements ISistemaDeValidacion {
    public boolean esValida() {
    }
}

```

4.04 Figura 9 (página 10) y Figura 10

De la dos figuras indicadas podemos deducir al menos el siguiente código:

```

public abstract class Prestamo {
    float mImporte, mMeses, mFechaApertura;
    Interes mInteres;

    public float calcularPagoMensual() {
    }

    /** Nótese que devolvemos el interés preguntándole al objeto mInteres,
        pero sin preocuparnos de si se trata de un Fijo o un Variable.
        Esta cuestión la resuelve el objeto mInteres **/
    protected float getInteres() {
        return mInteres.getInteres();
    }
}

public abstract class Interes {
    public abstract float getInteres();
}

public class Fijo extends Interes {
    protected float mTipo;

    public float getInteres() {
        return mTipo;
    }
}

public class Variable extends Interes {
    protected float mDiferencial;
    protected TipoBase mTipoEntidadExterna;

    public float getInteres() {
        return mDiferencial + mTipoEntidadExterna.getTipoBase();
    }
}

```

4.05 Figura 11 (página 11)

En este caso, recorreremos un *Vector* de préstamos y sobre cada uno calculamos su saldo mensual. En algún lugar habrá una operación que podría ser de este estilo:

```
public void pagosDePrestamos(Vector prestamos) {
    for (int i=0; i<prestamos.size(); i++) {
        Prestamo p=(Prestamo) prestamos.elementAt(i);
        float pago=p.calcularPagoMensual();
        System.out.println("Cantidad: " + pago);
    }
}
```

4.06 Figura 17 (página 15)

En este ejemplo, cada objeto de clase *Cuenta* conoce a 0 o más objetos de clase *Tarjeta*. La figura no nos indica en qué tipo de estructura de datos se almacenan las tarjetas (un *Vector*, un *array*, una *tabla hash*, una *pila*...).

Si se tratara de un *array*, el código de la clase *Cuenta* podría ser éste:

```
public class Cuenta {
    protected String mPKEntidad, mPKSucursal, mPKDC, mPKCCC;
    /** El array almacena referencias a objetos de clase tarjeta. */
    protected Tarjeta[] mTarjetasDomiciliadas;

    ...
}
```

Un error frecuente consiste en almacenar, en lugar de referencias a los objetos contenidos, valores de campos que los identifican. Suponiendo que el campo *mPKNumero* de *Tarjeta* identificara unívocamente a cada tarjeta, tendríamos lo siguiente:

```
public class Cuenta {
    protected String mPKEntidad, mPKSucursal, mPKDC, mPKCCC;
    /** El array almacena los números de tarjeta. Esta solución no
     es un reflejo fiel del diagrama de la figura, aunque podría
     estar justificado en ciertos casos por razones
     de rendimiento. */
    protected String[] mTarjetasDomiciliadas;

    ...
}
```

4.07 Figura 21 (página 20)

A partir de esta figura y de lo que sabemos de la Figura 19, podemos deducir un poquito de código:

```
public class FCuenta extends Frame { //Suponemos que se trata de un Frame
    ...
    /** De la Figura 19 deducimos que cada instancia de FCuenta conoce a un
     objeto de clase Cuenta. En la Figura 21, vemos que a este objeto
     se le llama "origen". */
    protected Cuenta origen;

    public void buttonTransferir_click() {
        /** En principio, no tenemos claro cómo se conocen los valores
         de importe y destino dentro de este método. */
        boolean correcto=origen.transferir(importe, destino);
        if (correcto) {
            mostrarMensaje("Transferencia realizada");
        }
    }
}
```

```

    }
}

public void mostrarMensaje(String texto) {
    // No sabemos qué hace esta operación para mostrar el mensaje
}

public class Cuenta {
    ...
    public boolean transferir(float importe, Cuenta destino) {
        float saldo=getSaldo();
        if (saldo>=importe) {
            retirar(importe);
            String SQL="...";
            mBD.executeUpdate(SQL);
            destino.ingresar(importe);
            return true;
        }
        /** Podemos suponer que si no entramos en el if, devolvemos
         false **/
        return false;
    }

    // No sabemos cómo funcionan las operaciones siguientes.
    public float getSaldo() {
    }

    public void ingresar(float importe) {
    }

    public float retirar(float importe) {
    }
}

```

5. Ejercicios

5.01 Biblioteca

Intente representar con diagramas de clases la capa de Dominio de un sistema encargado de la gestión de una biblioteca. Trate de identificar jerarquías de herencia, atributos y operaciones, asociaciones y agregaciones. Indique el nivel de visibilidad de cada operación.

5.02 Préstamo de un libro

Tomando como base el diagrama del ejercicio anterior, represente con un diagrama de secuencia el flujo de eventos que se produce cuando se presta un libro (tal vez detecte que necesita más operaciones o atributos que los identificados en el ejercicio anterior).

5.03 Traducción a código

Traduzca a código los diagramas anteriores.

6. Procesos de desarrollo de software orientado a objetos

En las secciones anteriores hemos visto algunos de los *artefactos* software que se producen durante la construcción de un sistema software, pero hay muchos más: otros tipos de diagramas, documentos de requisitos, manuales de usuario, manuales técnicos, manuales de seguridad, etc. Todos estos elementos son software o, más bien, juntos conforman lo que llamamos un *producto software*.

La construcción de productos software con cierto nivel de complejidad hace necesario dotar a los equipos de desarrollo de métodos estandarizados de comunicación y trabajo.

Para lo primero, existen multitud de notaciones que sirven para describir los diferentes elementos que componen un producto software: diagramas entidad-interrelación para los esquemas conceptuales de bases de datos; redes de Petri para sistemas paralelos; diagramas de flujos de datos para sistemas estructurados, etc. Para la descripción de sistemas orientados a objeto ocurre exactamente lo mismo, si bien desde hace algunos años se ha impuesto con claridad el Lenguaje Unificado de Modelado (UML), que utilizaremos en estos apuntes.

Para lo segundo, a lo largo de la breve historia de la Ingeniería del Software se han propuesto también numerosas metodologías para el desarrollo de sistemas orientados a objeto (por ejemplo: Catalysis, Objectory, OMT, OOSE), aunque parece también imponerse el llamado Proceso Unificado (UP). Recientemente, comienza a implantarse con mucha fuerza el enfoque llamado Programación Extrema.

De acuerdo con Larman (2001), un proceso de desarrollo de software describe un mecanismo para construir, distribuir y, deseablemente, mantener software.

6.01 Un vistazo al Proceso Unificado

La característica más importante del UP es su carácter iterativo: el desarrollo de un producto software se organiza en iteraciones, que vienen a ser miniproyectos de duración determinada (que suele prefijarse en periodos de 2 a 6 semanas). En cada iteración se realiza una fase de análisis de requisitos, diseño, implementación y pruebas, de tal manera que se obtiene un sistema ejecutable probado e integrado.

Por lo general, en cada iteración se selecciona un pequeño conjunto de requisitos que se diseña, se implementa y se prueba. Esto supone un refinamiento progresivo del sistema, que va además aumentando de tamaño, aunque alguna iteración puede dedicarse a mejorar software desarrollado con anterioridad.

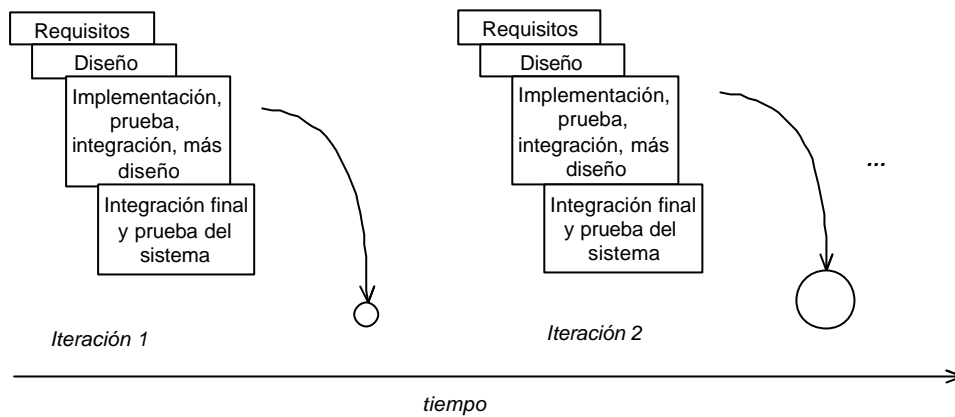


Figura 23 (adaptada de Larman, 2001). El UP es iterativo e incremental (los círculos blancos representan el tamaño del sistema).

El desarrollo completo de los subconjuntos de requisitos favorece la realimentación casi continua por parte de todas las personas involucradas en el proceso: analistas, programadores, ingenieros de pruebas, usuarios... Puede ocurrir que se fracase estrepitosamente en una iteración, pero su impacto es desde luego mucho menor que encontrarnos con esta situación al final del proceso. En este sentido, lo habitual es que el riesgo disminuya conforme se avanza en la ejecución del proyecto.

6.01.1 Fases del Proceso Unificado

En el Proceso Unificado se distinguen cuatro grandes fases:

1. *Comienzo*: en esta fase se realiza una planificación del proyecto. Al final, se habrá realizado un *Plan de iteraciones*, en el que se indican qué *casos de uso* se van a desarrollar en cada iteración.
2. *Elaboración*: se analiza el dominio del problema, se establece una base arquitectónica sólida, se desarrolla el plan del proyecto y se eliminan sus elementos de más alto riesgo.
3. *Construcción*: se desarrolla y se prueba el software.
4. *Transición*: se entrega el software a los usuarios.

El UP describe el conjunto de *disciplinas* que se muestra en la Figura 24. Una disciplina es un conjunto de actividades que se ejecutan para realizar alguna cosa. También se especifican los artefactos que deben generarse con cada disciplina.

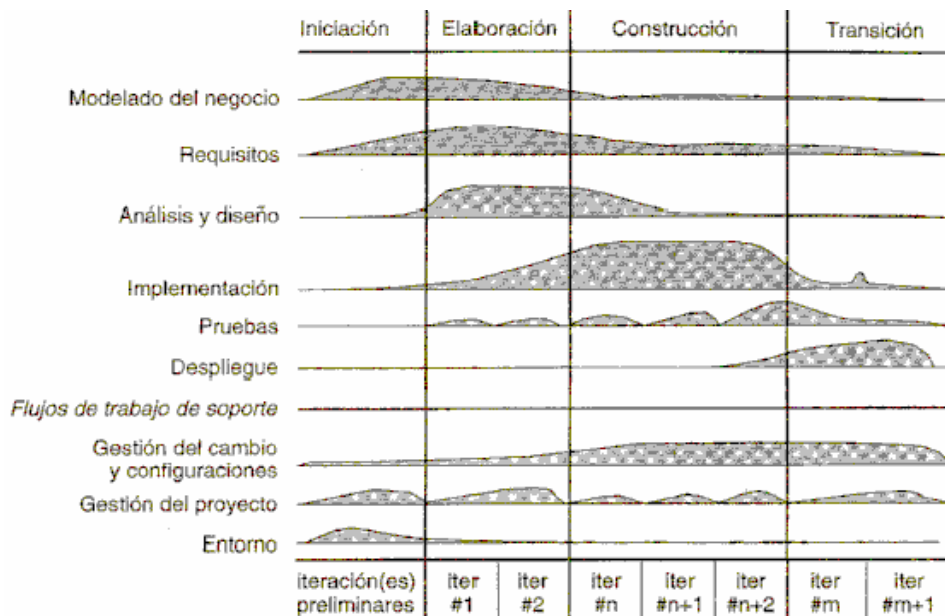


Figura 24. Disciplinas del Proceso Unificado.

Como se observa, en cada iteración se trabaja prácticamente en todas las disciplinas, si bien el esfuerzo dedicado a cada una va cambiando conforme avanza el proyecto.

6.02 Programación Extrema (*eXtreme Programmingo XP*)

XP¹ es una metodología de desarrollo de software cuyo principio básico consiste en llevar sus técnicas y principios a niveles extremos:

- El código será revisado continuamente, mediante la programación por parejas (*pair programming*), de manera que habrá dos personas por máquina.
- Se harán pruebas continuamente: los programadores todas las clases que vayan escribiendo (*pruebas unitarias*), y los clientes irán comprobando que el proyecto va satisfaciendo sus requisitos (*pruebas funcionales*).
- Se harán pruebas de integración cada vez que deba añadirse una nueva clase al proyecto o después de modificar cualquier clase existente. A esto se le llama *integración continua*.

¹ En <http://www.extremeprogramming.org> puede encontrarse mucha documentación sobre XP.

- Se pretende que el código esté siempre en su forma más simple posible, para lo que se utilizarán técnicas de *refactoring*.
- Las iteraciones serán más cortas que en otros métodos (como el UP), con el objetivo de beneficiarnos de la realimentación tanto como sea posible.

Para conseguir todo esto, XP propone una serie de *prácticas y reglas* con las que abordar los procesos, que están relacionadas con la Planificación, el Diseño, la Codificación y las Pruebas:

- La *Planificación* toma como base las historias del usuario (*user stories*), en las que éste explica con su lenguaje las cosas que desea que realice el sistema. A partir de ellas, los desarrolladores estiman el tiempo que les llevará realizar la implementación y prueba de la historia (según XP, entre 1 y 3 semanas), que luego son priorizadas por el cliente. Se crea entonces un *Plan de entrega*, que especifica con exactitud las historias que se van a implementar en cada entrega y sus fechas.
- El *Diseño* debe ser o más simple posible y debe resolver sólo el problema en el que estamos trabajando actualmente. Nunca añadiremos funcionalidades que aún no sean necesarias. Por otro lado, refactorizaremos el sistema para conservar siempre su simplicidad.
- En la *Codificación*, destaca el hecho de que las pruebas unitarias deben ser codificadas antes que las propias clases. Todos los programas se desarrollan mediante programación por pares, siguiendo estándares de codificación, de nombrado, etc.
- En las *Pruebas* se requiere realizar pruebas unitarias de todo el código, que deben ser superadas antes de éste sea entregado al cliente.

7. La importancia de la verificación y validación

Como hemos visto en las secciones dedicadas al Proceso Unificado y a Programación Extrema, ambas metodologías ponen un énfasis muy especial en las pruebas. Las estrategias de prueba propuestas en estos modelos difieren bastante de las técnicas clásicas de prueba, en las que éstas eran relegadas, sin darles mucha importancia, a las fases finales del desarrollo.

Como respuesta a la escasa importancia que se les daba, a comienzos de los años ochenta se propuso el conocido como “modelo en V” (Figura 25), que expresa que a

cada etapa del proceso de desarrollo debe corresponderle un nivel bien determinado de prueba.

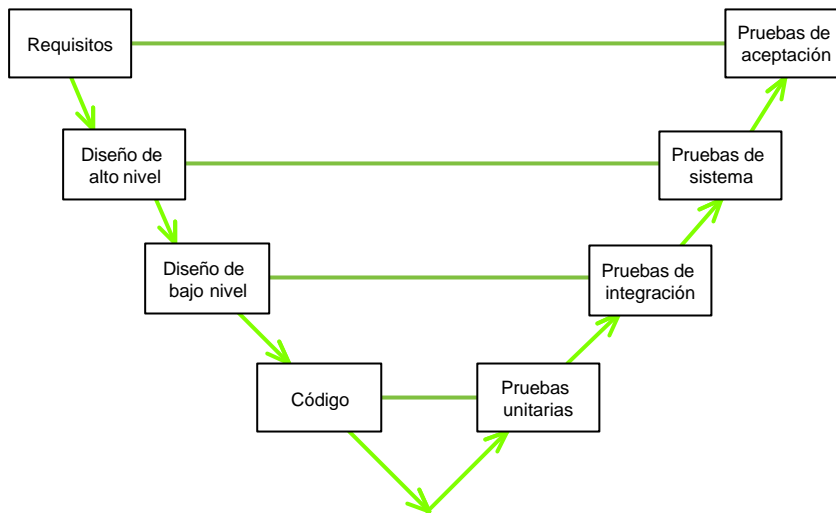


Figura 25. El modelo en V para pruebas del software

Las pruebas propuestas por el modelo en V, sin embargo, se ejecutan una vez se dispone de código fuente. En la actualidad, sin embargo, existe un claro convencimiento de la necesidad de verificar y validar todos los artefactos producidos a lo largo del desarrollo de un sistema software, ya sean código, documentos escritos o diagramas de cualquier tipo. Además, el carácter iterativo del desarrollo orientado a objetos invita desde luego a seguir un enfoque de pruebas de este tipo.

Es preciso, entonces, realizar *verificación* y *validación* (el concepto de “pruebas” es probablemente más restringido) no sólo del código fuente, sino también de los artefactos generados en las fases de análisis y diseño. Para la prueba de artefactos no ejecutables suelen utilizarse inspecciones y revisiones, listas de comprobación, etc. (que también pueden aplicarse al código fuente).

CAPÍTULO 2. DIAGRAMAS DE CLASES

1. Introducción

Los diagramas de clases son uno de los diagramas incluidos en UML para describir la estructura estática de los sistemas orientados a objetos, siendo sin duda el más importante de éstos.

Pueden construirse diagramas de clase en la fase de análisis del sistema (en donde manipularemos clases que no tendrán por qué corresponderse exactamente con las clases que tendremos más adelante), o en la fase de diseño (en donde las clases se corresponderán con conceptos bien definidos del sistema de información).

Los diagramas de clases están compuestos principalmente de clases, interfaces y relaciones entre éstos, pero pueden contener también paquetes (subsistemas) e instancias.

2. Clases

En su formato más básico, una clase se representa mediante un rectángulo con tres compartimentos separados por líneas horizontales: el superior contiene el nombre de la clase y, tal vez, información adicional; el del centro contiene la lista de atributos de la clase; el inferior tiene la lista de operaciones.

No obstante, opcionalmente podemos suprimir compartimentos:

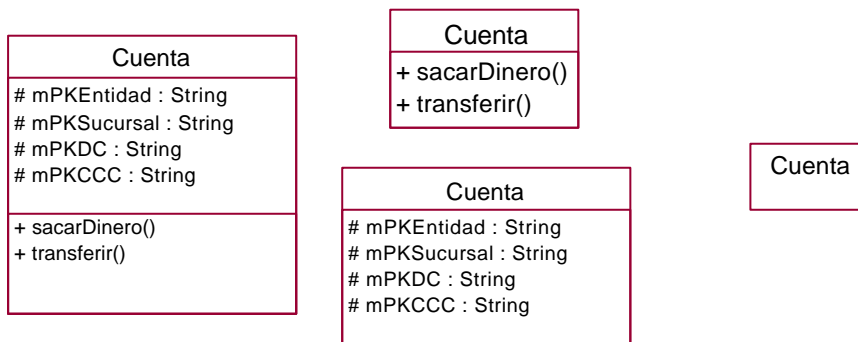


Figura 26. Cuatro formas diferentes de representar la misma clase.

Igualmente, también pueden mostrarse más o menos detalles de atributos y operaciones:

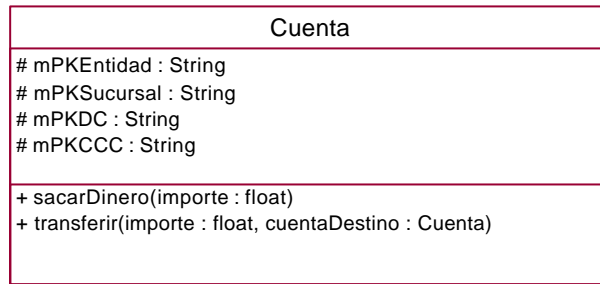


Figura 27. En este caso, se muestra la cabecera de las operaciones

También se puede dotar a las clases de *estereotipos*. En la siguiente figura se representa la clase *Cuenta* estereotipada:

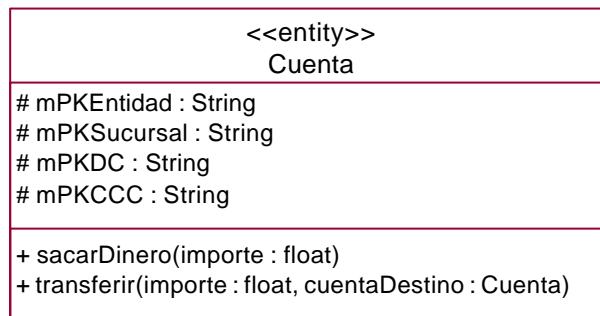


Figura 28. La clase Cuenta con el estereotipo «entity»

Los atributos y operaciones de una clase también pueden tener estereotipo: en la siguiente figura, indicamos que los atributos son parte de la clave primaria en la tabla de la base de datos en la que se almacenan las cuentas, y que las dos operaciones actualizan la información de la base de datos.

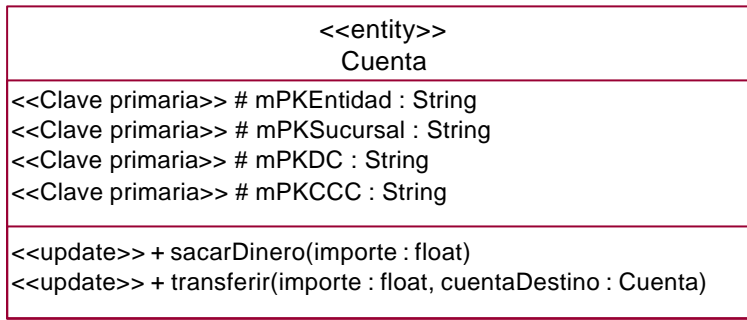


Figura 29. Atributos y operaciones estereotipados

También podemos ampliar el número de compartimentos de una clase, añadiendo por ejemplo uno que contenga las posibles excepciones que pueden arrojar sus operaciones. La Figura 30 muestra la clase *Cuenta* con un compartimento adicional al que, además, hemos puesto nombre.

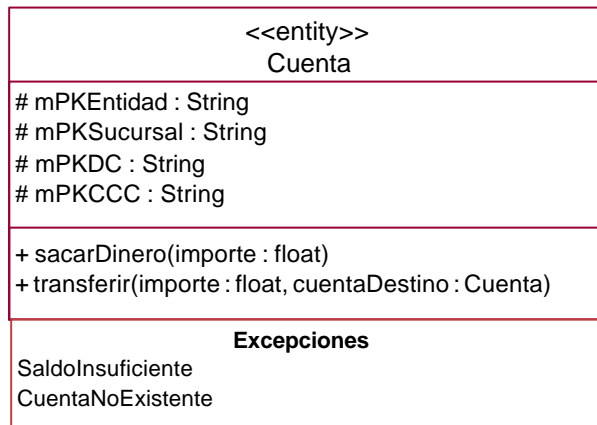


Figura 30. La clase Cuenta, con un compartimento adicional de excepciones

2.01 Notación para atributos

La notación para representar un atributo de una clase debe cumplir la siguiente regla:

visibilidad nombre : tipo [multiplicidad orden] = valorInicial {propiedades}

También pueden representarse atributos *estáticos* (aquellos cuyo valor es compartido por todos los objetos de esa clase, de manera que si cambia el valor del atributo en un objeto, cambia en todos) y *derivados* (aquellos cuyo valor depende del valor del resto de atributos). Para el primer caso, UML propone subrayar el campo estático, y anteponer el símbolo “/” al atributo derivado. No obstante, la herramienta CASE que utili-

zamos para confeccionar estos apuntes (Rational Rose 2000) utiliza el símbolo “\$” para los atributos estáticos, aunque mantiene el símbolo “/” para los derivados.

La siguiente figura muestra mucho más detalladamente la estructura de la clase *Cuenta*:

<<entity>> Cuenta
<<Clave primaria>> #\$ mPKEntidad : char[4] <<Clave primaria>> # mPKSucursal : char[4] <<Clave primaria>> /# mPKDC : char [2] <<Clave primaria>> # mPKCCC : char [10] = 0000000000
<<update>> + sacarDinero(importe : float) <<update>> + transferir(importe : float, cuentaDestino : Cuenta)

Figura 31. Mayor detalle en los atributos. Obsérvese el \$ en mPKEntidad y la / en mPKDC

2.02 Notación para operaciones

La notación por defecto para una operación es la siguiente:

visibilidad nombre (listaDeParámetros) : tipoDeRetorno {propiedades}

El único elemento que puede inducir a confusión es la *listaDeParámetros*; cada parámetro tendrá la siguiente sintaxis:

clase nombre : tipo = valorPorDefecto

El elemento *clase* hace referencia a la forma en que se pasa el parámetro a la operación:

- *in*, que se asume por defecto e indica que el parámetro es de entrada
- *out*, que indica que el parámetro es de salida
- *in-out*, que indica que el parámetro es de entrada/salida

Las operaciones de clase (estáticas) deben subrayarse. Por otro lado, las operaciones de consulta, que no afectan el estado del sistema, pueden especificarse con la propiedad *{query}* o *{consulta}*. Las operaciones abstractas se escribe en cursiva o, bien, se adornan con la propiedad *{abstracta}*.

Puede describirse el cuerpo de una operación con un lenguaje de programación, pseudocódigo o texto mediante una *nota* unida a la operación.

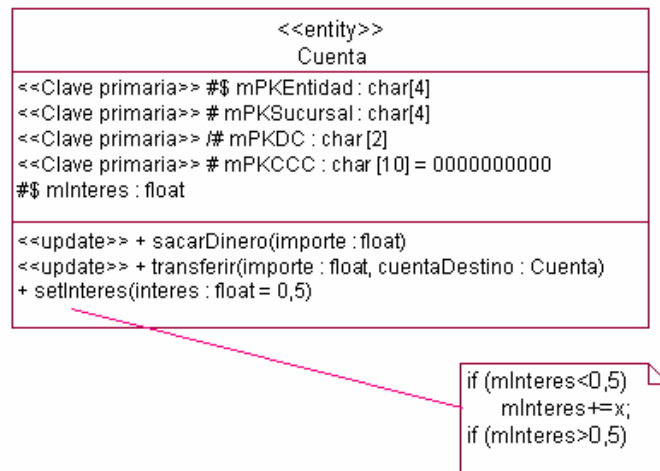


Figura 32. Describimos la operación con una nota

3. Interfaces

Las interfaces son conjuntos de operaciones sin implementación. Tampoco pueden tener atributos ni ser el origen de asociaciones o agregaciones. Podemos entender que una interfaz representa un subconjunto de operaciones externamente visibles de una clase, y también que una interfaz es una clase abstracta sin atributos y compuesta sólo de operaciones abstractas.

Las interfaces pueden representarse con varias notaciones, como por ejemplo:



Figura 33. Diferentes formas de representar interfaces

Las clases que implementan las operaciones enumeradas en las interfaces se unen a éstos mediante relaciones de *implementación* o de *realización*, diciéndose entonces que “tal clase implementa tal otro interfaz”. Estas relaciones se representan mediante líneas punteadas cuyos extremos son puntas de flecha con forma de triángulo:

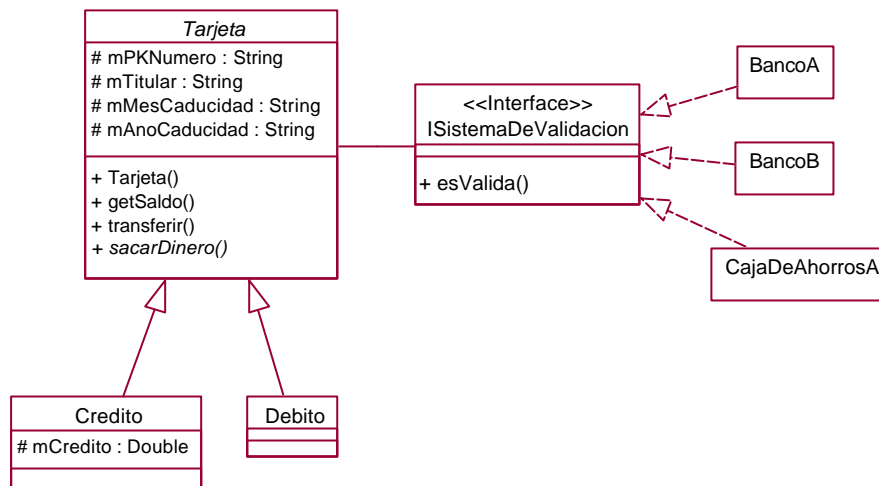


Figura 34. Relaciones de implementación

4. Relaciones entre clasificadores

De forma general, con el término *clasificador* podemos referirnos a clases, interfaces y tipos de datos. Estos tres elementos se representan en UML con la misma notación.

4.01 Asociaciones

Una asociación entre dos clasificadores representa el hecho de que al menos uno de los clasificadores tiene conocimiento de la otra (véase epígrafe 2.01, página 13). Se indica con una línea sólida que los une. Una asociación puede tener nombre, navegabilidad, nombres de rol, estereotipos, restricciones, atributos y operaciones.

En la siguiente figura, que muestra un fragmento del sistema bancario, estamos representando muchos hechos:

- Un cliente puede ser titular de varios productos (nótese que *Producto* es abstracta, ya que el banco oferta muchos tipos de productos, como tarjeta, cuentas, préstamos, fondos de inversión, etc.); a su vez, cada producto debe tener al menos un titular, aunque admite más (por ejemplo, un préstamo hipotecario del que son titulares los dos cónyuges de un matrimonio). A esta asociación la hemos llamado *ProductosDeCadaTitular*: como se observa por la navegabilidad de la asociación, dado un cliente podemos conocer los productos de los que es titular, pero no al

revés (por lo cual, el rol *mTitulares* no sirve para nada). Además, a la colección de productos de lo que un cliente es titular le hemos asignado el nombre de rol *mProductos*, que es protegido. Cuando pasáramos a código, esto se traduciría por la presencia de un atributo de tipo *Vector*, *Enumeration*, *Hashtable*, array o de otro tipo que se llamaría *mProductos*.

- Por otro lado, tendremos clientes en el banco que son firmas autorizadas de productos (una firma autorizada es una persona que, con ciertas limitaciones, puede operar con el producto porque lo ha autorizado un titular). La relación en este caso es bidireccional. En este caso nos interesa almacenar la fecha en que el cliente se convirtió en firma autorizada del producto, para lo que creamos la clase *FechaAutorizacion* (a este tipo de clases se les llama *association class* o *clase asociativa*).

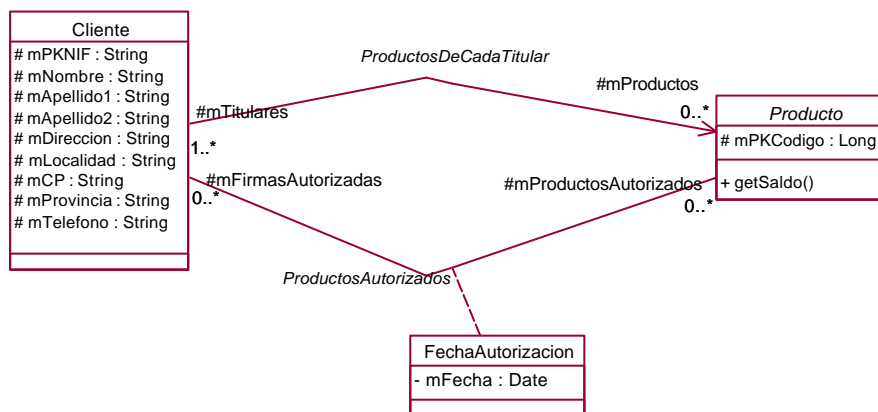


Figura 35

4.01.1 Traducción a código de clases asociativas

No hay una traducción directa de una clase asociativa a código, al menos en lenguaje Java. Lo que haremos será reconsiderar el diagrama de la Figura 35 y dibujarlo de esta forma:

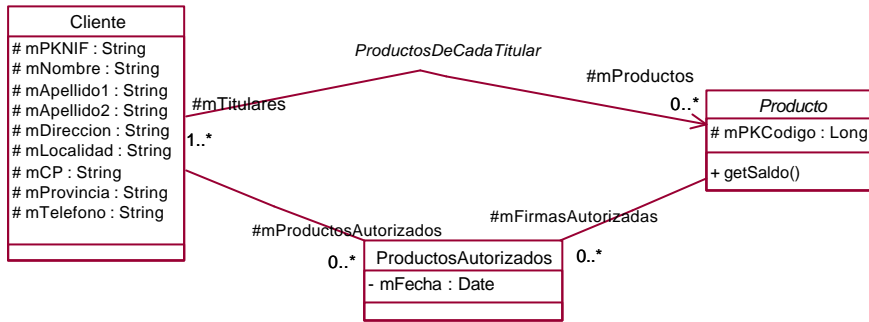


Figura 36. El diagrama de la Figura 35, adaptado para generar código

Ahora podemos traducirlo al siguiente código Java:

```

public class Cliente {
    protected String mPKNIF,
        mNombre,
        ...;
    protected Hashtable
        mProductos,
        mProductosAutorizados;
    ...
}

public class ProductosAutorizados {
    protected Date mFecha;
    Producto mProducto;
    ...
}

public abstract class Producto {
    protected Long mPKCodigo;
    protected Hashtable
        mFirmasAutorizadas;
    ...
}

```

4.02 Agregaciones

Una agregación representa una relación “es parte de” entre instancias de dos clases.

La composición es una forma fuerte de agregación en la que, además, los objetos agregados se crean después que el agregante y se destruyen antes. Es decir, no existen objetos agregantes si no existe su correspondiente instancia del agregado. En las relaciones de composición, la multiplicidad de la clase agregante no puede ser mayor que uno, ya que no se permite que el objeto agregado esté dentro de más de un objeto agregante.

La composición se denota con el mismo símbolo que la agregación, pero con el rombo relleno de color.

4.03 Asociaciones n-arias

Un asociación n-aria es una asociación entre tres o más clasificadores. Se representa mediante un rombo que une los clasificadores. Las asociaciones ternarias pueden ser clases asociativas.

La Figura 41 muestra una asociación ternaria entre *Producto*, *Empleado* y *Cliente*, que puede representar el hecho de el empleado del banco está autorizado para ofrecer unas ciertas condiciones de cada producto según quién sea el cliente.

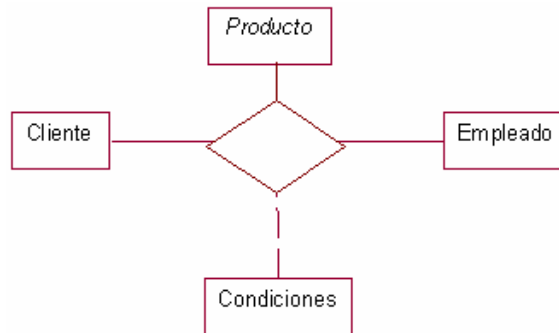


Figura 37. Una asociación ternaria

4.04 Generalización

Una relación de generalización muestra una relación de herencia entre un elemento general y uno o más elementos especializados. Para que una relación de generalización sea correcta deben cumplirse las reglas del 100% y es-un (véase epígrafe 1.08, página 4).

La relación puede tener un *discriminador*, que representa el nombre del subconjunto de especializaciones. También pueden tener *restricciones*; las restricciones predefinidas son:

- *con solapamiento*, que indica que una instancia puede ser de dos o más tipos de los especializados.
- *disjunta*, que indica que cada especialización es exclusivamente de uno de los tipos.
- *completa*, con lo que decimos que ya hemos representado todas las posibles especializaciones.
- *incompleta*, con lo que decimos que hemos especificado algunas especializaciones, pero que hay algunos pendientes de especificar.

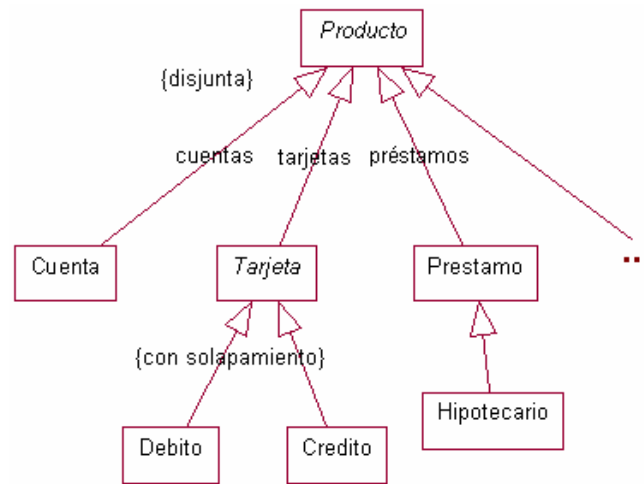


Figura 38. Un árbol de herencia con bastante detalle

En la figura anterior representamos que nuestro banco ofrece diversos tipos de productos, de los que mostramos sólo tres (los puntos suspensivos denotan que hay más especializaciones identificadas, pero que no las estamos mostrando). También decimos que un producto o es una tarjeta, o una cuenta, o un préstamo, o de otro tipo de los “ocultados” por los puntos suspensivos. También decimos que una tarjeta puede ser de crédito y de débito.

4.05 Dependencias

Una dependencia viene a representar el hecho de que un cambio en el elemento origen puede suponer un cambio en el elemento destino.

De forma general, una dependencia representa una relación temporal entre dos clasificadores.

5. Otros elementos de modelado

Algunos elementos que aparecen menos habitualmente se explican en las siguientes subsecciones.

5.01 Utilidades (*utilities*)

Una utilidad es una agrupación de variables y funciones globales, que se ubican en el símbolo de una clase con el estereotipo «*utilidad*», como en la siguiente figura:

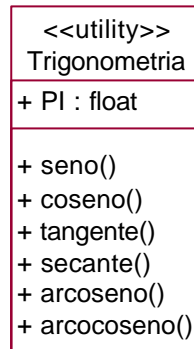


Figura 39. Una utilidad.

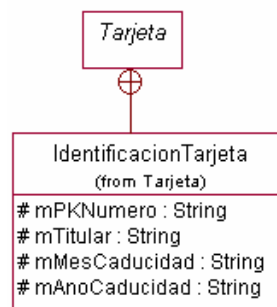
5.02 Enumeraciones

Una enumeración es un tipo de datos definido por el usuario cuyas instancias están en un conjunto predefinido de valores.

Se representan con el símbolo de una clase estereotipado como *enumeration*. Los valores de la enumeración se sitúan en el compartimento central, y las operaciones que actúan sobre los valores se sitúan en el compartimento inferior.

5.03 Clases anidadas

Una clase anidada está declarada dentro de otra clase y es accesible sólo a ésta.

Figura 40. La clase *IdentificacionTarjeta* se ha definido dentro de *Tarjeta*

5.04 Tipos e implementaciones

5.05 Clases parametrizadas (plantillas)

Una clase parametrizada es una clase cuyos atributos son de tipos genéricos que se pasan como parámetros. Puede definirse, por ejemplo, una clase parametrizada *Pila* con

las operaciones habituales *push(Tipo)* y *pop():Tipo*; en este caso, el tipo de datos con el que la clase trabaja (*Tipo*) se pasa como parámetro, de manera que a partir de una clase parametrizada podemos definir una familia de clases, cada una de las cuales trabajará con tipos de datos diferentes pero con un comportamiento idéntico. En nuestro ejemplo, podríamos definir la *pila de enteros*, *pila de cadenas* o *pila de personas*.

La siguiente figura muestra una *ListaOrdenada* que actúa sobre cualquier tipo de datos, así como las listas específicas de cuentas y de tarjetas. Nótese que los tipos concretos se ponen junto al estereotipo *bind*.

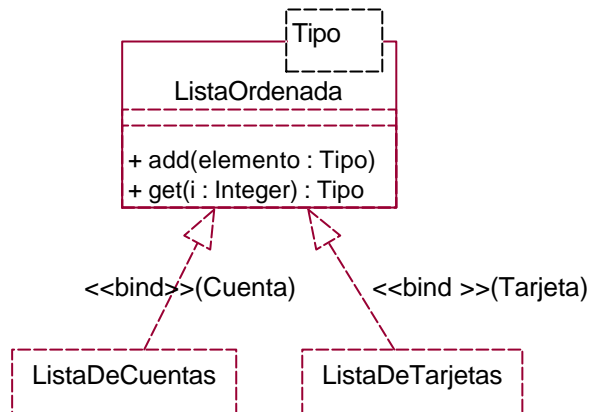


Figura 41

Las clases parametrizadas tienen un reflejo directo en el lenguaje C++, que permite la definición de plantillas de clases (*templates*), en las que los tipos de datos sobre los que actúa la clase son pasados como parámetros. Al compilar, se generan tantas versiones de la plantilla de clase como versiones diferentes de ella se hayan utilizado.

6. Construcción de diagramas de clases en OMT

En la metodología OMT (*Object Modeling Technique*), que es una de las precursoras de UML, se proponen los siguientes pasos para la elaboración inicial de un diagrama de clases²:

1. *Identificación de clases*. Se extraen todos los sustantivos de la descripción del problema, convirtiendo cada sustantivo en una clase candidata.

² En <http://cannes.rhon.itam.mx/Alfredo/Espaniol/Publicaciones/Omt.htm>, el profesor Alfredo Weitzenfeld tiene publicado un amplio manual en español.

2. *Selección de clases.* Se seleccionan las clases importantes para el análisis, eliminando clases que no tengan sentido en el área de aplicación, las clases redundantes, las clases que serán realmente atributos, operaciones o roles en vez de clases, las clases inespecíficas, etc.
3. *Identificación de asociaciones y agregaciones.* Tendremos en cuenta los verbos o proposiciones verbales encontradas en el enunciado del problema, si bien eliminaremos aquellas que relacionan clases ya eliminadas, las relaciones de implementación (el mantenimiento de éstas es un error muy frecuente) y las asociaciones derivadas o redundantes.
4. *Identificación de atributos.* Éstos tienen menor probabilidad de ser descritos por el problema. Algunos atributos se encuentran en el conjunto de clases candidatas eliminadas con anterioridad.
5. *Adición de multiplicidades.*
6. *Adición de herencia.* Se identifican características comunes de las clases existentes, que se agrupan en superclases. Se puede utilizar herencia múltiple para refinar el modelo, si bien hay que tener cuidado con ella porque aumenta la complejidad conceptual y la de implementación.

7. Verificación y validación de diagramas de clases

Podemos utilizar la siguiente lista de comprobación para los diagramas de clases:

- Todas las clases tienen su nombre en singular
- Todos los atributos tienen tipo y visibilidad
- No hay atributos que estén representando asociaciones o agregaciones
- Las cardinalidades de asociaciones y agregaciones son correctas
- La navegabilidad de asociaciones y agregaciones son correctas
- Se han identificado adecuadamente “asociaciones como clases”
- Se han identificado claramente relaciones de herencia
- Se usa la notación adecuada para clases abstractas
- Se usa la notación adecuada para operaciones abstractas
- Se usa la notación adecuada para especificar operaciones
- En caso de que haya “asociaciones de implementación”, se encuentran claramente estereotipadas

8. Ejercicios

En los siguientes casos se pide que construya, verifique y valide los diagramas de clase.

8.01 Gestión de PFC

La subdirección académica y de gestión de calidad de la Escuela Superior de Informática de la UCLM desea construir una intranet que permita gestionar fácilmente los Proyectos de Fin de Carrera (PFC). Los profesores de los departamentos proponen cada curso un conjunto de PFC. Más adelante, un alumno y un profesor se ponen de acuerdo para su realización. El alumno redacta un anteproyecto que presenta en la Secretaría de la ESI con el Vº Bº del profesor, que actuará como director, así como una propuesta de tribunal formada por seis profesores.

Más adelante, la Comisión Académica de la ESI suele aprobar el anteproyecto y nombra presidente, secretario, vocal y tres suplentes para componer el tribunal. Después, el alumno presenta en la Secretaría de la ESI su PFC; la Dirección de la ESI lo comunica al presidente del tribunal, que convoca a los miembros y al interesado para su defensa. El PFC se califica con MdH, Sb, Nt, Ap o Suspenso.

Si la Comisión Académica no aprueba el anteproyecto, se anulan los trámites realizados hasta la fecha.

El sistema debe ser accesible sólo a profesores. El usuario se identificará ante el sistema introduciendo el mismo nombre de usuario y contraseña que utiliza con el servidor de correo electrónico.

8.02 Gestión de solicitudes de ayuda por asistencia a eventos

La subdirección académica y de gestión de calidad de la Escuela Superior de Informática de la UCLM desea construir una intranet que permita gestionar fácilmente las solicitudes de ayuda por asistencia a eventos de carácter docente: un profesor solicita a la ESI ayuda económica para asistir a algún congreso o curso de este tipo, en el que tal vez presente alguna ponencia. La Comisión de Calidad resuelve en cierta fecha las solicitudes presentadas, asignando una cantidad de dinero a cada profesor que lo solicitó (o rechazando las solicitudes).

Cuando el profesor vuelve, presenta en la Administración del centro las facturas y justificantes necesarios para pagarle el importe que solicitó (habiendo casos en los que se justifica menos dinero y casos en que el profesor no acude finalmente al evento),

momento en el que se la hace una transferencia por la cantidad solicitada y se cierra el expediente.

Además, es posible solicitar un anticipo, que es transferido al profesor antes del evento y justificado posteriormente. En caso de inasistencia, este dinero es devuelto a la ESI.

El sistema debe ser accesible sólo a profesores. El usuario se identificará ante el sistema introduciendo el mismo nombre de usuario y contraseña que utiliza con el servidor de correo electrónico.

8.03 Integración

Integre los dos sistemas descritos anteriormente.

CAPÍTULO 3. ARQUITECTURA MULTICAPA (I)

1. Introducción

Los diagramas de clase que hemos visto en el capítulo anterior se centran en el modelado del dominio del sistema; es decir, en la descripción de las relaciones estructurales entre los elementos que encontramos en el enunciado del problema. En la práctica, los sistemas de información interactúan con personas, bases de datos, dispositivos hardware, otros sistemas, etc. Es deseable separar la lógica de la aplicación (es decir, los elementos que poseen la mayor complejidad del sistema) del resto de cosas, lo que puede conseguirse mediante un correcto diseño arquitectónico del sistema.

Una de las propuestas más interesantes en este sentido es la arquitectura multicapa, en la que (como su propio nombre indica) se distinguen un conjunto de capas, a cada una de las cuales se le asigna un conjunto bien definido de responsabilidades. Algunos de los beneficios de estas arquitecturas son los siguientes (Larman, 1998):

- Aislamiento de la lógica de la aplicación en componentes separados, que pueden ser reutilizados en otros sistemas.
- Distribución de capas en diferentes máquinas o procesos, lo que puede mejorar el rendimiento, aumentar la coordinación y la compartición de información en sistemas cliente/servidor.
- Dedicación de recursos humanos específicos para cada capa (excelentes diseñadores gráficos para la capa de Presentación, p.ej.) y posibilidad de desarrollarlas en paralelo.

En este capítulo estudiaremos en detalle esta arquitectura.

2. Arquitectura de tres capas

Dentro de las arquitecturas multicapa, una de las más utilizadas es la de tres capas o niveles, de la que ya hablamos en el epígrafe 3, página 16. Cada capa compone un subsistema en el que ubicamos clases con responsabilidades propias de la capa a la que pertenecen.

La siguiente figura muestra un diagrama de paquetes con la estructura general de una aplicación de tres capas:

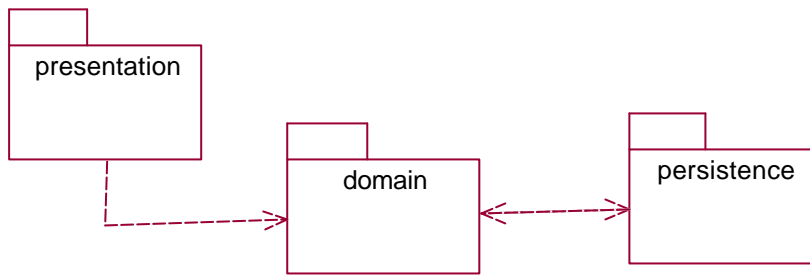


Figura 42. Estructura general de un sistema multicapa

Cada paquete representa un grupo de elementos del sistema, entre los que puede haber clases u otros paquetes (y otros tipos de elementos, aunque no viene al caso). Las relaciones más habituales entre paquetes son las de dependencia, con lo que significamos que dentro un paquete hay elementos que necesitan de los elementos del otro paquete. En la Figura 42, por ejemplo, estamos diciendo que los elementos del paquete *Presentation* necesitan a los elementos del paquete *Domain* para funcionar, y que los de *Domain* y *Persistence* se necesitan mutuamente.

La figura anterior es bastante representativa de las dependencias deseables en un sistema de tres capas. En efecto:

- Los objetos de la capa de Presentación son pantallas (*frames*, diálogos, páginas web, pantallas en formato texto o de otro tipo) que reciben mensajes del usuario, realizan quizás algún tipo de validación sencilla, los pasan hacia dentro (en donde se ejecutan) y muestran resultados y el estado de los objetos de la capa de Dominio. Es decir, que al menos unas pocas clases de la capa de Presentación necesitan conocer a, al menos, algunos objetos de la capa de Dominio; esto se representa con la relación de dependencia desde Presentación (quien depende) hacia Dominio (de quien se depende).
- Los objetos de la capa de Dominio, que son los realmente complejos, deben ser diseñados de manera que “haya que cambiarlos poco”, para que “puedan funcionar con cualquier tipo de pantalla”. Se desea, entonces, que estos objetos no dependan del tipo de pantalla que utiliza el usuario para interactuar con ellos, por lo que la capa de Dominio no debe depender de la de Presentación. Por ello, en la figura anterior no hay dependencia en este sentido.

- En algún momento, los objetos de la capa de Dominio querrán guardarse en la base de datos, para lo que tendrán que comunicarse con ella de alguna manera. Por eso, y aunque es poco deseable que la capa de Dominio dependa de nadie, no suele haber más remedio que tener que representar la relación de dependencia desde Dominio hacia Persistencia. Por otro lado, cuando en Persistencia se recibe un mensaje de un objeto de Dominio que quiere, digamos, insertarse, la clase correspondiente de Persistencia tendrá (habitualmente) que acceder al objeto, leer su estado (formado por su conjunto de atributos) y guardarlo en la base de datos. Por tanto, como Persistencia conoce a Dominio, hemos marcado la dependencia desde la capa derecha hacia la capa central. Hay por tanto una dependencia mutua, ya que una capa depende de la otra.

Dentro de una capa puede haber subcapas. Por ejemplo, en la de Presentación tendremos probablemente un conjunto de ventanas que utilizaremos, sin cambios, en muchas aplicaciones, como pueden ser diálogos que muestran mensajes de error, de “acerca de...”, de solicitud de confirmación, etc.; en la de Dominio tal vez distingamos entre los diferentes subsistemas de gestión de cuentas, préstamos, tarjetas, fondos de inversión, etc. Esto se muestra en la siguiente figura:

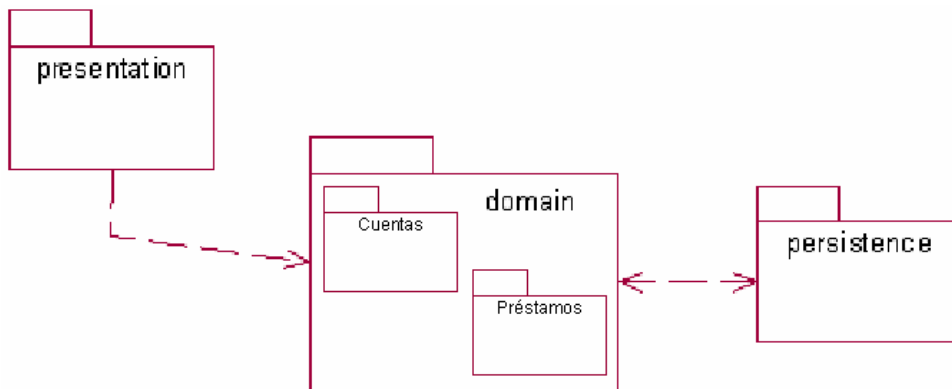


Figura 43

Podemos detallar a nivel de clases el diagrama de la figura anterior. Como se observa, prácticamente todo el esfuerzo de modelado se dedica a la capa de Dominio.

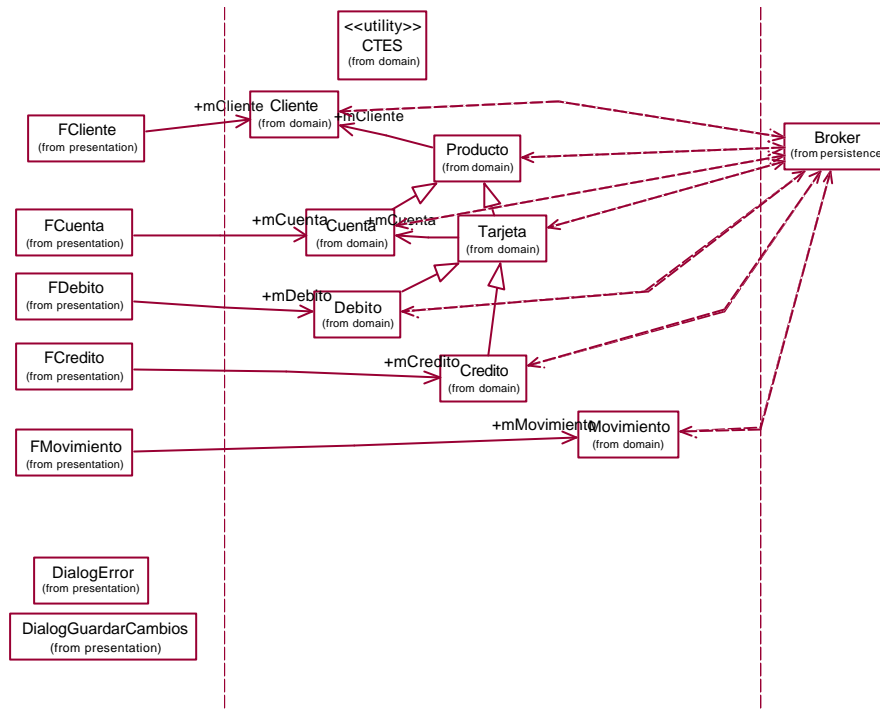


Figura 44. La Figura 43, detallada a nivel de clases

3. Construcción de la base de datos

Es muy habitual utilizar el diseño de la capa de Dominio (o parte de él) como esquema conceptual de la base de datos en la que se almacenarán las instancias de las clases persistentes. Esta es una de las razones por las que se dice que el paradigma orientado a objetos unifica el modelado de datos y el de procesos (recuérdese que, en el paradigma estructurado, se trabaja por separado utilizando DFD's para los procesos, y diagramas E/R o E/ER para los datos). Cada vez más gente utiliza notación de objetos para construir los esquemas conceptuales de datos, sustituyendo así a los E/R y E/ER.

El modelo de datos más extendido en las empresas es el relacional (a pesar de la aparición en los últimos años de otros modelos, como el orientado a objetos o el objeto-relacional), por lo que nos centraremos en el diseño de bases de datos relacionales. Para construir una base de datos relacional a partir de un diagrama de clases, podemos utilizar tres patrones³ de transformación bien conocidos, o bien combinaciones de ellos.

³ Un patrón es una "solución buena a un problema frecuente".

3.01 Patrón “una clase, una tabla” (1C1T)

Mediante este patrón, se construye una clase por cada tabla. Las relaciones de herencia se transforman en restricciones de clave externa 1 a 1, y las asociaciones y agregaciones se traducen a relaciones de clave externa cuya cardinalidad depende de la multiplicidad de la asociación o agregación. Una posible distinción entre asociaciones y agregaciones podría estar en que, en éstas, especificáramos disparadores del tipo *on delete cascade* y *on update cascade*.

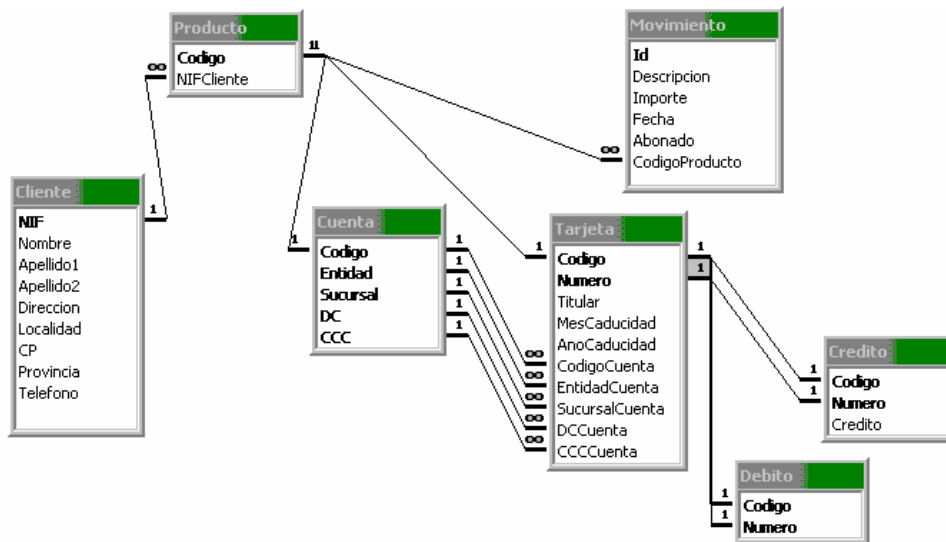


Figura 45. Base de datos construida al aplicar el patrón una clase, una tabla a la capa de Dominio de la Figura 44

3.02 Patrón “un árbol de herencia, una tabla” (1AH1T)

Este patrón no suele ser aplicado para generar la base de datos completa, sino que se aplica a pequeños subconjuntos del diagrama de clases. Consiste en construir una sola tabla para representar una generalización. En nuestro ejemplo (véase Figura 44) hay muy poca diferencia entre las clases *Debito* y *Credito*, por lo que podríamos almacenar todos los datos de tarjetas en una sola tabla que se llamara *Tarjeta*.

Con este cambio, nuestra base de datos quedaría con el diseño que se muestra en la Figura 46. En la nueva tabla guardaremos todos los registros correspondientes a tarjetas, tanto de crédito como de débito. La tabla contiene todos los campos existentes en el árbol de herencia del cual procede. Evidentemente, las tarjetas de débito tendrán su columna Crédito a NULL.

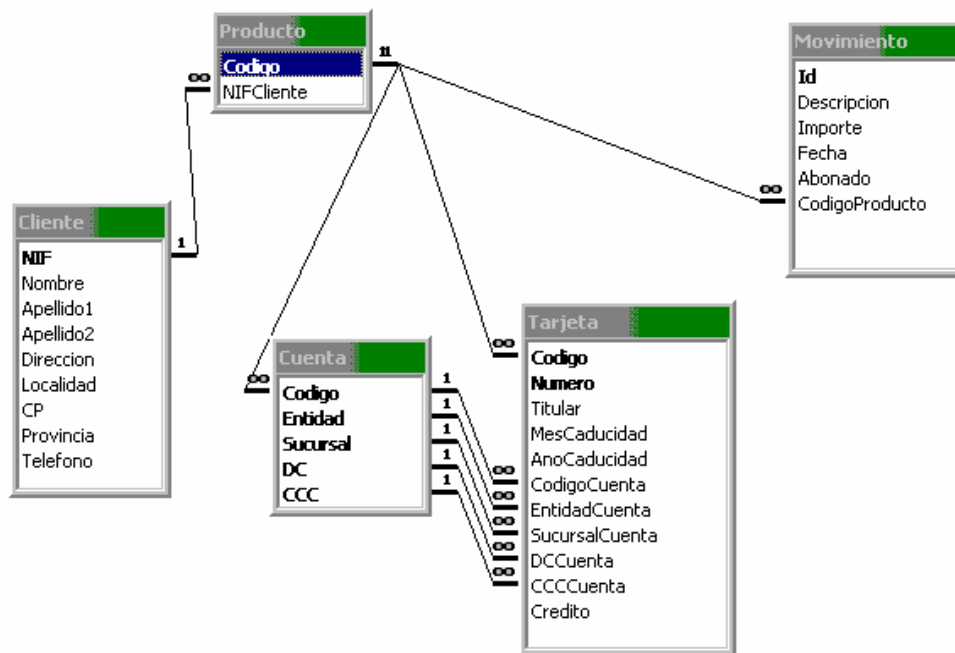


Figura 46. Base de datos construida al aplicar el patrón un árbol de herencia, una tabla al fragmento de tarjetas de la capa de Dominio de la Figura 44

Podríamos haber resumido los datos de todos los productos en una sola tabla *Productos*, pero entonces el número de columnas con valor NULL sería enormemente grande. Además, sería difícil realizar algunos tipos de consultas (p. ej., datos de tarjetas de crédito), ya que están mezclados en la misma tabla los datos de todos los productos. Una solución a este problema sería añadir una columna adicional de información con el tipo de producto, que empeora la calidad del diseño de la base de datos.

3.03 Patrón “un camino de herencia, una tabla” (1CH1T)

Con este patrón se crea una tabla por cada camino identificable en un árbol de herencia. Igual que ocurre con el anterior, es raro que se aplique solo a un diagrama de clases, siendo lo habitual que se utilice para un fragmento.

La siguiente figura muestra el resultado de aplicar este patrón al árbol de herencia completo. El número de tablas disminuye respecto del “una clase, una tabla”. Hay columnas duplicadas en todos los productos, y múltiples restricciones de clave externa desde *Cliente* y hacia *Movimiento*. Por otro lado, resulta difícil garantizar la unicidad del código de cada producto, que debe ser único.

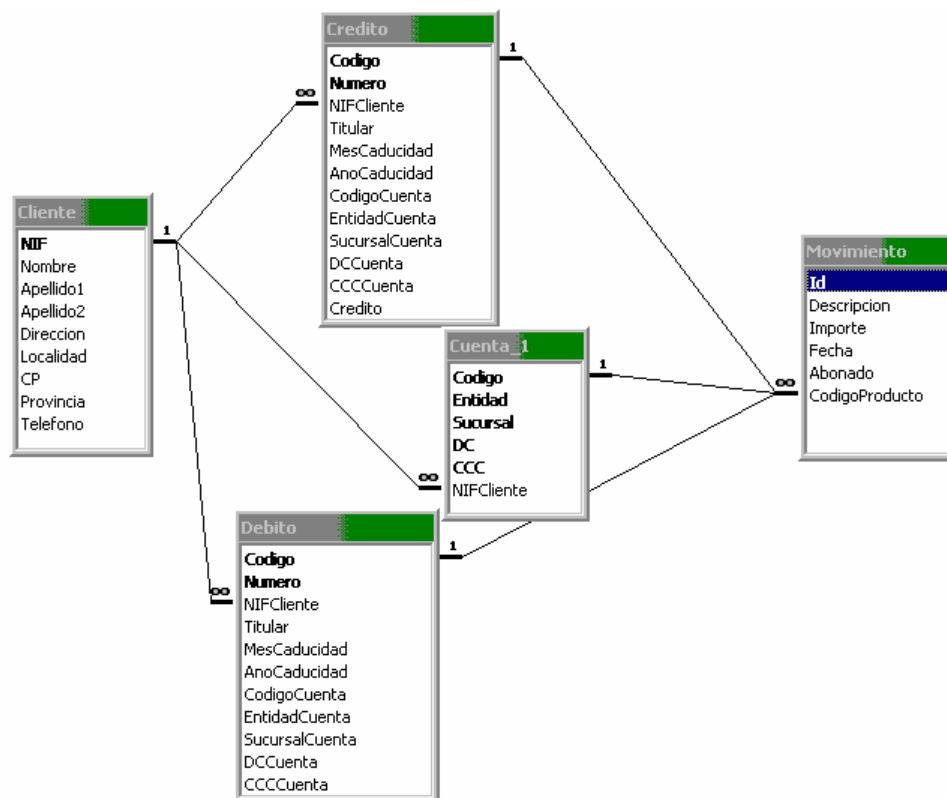


Figura 47. Base de datos producida al aplicar el patrón “un camino de herencia, una tabla”

Una dificultad adicional de este patrón se da cuando una de las clases intermedias en un camino de herencia no es abstracta. En nuestro caso, si *Tarjeta* no lo fuera, tendríamos que haber construido una tabla adicional, o bien haber aplicado 1AH1T a la jerarquía de tarjetas.

4. Operaciones de persistencia

Las cuatro operaciones básicas de persistencia se agrupan bajo la denominación *CRUD* (página 75), que sirven para insertar, modificar o borrar instancias de la base de datos, y para crear instancias a partir de la información contenida en ella.

Cuando se ha utilizado el patrón 1C1T para construir la base de datos, es muy fácil escribir el código básico de las operaciones CRUD. Suponiendo que se ha mantenido una adecuada convención de nombrado de clases con respecto a sus tablas, y de campos con respecto a columnas, el código SQL necesario para insertar al cliente Paco Pil será:

```
Insert into Cliente
(NIF, Nombre, Apellido1, Apellido2, Direccion,
 Localidad, CP, Provincia, Telefono)
values ('12345678A', 'Paco', 'Pil', 'López', 'Plaza Mayor, 5',
```

```
'Ciudad Real', '13001', 'Ciudad Real', '926223344')
```

Evidentemente, en tiempo de ejecución deberemos acceder a los auténticos valores de la instancia, que están almacenados en sus atributos *mPKNIF*, *mNombre*, *mApellido1*, etc. El método *insert()*, entonces, tendrá que componer una cadena adecuada a la clase, a la base de datos y al gestor de base de datos (hay que tener en cuenta detalles como los tipos de comillas para las cadenas de texto, quizás almohadillas para las fechas, valores como *true*, *false* o *1*, *0* para los lógicos, etc.). Una posible forma de componer la cadena para insertar a un cliente cualquiera sería:

```
String SQL="Insert into Cliente " +
  "(NIF, Nombre, Apellido1, Apellido2, Direccion, " +
  "Localidad, CP, Provincia, Telefono) " +
  "values (" +
  "\"" + mPKNIF + "\", \"" + mNombre + "\", \"" + mApellido1 + "\", \"" +
  mApellido2 + "\", \"" + mDireccion + "\", " +
  "\"" + mLocalidad + "\", \"" + mCP + "\", \"" + mProvincia + "\", \"" +
  mTelefono + "\")";
```

A veces, la instancia que queremos insertar no tiene directamente en sus campos el valor de alguna columna que sí está en su tabla asociada. Por ejemplo, la clase *Producto* no tiene información del NIF del cliente que la posee, pero sí que tiene conocimiento de la instancia de tal cliente (véase Figura 44). Para componer la inserción de una producto necesitaremos acceder al campo *mPKNIF* de la clase cliente, que es protegido y por tanto no accesible a *Producto*. Afortunadamente, *Cliente* tendrá una operación *getNIF():String* que lo devuelve:

```
String SQL="Insert into Producto " +
  "(Codigo, NIFCliente) values (" +
  mPKCodigo + ", \"" + mCliente.getNIF() + "\")";
```

En otras ocasiones, la inserción de un registro en una tabla implica la inserción de un registro en otra. Por ejemplo, la restricción de integridad referencial entre *Producto* y *Cuenta* nos obliga a insertar un registro en la primera antes de insertarlo en ésta. Puede ocurrir, además, que la inserción del primer registro (en *Productos*) falle, por lo que ni siquiera debería intentarse la inserción en *Cuenta*. En estos casos, ambas operaciones deben ejecutarse dentro de una transacción.

4.01 Acceso a la base de datos: patrón Agente (*Broker*)

La clase *Broker* que aparece en la capa de Persistencia de la Figura 44 corresponde a un patrón *agente de base de datos (database broker)*. Un agente de base de datos es una clase que se encarga de dar acceso a la base de datos a los objetos de la capa de Dominio, de manera que se consigue cierto desacoplamiento de éstos.

La única clase con conocimiento directo de la base de datos es el agente, que ofrece una interfaz adecuada a los objetos de la capa de Dominio. Aunque hay varias alternativas (véase página 463 de Larman, 1998), una forma sencilla pero efectiva de utilizar un agente consiste en dotarlo de un conjunto de operaciones públicas que ejecuten instrucciones SQL (para el caso relacional):

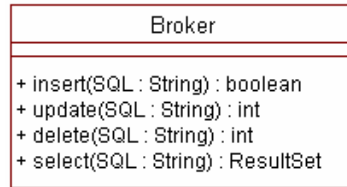


Figura 48. Algunas de las operaciones ofrecidas por el Agente

El Agente debe tener un medio de acceso a la base de datos. En Java, la interfaz *Connection* incluye todas las operaciones necesarias para acceder y manipular la base de datos. La interfaz *Connection* es implementada por el gestor de base de datos (esto es, por Microsoft Access, Oracle, etc.):

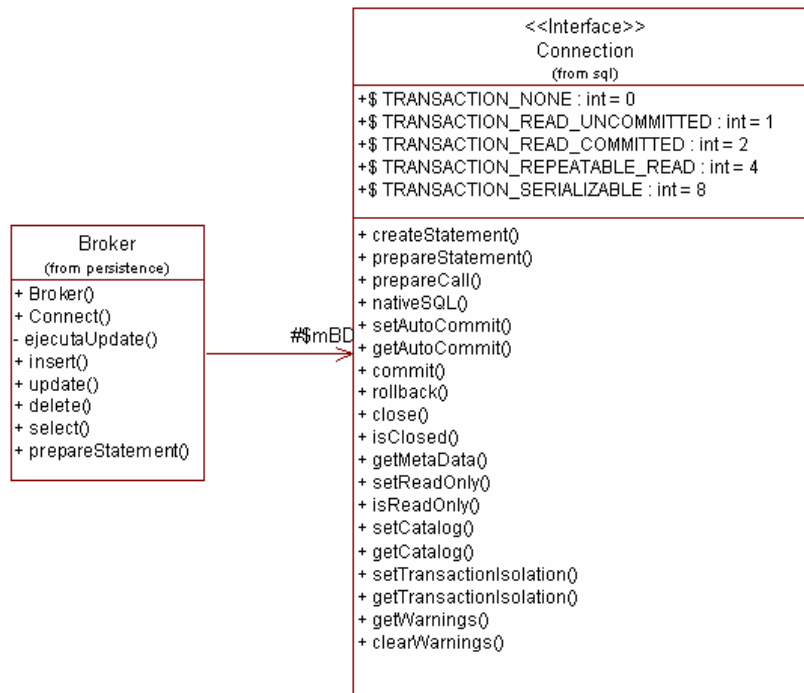


Figura 49. El Agente es el único que conoce a la base de datos a través una *Connection*

4.01.1 El patrón “Singleton”

Como se observa en la figura anterior, el agente tiene un atributo estático *mBD* que representa el acceso a la base de datos. Si todas las clases de la capa de Dominio acceden a la base de datos a través de este agente, resulta que, en todo momento, existe un único canal de acceso a la base de datos, que es utilizado por cualquier objeto. Esta forma de acceso es habitual, pero pueden desde luego utilizarse otras (piénsese, por ejemplo, qué ocurriría si compartiésemos una sola conexión a la base de datos en la “oficina virtual” de nuestra entidad bancaria).

Cuando en un sistema imponemos la sola existencia de una única instancia de cierta clase (como es éste caso), podemos hacer uso del patrón *Singleton*, que nos dice que definamos un método que devuelva la única instancia de la clase. Con este patrón, en lugar de tener la clase Agente como en la figura anterior, tendría esta forma:

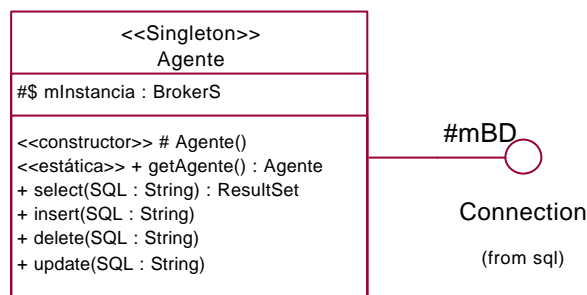


Figura 50. El Agente de base de datos, como un Singleton

4.01.2 Detalles de implementación

Un posible código Java de la clase Agente como un Singleton es el siguiente:

```

package persistencia;

import java.sql.*;

public class Agente {
    protected static Agente mInstancia=null;
    protected Connection mBD;

    /**
     * Nótese que el constructor es protegido. Cuando un objeto necesita
     * al Agente, lo pide a través de su método público getAgente
     */
    protected Agente() throws Exception {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        String url="jdbc:odbc:JdbcOdbcDriver";
        mBD=DriverManager.getConnection(url);
    }

    public static Agente getAgente() throws Exception {
        if (mInstancia==null) {
            mInstancia=new Agente();
        }
    }
  
```

```

    }
    return mInstancia;
}

public ResultSet select(String SQL) throws Exception {
    return mBD.createStatement().executeQuery(SQL);
}

public ResultSet insert(String SQL) throws Exception {
    return mBD.createStatement().executeUpdate(SQL);
}

public ResultSet delete(String SQL) throws Exception {
    return mBD.createStatement().executeUpdate(SQL);
}

public ResultSet delete(String SQL) throws Exception {
    return mBD.createStatement().executeUpdate(SQL);
}
}

```

4.02 Asignación de responsabilidades de persistencia

Como ya hemos dicho, cada clase persistente debe implementar, al menos, las operaciones CRUD que gestionarán la persistencia de sus instancias. Puesto que queremos que nuestra capa de Dominio sea lo más reutilizable posible y, por tanto, tan independiente como se pueda de las capas adyacentes, debemos pensar bien el lugar en el que colocaremos estas operaciones.

4.02.1 Patrón “Experto”

Este patrón (Larman, 1998) nos dice que asignemos las responsabilidades al experto en la información; es decir, a la clase que posee la información necesaria para cumplir con tal responsabilidad.

Si seguimos este criterio, cada clase persistente será responsable de implementar por completo sus métodos de persistencia. De este modo, la implementación de la clase *Cliente* podría ser:

```

package dominio;

import persistencia.Agente;

public abstract class Producto {
    protected long mPKCodigo;
    protected Cliente mCliente;

    public Producto() {
        mCodigo=0;
        mCliente=new Cliente();
    }

    /** Constructor materializador */
    public Producto(long codigo) throws Exception {
        String SQL="Select * from Producto where Codigo=" + codigo;
        ResultSet r=Agente.getAgente().select(SQL);
        if (r.next()) {
            mPKCodigo=r.getLong(1);
        }
    }
}

```

```

        mCliente=new Cliente(r.getString(1));
    }
    r.close();
}

public void insert() throws Exception {
    String SQL="Insert into Producto (Codigo, NIFCliente) values (" +
        mPKCodigo + ", '" + mCliente.getNIF() + "')";
    Agente.getAgente().insert(SQL);
}

public void delete() throws Exception {
    String SQL="Delete from Producto where Codigo= " + mPKCodigo;
    Agente.getAgente().delete(SQL);
}

public void update(long codigoAntiguo) throws Exception {
    String SQL="Update Producto set Codigo=" + mCodigo +
        ", NIFCliente='" + mCliente.getNIF() + "' " +
        "where Codigo=" + codigoAntiguo;
    Agente.getAgente().update(SQL);
}
}

```

4.02.2 Patrón “Fabricación pura”

De forma general, una fabricación pura (Larman, 1998) es una clase a la que se asignan un conjunto de responsabilidades que, si se asignaran a la clase a la que realmente corresponden, quedaría excesivamente grande, demasiado acoplada, poco cohesionada o, en general, con un detrimento de su calidad. Las clases de Dominio delegan un conjunto de responsabilidades a estas fabricaciones puras. En el caso que nos ocupa, las responsabilidades que pueden delegarse son las relacionadas con la persistencia.

Hay varias estrategias para implementar las fabricaciones puras (Polo et al., 2002):

a) La clase de Dominio puede conocer permanentemente a una instancia de la fabricación pura; cuando la instancia quiere insertarse, ejecuta su método *insert()*, que lo único que hace es decirle a su fabricación pura que le inserte: *insert(this)*.

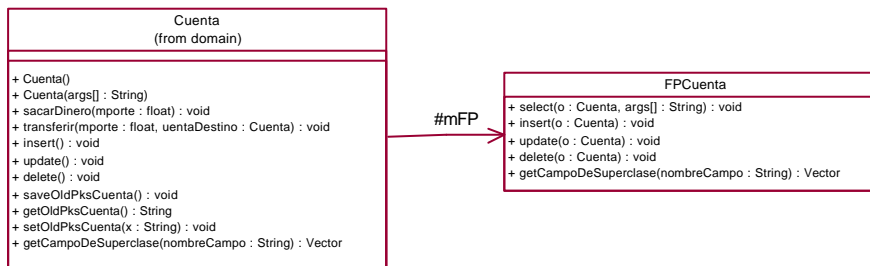


Figura 51. La fabricación pura es conocida en todo momento por la clase de Dominio

En este caso, un fragmento de las dos clases es:

```

public class Cuenta {
    ...
    protected FPCuenta mFP;
    public void insert()
        throws Exception {
        mFP.insert(this);
    }
    ...
}

public class FPCuenta {
    ...
    public void insert(Cuenta o)
        throws Exception {
        String SQL="Insert into Cuenta ()"+
            ... + o.getCCC() + ...;
        Agente.getAgente().insert(SQL);
    }
    ...
}

```

b) La clase de Dominio y la fabricación pura pueden conocerse mutuamente. Cuando la instancia de Dominio quiere insertarse, ejecuta la operación *insert()* de la fabricación pura (nótese que ahora no se pasa como parámetro la propia instancia de Dominio, ya que es conocida por la fabricación pura).

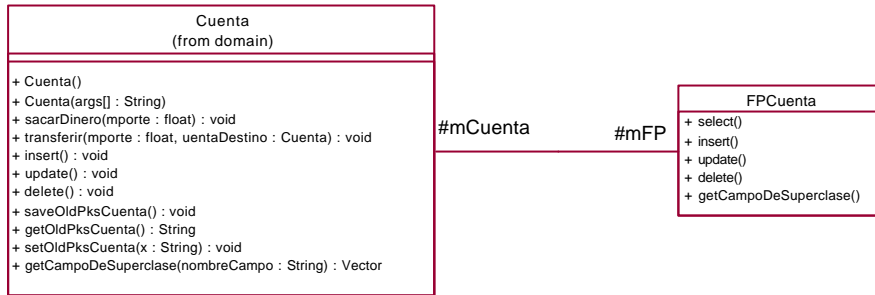


Figura 52. La fabricación pura y la clase de Dominio se conocen mutuamente

Un fragmento ilustrativo del código de estas clases es:

```

public class Cuenta {
    ...
    protected FPCuenta mFP;
    public void insert()
        throws Exception {
        mFP.insert();
    }
    ...
}

public class FPCuenta {
    ...
    protected Cuenta mCuenta;
    public void insert() throws Exception {
        String SQL="Insert into Cuenta ()"+
            ...+ mCuenta.getCCC() + ...;
        Agente.getAgente().insert(SQL);
    }
    ...
}

```

c) La clase de Dominio no conoce a ninguna instancia de la fabricación pura, ya que ésta consta exclusivamente de un conjunto de operaciones estáticas.

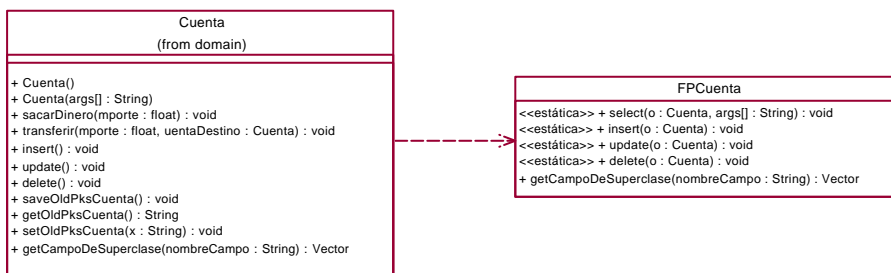


Figura 53. Delegación a una clase con operaciones estáticas

El código, en este caso, podría ser de este estilo:

```
public class Cuenta {
    ...
    protected FPCuenta mFP;
    public void insert()
        throws Exception {
        FPCuenta.insert(this);
    }
    ...
}

public class FPCuenta {
    ...
    public static void insert(Cuenta o)
        throws Exception {
        String SQL="Insert into Cuenta ()"+
            ...+ o.getCCC() + ...;
        Agente.getAgente().insert(SQL);
    }
    ...
}
```

Las ventajas evidentes de las fabricaciones puras son el incremento de cohesión en las clases de la capa de Dominio, que se descargan de un conjunto de responsabilidades que no tienen directamente que ver con el adecuado cumplimiento de las funciones que les corresponden según el enunciado del problema. A la vez, se decrementa el acoplamiento de las clases de Dominio, que dejan de depender de los detalles específicos del Sistema Gestor de Base de Datos (cuantitativamente, el acoplamiento es igual o mayor, pero se trata de un acoplamiento “menos malo”). Las fabricaciones puras son un medio importante para lograr el *bajo acoplamiento* y la *alta cohesión*, que son dos principios básicos del diseño orientado a objetos, que además se suelen “vender” en forma de patrones.

4.02.3 Patrón RCRUD (*Reflective CRUD*)

El patrón RCRUD (Polo et al., 2001) proporciona persistencia a las clases de la capa de Dominio mediante reflexión o introspección, implementando las operaciones CRUD. La reflexión es una característica de algunos lenguajes de programación que permite a una instancia conocer algunos detalles de la clase a la que pertenece en tiempo de ejecución.

En Java, la reflexión la proporciona el paquete *java.lang.reflect.**, que define, entre otras, las clases mostradas en la Figura 54.

El principio básico para el uso de reflexión con el fin de dar persistencia consiste en que, en muchas ocasiones, el código de las operaciones CRUD tiene muy pocos cambios de una clase a otra: podemos componer la instrucción para insertar una instancia de cierta clase de este modo:

```
Insert into Nombre_de_la_Tabla
(column1, columna2, ...)
values
(valor_del_campo_1, valor_del_campo_2, ...)
```

Con adecuadas convenciones de nombrado, podemos extraer el *Nombre_de_la_Tabla* a partir del nombre de la clase de la instancia que se quiere insertar (lo cual puede obtenerse mediante *getClass().getName()*), los nombres de las columnas a

partir de los nombres de los campos (mediante *getFields():Field[]*, y con *getName()*), y los valores de los campos mediante los métodos *get()* de la clase *Field*.

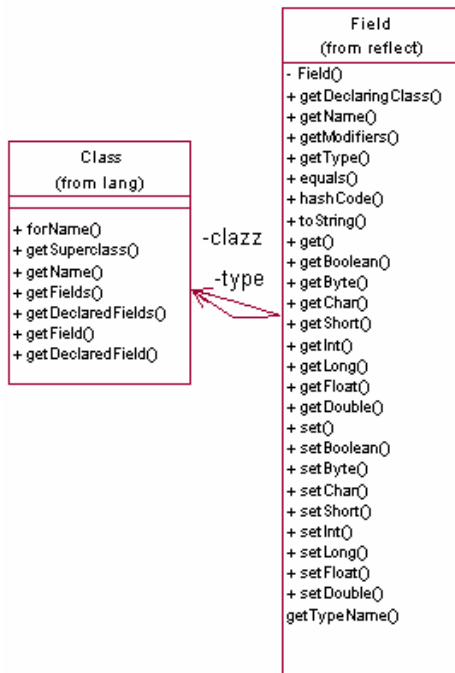


Figura 54. Algunas clases útiles para la reflexión

Todas las clases persistentes de la capa de Dominio heredan de una superclase RCRUD en la que, mediante reflexión, se generan en tiempo de ejecución las operaciones CRUD. Si alguna operación CRUD de alguna clase tiene un comportamiento diferente del comportamiento por defecto, la redefine. De este modo, las clases de la capa de Dominio no implementan directamente ninguna operación de persistencia (todas las heredan de RCRUD) ni necesitan fabricaciones puras que se encarguen de ella. Además, RCRUD puede actuar de agente de base de datos.

Además, un cambio en una tabla de la base de datos (adición de una columna, por ejemplo), sólo requiere que se añada ese campo a la correspondiente clase, sin que sea necesario modificar sus métodos, ya que la instrucción de persistencia se genera automáticamente en tiempo de ejecución en función de la estructura de la clase.

La estructura de la capa de Dominio queda como se muestra en la Figura 55.

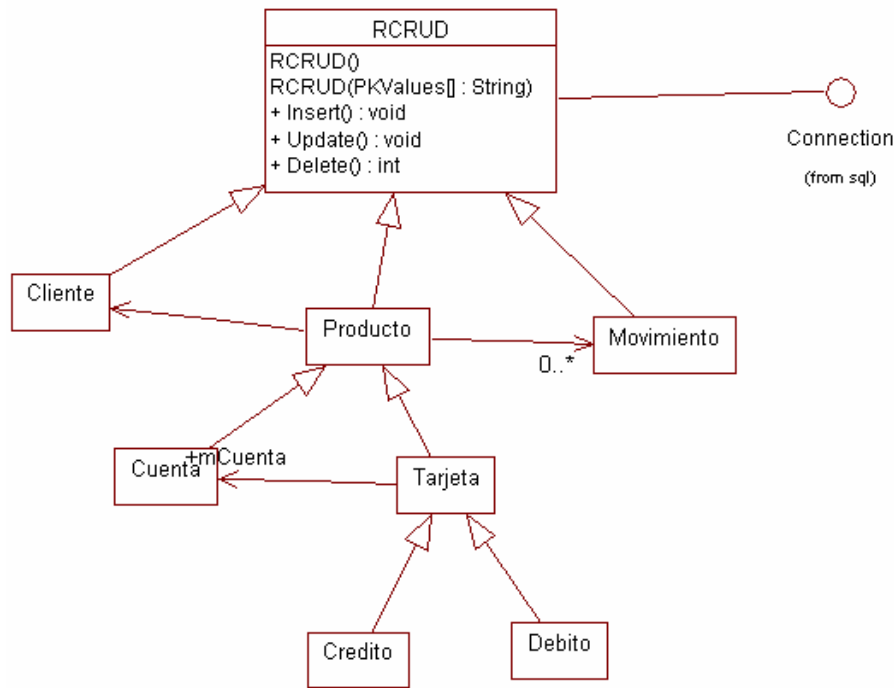


Figura 55. Asignación de persistencia mediante RCRUD.

5. Ejercicios

5.01 Tolerancia a fallos

Suponga que tenemos un sistema en el que los datos se guardan, borran, actualizan, etc. simultáneamente en dos bases de datos diferentes, situadas en máquinas distintas. Proponga un diseño de la capa de Persistencia que lo haga posible.

5.02 Diferentes modelos

En el caso anterior, suponga que una de las bases de datos es relacional y la otra es orientada a objetos.

5.03 Gestión de PFC

Diseñe la aplicación del ejercicio 8.01 con estructura de tres capas.

5.04 Ayudas por asistencia a eventos

Diseñe la aplicación del ejercicio 8.02 con arquitectura de tres capas.

5.05 Integración (I)

Integre las aplicaciones de los dos ejercicios anteriores.

5.06 Integración (II)

Integre las aplicaciones de los ejercicios 5.03 y 5.04, suponiendo que se utilizan dos diferentes bases de datos.

5.07 Construcción de bases de datos

Obtenga el diseño relacional de las bases de datos de los ejercicios anteriores, utilizando los patrones vistos en la sección 3 y combinaciones de ellos.

5.08 Escritura de código

Suponiendo que se ha utilizado el patrón Experto para asignar las responsabilidades de persistencia, escriba el código de algunos de estos métodos para algunas de las clases, dependiendo del patrón de transformación a base de datos elegido.

CAPÍTULO 4. DIAGRAMAS DE INTERACCIÓN

1. Introducción

Los diagramas de interacción (que ya se mencionaron en el capítulo 1) muestran el paso de mensajes entre objetos que se produce cuando el sistema realiza una función determinada en una situación predefinida. Nótese que, en este caso, no hablamos de clases, sino de objetos, ya que son instancias de clases quienes *interaccionan* para ejecutar cierta tarea. Una interacción consta de un conjunto de mensajes parcialmente ordenados.

Hay dos tipos de diagramas de interacción: los de secuencia y los de colaboración.

2. Diagramas de secuencia

Un diagrama de secuencia tiene dos dimensiones: la vertical, que representa el paso del tiempo de arriba abajo, y la horizontal, que muestra objetos y mensajes entre éstos. No obstante, pueden intercambiarse los ejes y representar el tiempo en horizontal y los objetos en vertical.

Supongamos, tomando como base el diagrama de clases de la Figura 44, que un usuario ha rellenado en una ventana de tipo *FCliente* los datos de un cliente nuevo que va a dar de alta en la base de datos. En un escenario normal (esto es, en el que el puede darse de alta al cliente con normalidad), el curso típico de eventos sería el siguiente:

- 1) El usuario rellena los datos del cliente en la ventana
- 2) El usuario pulsa el botón *Guardar* en la ventana
- 3) Puesto que la ventana no tiene acceso directo a la base de datos, la ventana le dice a su correspondiente objeto de clase *Cliente* que le inserte. Este objeto compone la instrucción SQL adecuada y...
- 4) ...le pide al *Broker* que ejecute la instrucción que le pasa como parámetro

Esta situación la podemos representar con el siguiente diagrama de secuencia:

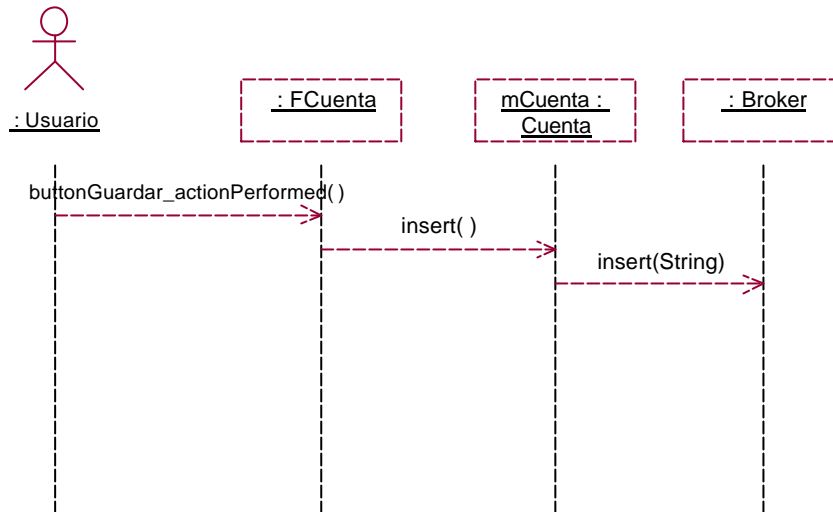


Figura 56. Diagrama de secuencia para insertar un cliente en su escenario normal

Suponiendo que se ha utilizado el patrón 1C1T para construir la base de datos (véase Figura 45), los eventos que se producen para insertar una tarjeta de crédito es bastante diferente, ya que se requiere que previamente se haya insertado su registro correspondiente en la tabla *Tarjeta*, pero antes aún se requiere que se haya insertado en *Producto*. El diagrama de secuencia correspondiente podría ser éste:

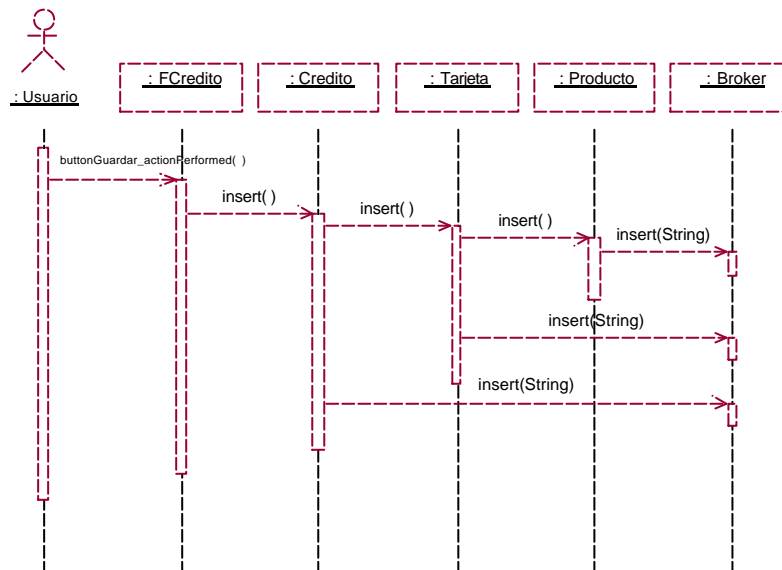


Figura 57. Diagrama de secuencia para insertar una tarjeta de crédito

Una diferencia de notación de la Figura 57 con respecto de la anterior es que ahora hemos incluido *focos de control*, que son las barras verticales situadas sobre las *líneas de la vida* de cada objeto. El foco de control muestra las relaciones existentes entre unos mensajes y otros de la secuencia.

2.01 Condiciones y bucles

Un diagrama de secuencia puede incluir condiciones. En el diagrama siguiente, ilustramos el hecho de que se crea y se muestra un diálogo con un mensaje de error cuando la inserción falla:

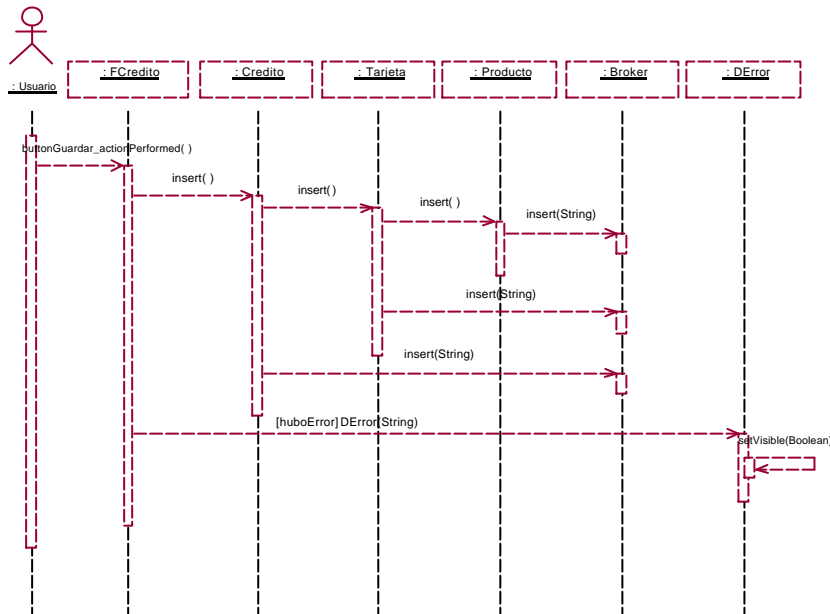


Figura 58. Un diagrama de secuencia con una condición

También se pueden representar bucles: supongamos que, a fin de mes, se procesan todos los movimientos pendientes de las tarjetas de crédito, se hace el cargo a la cuenta asociada y se ponen los movimientos como “pagados”; si la cuenta no tiene saldo suficiente, entonces no se hace el cargo y se anota al cliente como moroso.

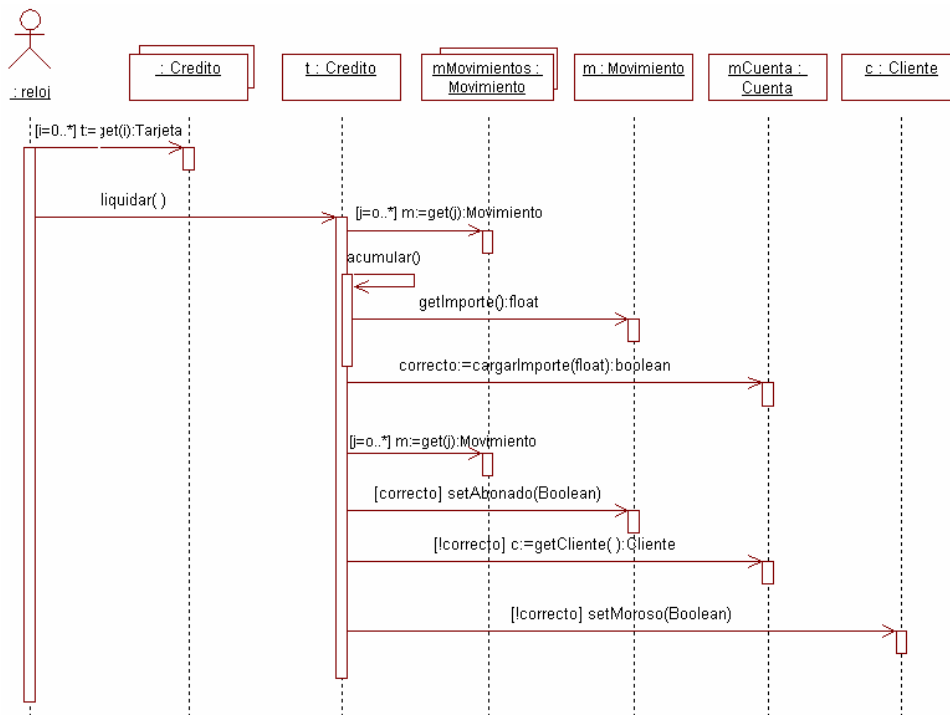


Figura 59. Diagrama de secuencia con bucles, condiciones y múltiples instancias.

3. Diagramas de colaboración

Los diagramas de colaboración muestran con otro tipo de vista exactamente la misma información que los diagramas de secuencia.

Los diagramas de colaboración de los diagramas de secuencia de algunas de las figuras anteriores se muestran a continuación. La ordenación temporal de las operaciones y sus interdependencias se representan mediante números.

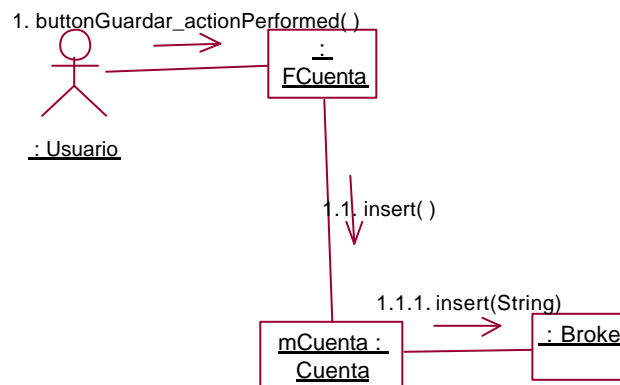


Figura 60. Diagrama de colaboración equivalente al de la Figura 56

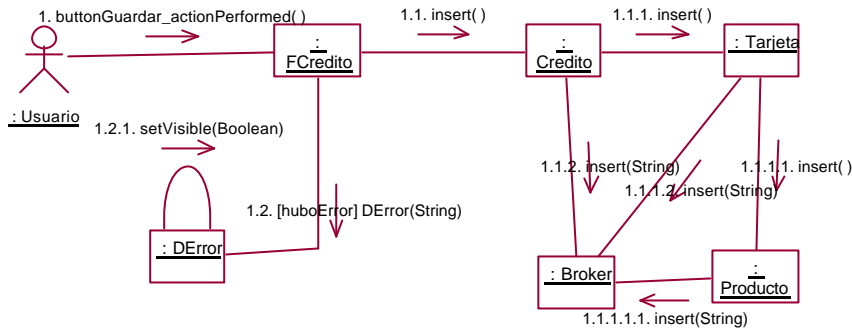


Figura 61. Diagrama de colaboración equivalente al de la Figura 57

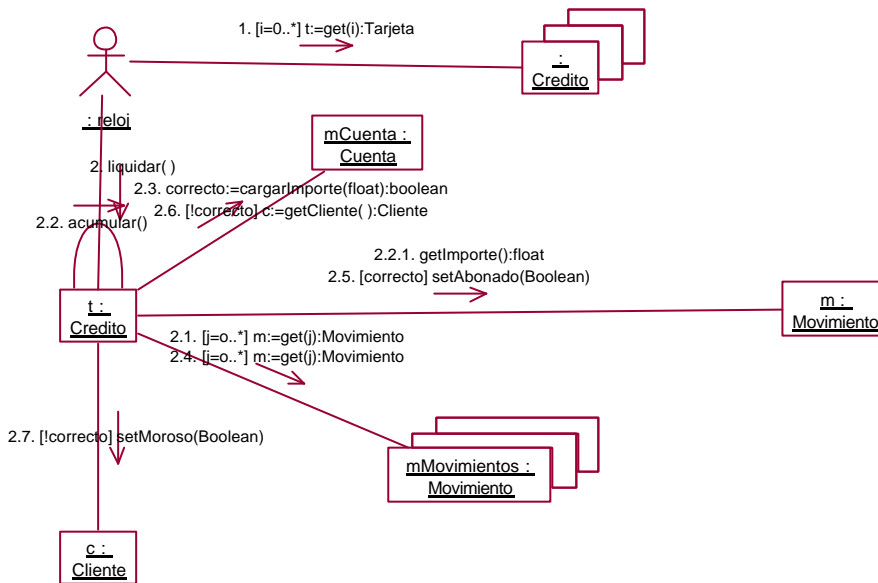


Figura 62. Diagrama de colaboración equivalente al de la Figura 59

4. Nivel de detalle

Dependiendo del nivel de abstracción o de la fase del desarrollo en la que estemos, los mensajes entre objetos pueden tener mayor o menor nivel de detalle. Así, por ejemplo, si únicamente queremos dar una descripción a grandes rasgos de un cierto escenario, podemos utilizar frases en castellano en lugar de nombres de operaciones, y escribir por ejemplo descripciones de mensajes tales como “el usuario rellena los datos” o “la cuenta se guarda en la base de datos”.

Del mismo modo, un diagrama de secuencia no tiene por qué mostrar la secuencia completa de mensajes, pues pueden ocultarse algunos que no sean muy significativos

para el escenario que estamos describiendo. En la Figura 56 y siguientes, por ejemplo, no hemos puesto lo que el *Broker* le dice a la *Connection*.

5. Verificación y validación de diagramas de interacción

Antes de dar por bueno un diagrama de interacción, podemos repasar la siguiente lista de comprobación:

- Los mensajes están representados con un nivel de detalle adecuado
- No hay mensajes inconexos o espontáneos (es decir, no hay mensajes cuyo origen esté situado fuera del foco de control del mensaje que lo lanza)
- Todos los mensajes son operaciones de la clase “destino de la flecha”
- Las clases de las instancias emisora y receptora de los mensajes se conocen en el diagrama de clases
- Los mensajes de segundo nivel y posteriores tiene información suficiente para ejecutarse (es decir, ningún mensaje se olvida de pasar a su siguiente la información que éste necesita)
- No se dirigen a colecciones de elementos mensajes que pertenecen a instancias
- Hay mensajes de creación de instancias cuando es necesario
- El diagrama describe completamente el escenario

6. Ejercicios

6.01 Transacciones

Como se mencionó en el capítulo anterior, hay veces en que un conjunto de operaciones de persistencia deben realizarse en una transacción. Sabiendo que la interfaz *Connection* tiene las operaciones *setAutoCommit(boolean)*, *commit()* y *rollback()*, modifique el diagrama de la Figura 57 para que todas las operaciones de inserción se realicen en una transacción.

6.02 Gestión de PFC

Represente con diagramas de interacción diferentes escenarios de la aplicación cuya estructura de tres capas diseñó en el ejercicio 5.03 (página 62).

CAPÍTULO 5. ANÁLISIS Y ESPECIFICACIÓN DE REQUISITOS

1. Diagramas de casos de uso

Un diagrama de casos de uso muestra las relaciones entre actores, casos de uso y, tal vez, interfaces. Un caso de uso representa un requisito funcional del sistema.

El siguiente diagrama muestra el posible diagrama de casos de uso de un cajero automático. El origen de las flechas indica quién desencadena la acción, pero no el sentido de la comunicación que, una vez establecida, puede ser bidireccional (un error muy frecuente es representar con notación de diagramas de casos de uso diagramas de flujo de datos o diagramas de estados).

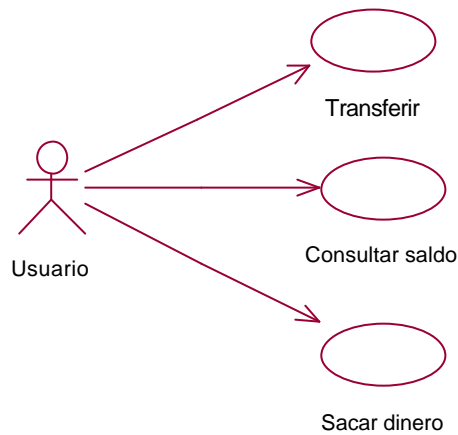


Figura 63. Un sencillo diagrama de casos de uso

Las relaciones estándares que pueden existir en los diagramas de casos de uso son las siguientes:

1) *Asociaciones*: muestran el origen de las comunicaciones entre casos de uso y actores y viceversa. En la figura anterior, hay tres asociaciones. Se representan con líneas continuas.

2) *Extensiones*: una relación de extensión desde un caso de uso A hacia otro B representa que la funcionalidad del caso B puede ser extendida por el A. Se representa con una línea punteada estereotipada «extiende» o «extends».

3) *Inclusiones*: muestran que un caso de uso contiene la funcionalidad incluida en otro caso de uso. Se representan con una flecha punteada estereotipada «incluye» o «include».

4) *Generalizaciones de casos de uso y generalizaciones de actores*: muestran que un caso de uso es una especialización de otro, o que un actor es especialización de otro. Ambos tipos de relación se representan con la flecha habitual de herencia: continua y con la punta en forma de triángulo, dirigida desde el hijo hacia el padre.

En la siguiente figura se muestra el posible diagrama de casos de uso del sistema de tarjetas de nuestra aplicación bancaria. En ocasiones, se representan los límites del sistema (es decir, lo que nosotros informatizamos) encerrando los casos de uso en un rectángulo que se etiqueta con el nombre del sistema. Se observa que en los casos de uso no se muestra el orden de ejecución de las funcionalidades: simplemente se describe *qué* cosas hace el sistema (es decir, qué requisitos funcionales sirve), pero no *cómo*.

La generalización entre los actores que aparecen a la izquierda denota que los clientes pueden realizar ciertas operaciones, pero que hay un tipo especial de clientes que pueden realizar algunas operaciones más. Los casos de uso *Transferir* y *Sacar dinero* incluyen la funcionalidad *Consultar saldo*, que es además una funcionalidad adicional que puede ejecutarse aisladamente.

A la derecha se observa el actor *BD*, que representa realmente la base de datos de nuestro banco; a pesar de que no se trata de un humano, se mantiene el símbolo del monigote porque es el icono estándar que UML propone para representar actores.

Por otro lado, cuando se paga a crédito, nuestro sistema se pone en contacto con un sistema externo, que perfectamente podríamos haber representado mediante un actor, pero que, por las razones que sean, hemos preferido representarlo con un símbolo de interfaz.

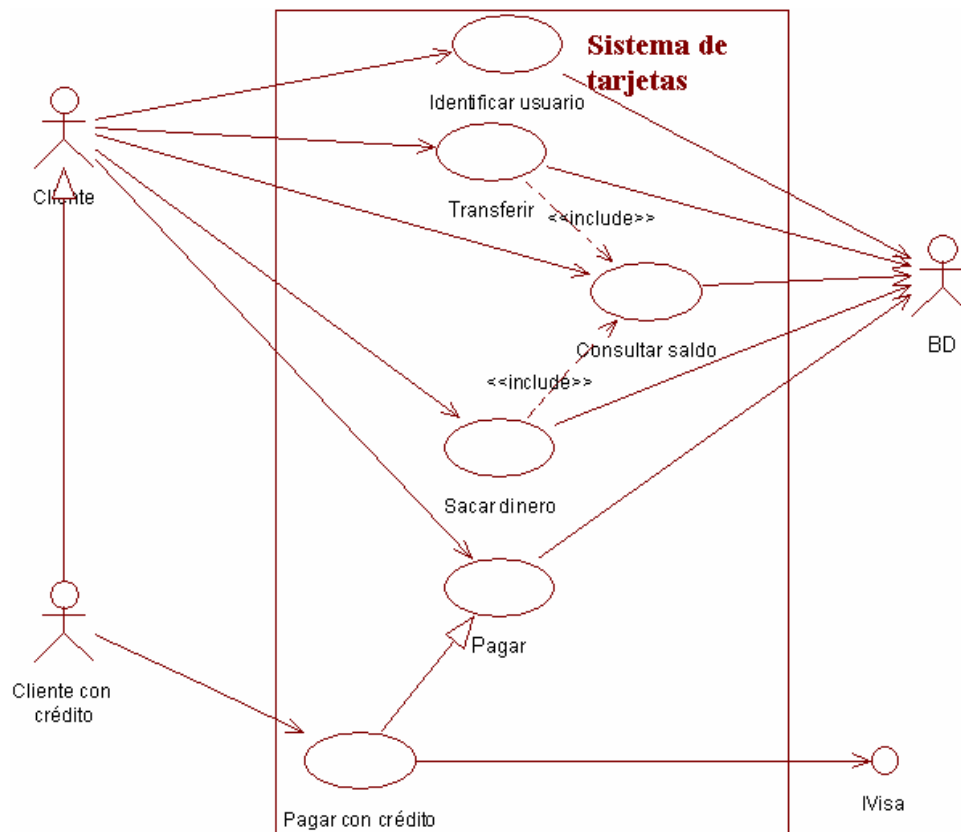


Figura 64. Un diagrama de casos de uso con varios tipos de elementos

1.01 Verificación y validación de diagramas de casos de uso

La siguiente es una lista de comprobación que nos sirve para dar por buenos o por malos diagramas de casos de uso:

- No hay casos de uso excesivamente simples
- Se utiliza la notación adecuada, especialmente para las relaciones
- No se representan “pasos” en los diagramas, del estilo de DFDs o DTEs
- Los nombres de los casos de uso son o tienen un verbo o dan idea de que realizan una funcionalidad
- Los nombres de los casos de uso y de los actores son significativos con el papel que juegan en el sistema
- Las generalizaciones identificadas son adecuadas

2. Casos de uso y diagramas de interacción

Una vez identificados los casos de uso, se deben identificar los posibles *escenarios* que se pueden dar en su ejecución. Un escenario es una situación que puede darse en la ejecución de una funcionalidad del sistema. En el diagrama anterior, posibles escenarios del caso de uso *Transferir* serían:

- 1) La transferencia se realiza con éxito (que podemos llamar *escenario normal*).
- 2) La cuenta origen no existe.
- 3) La cuenta destino no existe.
- 4) La cuenta origen no tiene saldo suficiente.
- 5) La cuenta origen está bloqueada (por orden judicial, p.ej.).
- 6) La cuenta destino está bloqueada.

Cada escenario de cada caso de uso debe detallarse con diagramas de interacción (véase capítulo 4), es decir, diagramas de secuencia o diagramas de colaboración).

3. Derivación de una estructura de clases a partir de los diagramas de interacción

Los diagramas de interacción muestran, con mayor o menor nivel de detalle, el paso de mensajes entre objetos que se produce en un determinado escenario de un caso de uso. Para que un objeto le diga algo a otro objeto, es preciso que lo conozca; y para que lo conozca, debe tener algún tipo de relación con él: una asociación, una agregación, una dependencia o una relación filial.

Por tanto, es factible ir construyendo una estructura de clases a partir de uno o más diagramas de secuencia. En el diagrama de la Figura 59, es claro que existe algún tipo de relación de *Credito* con *Movimiento*, ya que la instancia *t* le “dice cosas” a los objetos *mMovimientos* y *m*.

4. Los casos de uso en el Proceso Unificado

Como recordará el lector de la lectura del epígrafe 6 del segundo capítulo (página 42), el método de desarrollo orientado a objetos propuesto por OMT comenzaba por la construcción del diagrama de clases a partir del análisis detallado del enunciado del problema. Del Proceso Unificado, por el contrario, se dice que es “centrado en la arquitectura y dirigido por casos de uso”; lo que esto último viene a significar que los casos de uso se van desarrollando de acuerdo al *Plan de iteraciones* que se elaboró en la pri-

mera etapa (*Comienzo*, véase sección 6.01 del primer capítulo en la página 26), por lo cual el desarrollo de software orientado a objetos no comienza con la construcción de diagramas de clase.

Al contrario, según esta metodología debe comenzarse realizando una identificación de los requisitos funcionales del sistema que se debe desarrollar, de manera que se haga corresponder, más o menos, un requisito funcional con un caso de uso. Entonces, se elabora el Plan de iteraciones, en el que se indica el orden en que se desarrollarán los casos de uso (atendiendo a criterios de precondition, complejidad, urgencia, etc.).

Ya en la fase de *Elaboración*, se plantea una arquitectura base con la que montar el sistema (de tres capas, por ejemplo). Posteriormente, en la fase de *Construcción* se desarrolla cada caso de uso en el orden establecido en el Plan de iteraciones.

Cada caso de uso se elabora mediante la identificación de sus escenarios y detallando éstos con diagramas de interacción. Es entonces cuando vamos derivando la estructura de clases del sistema, tal y como hemos indicado en la sección anterior. La diferencia, entonces, con OMT, es desde luego muy notable. Una de las ventajas del Proceso Unificado es que se minimizan muchos de los riesgos de OMT: en ésta, la corrección de un error cometido al construir el modelo de clases resultaba muy costosa si el error se encontraba tarde; con el Proceso Unificado, el riesgo se limita a la pérdida del trabajo realizado en una iteración o, a lo sumo, en un caso de uso.

5. Contratos a nivel de análisis

Ya estemos tratando con diagramas de clases o con diagramas de interacción, habremos identificado un conjunto de operaciones asociadas a las clases. Para que una operación se ejecute correctamente, es muy probable que necesite un conjunto de precondiciones, de manera que la operación no se ejecutará o producirá un fallo en tiempo de ejecución en caso de que no se cumplan.

Los *contratos* son documentos que describen lo que hace una *operación del sistema*, pero no cómo (algo parecido a lo que ocurre con los diagramas de casos de uso). Una operación del sistema es una operación que el sistema ejecuta en respuesta a un *evento del sistema*, que no es ni más ni menos que un evento generado por un actor.

Entre otra información, los contratos incluyen las precondiciones necesarias para la ejecución de operaciones del sistema (es decir, el estado del sistema antes de su ejecución) las y postcondiciones (el estado después de la ejecución).

Así, por ejemplo, un posible contrato para la operación *transferir(Cuenta, double)* podría ser el siguiente:

Nombre	<i>transferir(destino:Cuenta, importe:double)</i>
Responsabilidades	Transferir el <i>importe</i> indicado desde la cuenta actual a la cuenta <i>destino</i>
Referencias cruzadas	Otras funciones: <i>consultarSaldo()</i> , <i>ingresar(float)</i> , <i>retirar(float)</i> Casos de uso: <i>Transferir</i>
Notas	Por razones de rendimiento, el saldo de la cuenta debe ser un atributo de la clase <i>Cuenta</i> , y no ser calculado buscando registros en la tabla de movimientos
Excepciones	Se solicitará confirmación cuando <i>importe</i> > 6000 €
Precondiciones	La cuenta origen debe tener saldo suficiente Ambas cuentas deben existir
Postcondiciones	El saldo de la cuenta origen es el saldo original menos el importe El saldo de la cuenta destino es el saldo original más el importe

Figura 65. Ejemplo de un contrato

5.01 Cómo hacer contratos

Larman (1998) da las siguientes recomendaciones para crear contratos:

- 1) Identificar las operaciones del sistema a partir de los eventos del sistema que aparezcan en los diagramas de interacción. Los diagramas de casos de uso también son un buen lugar para buscar operaciones del sistema.
- 2) Para cada operación del sistema, crear un contrato.
- 3) Escribir primero el nombre de la operación, y luego la sección *Responsabilidades*.
- 4) Describir a continuación las *Postcondiciones* utilizando las siguientes categorías de cambios de estado: instancias que se crean y se destruyen, atributos que se modifican, asociaciones que se establecen y se deshacen.
- 5) Completar las *Precondiciones*.
- 6) Rellenar la sección de *Notas* con alguna discusión sobre el diseño, algún algoritmo, etc.
- 7) Indicar cómo debe reaccionar el sistema el caso de situaciones excepcionales en la sección *Excepciones*.

Desde luego, puede ocurrir que consideremos que alguna de estas secciones es innecesaria y dejarla en blanco.

5.02 Especificación de precondiciones y postcondiciones

Es importante notar que las precondiciones y las postcondiciones describen el estado del sistema, por lo que no deben incluir acciones por realizar. Sería incorrecto, entonces, dejar como postcondición en la figura anterior el siguiente texto:

La cuenta origen decrementa su saldo restándole el importe

La cuenta destino incrementa su saldo sumándole el importe

Con respecto a las precondiciones, es muy fácil identificar un gran número de ellas que, sin embargo, aportarán muy poco a la descripción de la operación. En general, debemos incluir como precondiciones aquellas características que, en algún momento, vayan a ser útiles para la realización de pruebas y hallazgo de errores.

5.03 Verificación y validación de contratos

Antes de dar por buena la especificación de un contrato, deberíamos realizar las siguientes comprobaciones:

- No hay precondiciones innecesarias o inútiles
- No se han olvidado precondiciones importantes
- No se han olvidado postcondiciones importantes
- Las postcondiciones no incluyen operaciones que deban realizarse
- Se han indicado las asociaciones que se establecen y se deshacen en las postcondiciones (cuando proceda)

6. Contratos a nivel de diseño y codificación

Se pueden incorporar contratos a todas las clases o interfaces de un sistema orientado a objetos. Para ello, y dada una clase o interfaz, seguiremos los siguientes seis principios:

1) Separar comandos y consultas: un *comando* es una operación que modifica el estado de la instancia; una *consulta* es una operación que devuelve algún resultado sin modificar el estado de la instancia.

2) Separar consultas básicas y derivadas: una *consulta derivada* es aquella que puede expresarse como una o más *consultas básicas*.

3) Especificar las consultas derivadas en función de consultas básicas.

4) Especificar los comandos en función de consultas básicas.

5) Añadir precondiciones y postcondiciones en los lugares apropiados.

6) Añadir invariantes en los lugares apropiados.

6.01 Contratos y aserciones en Java

Supongamos que deseamos aplicar diseño a la construcción de una Cola, que representa el Tipo Abstracto de Datos “Cola”. Una posible interfaz para representar una cola es:

```
package dominio;

public interface ICola
{
    public void añadir(Object o);
    public void quitar();
    public int cardinal();
    public Object primero();
    public boolean esVacia();
}
```

Figura 66. Interfaz básica para una cola

En la figura anterior, serían *consultas* las operaciones *cardinal*, *primero* y *esVacia*, y comandos las operaciones *añadir* y *quitar*. Además, la operación *esVacia* puede expresarse en función de la consulta básica *cardinal* (número de elementos en la cola), por lo que la podemos considerar una *consulta derivada*.

Hecha esta distinción, podemos añadir a estas operaciones comentarios tipo *Java-doc* (<http://java.sun.com/j2se/1.4.2/docs/tooldocs/javadoc/index.html>) para representar sus precondiciones y postcondiciones. Una posible notación es la siguiente:

```
package dominio;

public interface ICola
{
    /**
     * @post size()==size()@pre+1
     */
    public void añadir(Object o);

    /**
     * @pre size()>0
     * @post size()==size()@pre-1
     */
    public void quitar();

    public int cardinal();

    public Object primero();

    /**
     * @return size()==0
     */
    public boolean esVacia();
}
```

@pre denota el valor de lo que tiene delante antes de ejecutar la operación

Figura 67. La interfaz *ICola*, con algunos comentarios especificando precondiciones y postcondiciones.

Las *consultas básicas* pueden considerarse *axiomas*, y podemos no dotarlas de precondiciones ni postcondiciones: por eso, en la Figura 67, no se han añadido comentarios a las operaciones *cardinal* y *quitar*.

A la hora de implementar la interfaz en una clase, debemos asegurar que las operaciones cumplen las precondiciones y postcondiciones especificadas en la interfaz. Al lenguaje Java 1.4 se le ha incorporado la posibilidad de utilizar aserciones para comprobar precondiciones y postcondiciones de operaciones, así como invariantes de clase. La siguiente figura muestra el código de la clase *Cola*, que implementa la interfaz *ICola*, a la que se ha añadido código de comprobación de aserciones:

```
package dominio;

import java.util.Vector;

public class Cola extends Vector implements ICola
{
    public Cola()
    {
        super();
    }

    public void añadir(Object o) throws AssertionError
    {
        int tam=size();
        super.addElement(o);
        assert cardinal()==tam+1 : "No se ha modificado el nº de elementos en la cola";
    }

    public void quitar() throws AssertionError
    {
        int tam=size();
        assert tam>0 : "No se puede quitar un elemento de una cola vacia";
        super.removeElementAt(0);
        assert cardinal()==tam-1 : "No se ha modificado el nº de elementos en la cola";
    }

    public int cardinal()
    {
        return size();
    }

    public Object primero()
    {
        return elementAt(0);
    }

    public boolean esVacia()
    {
        return cardinal()==0;
    }
}
```

Figura 68. Una implementación de la interfaz *ICola*, a la que se ha añadido código de comprobación de aserciones

De forma general, la comprobación de aserciones se realiza mediante una instrucción *assert*, que tiene uno de los dos siguientes formatos:

```
assert expresiónBooleana;
assert expresiónBooleana : expresiónConValor;
```

La *expresiónBooleana* representa la condición que se está verificando, mientras que la *expresiónConValor* representa el mensaje de error que se lanzará en caso de que la condición sea falsa. Obsérvese que las operaciones que incluyen asertos arrojan el

error *java.lang.AssertionError*, cuya ubicación en la estructura de clases de Java se muestra en la siguiente figura:

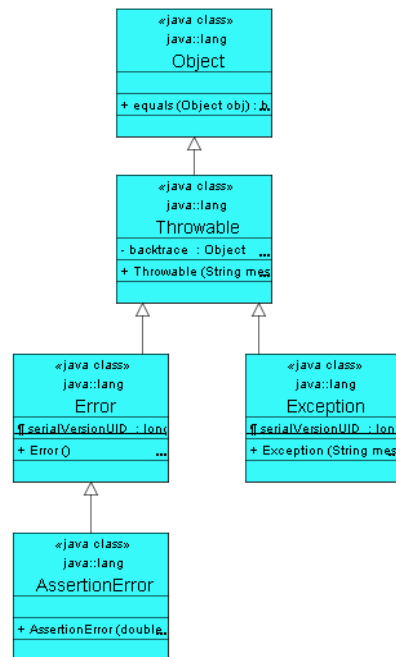


Figura 69. La *AssertionError* es un objeto *Throwable*, pero exactamente una excepción

Para dar un tratamiento adecuado a los posibles *AssertionError*'s que se puedan dar, es preciso que se escriba, en los lugares adecuados, código de captura y tratamiento de estos errores:

```

package dominio;

public class Ejecutor
{
    public static void main(String[] args)
    {
        try {
            Cola cola=new Cola();
            cola.añadir("Primero");
            cola.añadir("Segundo");
            cola.añadir("Tercero");
            cola.añadir("Cuarto");
            System.out.println(cola.primero().toString());
            cola.quitar();
            System.out.println(cola.primero().toString());
        }
        catch (AssertionError ex)
        {
            System.out.print(ex.getMessage());
        }
    }
}
  
```

Figura 70. Código de captura de errores de comprobación de aserciones

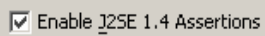
Puesto que, en caso de fallo, *assert* arroja un *AsssertionError*, no debería utilizarse esta instrucción para comprobar la validez de los argumentos de métodos públicos: esto debería resolverse con uno o más *if*, de manera que se arroje una excepción de tipo *IllegalArgumentExcepcion* y se actúe para tratar este error específico. Si se arroja un *AssertionError*, no estaremos dando un tratamiento adecuado al error que realmente se ha producido.

Para que el compilador de Java incluya el tratamiento de las instrucciones *assert* que hayamos introducido, es preciso compilar con la opción **–source 1.4**; del mismo modo, al ejecutar, debemos añadir el modificador **–ea** (o **–enableassertions**). Es decir:

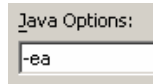
`javac –source 1.4 dominio.Cola.class` para compilar

`java –ea dominio.Cola` para ejecutar

Al entorno JDeveloper se le dice que compile aserciones marcando la opción



en las propiedades del proyecto (nodo *Compiler* del árbol del

proyecto). Para que desde el entorno ejecute aserciones, se añade  a las propiedades del proyecto (nodo *Runner* del árbol).

Por último, es importante notar que la *expresiónBooleana* puede ser tan compleja como se quiera, en el sentido de que puede consistir en la llamada a un método privado de tipo *boolean* que haga una comprobación exhaustiva de las condiciones de ejecución.

7. Ejercicios

7.01 Gestión de PFC

- Dibuje el diagrama de casos de uso de la aplicación de Gestión de PFC que se ha ido proponiendo en los ejercicios de los capítulos anteriores.
- Elabore un Plan de Iteraciones para este sistema.
- Asigne los diagramas de secuencia realizados en el ejercicio 6.02 (página 70) los casos de uso identificados en el apartado anterior.
- Identifique las operaciones del sistema y especifíquelas mediante contratos.

7.02 Ayudas por asistencia a eventos

- Construya el diagrama de casos de uso de la aplicación propuesta en el ejercicio 5.04 (página 62).
- Elabore un Plan de Iteraciones para este sistema.

- c) Identifique los escenarios de los casos de uso anteriores.
- d) Identifique las operaciones del sistema y especifíquelas mediante contratos.

7.03 Contratos y aserciones

Dado el siguiente diagrama de clases:

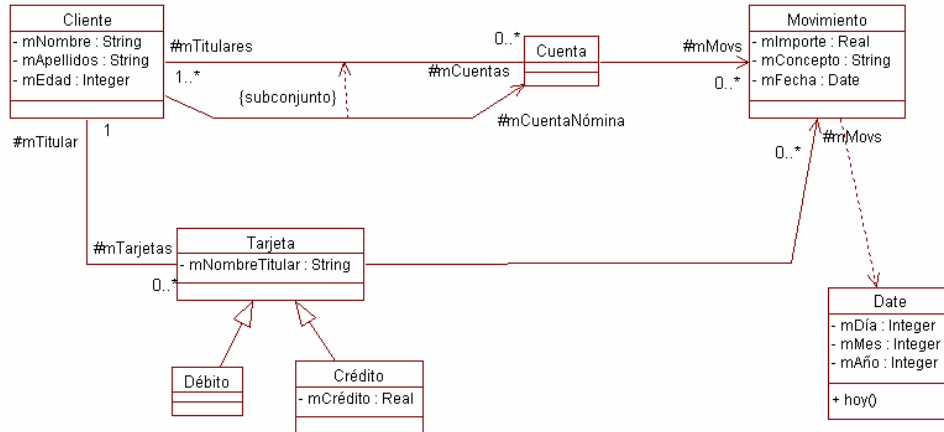


Figura 71. Un diagrama de clases

Represente en código Java el siguiente conjunto de restricciones, tanto especificando contratos como aserciones Java:

- 1) La edad de cualquier cliente es siempre mayor o igual a cero.
- 2) En toda cuenta debe haber al menos un titular de 18 años o más.
- 3) El saldo de toda cuenta debe ser siempre mayor o igual a cero (el saldo se calcula como la suma del importe de sus movimientos).
- 4) El titular de toda tarjeta de crédito debe tener la nómina domiciliada en la misma cuenta que está asociada a la tarjeta.
- 5) El nombre del titular que figura en toda tarjeta es la concatenación del nombre y apellidos del titular.
- 6) La cantidad gastada en cierto mes con una tarjeta de crédito es menor o igual al crédito de la tarjeta.
- 7) El importe que se ingresa o se retira de una cuenta es siempre positivo.
- 8) Tras hacer un ingreso, el saldo de la cuenta es el saldo que había antes más el importe ingresado.
- 9) Tras retirar de una cuenta, el saldo de la cuenta es el saldo que había antes menos el importe ingresado.

CAPÍTULO 5 BIS.

RECAPITULACIÓN

8. Introducción

Hasta este momento, en estos apuntes se han estudiado, de manera aparentemente desorganizada, ciertos aspectos del desarrollo orientados a objetos, como la elaboración de diagramas de clases, algo de la arquitectura multicapa, de requisitos, de diagramas de casos de uso o de interacción. Ahora debemos poner en orden todos estos elementos de modelado e indicar en qué momentos del proceso deben elaborarse cada uno de ellos.

En este capítulo haremos un breve resumen de estas cosas, teniendo en cuenta lo que hasta ahora se ha visto en estos apuntes. Información más detallada puede obtenerse del libro “El Proceso Unificado de Desarrollo de Software”, de Jacobson, Booch y Rumbaugh (editorial Addison Wesley).

9. El Proceso Unificado, de un vistazo

La aplicación del Proceso Unificado al desarrollo de un proyecto supone la realización de una serie de *ciclos*, tras cada uno de los cuales se obtiene una *versión* nueva del producto, susceptible de ser entregada al cliente. Cada *versión* incluye código fuente, manuales de usuario y otros posibles artefactos. Un *ciclo* se descompone en *fases* (las ya conocidas de *Comienzo*, *Elaboración*, *Construcción* y *Transición*, véase pág. 27), y cada *fase* consta de varias *iteraciones*. La versión final del producto (el *producto terminado*) incluirá, además los requisitos, casos de uso, especificaciones no funcionales y casos de prueba. La Figura 72 representa estos comentarios de forma gráfica utilizando notación UML:

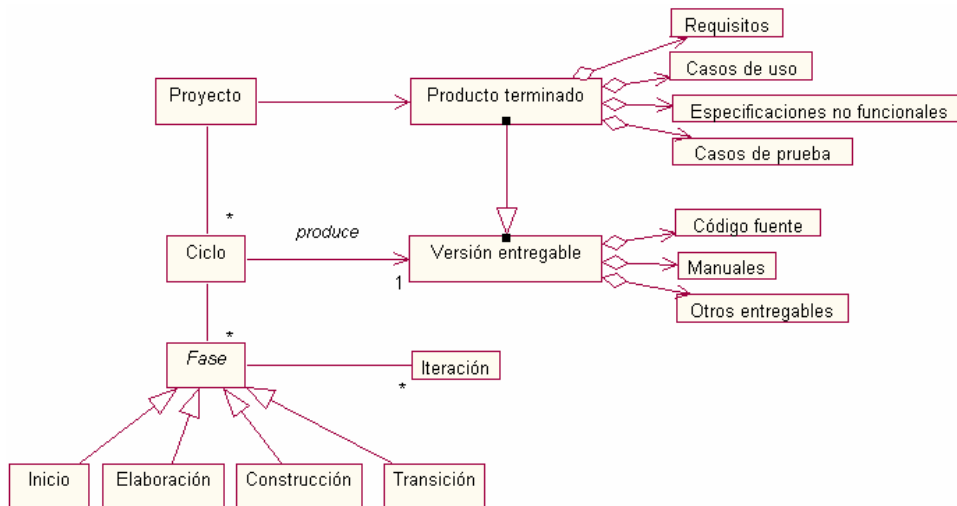


Figura 72. Estructura (muy general) del Proceso Unificado de Desarrollo

Además de los artefactos incluidos en las versiones entregables y en el producto terminado, durante el desarrollo del producto se construyen muchos otros tipos de elementos, los cuales representan el mismo sistema, pero desde diferentes puntos de vista:

1) De los requisitos obtenemos uno o más diagramas de casos de uso, que representan el sistema entero desde el punto de vista de las funcionalidades solicitadas por el usuario.

2) El diagrama de casos de uso (que representa, insistimos, el sistema entero) se *especifica* mediante un *modelo de análisis*.

3) El diagrama de casos de uso se *realiza* mediante un *modelo de diseño*.

4) El sistema (que podemos representar mediante el diagrama de casos de uso) se *distribuye* (o se instala en máquinas, ordenadores, etc.) a partir de un *diagrama de despliegue*.

5) El sistema se *implementa* mediante un *diagrama de componentes*.

6) El sistema se *prueba* mediante un *modelo de prueba*.

En la enumeración anterior aparecen varios términos que pueden resultar nuevos. La siguiente tabla explica brevemente su significado:

Término	Descripción
Modelo de análisis	Cumple dos propósitos: por un lado, explica los casos de uso con más detalle; por otro, asigna de manera inicial las diferentes funcionalidades del sistema a un conjunto de clases
Modelo de diseño	Representa la estructura estática del sistema en forma de subsistemas, clases e interfaces. Básicamente, consistiría en el conjunto de diagramas de clases y de paquetes, tal y como los hemos visto hasta ahora.
Diagrama de componentes	Como se verá más adelante, un <i>componente</i> viene a corresponderse con un fichero físico (más o menos, un fichero de disco). Un diagrama de componentes muestra qué ficheros contiene el sistema, así como qué clases se corresponden con qué componentes.
Diagrama de despliegue	Representa el conjunto de nodos (ordenadores) en los que se ejecutará la aplicación, las relaciones entre estos nodos y las relaciones entre los diferentes componentes y los nodos
Modelo de prueba	Especifica los casos de prueba que verifican los casos de uso: si entendemos los casos de uso como requisitos solicitados por el cliente, en este modelo de prueba se incluirán casos de prueba que sirvan para comprobar que las funcionalidades se ejecutan de manera correcta

Tabla 1. Descripción de algunos términos

De manera ideal, debería ser posible realizar un seguimiento desde un modelo a otro: en particular, desde los requisitos deberíamos poder ir a los casos de uso, de éstos al modelo de análisis, de aquí del modelo de diseño, de aquí al de implementación, y de aquí al de despliegue. Si esto se consigue, no sería difícil conocer qué componente, clase, método, ordenador... está ejecutando una funcionalidad determinada.

10. El modelo de análisis

El modelo de análisis se construye a partir del diagrama de casos de uso: para cada caso de uso identificamos, de manera preliminar, qué posibles clases van a intervenir en su realización.

Por ejemplo, a partir del siguiente sencillo diagrama, que representa una pequeña funcionalidad de un cajero automático...



Figura 73

...podríamos deducir que, para su ejecución, va a ser necesario disponer de una ventana que actúe de interfaz con el usuario, una clase que se ocupe de verificar la identidad del usuario y otra que actúe sobre la cuenta, modificando su saldo. En otras palabras o, mejor dicho, gráficamente:

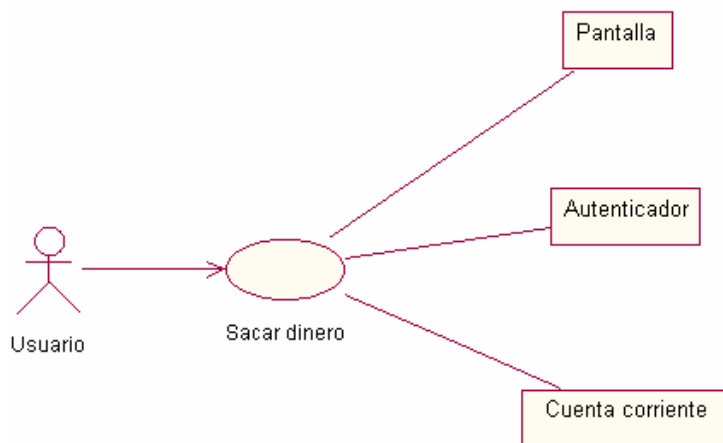


Figura 74. Clases que intervienen en la realización del caso de uso

UML dispone de una notación gráfica específica para representar clases de análisis, que son las tres que se muestran en la figura anterior:

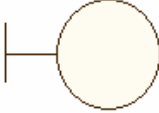
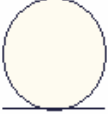

Símbolo	Significado
	<i>Interfaz</i> : se utiliza para representar aquellos objetos que se van a comunicar con el entorno, que van a recibir estímulos y a enviar resultados a elementos situados fuera de los límites de nuestro sistema.
	<i>Entidad</i> : denotan objetos que perviven. Suelen ser los objetos persistentes de la capa de dominio.
	<i>Control</i> : son objetos efímeros. Suele haber un objeto de control por cada caso de uso, aunque puede haber más de uno. Se les suelen asignar responsabilidades de control de transacciones, de secuencias, de aislamiento de objetos entidad, etc.

Figura 75. Notación para representar clases de análisis

Con esta notación, podemos redibujar la Figura 74 de la siguiente manera:

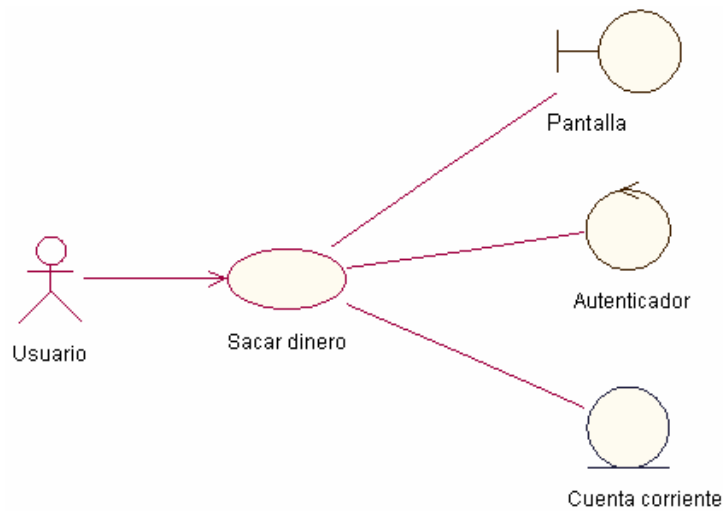


Figura 76. La Figura 74, pero usando ahora notación específica de clases de análisis

Si lo preferimos, podemos representar la misma figura utilizando estereotipos en lugar de iconos:

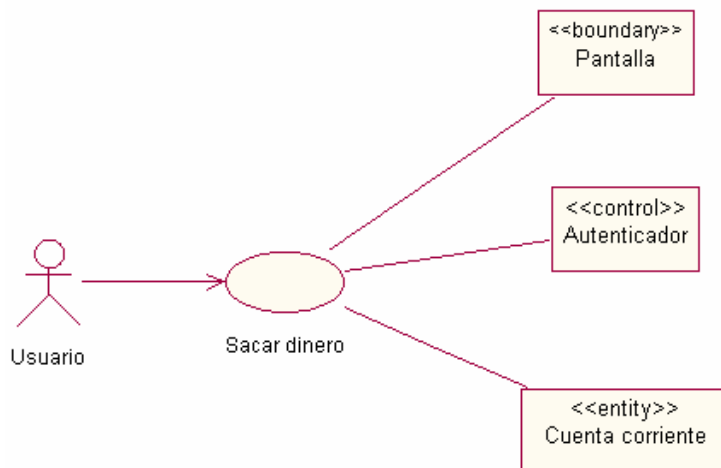


Figura 77. La Figura 74, pero usando estereotipos en lugar de iconos

También puede precisarse más en qué forma intervienen las clases de análisis anteriores en la realización del caso de uso: en la figura siguiente estamos diciendo que el usuario interactúa con la *Pantalla* y que ésta utiliza el *Autenticador* y la *Cuenta corriente*.

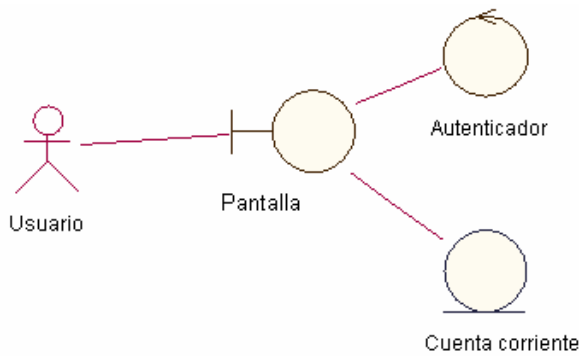


Figura 78. Descripción muy general de cómo colaboran las diferentes clases para realizar el caso de uso *Sacar dinero*

La descripción de la figura anterior es desde luego muy genérica (en parte debido a que estamos en la etapa de análisis), por lo que debemos completarla con algún tipo de diagrama de interacción (de secuencia, por ejemplo) que ya incluya paso de mensajes, de modo que se describa cómo intervienen las instancias de estas clases para servir la funcionalidad.

11. Modelo de diseño

El modelo de diseño se construye a partir del modelo de análisis. Como ya mencionamos, es deseable mantener la posibilidad de “trazar” o seguir la pista de todos los artefactos que se van produciendo. Es importante, entonces, que los elementos que añadamos a este modelo procedan de verdad del modelo análisis.

Para ello, y ya con ciertas restricciones por el entorno de distribución, iremos derivando clases de diseño a partir de las clases de análisis, como se ve en la Figura 79. Hemos estereotipado las relaciones con el estereotipo *traza* para denotar qué clases de diseño proceden de qué clases de análisis.

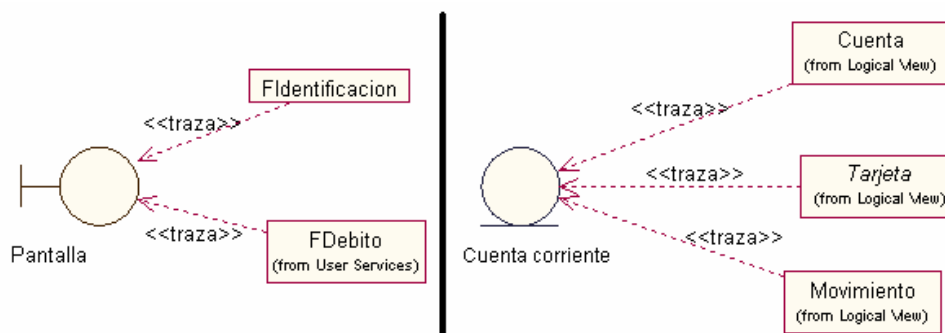


Figura 79. Derivación de clases de diseño a partir de clases de análisis

También podemos ir construyendo un diagrama que muestre cómo interactúan los actores no ya con las clases de análisis (como hacíamos en la Figura 78), sino con las clases de diseño:

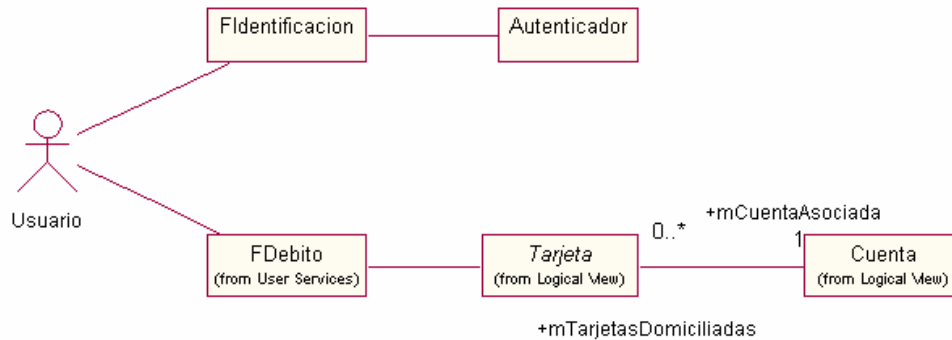


Figura 80. Descripción muy general de las relaciones entre actores y clases de diseño, y entre clases de diseño y clases de diseño

A partir de todos estos diagramas que vamos mencionando, iremos identificando escenarios y los detallaremos mediante diagramas de interacción. Con éstos vamos enriqueciendo progresivamente las clases, a las que añadimos operaciones y atributos (entre los que se encuentran relaciones con otras clases que previamente no hayamos identificado), de modo que podremos ir generando código. Cuando se haya finalizado un conjunto de iteraciones que constituyan un *ciclo* (se reproduce de nuevo la Figura 72), dispondremos de una *versión entregable*.

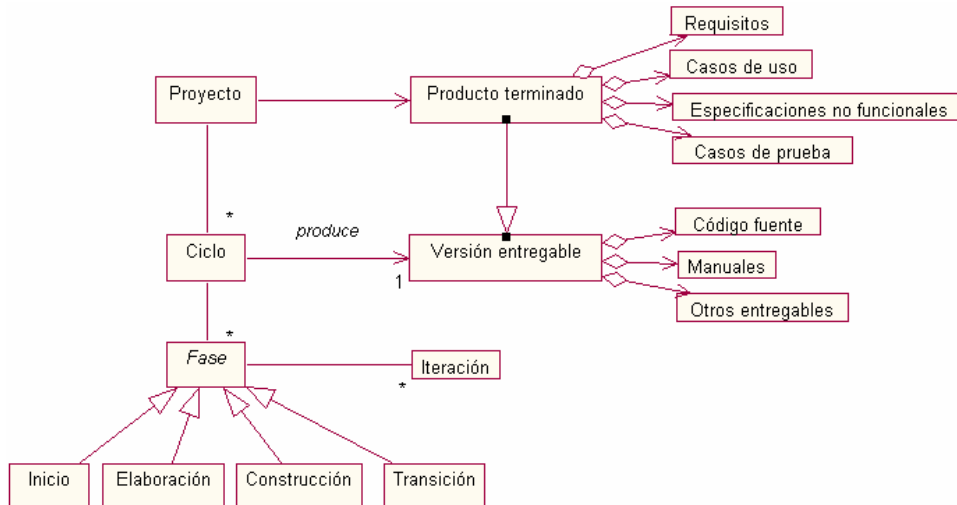


Figura 81. Estructura (muy general) del Proceso Unificado de Desarrollo

CAPÍTULO 6. ARQUITECTURA MULTICAPA (II)

1. Introducción

En el capítulo 4 hemos visto, fundamentalmente, algunos patrones para relacionar la capa de Dominio con la capa de Persistencia. Es importante también mantener el bajo acoplamiento de las clases de Dominio con la de Presentación. En este capítulo veremos patrones para que los objetos de Dominio puedan comunicar, sin acoplarse, sus cambios de estado a los de Presentación, así como algunas estrategias para diseñar la capa de Persistencia.

2. El patrón Modelo-Vista-Controlador

La solución que nos da este patrón (Buschmann et al., 1996) para desacoplar la capa de Dominio de la de Presentación es crear una clase intermediaria (un *Observador*) que recibe las notificaciones de los cambios de estado de los objetos de la capa de Dominio y los transmite a los de Presentación.

La estructura de la solución más burda para comunicar ambas capas es la siguiente:

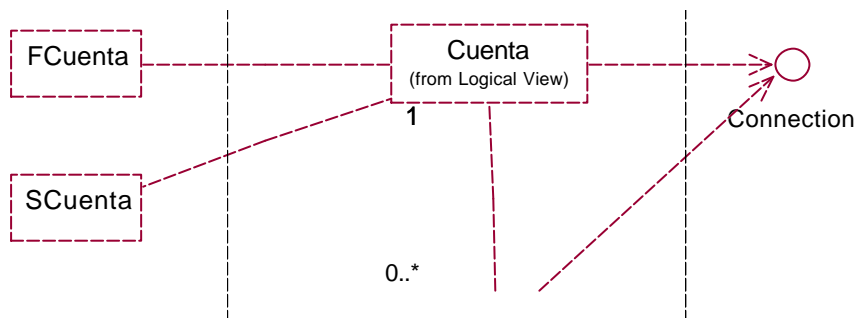


Figura 82. Una mala solución para comunicar Dominio con Presentación

En la figura anterior, observamos que pueden manipularse cuentas desde dos tipos de ventanas, que pueden corresponderse con *Frames* y con *Servlets*. Supongamos un escenario en el que dos titulares de la misma cuenta están trabajando con ella simultáneamente. En un mismo intervalo de tiempo, uno de ellos realiza una consulta de saldo y el otro hace una transferencia. Se desea que el sistema actualice automáticamente su estado en las dos ventanas que le apuntan. Con la estructura de clases de la figura anterior, el funcionamiento podría ser el siguiente:

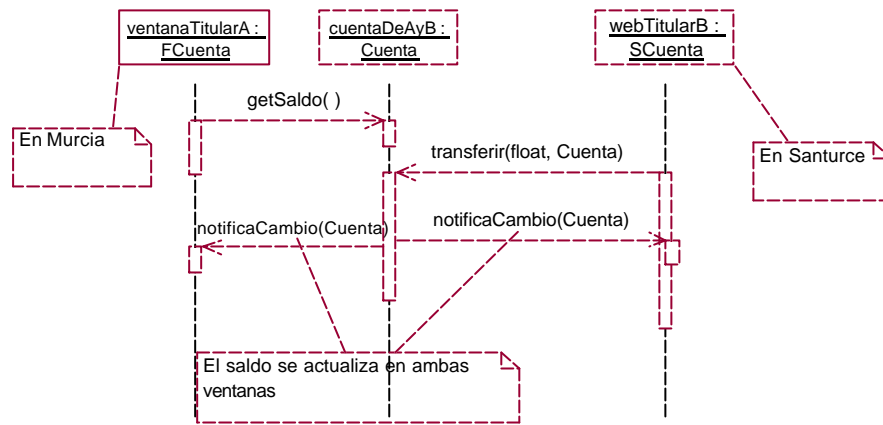


Figura 83. Actualización de las dos ventanas que “miran” a un objeto

Esa solución es mala porque la clase *Cuenta* debe mantener relaciones con las dos ventanas a las que conoce. La clase *Cuenta* podría ser utilizada en aplicaciones en las que no se necesiten estas ventanas; sin embargo, aunque sólo sea para compilar, dichas clases tendrían que estar presentes. Por otro lado, la adición de un tipo de ventana nuevo necesitaría la modificación de *Cuenta*, lo que no es deseable.

Una posible solución es la creación de una clase más simple a la que *Cuenta* puede conocer; cuando *Cuenta* cambia su estado, lo notifica a esta clase observadora (llamémosla *ObCuenta*), que a su vez lo notifica a todas las ventanas que apunten a la *Cuenta*. Gráficamente:

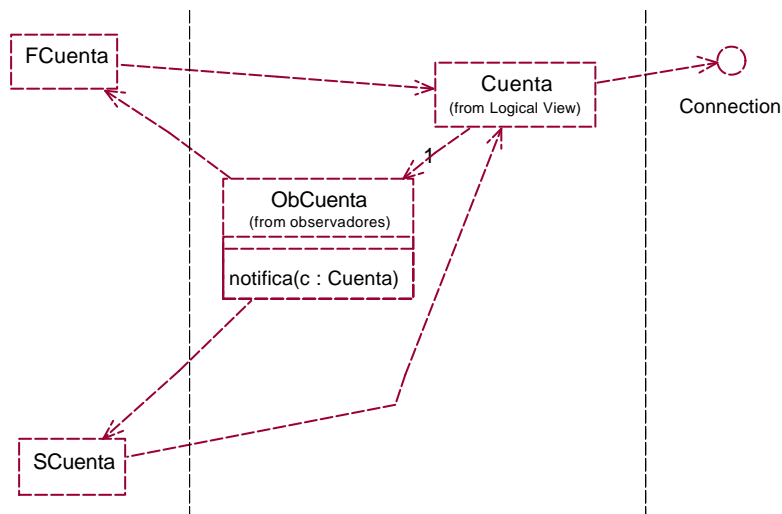


Figura 84. Una clase sencilla conoce a las clases interesadas de la capa de Presentación

El mismo escenario que en la Figura 83 pero con esta última solución sería el siguiente:

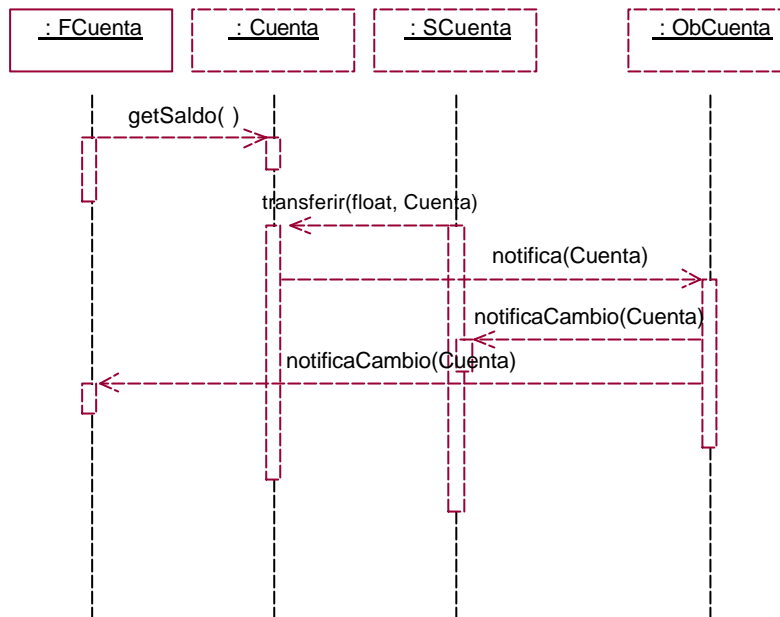


Figura 85. Actualización de las dos ventanas mediante la solución de la Figura 84

Esta solución decrementa el acoplamiento y consigue más cohesión, ya que alguna de las cosas que antes hacía *Cuenta* y que en verdad no le corresponden (actualizar ventanas), pasa a hacerlas una clase nueva que se dedica exclusivamente a eso.

De todos modos, algunos de los problemas de la primera solución (Figura 82) siguen existiendo, aunque tendrían menos importancia ya que *ObCuenta* es “menos importante” y tiene menos complejidad que *Cuenta*. Así, la adición de un tipo de ventana nuevo no obligaría a tocar *Cuenta*, sino sólo *ObCuenta*.

Podemos acercarnos a la situación ideal, en la que no haya que tocar nada, si nos fijamos en que ambas ventanas responden exactamente al mismo mensaje, *notificaCambio(Cuenta)*, aunque cada una lo implementa de manera diferente. En efecto, podemos crear una interfaz *IVentanaCuenta* que contenga operación, y que sea implementada por ambas ventanas:

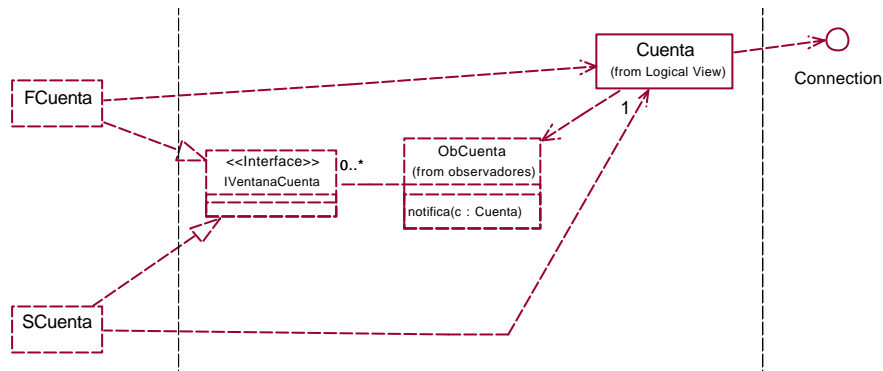


Figura 86. Una solución mucho mejor

En la Figura 86, la adición de un nuevo tipo de pantalla no supone cambio alguno en *Cuenta*. Obsérvese que *ObCuenta* puede conocer a muchas instancias de tipo *IVentanaCuenta*, con lo que puede comunicar el cambio de estado de *Cuenta* a múltiples ventanas, sin importarle de qué tipo sean:

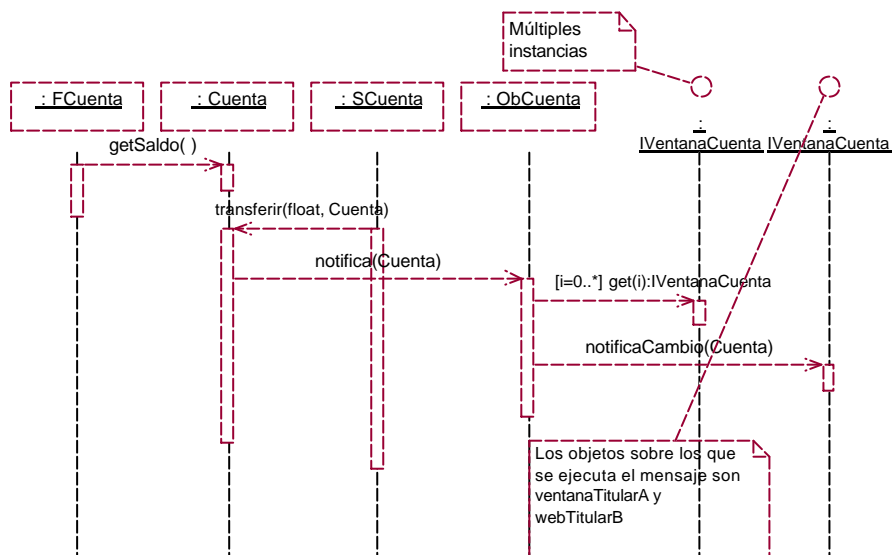


Figura 87. Notificación a múltiples ventanas, sin importarnos el tipo de ventana

Dependiendo del ingeniero de software, es posible que se considere innecesaria la presencia de *ObCuenta*, ya que tiene asignadas muy pocas responsabilidades. En este caso, podríamos conectar directamente *Cuenta* a las interfaces, debiendo asumir *Cuenta* la responsabilidad de notificar sus cambios a las ventanas que le apuntan.

2.01 El patrón Observador

En los casos en que hay un único objeto encargado de comunicar cambios de estado a múltiples objetos (como en la Figura 85), se habla también del patrón *Observador* (Gamma et al., 1996) o *Productor-Consumidor* (Buschmann et al., 1996).

Todos los objetos que quieren ser notificados de un evento se deben *suscribir* al observador, que tendrá una operación del tipo *suscribir(Object)* o, en nuestro ejemplo, *suscribir(IVentanaCuenta)*; cuando se produce el cambio de estado, el observador lo publica mediante un método *publicar(LoQueSea)*.

2.02 Otros usos de los observadores

Evidentemente, el uso de observadores no está relegado a comunicar la capa de Dominio con la de Presentación. Siempre que la modificación del estado de un objeto deba ser comunicada a otros objetos, el uso de un observador es una solución recomendada.

3. Ejercicios

3.01 Escritura de código para conectar el observador y el observado (I)

Partiendo del diagrama de la Figura 84, escriba el código para que los cambios de estado en el objeto *Cuenta* se comuniquen a *FCuenta*.

3.02 Escritura de código para conectar el observador y el observado (II)

Partiendo del diagrama de la Figura 86, escriba el código para que los cambios de estado en el objeto *Cuenta* se comuniquen a *FCuenta*.

3.03 La propia pantalla actúa de observador

Suponga, en la siguiente figura, que cada cambio de estado en una instancia de *Cuenta* se va a transmitir sólo a una ventana que lo observa. ¿Sería posible eliminar la clase *ObCuenta* y hacer que la propia ventana sea la observadora?

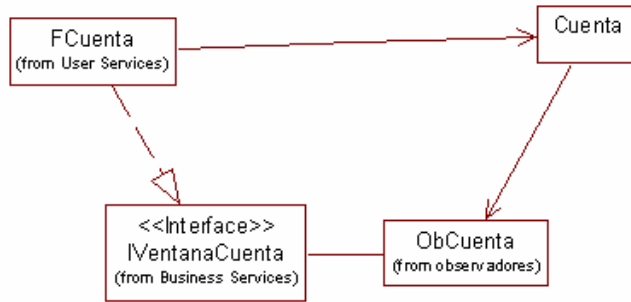


Figura 88. ¿Es posible eliminar la clase *ObCuenta*?

4. Cachés de objetos

Una caché es un mecanismo que almacena en memoria datos de uso frecuente. En nuestro caso, guardaremos en cachés algunas de las instancias que vayamos materializando, con el fin de mejorar el rendimiento de la aplicación.

Cuando se desea materializar una instancia, se mira primero en una caché de objetos creados si la instancia ya ha sido materializada. En caso afirmativo, en lugar de acceder físicamente a la base de datos, se devuelve una referencia a la instancia, que se recupera de la caché.

Al utilizar cachés, el diseño del sistema cambia, ya que el Agente, que encamina las operaciones hacia la base de datos, debe buscar el objeto que se quiere instanciar en la caché. Un posible diseño alternativo delegaría la responsabilidad de buscar en la caché a cada clase persistente o a una Fabricación Pura; sin embargo, de acuerdo con los patrones Alta Cohesión y Bajo Acoplamiento, es más recomendable que el propio Agente sea responsable de buscar en la caché.

La Figura 89 muestra el uso de una caché simple para almacenar instancias de *Cuenta*. A la clase que implementa la caché se la puede estereotipar como “tabla hash” para indicar que los objetos son almacenados en una tabla hash, con el objetivo de acelerar las búsquedas de objetos.

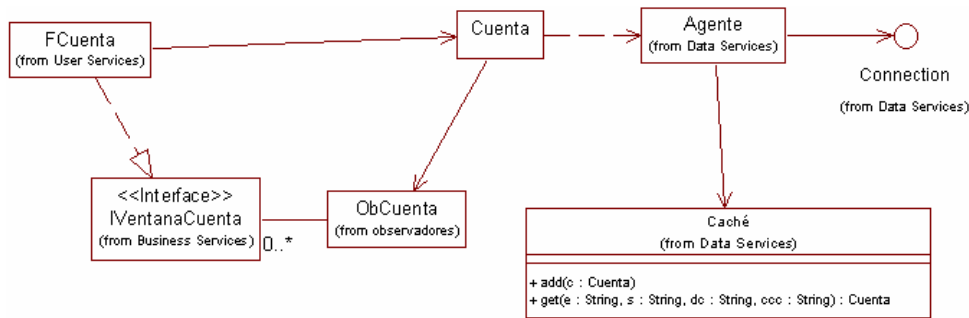


Figura 89. Una caché sencilla para cuentas

Con la solución mostrada en la figura anterior, la caché sólo serviría para almacenar objetos de clase *Cuenta*, cuando lo ideal es que podamos almacenar objetos de cualquier clase persistente. Esto podremos conseguirlo al menos de dos maneras:

- Añadiendo a la clase Caché tantos atributos de tipo *TablaHash* como clases de objetos puedan almacenarse en ella.
- Generalizando todas las clases persistentes mediante el uso de una superclase (*RCRUD*, *Persistente*, etc.) o haciendo que implementen una interfaz específica (*IPersistente*, p. ej.).

En principio, esta segunda solución parece bastante más lógica, ya que debemos gestionar exclusivamente una única estructura de datos. Sin embargo, si el número de registros materializados es muy grande, el número de colisiones de la tabla hash puede afectar negativamente al rendimiento del sistema. Desde luego, optar por la utilización de un Vector, array u otra estructura de datos lineal y no ordenada no estaría recomendado en ningún caso por el coste de las búsquedas de objetos.

Gracias a que Java incorpora la clase *TablaHash*, el código Java de la clase Caché es muy sencillo:

```

import Dominio.*;

public class Cache
{
    public static Hashtable mObjetos;

    Cache() {
        mObjetos=new Hashtable();
    }

    public static void add(Cuenta c) {
        c.incrementaInstancias();
        mObjetos.put(c.getEntidad()+c.getSucursal()+c.getDC()+c.getCCC(), c);
    }

    public static Cuenta get(String e, String s, String dc, String ccc) {
        return (Cuenta) mObjetos.get(e+s+dc+ccc);
    }
}

```

```
}
}
```

Figura 90. Código de la clase Caché

4.01 Compartición de instancias

Hasta ahora, cuando desde dos ventanas se quería mostrar la información de un mismo registro de la base de datos, se creaban en memoria dos instancias diferentes de la misma clase: por ejemplo, si un titular de una cuenta se conecta mediante Internet a su *Cuenta*, y un empleado está viendo la información de la misma cuenta desde la sucursal bancaria, realmente están observando dos instancias diferentes de la clase, aunque ambas hayan sido materializadas desde los mismos datos de la base de datos.

El uso de cachés soluciona este problema, al permitir, por ejemplo, que varias pantallas apunten a la misma instancia.

El siguiente código es un fragmento de la clase “AgenteConCache”, resuelve parcialmente este problema: cuando una Cuenta desea materializarse, se lo dice al agente, que ejecuta el método *materializaCuenta(String e, String s, String dc, String ccc)*. El agente busca la instancia solicitada en la caché:

- En caso de que exista, devuelve una referencia a ella.
- En caso de que no exista, la recupera de la base de datos, le crea un observador y se lo pone. Acto seguido, añade la instancia a la caché y devuelve su referencia.

La siguiente figura muestra el posible código de la clase AgenteConCache, que representa un Agente de base de datos con una caché para cuentas.

```
import Dominio.Empleado;
import ServPres.Observador;

public class AgenteConCache
{
    static Connection mBD;
    static Cache mCache;

    public AgenteConCache(Connection bd) throws Exception {
        mBD=bd;
        mCache=new Cache();
    }

    public static Cuenta get(String e, String s, String dc, String ccc) throws Exception {
        Cuenta c=mCache.get(e+s+dc+ccc);
```

```

    if (c!=null) {
        c.incrementaInstancias();
        return c;
    } else {
        String SQL="Select * from Cuenta where Entidad=? and ...";
        PreparedStatement p=mBD.prepareStatement(SQL);
        p.setString(1, e);
        ...
        ResultSet r=p.executeQuery();
        if (r.next()) {
            c=new Cuenta();
            c.setEntidad(r.getObject(1).toString());
            ...
            ObCuenta o=new ObCuenta();
            c.setObservador(o);
            mCache.add(c);
        } else throw new Exception("No existe ninguna cuenta con esos datos");
    }
    return c;
}
.....

```

Figura 91. Fragmento del código de la clase AgenteConCaché

El código del constructor materializador de la clase *Cuenta* podría ser éste:

```

public Cuenta(String e, String s, String dc, String ccc) throws Exception {
    Cuenta c=AgenteConCache.get(s, e, dc, ccc);
    mPKCuenta=e.mPKCuenta;
    ...
    this.setObservador(c.getObservador());
}

```

Figura 92. El constructor materializador de la clase Empleado.

Como se ha dicho en el párrafo anterior, el AgenteConCaché devuelve la referencia al objeto Cuenta, que tal vez acaba de ser materializado (si no residía en la caché) o tal vez no (si ya residía):

- Si acaba de ser materializado, el AgenteConCaché le habrá asignado un observador.
- Si simplemente se ha devuelto una referencia al objeto que residía en la caché, ésta ya tenía observador, y lo único que tenemos que hacer es asignárselo, que es lo que se hace en la última instrucción de la Figura 92.

Tras ejecutarse el código de la Figura 92, lo siguiente que tenemos que hacer es que la pantalla que va a mostrar la información de la instancia de clase *Cuenta* recién creada (o recuperada de la caché) se suscriba al observador de la instancia. Esto se muestra en la última instrucción de la Figura 93.

```

public FrameCuenta(String title, Cuenta c) {
    this(title);
    mCuenta=c;
    textFieldEntidad.setText(e.getEntidad());
    textFieldSucursal.setText(e.getSucursal());
    ...
    setVisible(true);
    Operaciones.habilita(this, false);
    e.getObservador().suscribir(this);
}

```

Figura 93

El observador notifica a objetos que pueden ser frames, diálogos o, por qué no, objetos no visuales. Dado que notifica a una variedad de objetos que puede ser muy amplia, es bueno que el observador no dependa del tipo de objeto al que tiene que notificar. Esto podemos conseguirlo, como ya vimos al principio de este capítulo, diciendo que el observador notifica a objetos genéricos (interfaces o clases abstractas) de tipo `IObservable`, `INotificable` u otro nombre con algo de coherencia que se nos pueda ocurrir. La Figura 94 muestra la cabecera de dos clases distintas de la capa de Presentación, a las que el observador genérico puede notificar: un `Frame` y un `Dialog`.

```

public class FrameCuenta extends Frame implements IObservable {
    ....
}

public class DialogCuenta extends Frame implements IObservable {
    ....
}

```

Figura 94. Código del observador.

4.02 Tipos de cachés

De acuerdo con Larman (1999), en nuestro sistema puede haber, al menos, hasta seis cachés diferentes:

- *New Clean Caché*: en esta caché se almacenan los objetos “nuevos y limpios”; es decir, aquellos objetos que se han creado durante esta sesión (no se han materializado desde la base de datos) y que aún no han sido modificados.
- *Old Clean Caché*: se guardan los objetos “viejos y limpios”, que son los objetos que se han materializado desde la base de datos pero que aún no han sido modificados.

- *New Dirty Caché*: guarda objetos “nuevos y sucios”, que son aquellos que se han materializado en esta sesión (no se han materializado desde la base de datos) y que han sido modificados.
- *Old Dirty Caché*: guarda objetos “viejos y sucios”, que son los objetos que se han materializado desde la base de datos y que han sido modificados en esta sesión.
- *New Delete Caché*: guarda los objetos creados en esta sesión (no se han materializado desde la base de datos) y que han sido borrados.
- *Old Delete Caché*: guarda los objetos materializados desde la base de datos y que han sido borrados.

5. Ejercicios

5.01 Algoritmos de gestión de cachés (I)

Suponga un sistema en el que se dispone de una caché con capacidad para almacenar n instancias de cada clase. Proponga un diseño de clases para la capa de persistencia suponiendo que se desea gestionar esta caché usando los algoritmos:

- LRU (*Least Recently Used*): se elimina de la caché el objeto usado menos recientemente.
- FIFO (*First-In, First-Out*): se elimina de la caché el objeto que lleve más tiempo en ella, independientemente del momento en que se accedió a él por última vez.
- LIFO (*Last-In, First-Out*): se elimina el objeto que lleve menos tiempo en la caché, independientemente del momento en que se accedió a él por última vez.
- Otros algoritmos.

5.02 Algoritmos de gestión de cachés (II)

En el ejercicio anterior, represente el comportamiento del sistema mediante diagramas de secuencia.

CAPÍTULO 7. ACCESO A INSTANCIAS REMOTAS

1. Introducción

En la Sección 4.01 del capítulo anterior vimos cómo se podían utilizar cachés para facilitar la compartición de instancias entre diversas aplicaciones: es decir, conseguir que dos usuarios que están trabajando con, por ejemplo, la misma cuenta corriente de la base de datos, estén también trabajando con la misma instancia del clase *Cuenta*. El asunto no es muy complicado si ambos usuarios –cosa difícil– están trabajando en la misma máquina, pero se vuelve un poco más engorroso si, como es habitual, se encuentran en máquinas separadas. De esto nos ocuparemos en este capítulo, así como del almacenamiento de instancias mediante *Serialización*. Como veremos, la serialización y la compartición de instancias remotas nos ayudará a comprender la tecnología de componentes de los *Enterprise Java Beans* (EJB).

2. Compartición de instancias remotas

La siguiente figura muestra parte de la estructura de clases de nuestro sistema bancario. De ella se deduce que cada ventana (sea de clase *FCuenta* o *SCuenta*) conoce a un objeto de tipo *Cuenta*.

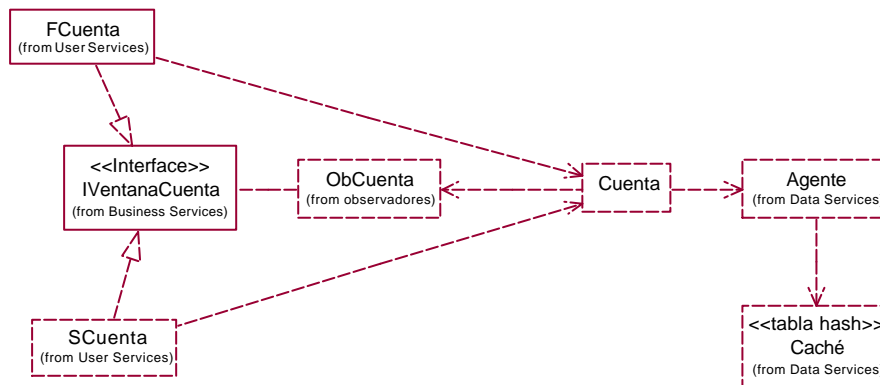


Figura 95. Estructura (parcial) de clases del sistema bancario, que no permite compartir instancias remotas

Este modelo de clases es válido para una aplicación que corre en una sola máquina, ya que permite compartir entre diferentes ventanas las mismas instancias de las clases de dominio. Sin embargo, si las instancias de las clases de dominio residen en una máquina diferente de las ventanas, éstas no pueden mantener la referencia a la instancia de la clase de dominio y se hace necesario el uso de alguna argucia que permita, por un lado, mantener la referencia remota y, por otro, compartir la instancia entre diferentes ventanas.

En este caso, nos enfrentamos realmente al desarrollo de dos aplicaciones diferentes: la aplicación cliente (que corre en una máquina) y la aplicación servidora (que corre en una máquina diferente). En nuestro ejemplo, habrá dos versiones diferentes de la aplicación cliente (en una se usan pantallas de clase *FCuenta* y en otra de clase *SCuenta*) y una sola versión de la aplicación servidora. Para que las instancias de *FCuenta* y de *SCuenta* puedan conocer a una instancia remota de *Cuenta* (que, además, a veces van a compartir), es necesario conectar ambas pantallas no a un objeto *Cuenta*, sino a un interfaz que ofrezca los servicios solicitados por ambas pantallas.

Dicho de otro modo: supongamos que los usuarios de las dos aplicaciones cliente que vamos a tener ejecutarán, sobre la instancia de clase *Cuenta* asociada, las operaciones *consultarSaldo():float* y *transferir(Cuenta destino, float importe)*. Lo que haremos será conectar ambas pantallas a una interfaz *ICuenta* en la que se habrán declarado las dos operaciones mencionadas. Esta interfaz será implementada por la clase *Cuenta* que, ojo, reside en otra máquina.

Con estas consideraciones, el diseño de este sistema iría siendo de este estilo:

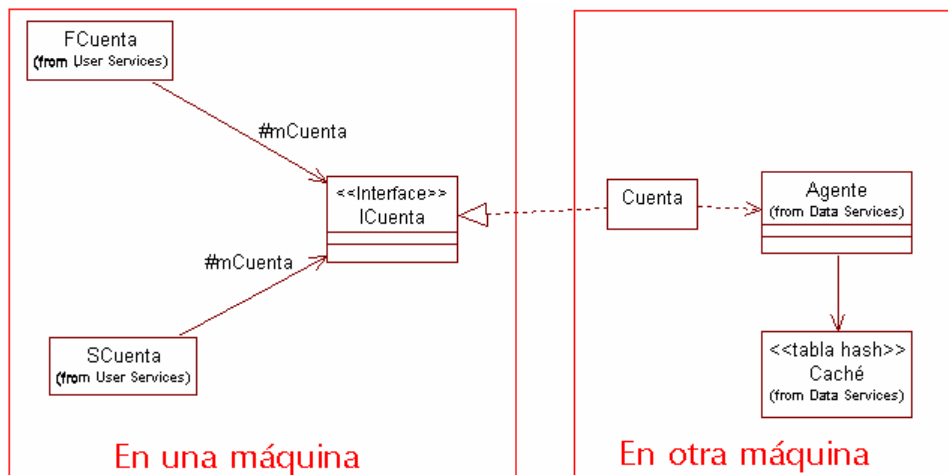


Figura 96. Modificación del diseño de la Figura 95

La interfaz *ICuenta* es una interfaz *remota*, que debe ser conocida tanto por las clases de la aplicación cliente como por la aplicación servidora. Los objetos de la clase *Cuenta* son objetos a los que se accede también de forma *remota*. En Java, esto se denota de la siguiente forma:

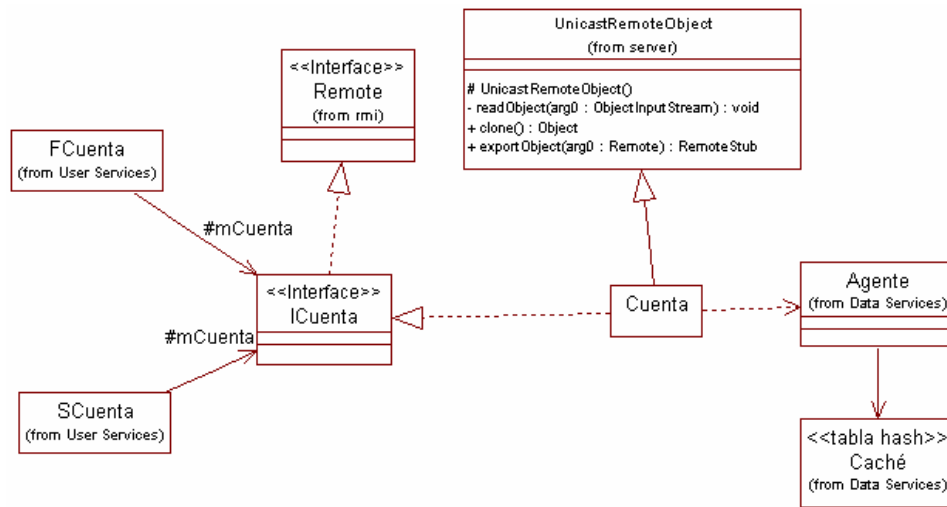
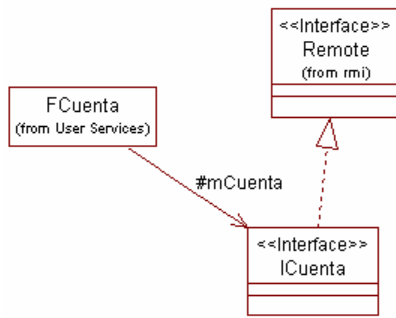
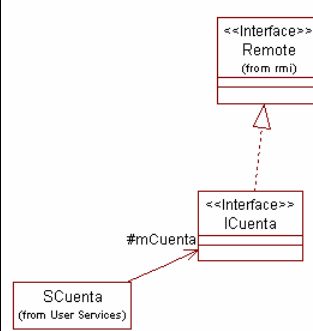


Figura 97. Diseño de la Figura 96, pero explicitando ahora el carácter remoto de la interfaz *ICuenta* y de la clase *Cuenta*.

En la figura anterior, nótese que hemos indicado que la interfaz *ICuenta* es una interfaz remota mediante la relación de implementación que va desde *ICuenta* hasta *Remote*. *Remote* es una interfaz incluida en el paquete *java.rmi*. En la misma figura, también hemos añadido la relación de especialización desde *Cuenta* hasta *UnicastRemoteObject*, clase incluida en el paquete *java.rmi.server*.

2.01 Representación real del diseño

Las figuras anteriores nos han servido para ilustrar la forma de construcción de nuestro sistema. Como en realidad se trata de tres aplicaciones diferentes (dos clientes y una servidora), debemos construir tres diagramas de clases distintos. Las aplicaciones clientes son las siguientes:

Figura 98. Aplicación cliente que usa *FCuenta*Figura 99. Aplicación cliente que usa *SCuenta*

El esquema de clases de la aplicación servidora es el siguiente:

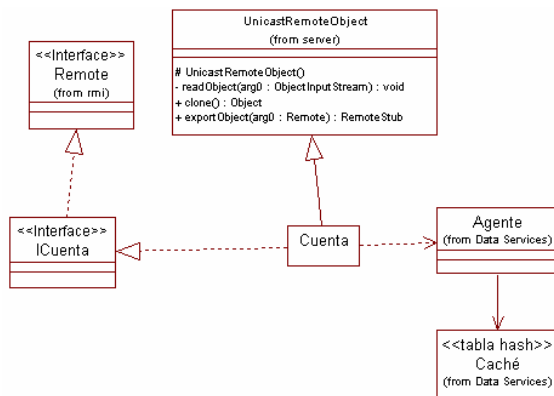


Figura 100. Aplicación servidora

Nótese, en esta última figura, que la aplicación servidora también requiere conocer a la interfaz *ICuenta*, que debe residir en ambas máquinas (en la práctica, esto se traduce en que su correspondiente fichero *.class* se copia en ambas máquinas).

2.02 Adición de observadores remotos

Lamentablemente, los diseños anteriores permiten sólo la compartición de instancias remotas, pero no la notificación remota: es decir, cuando se realiza una transferencia sobre una *Cuenta* a la que “miran” dos ventanas distintas, el cambio de estado no es notificado.

Podemos conectar la clase *Cuenta* a un observador, y que el observador notifique a todas las ventanas que estén apuntando a la instancia. Puesto que las ventanas residen en las máquinas cliente y el observador en la servidora, se hace necesario utilizar también interfaces y objetos remotos.

La siguiente figura muestra una primera aproximación a la resolución de este problema (por claridad, se mantiene un único diagrama de clases para una de las aplicaciones cliente y la aplicación servidora).

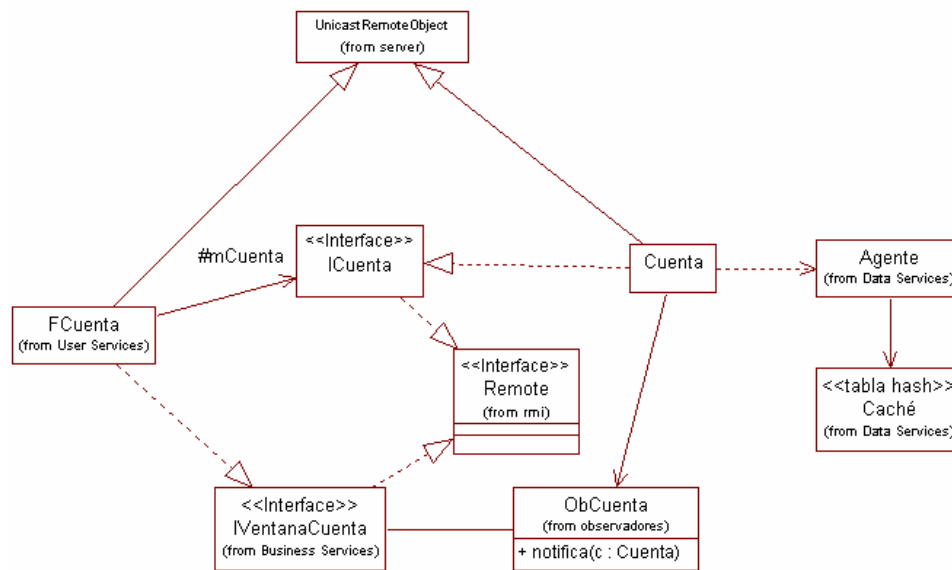


Figura 101. La interfaz *IVentanaCuenta* se hace también remota, así como la clase *FConta*

Como se ve en la figura anterior, la interfaz *IVentanaCuenta* se ha hecho también remota, ya que se accederá a ella desde la aplicación servidora; la clase *FConta*, que implementa la interfaz remota, se ha convertido en especialización de *UnicastRemoteObject*.

Existe sin embargo un importante problema en el diagrama de clases de la Figura 101: *FConta* es una ventana de tipo *Frame*, por lo que *FConta* es una especialización de *Frame*; en Java no existe herencia múltiple (es decir: una clase no puede heredar de más de una superclase), por lo que no es posible que *FConta* sea, a la vez, un *Frame* y un *UnicastRemoteObject*. Una posible solución pasa por crear una clase adicional que sea la encargada de implementar la interfaz remota:

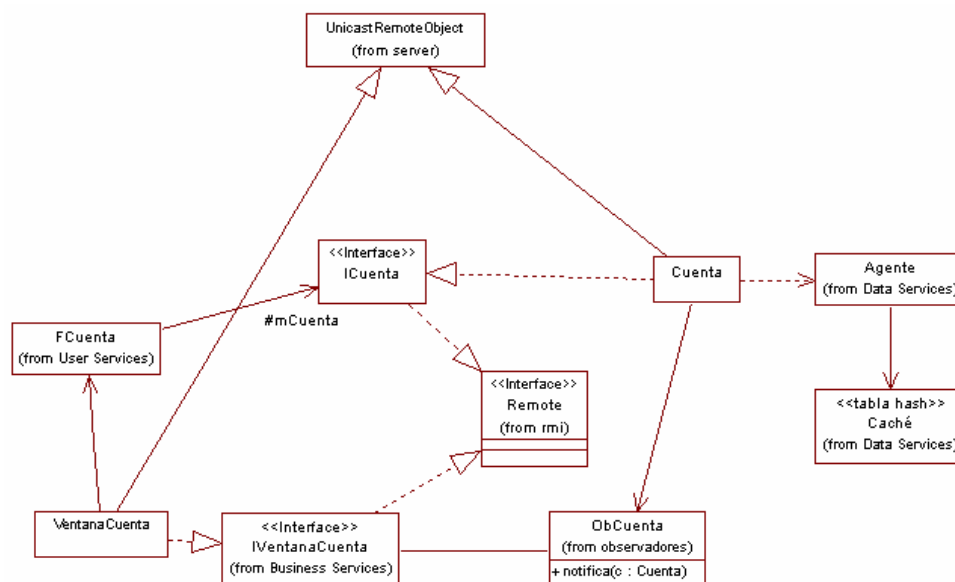


Figura 102. Se añade una clase auxiliar (*VentanaCuenta*) para resolver el problema producido por la imposibilidad de utilizar herencia múltiple

2.03 Detalles de implementación

Cuando un objeto quiere conectarse a un objeto remoto (por ejemplo, una ventana de tipo *FCuenta* se quiere conectar a un objeto remoto de tipo *Cuenta*), debe *buscarlo en el ciberespacio*. Para ello se utiliza el método estático *lookup(String)* de la clase *java.rmi.Naming*:

```
public void conectarACuenta (String ipServidor) throws Exception {
    mCuenta=(ICuenta) Naming.lookup("rmi://" + ipServidor + "/cuenta" +
        textFieldEntidad.getText() +
        textFieldSucursal.getText() +
        textFieldDC.getText() +
        textFieldCCC.getText());
}
```

El objeto buscado debe estar publicado por el servidor al cual se conecta el cliente. Para publicar objetos se utiliza el método estático *bind(String, Remote)* de la clase *java.rmi.Naming*. Un objeto de clase *Cuenta* que desee publicarse ejecutará el método resaltado en negrilla:

```
public class Cuenta extends UnicastRemoteObject implements ICuenta {
    ....
    public void publicarme(String ip) throws Exception {
        Naming.bind("rmi://" + mIP + "/servidor", this);
    }
}
```


Antes de poder ejecutar el *bind*, debe haberse lanzado el servicio *rmi*, que se hace ejecutando la operación *LocateRegistry.createRegistry(puerto)*, en donde *puerto* es el número de puerto en el que quiere lanzarse el servicio (por defecto, el 1099).

Como se ha dicho, el segundo parámetro de *bind* es un *Remote*. En el código que acabamos de ver, el objeto *Cuenta* se pasa a sí mismo al método *bind*. ¿Es *Cuenta* un *Remote*? Sí, porque *Cuenta* implementa la interfaz *ICuenta*, que es *Remote* (véase la Figura 102).

2.04 Alternativas de implementación

Como hemos visto en el epígrafe anterior, para acceder de forma remota a nuestra cuenta, ésta debe estar publicada por el servidor. Sin embargo, resulta completamente inviable mantener simultáneamente publicada la totalidad de las cuentas almacenadas en la base de datos (podemos tener millones de clientes); pero como debemos garantizar el servicio de acceso remoto a las cuentas, hemos de buscar una solución viable.

Algunas posibles soluciones son las siguientes:

- Creación y publicación de un objeto *Servidor*, que sea un *Singleton* (véase sección 4.01.1, página 56) y que, bajo demanda, vaya materializando y publicando instancias de clase *Cuenta*. Cada vez que se materialice una *Cuenta*, se publica (método *bind*) y se devuelve su referencia al cliente que la solicitó, dándole la posibilidad de utilizarla a través de la interfaz remota *ICuenta*. Cuando el cliente cierra la conexión, se pone fin a la publicación de la cuenta (método *unbind*).

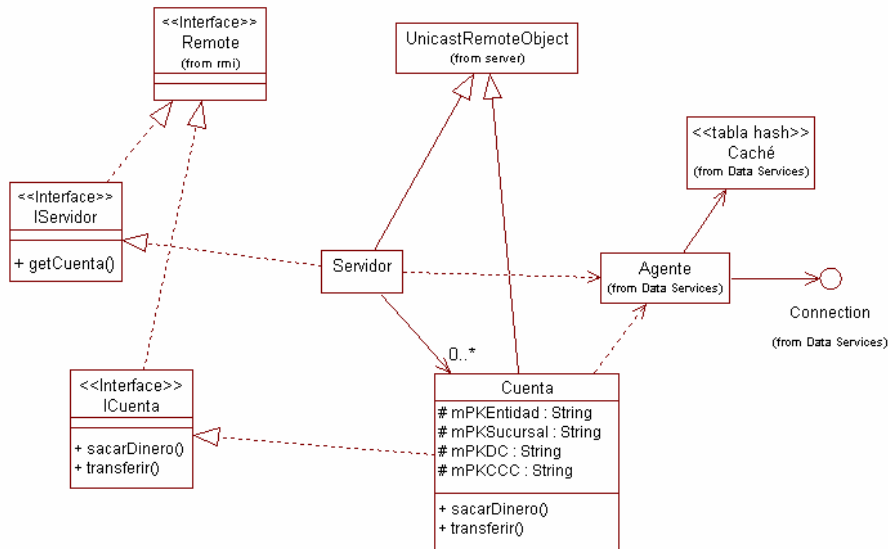


Figura 103. El *Servidor* se pone como objeto remoto y da acceso a instancias, también remotas, de clase *Cuenta*

- Creación y publicación de un objeto *Servidor*, que sea un *Singleton*, y que actúe de interfaz entre los clientes y las cuentas, pero de manera que éstas no sean publicadas. En este caso, los clientes podrían enviar al servidor las acciones deseadas en forma de mensajes codificados (por ejemplo, en lugar de ejecutar el método *transferir* sobre su *Cuenta* asociada, el cliente enviaría al servidor una cadena de caracteres con toda la información necesaria para ejecutar la operación debidamente codificada). Para esta alternativa, el servidor podría ser o tener un *ServerSocket* que iría creando un “dispatcher” para atender a cada cliente.

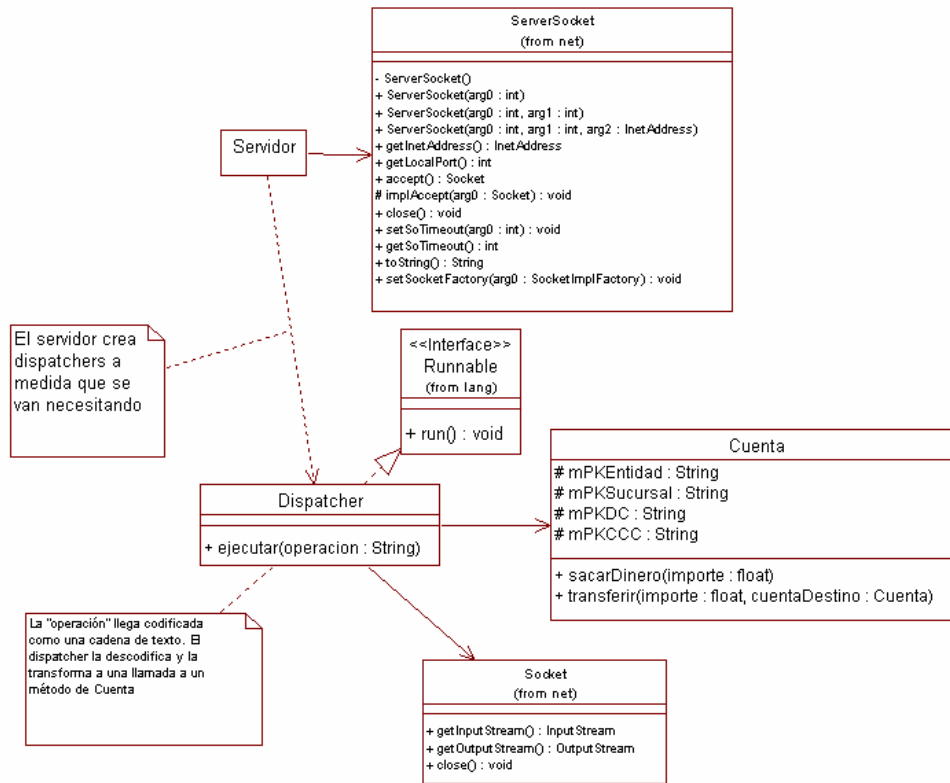


Figura 104. El *Servidor* dispone de un *ServerSocket* que acepta peticiones de los clientes.

3. Gestión de la persistencia en el servidor

Puede configurarse el servidor para que admita un número máximo de instancias de *Cuenta* accesibles simultáneamente, pero manteniendo un número mayor de instancias publicadas. Es decir: permitimos 100 usuarios distintos conectados a la vez a nuestro sistema, cada uno con su cuenta, pero sólo mantenemos en memoria 50 cuentas activas. Las 50 que no se están usando se guardan en un *pool* (piscina), que es una zona del disco en la que guardamos los objetos activos pero que no se están usando.

Supongamos el siguiente escenario:

- 100 usuarios conectados a nuestro sistema, cada uno realizando operaciones con su *Cuenta* correspondiente.
- El sistema está configurado para mantener hasta 50 cuentas en memoria principal, y otras 50 en el *pool*.
- Un usuario cuya cuenta está en el *pool* hace una transferencia.

El sistema realizará lo que sigue:

1. Localiza en memoria una *Cuenta* sobre la que no se esté realizando ninguna operación. Esta búsqueda puede basarse en un algoritmo de tipo *Least Recently Used* o *More Recently Used*.
2. Pasa al *pool* la cuenta localizada en el paso anterior.
3. Pasa a memoria la cuenta del usuario que quiere realizar la transferencia.
4. Realiza la transferencia.

El *pool* almacena objetos *serializados*. Al serializar un objeto, se almacena en disco de tal manera que puede ser posteriormente recuperado en el mismo estado. En Java, son serializables los objetos que implementan la interfaz *java.io.Serializable*. Un objeto *Serializable* se serializa guardándolo mediante un objeto *ObjectOutputStream*, y se recupera mediante un objeto *ObjectInputStream*.

4. Ejemplo

A continuación presentamos un sencillo ejemplo, en el que un cliente se identifica en un servidor remoto mediante RMI. La ventana del cliente tiene dos campos para introducir el login y la password, y un botón de validación:

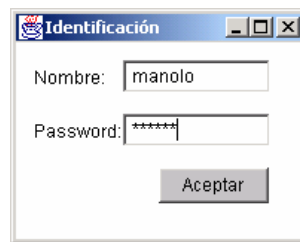


Figura 105. Ventana del cliente

La aplicación servidora muestra únicamente una ventana en la que se muestra lo que va ocurriendo (conexiones correctas e incorrectas de clientes):

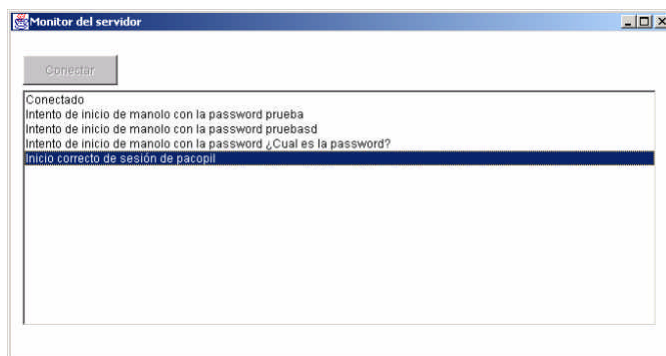


Figura 106. Aspecto de la única ventana de la aplicación servidora

4.01 Aplicación servidora

Esta aplicación consta de un objeto de clase *Servidor*, que es un *UnicastRemoteObject* y que, por tanto, implementa una interfaz *IServidor* que es a su vez *Remote*. En esta interfaz se encuentra la declaración de la operación *public validar(String login, String password)*, cuya implementación se encuentra en *Servidor*. El código de esta clase es el siguiente:

Servidor.java
<pre> package dominio; import java.rmi.server.UnicastRemoteObject; import java.rmi.Naming; import java.rmi.registry.LocateRegistry; import java.rmi.RemoteException; import java.sql.PreparedStatement; import java.sql.ResultSet; import dominio.servpres.IMonitor; import persistencia.Agente; public class Servidor extends UnicastRemoteObject implements IServidor { protected static Servidor mYo=null; // Es un Singleton protected IMonitor mOb; // Tiene un observador // Como es Singleton, el constructor no es accesible desde fuera de la clase protected Servidor() throws Exception { LocateRegistry.createRegistry(1200); Naming.bind("rmi://161.67.27.108:1200/servidor", this); } public static Servidor getServidor() throws Exception { if (mYo==null) mYo=new Servidor(); return mYo; } public void setOb(IMonitor o) { mOb=o; } // Los métodos declarados en la interfaz IServidor, que es Remote, // deben arrojar obligatoriamente la excepción RemoteException public boolean validar(String login, String pwd) throws RemoteException, Exception { String SQL="Select count(*) from Usuarios where Login=? and Password=?"; PreparedStatement p=Agente.getBD().prepareStatement(SQL); p.setString(1, login); p.setString(2, pwd); ResultSet r=p.executeQuery(); if (r.next()) { if (r.getInt(1)==1) { mOb.notifica("Inicio correcto de sesión de " + login); return true; } else { mOb.notifica("Intento de inicio de " + login + " con la password " + pwd); return false; } } p.close(); return false; } } </pre>

4.02 Aplicación cliente

Sólo necesita, además del Frame, tener acceso a la interfaz remota del Servidor y a dos archivos, que se deben generar explícitamente en el servidor. Estos archivos, llamados *skeleton* y *stub*, se construyen compilando con el comando *rmic* los objetos que implementan interfaces remotas.

Estos dos archivos, junto a la interfaz remota ya compilada, pueden empaquetarse en un .zip o en un .jar y ser puestos en el *CLASSPATH* de la aplicación cliente.

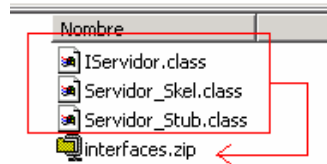


Figura 107. El skeleton, el stub y la interfaz remota deben empaquetarse y ser accesibles a la aplicación cliente

El código de interés de la aplicación cliente es el siguiente:

FSesion.java (en el cliente)

```
package presen;

import java.awt.*;
import java.beans.*;
import java.rmi.Naming;
import java.rmi.RemoteException;
import dominio.IServidor;

public class FSesion extends java.awt.Frame
{
    IServidor mServidor;

    public FSesion()
    {
        ...
    }
    protected void conectar() throws Exception {
        mServidor=(IServidor) Naming.lookup("rmi://161.67.27.108:1200/servidor");
        if (mServidor.validar(this.textFieldNombre.getText(),
            textFieldPassword.getText()))
            showMensaje("Bienvenido");
        else
            showMensaje("Malvenido");
    }
}
```

CAPÍTULO 8. MÁQUINAS DE ESTADOS

1. Introducción

Las máquinas de estados (*state machines*) se utilizan para representar el comportamiento de entidades individuales (instancias de clases) o para definir interacciones entre entidades. Las tres siguientes figura muestran gráficamente la sintaxis abstracta de las máquinas de estados. Están extraídas de la página web de Mario Jeckle (http://www.jeckle.de/uml_spec.htm), que las ofrece en formato Rational Rose a partir del documento de especificación de UML 1.4, disponible en la página web del OMG (www.omg.org).

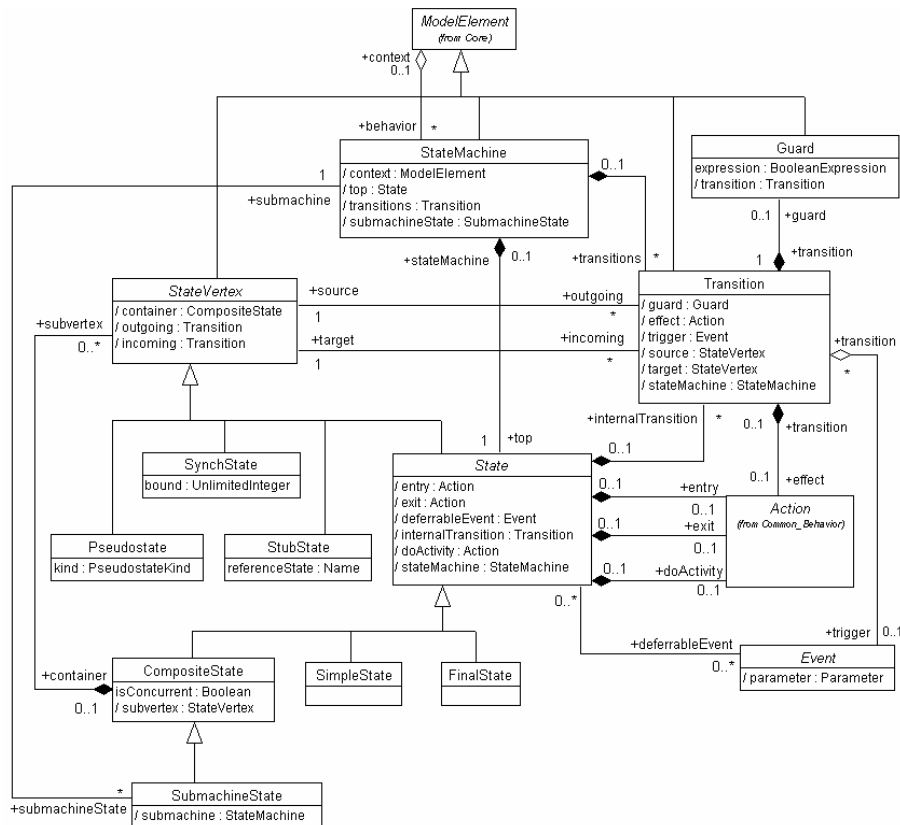


Figura 108. Sintaxis abstracta de las máquinas de estados (I)

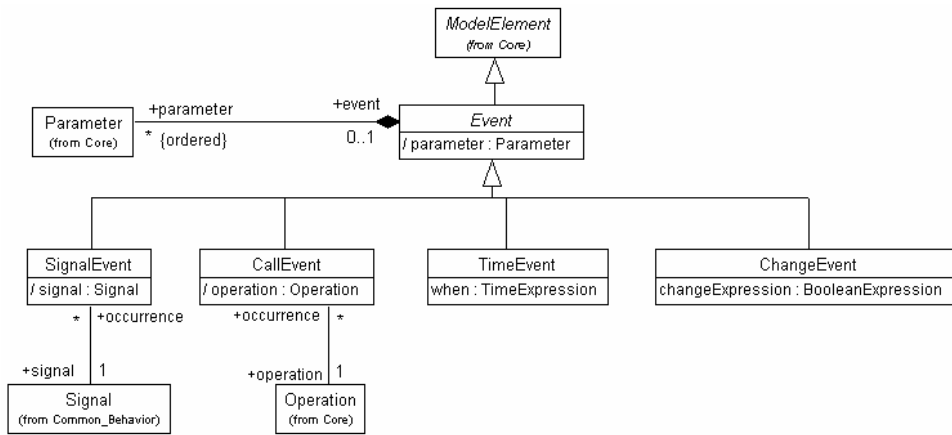


Figura 109. Sintaxis abstracta de las máquinas de estados (II): eventos

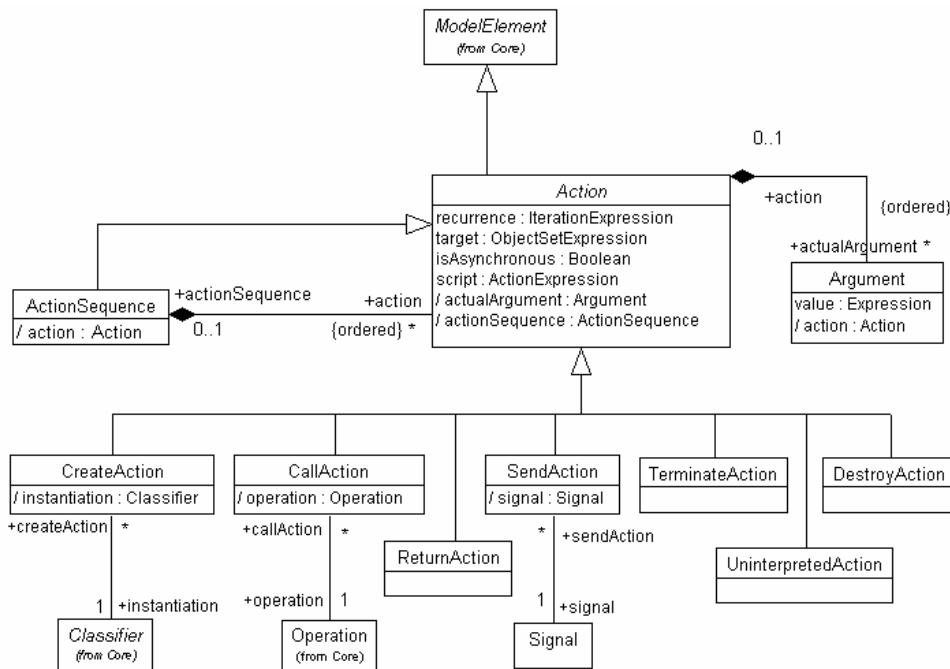


Figura 110. Sintaxis abstracta de las máquinas de estados (y III): acciones

Como puede deducirse (no sin cierto esfuerzo) de las figuras, una máquina de estados (clase *StateMachine* en la Figura 108) está formada por estados y transiciones. Existen diferentes tipos de estados, y cada transición consta, entre otras cosas, de cero o un eventos, de los que existen a su vez varios tipos. Gráficamente, una máquina de estados consistirá en un grafo dirigido cuyos nodos serán los estados y cuyos arcos serán las transiciones.

2. Elementos de las máquinas de estados

A continuación se describen los elementos más representativos de las figuras anteriores, explicando además el significado de los roles de las asociaciones de cada elemento. Préstese atención a las multiplicidades de las asociaciones, ya que son importantes para saber cuántos elementos de modelado pueden admitirse.

Elemento	Descripción	Asociaciones y otros atributos
Máquina de estados (<i>StateMachine</i>)	Describe todos los posibles comportamientos de algún elemento dinámico.	<i>context</i> : elemento del modelo cuyo comportamiento se describe. Obsérvese que un elemento puede describirse con más de una máquina, aunque normalmente basta con una. <i>top</i> : denota el estado raíz o inicial. Hay sólo uno. <i>transitions</i> : conjunto de transiciones de la máquina.
Transición (<i>Transition</i>)	Relación dirigida entre un estado origen y un estado destino. Puede formar parte de una transición compuesta.	<i>trigger</i> : evento que dispara la transición. <i>guard</i> : condición que debe ser cierta para que se ejecute la transición. <i>effect</i> : procedimiento que se ejecuta al dispararse la transición. <i>source</i> y <i>target</i> representan los estados origen y destino.
Estado (<i>State</i>)	Modela una situación durante la que se está verificando alguna invariante (que normalmente es implícita: es decir, que no se encuentra descrita explícitamente). También puede representar una actividad, lo que significa que el elemento que se describe entra en el estado cuando la actividad comienza y sale de él cuando termina).	<i>entry</i> : procedimiento que se ejecuta al entrar en el estado. <i>exit</i> : procedimiento que se ejecuta al salir del estado. <i>doActivity</i> : procedimiento que se ejecuta mientras se está en el estado. La operación comienza a ejecutarse al entrar en el estado, y se termina cuando termina por sí misma o cuando se sale del estado por el disparo de alguna transición. <i>internalTransition</i> : conjunto de transiciones que, si se disparan, no provocan entrada o salida del estado, con lo que no se produce cambio de estado, ni se invocan a las cláusulas <i>entry</i> y <i>exit</i> . <i>deferrableEvent</i> : lista de eventos que pueden quedar retenidos por la máquina de estados si no disparan transiciones hacia fuera del estado.
Estado simple (<i>SimpleState</i>)	Representa un estado que no tiene subestados.	
Estado compuesto (<i>CompositeState</i>)	Estado que contiene otros estados de cualquier tipo (nótese la relación de composición)	<i>subvertex</i> : estados incluidos en este estado. <i>isConcurrent</i> : valor booleano que, si es <i>true</i> , denota que hay dos o más regiones de ejecución concurrente; si es <i>false</i> , no hay concurrencia.
Estado final (<i>FinalState</i>)	Estado distinguido que denota que el estado compuesto en el que está incluido ha finalizado; si el estado en el que está incluido es el raíz, significa que la máquina de estados ha finalizado.	
Guarda o condición de guarda (<i>Guard</i>)	Expresión booleana que complementa una transición (véase <i>Transición</i>). La guarda no puede tener efectos colaterales. El camino que marca el evento	<i>expression</i> : expresión booleana

Elemento	Descripción	Asociaciones y otros atributos
	se sigue si la guarda es <i>true</i> , por lo que las guardas representan precondiciones.	
Estado-submáquina (<i>Sub-machineState</i>)	Convención semánticamente equivalente a un estado compuesto, pero que representa un fragmento de comportamiento reutilizable. De este modo, se puede utilizar la máquina de estados insertada en este estado como si se tratara de una referencia.	<i>submachine</i> : máquina de estados que sustituye a este estado o, dicho de otro modo, referencia a la máquina de estados que se ejecuta al llamar a este estado
Pseudoestado (<i>PseudoState</i>)	Elemento usado para representar varios tipos de nodos en las máquinas de estados:	Observaciones: Tipos de nodos: <i>estado inicial</i> <i>nodos "join"</i> : unen varias transiciones procedentes de regiones concurrentes; tales transiciones no pueden tener guardas. <i>nodos "fork"</i> : separan una transición en varias, cada una a una región concurrente; tales transiciones no pueden tener guardas. <i>nodos "junction"</i> : se usan para unir varias transiciones procedentes de regiones no concurrentes en una sola, o para separar una transición en varias no concurrentes según la guarda. <i>nodos "choice"</i> : nodos que, al ser alcanzados, realizan una evaluación dinámica de las guardas de sus condiciones de salida. La "evaluación dinámica" significa que la comprobación de las guardas se puede calcular en función de computaciones anteriores. Si más de una guarda es <i>true</i> , se va por un camino arbitrario; si ninguna es <i>true</i> , entonces la máquina se considera mal formada, por lo que se aconseja poner siempre una guarda etiquetada "else". Con estos nodos podemos representar postcondiciones sobre el evento. Otros atributos: <i>kind</i> : tipo de pseudoestado (<i>initial</i> , <i>join</i> , <i>fork</i> , <i>junction</i> , <i>choice</i> , <i>deepHistory</i> o <i>shallowHistory</i> ; estos dos últimos no se han explicado).
Estado de sincronización (<i>SynchState</i>)	Nodo que se utiliza para sincronizar regiones concurrentes de una máquina de estados. Aunque hereda de <i>State</i> , no representa realmente un estado en el que pueda encontrarse el elemento que se está modelando.	<i>bound</i> : valor entero positivo o valor ilimitado que especifica la diferencia entre el número de veces que se disparan las transiciones de entrada y de salida.
Estado "stub" (<i>StubState</i>)	En los "estados submáquinas" (véase en esta misma tabla), un <i>StubState</i> representa un estado origen o destino de la máquina de estados que se utiliza por referencia.	<i>referenceState</i> : nombre del estado al que se hace referencia, designado como un "path".
Evento de señal (<i>SignalEvent</i>)	Representa la recepción de una señal (que es un estímulo asíncrono).	<i>signal</i> : señal (véase en la fila inferior de esta misma tabla) específica asociada con este evento

Elemento	Descripción	Asociaciones y otros atributos
Señal (<i>Signal</i>)	Estímulo asíncrono transmitido entre instancias (una excepción, por ejemplo, u otro estímulo del que no se espera respuesta).	
Evento de llamada (<i>CallEvent</i>)	Representa la recepción de una solicitud a una operación específica. Dos casos especiales son el evento de creación de un objeto y el de su destrucción.	<i>operation</i> : operación cuya invocación lanzó el evento. Pueden usarse los estereotipos <code><<create>></code> y <code><<destroy>></code> .
Evento de cambio (<i>ChangeEvent</i>)	Representa un evento que ocurre cuando se hace cierta una condición sobre uno o más atributos o asociaciones. Se lanzan de manera implícita, no explícita.	<i>changeExpression</i> : expresión booleana que especifica la ocurrencia del evento.
Evento de tiempo (<i>TimeEvent</i>)	Representa que se ha llegado a un momento predeterminado.	<i>when</i> : expresión de tiempo que representa el momento en el que debe ocurrir el evento.
Acción (<i>Action</i>)	Especificación de una instrucción ejecutable que produce un cambio de estado en el modelo	<i>recurrence</i> : expresión que indica cuántas veces debe ejecutarse la acción <i>target</i> : objeto u objetos sobre los que se ejecuta la acción <i>isAsynchronous</i> : valor booleano que indica si el mensaje enviado es o no asíncrono <i>actualArgument</i> : parámetros reales de la acción
Creación (<i>CreateAction</i>)	Acción que resulta en la creación una instancia	<i>instantiation</i> : clasificador (véase página 36) del cual se crea la instancia. Nótese que no se usa el atributo <i>target</i> , heredado de <i>Action</i> , pues éste denota un objeto, mientras que <i>instantiation</i> representa (normalmente) una clase.
Llamada (<i>CallAction</i>)	Acción que resulta en la llamada a una operación de una instancia	<i>operation</i> : operación que se ejecuta, que pertenece a la clase de la instancia receptora del mensaje
Retorno (<i>ReturnAction</i>)	Acción que resulta en la devolución de un resultado al que la llamó	
Envío (<i>SendAction</i>)	Acción que resulta en el envío asíncrono de una señal	<i>signal</i> : señal que se envía al ejecutarse la acción
Terminación (<i>TerminateAction</i>)	Acción que denota la autodestrucción de un objeto	El <i>target</i> es el objeto que ejecuta la acción
Acción no interpretada (<i>UninterpretedAction</i>)	Acción que no está recogida explícitamente en UML. Se usan para representar acciones dependientes del lenguaje de programación.	
Destrucción (<i>DestroyAction</i>)	Acción que resulta en la destrucción de una instancia	El atributo <i>target</i> (heredado de <i>Action</i>) indica qué instancia se destruye

Tabla 2. Descripción de los elementos de las máquinas de estados

En el contexto de las máquinas de estados, las acciones (*Action* en la Figura 110) son consecuencias que se producen al ejecutarse la transición que pueden provocar un cambio de estado en el modelo (por ejemplo, enviando un mensaje a otro objeto o modificando el valor de un atributo). Cada transición tiene 0 o 1 acciones (véase Figura 108),

si bien la acción (que es abstracta) puede ser una *ActionSequence* (secuencia de acciones), lo que significa que puede a su vez existir un conjunto ordenado de acciones⁴.

3. Ejemplos y notación

Como se ha dicho al principio de este capítulo, las máquinas de estados se utilizan para describir el comportamiento de entidades individuales (instancias de clases) o para definir interacciones entre entidades. En el primer caso (descripción del comportamiento de instancias), se asume que el estado del objeto es una función de los valores de su conjunto de atributos, de manera que se agrupan en un mismo estado aquellas situaciones en que el objeto va a responder del mismo modo ante un determinado conjunto de estímulos; en el segundo caso (descripción del comportamiento de interacciones entre entidades, como podría ser el funcionamiento de casos de uso, interacciones entre objetos o funcionamiento de los métodos de una clase), se asume que un estado representa un paso en la ejecución del elemento que se está modelando. No obstante lo dicho, el uso habitual de las máquinas de estados es el primero de los dos mencionados.

La siguiente figura muestra un posible diagrama de estados para la clase Cuenta de nuestro sistema bancario. Destacan el estado inicial (un “punto gordo”) y el final (un “punto gordo” rodeado por una circunferencia). El tiempo que el objeto permanece en el estado inicial es despreciable. Se han identificado los estados en que la cuenta (más bien las instancias de esta clase) puede estar: recién creada, con saldo positivo o negativo.

En cualquiera de estos dos estados pueden realizarse operaciones de ingreso de efectivo, que llevarán al objeto al mismo estado en que se encuentran o a otro distinto (si se ingresa con saldo negativo y el saldo resultante pasa a ser positivo, la cuenta cambia de estado). Nótese las condiciones de guarda (indicadas entre corchetes) y los nodos con forma de rombo:

- Recuérdese que las guardas no pueden tener efectos colaterales (véase la descripción de “Guarda o condición de guarda” en la Tabla 2), por lo que estamos asumiendo que la operación *getSaldo()* no afecta a ningún atributo de la instancia de clase Cuenta.
- El nodo con forma de trombo representa un nodo pseudoestado de tipo “choice”. Los arcos de salida que muestra representan condiciones evalua-

⁴ Esta estructura se corresponde con el patrón “Composite” (página 145).

das tras la ejecución de la operación, aunque no son exactamente postcondiciones.

También es importante hacer notar que hay eventos que pueden llegar a la instancia y no ser procesados, como es el caso de la operación *retirar(importe:double)* cuando la cuenta está en el estado “Saldo negativo”.

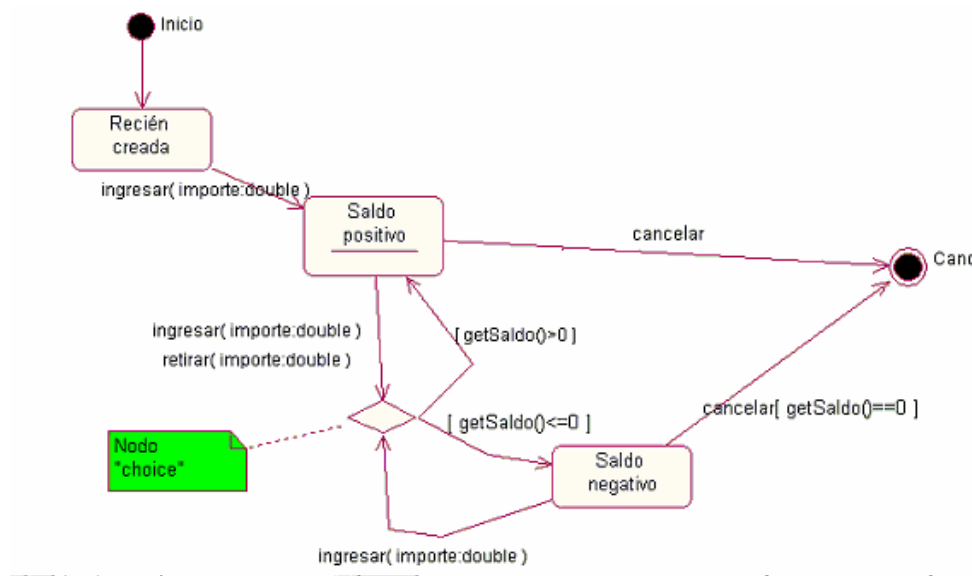


Figura 111. Máquina de estados para la clase Cuenta (ejemplo 1)

Puesto que toda cuenta tiene un titular, éste forma también parte del estado de la cuenta, y puede interesarnos utilizarlo de algún modo para representar el comportamiento de la cuenta. Supongamos que si el titular de la cuenta tiene un préstamo hipotecario, entonces le dejamos retirar dinero de su cuenta incluso con saldo negativo. Esto lo podemos representar añadiendo el evento *retirar(importe:double)* en el estado “Saldo negativo” y añadiendo una precondición con una guarda, como se muestra en la figura siguiente:

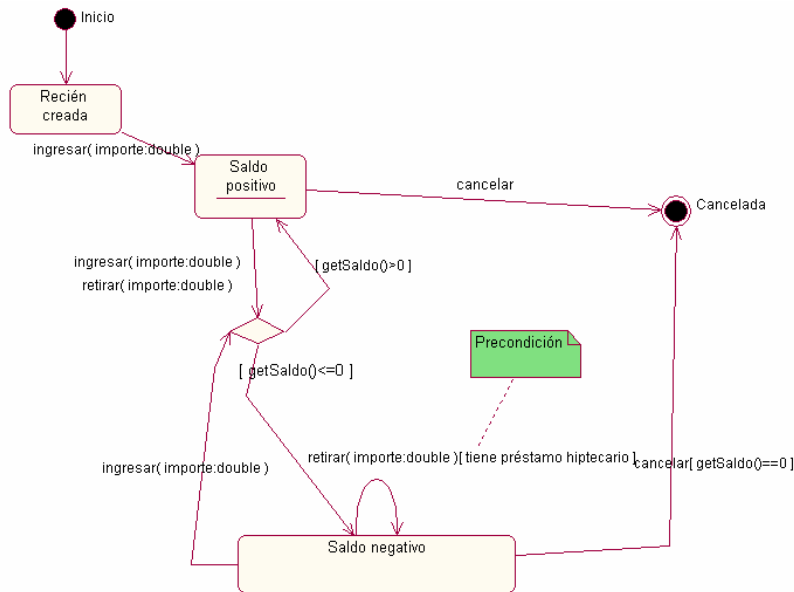


Figura 112. Máquina de estados para la clase Cuenta (ejemplo 2)

Con las acciones de entrada y salida (véase la definición de “Estado” en la Tabla 2) podemos especificar lo que hace la instancia nada más entrar a o salir de un estado. Podemos asumir que al ingresar o al retirar se suma, a un atributo *mSaldo* de la clase Cuenta, el importe ingresado o retirado (en el caso de *retirar*, el importe retirado debería ser negativo para que el saldo se actualizara correctamente). Esto lo representamos en la figura siguiente:

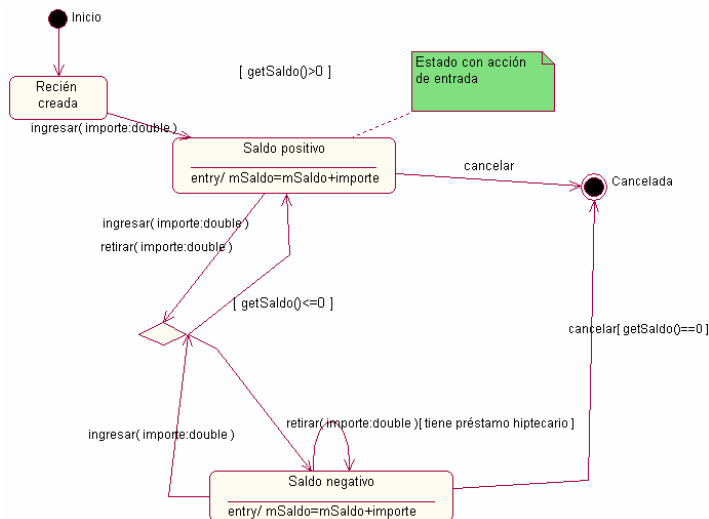


Figura 113. Máquina de estados para la clase Cuenta (ejemplo 3)

Si asumimos que tanto al ingresar como al retirar se pasa como argumento un *importe* con valor positivo, entonces la máquina de estados de la figura anterior no sería válida, ya que al retirar sumaríamos al saldo el importe retirado. Podemos entonces anotar las transiciones con efectos (véase el atributo *effect* en la definición de Transición en la Tabla 2):

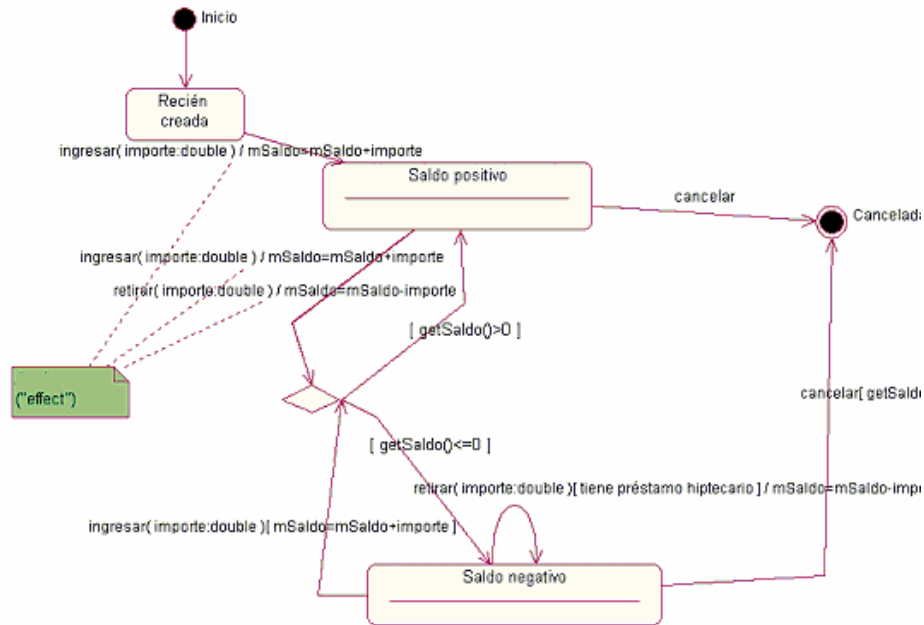


Figura 114. Máquina de estados para la clase Cuenta (ejemplo 4)

Podemos crear un estado “Activo” que represente el estado de la cuenta sobre la que se ha hecho al menos un ingreso: es decir, toda cuenta que tenga saldo negativo o positivo. En la siguiente figura se ha creado un Estado compuesto (véase definición en la Tabla 2) que alberga dos subestados. Obsérvese que puede haber transiciones desde fuera del estado compuesto hacia subestados, y que pueden salir transiciones desde subestados hacia el exterior.

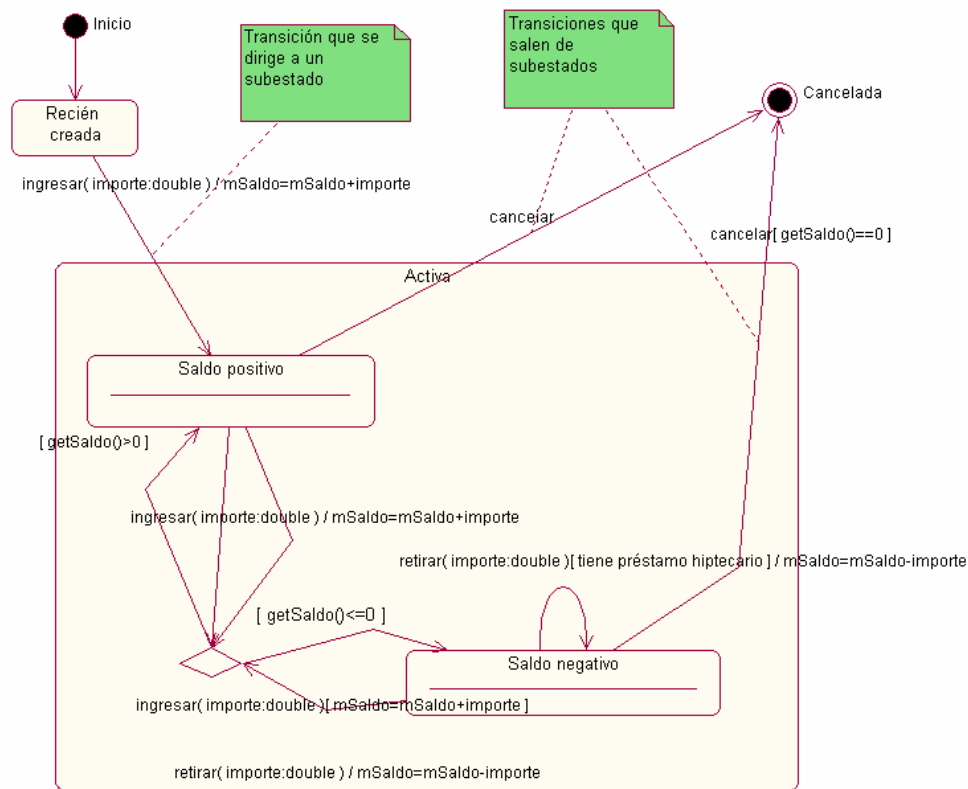


Figura 115. Máquina de estados para la clase Cuenta (ejemplo 5)

Otra forma de representar la máquina mostrada en la figura anterior es dirigir la transición desde “Recién creada” al estado “Activa”, y añadir entonces un estado inicial al estado compuesto que conduzca a “Saldo positivo”. Igualmente, podemos sustituir las dos transiciones etiquetadas “cancelar” por una sola, que salga de “Activa” a “Cancelada”:

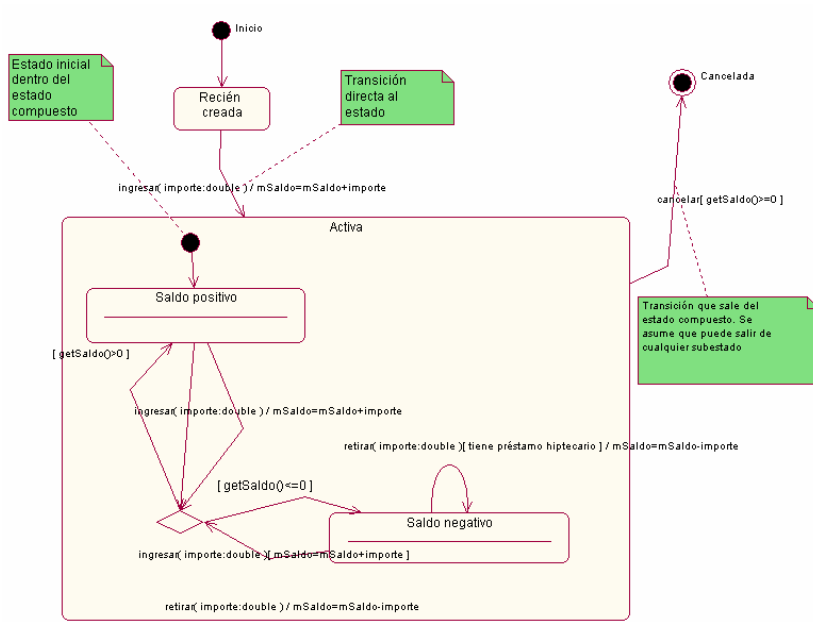


Figura 116. Máquina de estados para la clase Cuenta (ejemplo 6)

En la siguiente figura denotamos (por claridad sólo en una transición) que, al realizar un ingreso, se crea una instancia de clase “Movimiento” (podríamos encadenar acciones en una *ActionSequence*, pero la herramienta de diseño utilizada no lo permite).

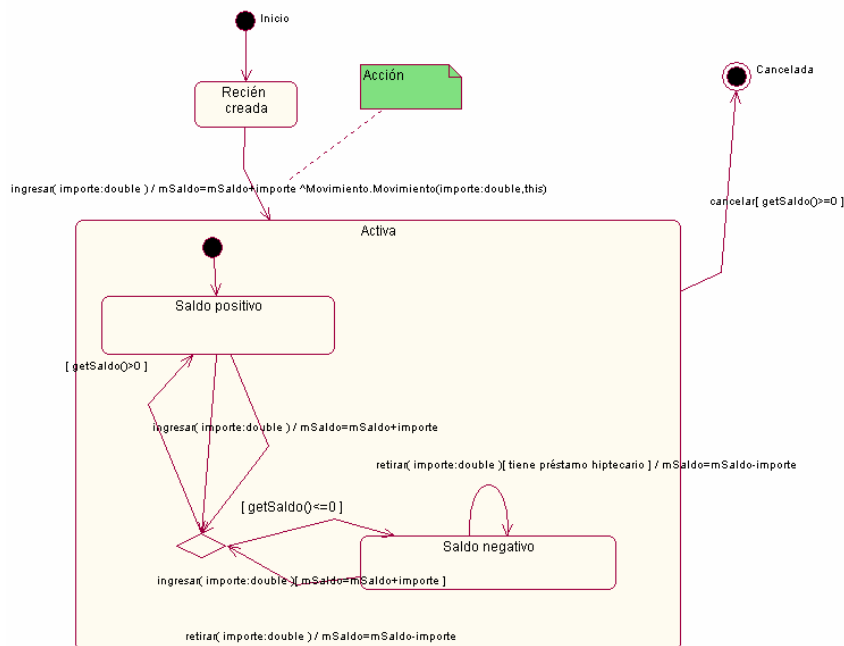


Figura 117. Máquina de estados para la clase Cuenta (ejemplo 7)

En la siguiente figura cambiamos de tercio para ilustrar con una máquina de estados el funcionamiento de un caso de uso (como podría ser el de “Retirar dinero con tarjeta de débito”), en lugar del comportamiento de una clase, como hasta ahora. Se observa que los estados vienen a representar ahora “pasos en la ejecución del caso de uso”, como se ha comentado al principio de esta sección (página 120). Las transiciones están etiquetadas en formato libre, aunque podrían muy bien corresponder a operaciones de las clases que intervienen en este caso de uso. Se ilustra el concepto de estados concurrentes en el estado compuesto “Entregando dinero”, en el que se ejecutan a la vez dos operaciones (la dispensación de billetes y la anotación del movimiento). Se asume que se sale de “Entregando dinero” cuando concluye la ejecución de los dos estados concurrentes. Hay transiciones sin etiquetar (desde “Tarjeta introducida” a “Solicitud de número secreto” a “Solicitud de número secreto”), que se ejecutan de manera automática. Igualmente, hemos añadido una acción de tipo “Actividad” (*doActivity*, véase la definición de “Estado” en la Tabla 2) al estado “Dispensando billetes”, así como una entrada.

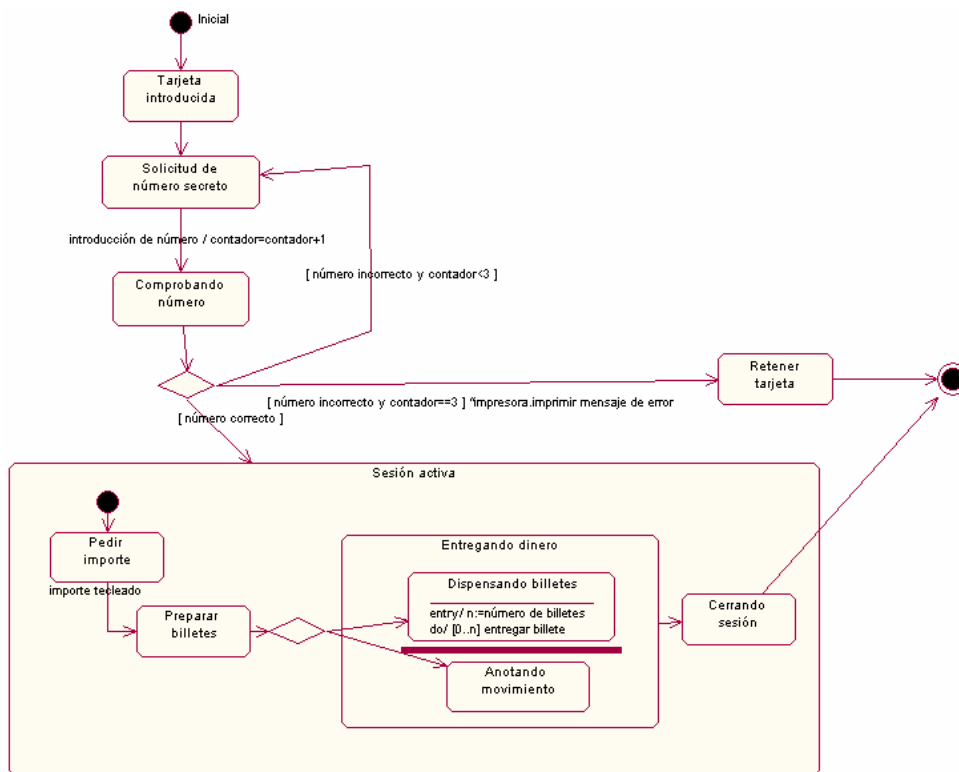


Figura 118. Descripción de un caso de uso con una máquina de estados

4. Ejercicios

4.01 PFC

La tramitación de un Proyecto de Fin de Carrera en esta Escuela es el siguiente:

1) El alumno presenta un anteproyecto en Secretaría junto a una propuesta de seis profesores para formar tribunal.

2) La Comisión Académica aprueba o desaprueba el anteproyecto, nombrando presidente, secretario, vocal y tres suplentes en el primer caso.

3) El alumno presenta cuatro ejemplares del Proyecto en Secretaría con al menos tres días de antelación sobre la fecha de lectura. La Secretaría envía dos copias al presidente, una al secretario y otra al vocal. El presidente convoca el acto de defensa del Proyecto. Si alguno de los miembros del tribunal no puede asistir, se elige un sustituto de entre los suplentes.

4) El alumno puede aprobar o suspender.

Se pide: represente con una máquina de estados el comportamiento de las instancias de clase "PFC".

4.02 Ascensor

Represente con una máquina de estados el comportamiento de un ascensor, cuya clase es la siguiente:

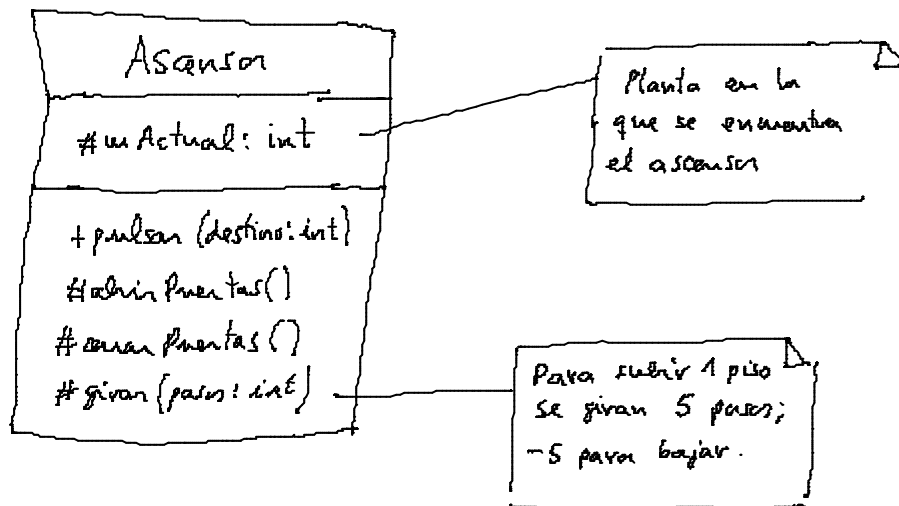
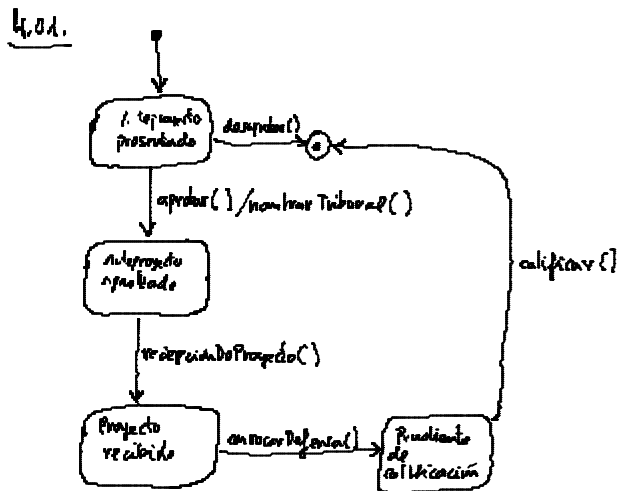


Figura 119



Como se trata de describir el comportamiento de las instancias de PFC, no sería correcto representar las operaciones de Secretaría, que no parecen afectar al estado del Proyecto. Tampoco afecta el hecho de q. se elija sustituto si alguno de los del tribunal no puede asistir.

4.02.

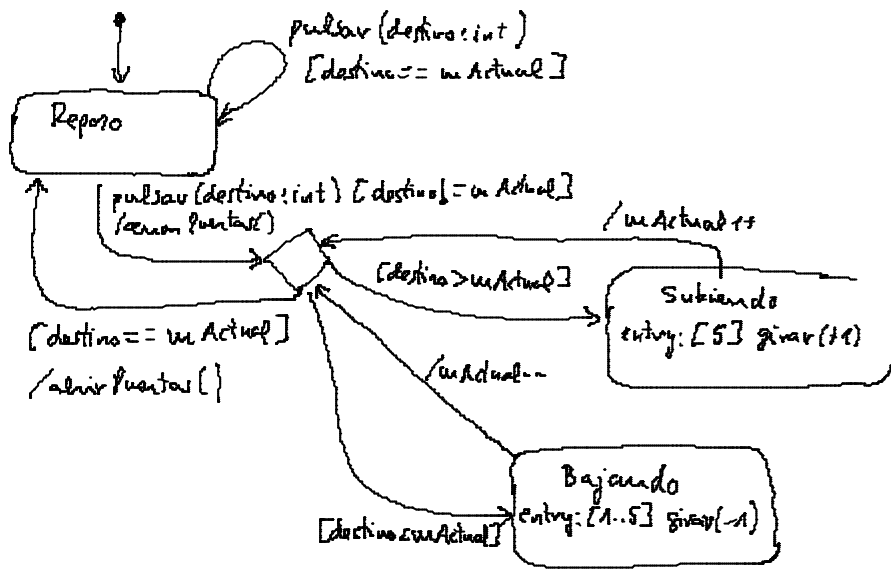


Figura 120

CAPÍTULO 9. DIAGRAMAS DE ACTIVIDAD

1. Introducción

Los diagramas de actividad son uno de los diagramas UML que muestran el comportamiento dinámico del sistema. Esencialmente, consisten en un diagrama de flujo en el que se muestran los pasos que deben ejecutarse para cumplir un proceso de cómputo, pudiendo también incluir aspectos de sincronización.

Los diagramas de actividad con un caso especial de las máquinas de estados. Si con éstas representábamos el comportamiento de un solo elemento, con los diagramas de actividad representamos procesos en los que (1) interviene más de un clasificador y (2) consideramos necesario representar cómo colaboran esos clasificadores entre sí.

La siguiente figura muestra la sintaxis abstracta de los diagramas de actividad. Como se observa, muchos de los elementos se incluían ya en la Figura 108 (página 115), en la que se describían las máquinas de estados. El contorno de los nuevos elementos se marca en color rojo y, para los que lo vean en blanco y negro, también en trazo más grueso.

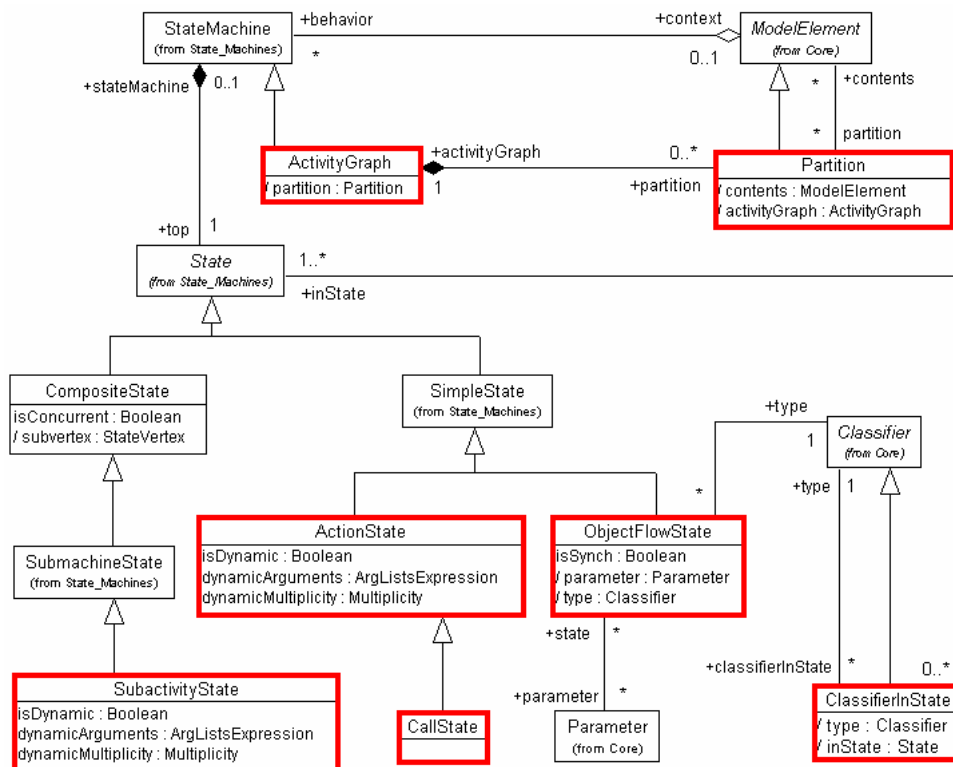


Figura 121. Sintaxis abstracta de los diagramas de actividad

2. Elementos de los diagramas de actividad

A continuación se describen los elementos nuevos que aparecen en la Figura 121. Complétense estas descripciones con las de la tabla Tabla 2 (página 119 y anteriores), en el capítulo anterior.

Elemento	Descripción	Asociaciones y otros atributos
Diagrama de actividad (<i>ActivityGraph</i>)	Tipo especial de máquina de estados que describe un proceso computacional en función del flujo de control y del flujo de objetos entre las actividades que lo forman el proceso de scritto.	
Partición (<i>Partition</i>)	Mecanismo que permite dividir los estados del diagrama en grupos. Los estados de cada partición son los conocidos a través del atributo <i>contents</i> .	
Estado de acción (<i>ActionState</i>)	Estado que representa un paquete atómico de una o más acciones (o se ejecutan todas las acciones en el paquete o no se ejecuta ninguna). Tiene una o más transiciones de salida “implícitas”: es decir, que se sale del estado de manera automática cuando el paquete de acciones termina de ejecutarse. Si hay más de una transi-	<i>isDynamic</i> : valor booleano que, si vale <i>true</i> , indica que las acciones del estado se pueden ejecutar de forma paralela. Se usa junto a los dos siguientes atributos. <i>dynamicArguments</i> : lista de objetos que determina en tiempo de ejecución el número de ejecuciones paralelas de las acciones del estado. Cada objeto sirve como parámetro para una de las ejecuciones concu-

Elemento	Descripción	Asociaciones y otros atributos
	ción de salida, debe etiquetarse cada una con una guarda. Los estados de acción no pueden tener actividades (<i>doActivity</i>) ni acciones de salida (<i>exit</i>).	rrentes. <i>dynamicMultiplicity</i> : número máximo de ejecuciones paralelas de esta actividad.
Objeto de flujo (<i>ObjectFlowState</i>)	Representa una instancia que es entrada o salida de una acción.	<i>isSynch</i> : valor booleano que, si vale <i>true</i> , indica que el objeto se usa como estado de sincronización. <i>type</i> : tipo de la instancia.
Clasificador con estado (<i>ClassifierInState</i>)	Representa una instancia que se encuentra en un cierto estado. Se usa cuando en la situación que describe el diagrama se manipula un objeto cambiando varias veces de estado.	<i>type</i> : tipo de la instancia <i>inState</i> : estado en el que se encuentra la instancia <i>parameter</i> : parámetros que el objeto suministra como salida o toma como entrada
Estado de llamada (<i>CallState</i>)	Es un estado de acción (<i>ActionState</i>) que llama a una única operación. Es una forma de separar las acciones de un <i>ActionState</i> en varios pequeños estados: esto a veces es interesante para remarcar, por ejemplo, las acciones que producen determinados resultados.	
Estado de subactividad (<i>SubactivityState</i>)	Estado que representa una secuencia no atómica de ejecución.	<i>isDynamic</i> : valor booleano que, si vale <i>true</i> , indica que las acciones del estado se pueden ejecutar de forma paralela. Se usa junto a los dos siguientes atributos. <i>dynamicArguments</i> : lista de objetos que determina en tiempo de ejecución el número de ejecuciones paralelas de las acciones del estado. Cada objeto sirve como parámetro para una de las ejecuciones concurrentes. <i>dynamicMultiplicity</i> : número máximo de ejecuciones paralelas de esta actividad.

Tabla 3. Descripción de los elementos de los diagramas de actividad

3. Ejemplos y notación

A continuación se muestra la descripción de la situación en que se realiza una transferencia con una tarjeta desde un cajero automático: intervienen tres instancias (la tarjeta, su cuenta asociada y la cuenta destino). Los diagramas de actividad permiten la creación de *swimlanes* (literalmente: “calles de piscina”). Cada calle se utiliza para representar los estados por los que pasa cada uno de los objetos que colabora activamente en el proceso que se está describiendo.

La Figura 122 muestra el diagrama de actividad correspondiente dibujado con Rational Rose. Esta herramienta no permite la adición de objetos al diagrama y permite construcciones ilegales, como la adición de acciones de salida a los estados de acción.

El mismo diagrama se muestra en la Figura 123, pero en esta ocasión con la herramienta Poseidon CE. En este caso, sólo existen estados de actividad y no de acción, y no es posible añadir *swimlanes*, aunque sí objetos.

La situación es bastante parecida con otras herramientas de análisis y diseño orientado a objetos (Fujaba, JDeveloper, Enterprise Architect...): todas dejan construir diagramas de actividad, pero ninguno de forma completa y fiel a la sintaxis abstracta.

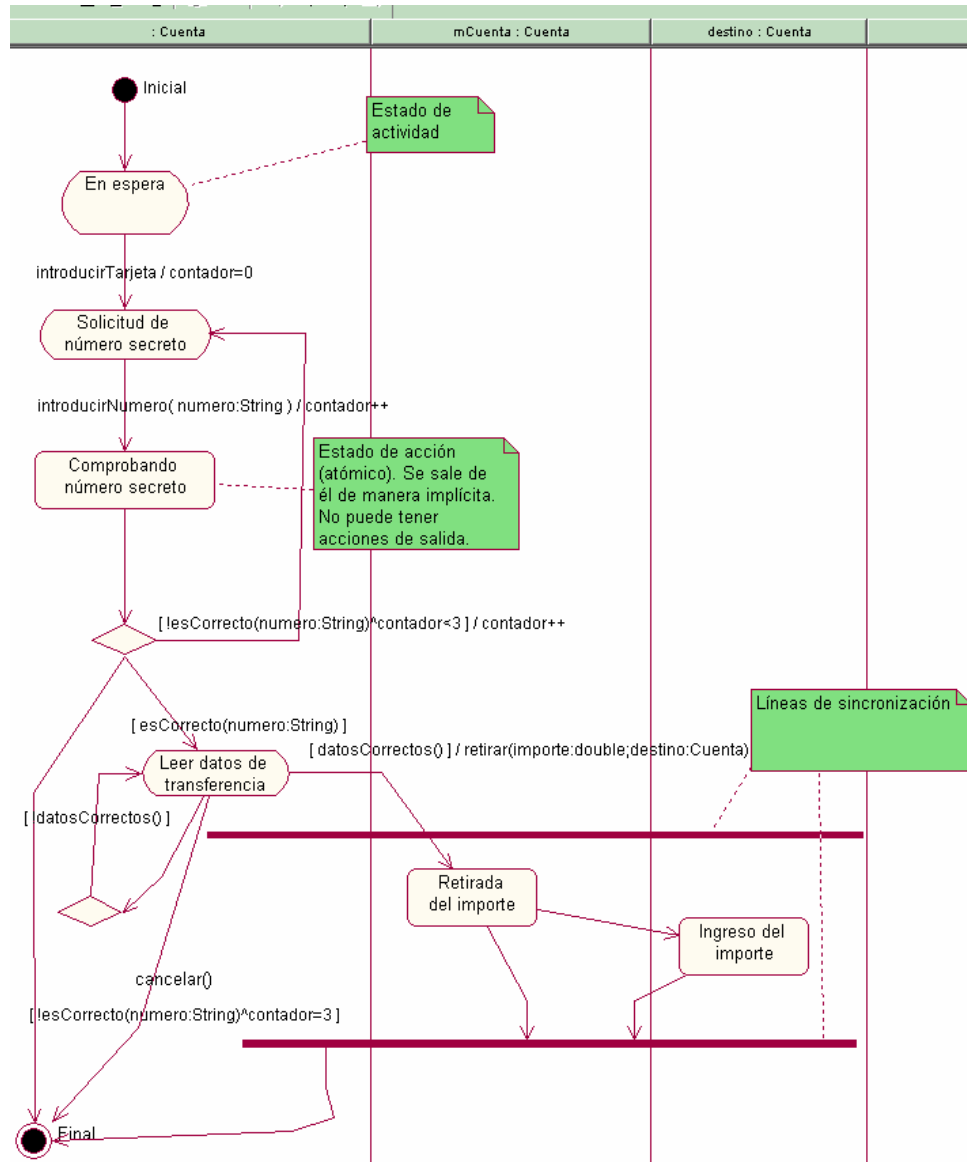


Figura 122. Un diagrama de actividad hecho con Rational Rose

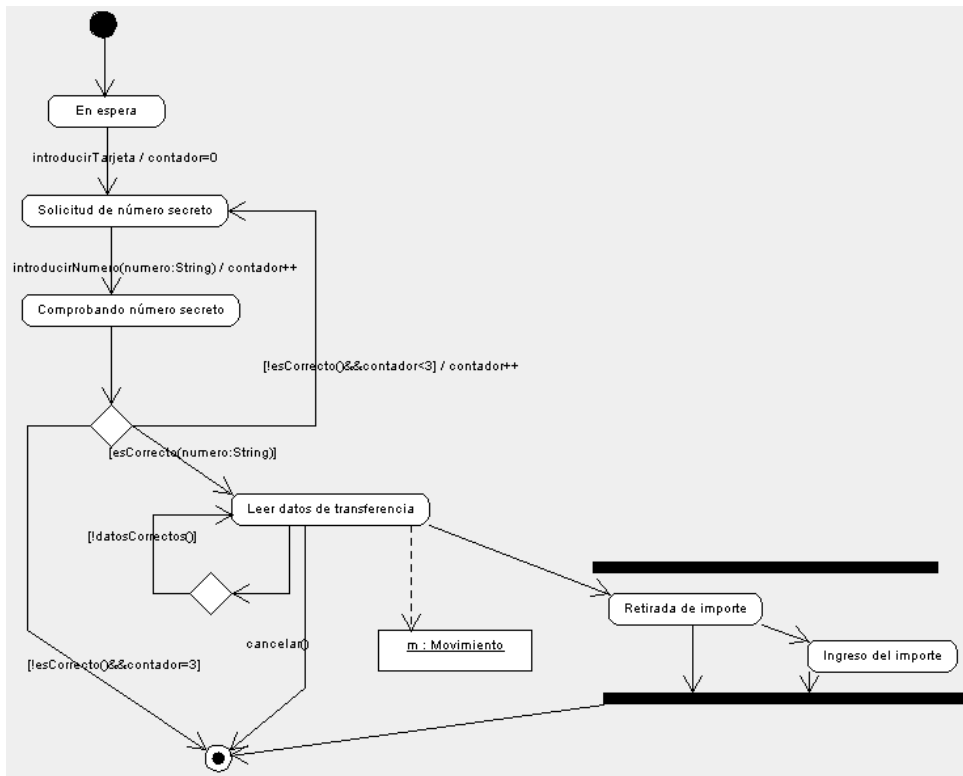


Figura 123. Un diagrama de actividad hecho con Poseidon CE

CAPÍTULO 10. PATRONES DE DISEÑO (I)

1. Introducción

En este capítulo veremos algunos de los patrones de Gamma, Helm, Johnson y Vlissides (1996)⁵. Estos autores agruparon sus patrones en tres familias:

- Patrones de creación, que “abstraen el proceso de creación de instancias”. Estos patrones ayudan a hacer un sistema independiente de cómo se crean, se componen y se representan los objetos que lo componen.
- Patrones estructurales, que permiten describir cómo se combinan estructuras y objetos para formar estructuras más grandes.
- Patrones de comportamiento, que permiten describir la forma en que se asignan las responsabilidades a los objetos del sistema.

2. Patrones de creación

2.01 *Abstract factory* (fábrica abstracta)

Supongamos que, en el banco que venimos describiendo, existen diferentes familias de productos asociados a diversos tipos de cuentas, que dependen del tipo de cliente que la abra. Así, por ejemplo:

- La “cuenta joven” se ofrece a los clientes de menos de 25 años y tiene un 2% de interés anual. Al abrirla, el banco regala un CD de un grupo de música y una tarjeta de débito sin cuota anual.
- La cuenta “10” se ofrece a los clientes entre 26 y 65 años que domicilian su nómina y tiene un 1% de interés anual. El banco ofrece los mismos productos que arriba. La tarjeta de débito no tiene coste; además, ofrece una tarjeta de crédito por 18 euros anuales, con un crédito igual al 60% de la nómina. La cuenta permite descubiertos de hasta el 50% de la nómina y, al abrirla, se regala un reproductor portátil de CD.

⁵ Existe traducción al castellano de este libro: Patrones de Diseño. Editorial Addison Wesley, 2003.

- La cuenta “oro” se ofrece a los clientes mayores de 65 años con pensión, e incluye tarjetas de débito y crédito gratuitas. Da un 1,5% de interés anual. El crédito es el 60% del importe de la pensión. Al abrir esta cuenta, el banco regala un seguro de amortización de crédito de hasta 12.000 euros.
- La “cuenta estándar” se ofrece a los clientes que no encajen en ninguno de los perfiles anteriores, o que no quieran ninguna de esas familias de productos. Da un 0,5% de interés anual. Permite disponer de una tarjeta de débito con una cuota anual de 5 euros, una cuenta corriente con un 0,5% de interés anual.

Además:

- En la Cuenta Joven: a fin de mes, por cada 100 euros que el saldo de la cuenta supere el saldo del mes anterior, se dan 10 “puntos jóvenes” al titular, que puede luego canjear por regalos. Igualmente, por cada 10 euros de compra con su tarjeta de débito se le dan 10 puntos jóvenes.
- En la Cuenta 10 y en la Cuenta Oro: cada vez que se recibe el ingreso de la nómina o de la pensión, se le dan 300 puntos al titular, que luego puede canjear por regalos; igualmente, por cada recibo domiciliado que llega a la cuenta, se le dan 100 puntos. Por cada 10 euros de compra con su tarjeta se le dan 10 puntos.
- La Cuenta estándar no da puntos.

Podemos resumir los productos ofertados en la siguiente tabla:

Tipo de cuenta	Cuenta	Tarjeta débito	Tarjeta crédito	Regalo
Joven	2% de interés	Gratuita	No	CD música
10	1% de interés 50% descubierto	Gratuita	18 euros 60% nómina	Reproductor CD
Oro	1,5%	Gratuita	Gratuita 60% pensión	Seguro
Estándar	0,5%	5 euros	No	No

En este ejemplo, se observa que, salvo algunas opciones, se ofrecen los mismos productos a los clientes, si bien cada uno de ellos tiene algunas particularidades que los diferencian de los demás. En notación de clases, podemos representar el conjunto de productos como se muestra en esta figura:

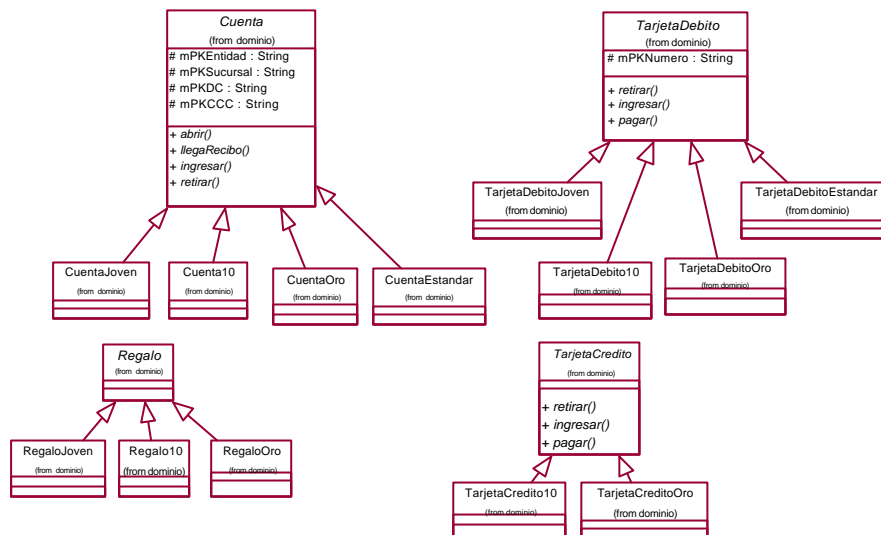


Figura 124. Familias de productos ofertados por el banco

Nos gustaría tener la opción de que, al crear una cierta cuenta, se crearan automáticamente los productos asociados al tipo de cuenta. Para esto podemos utilizar una fábrica abstracta, que nos permitirá crear de manera sencilla la familia de objetos (Figura 125). Como se observa, tenemos tantos tipos de fábricas como tipos de objetos. La fábrica abstracta (clase *Fábrica* en la figura) tiene tantas operaciones abstractas como tipos de productos hay en la familia. Estas operaciones son implementadas por las fábricas especializadas. Cada fábrica concreta conoce a los productos que puede fabricar; con el fin de mantener la legibilidad de la Figura 125, únicamente se han señalado las relaciones de dependencia de la *FabricaJoven* con sus respectivos productos: *CuentaJoven*, *TarjetaDebitoJoven* y *RegaloJoven*.

La creación de la familia de productos podría residir en el siguiente método:

```

public void crearLote(Cliente c) {
    Fabrica f;
    if (c.getTipo()==CLIENTE_JOVEN)
        f=new FabricaJoven(c);
    else if (c.getTipo()==CLIENTE_10)
        f=new Fabrica10();
    else if (c.getTipo()==CLIENTE_ORO)
        f=new FabricaOro(c);
    else
        f=new FabricaEstandar(c);
    f.crearCuenta();
    f.crearTDebito();
    f.crearTCredito();
    f.crearRegalo();
}
  
```

En este código existe un objeto f de tipo *Fabrica* (que es una clase abstracta); dependiendo del tipo de cliente, f se instancia al tipo de fábrica correspondiente. A continuación, se ejecutan las cuatro operaciones de creación del lote de productos. Estas operaciones son abstractas en *Fabrica*, pero concretas en sus subclases: puesto que, al llegar a la sentencia $f.crearCuenta()$ la variable f ya está instanciada a una fábrica concreta, se ejecutan la versión adecuada de la operación.

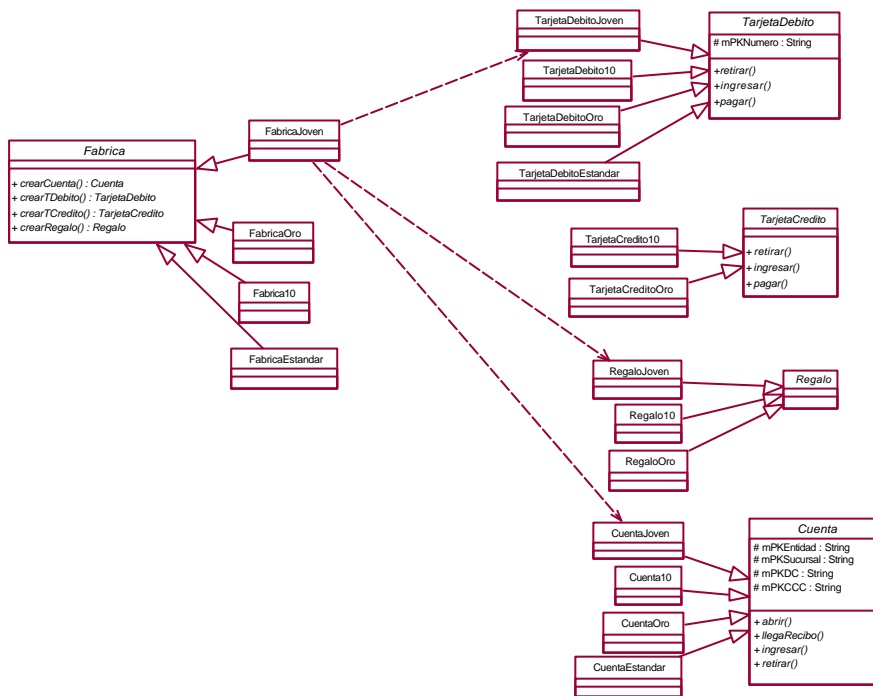


Figura 125. Una fábrica abstracta para crear familias de productos bancarios

2.01.1 Ejercicio: parchís

Suponga que debe Vd. desarrollar una versión electrónica de un juego de mesa (p.ej., del parchís), y que pretende que el jugador pueda elegir el aspecto de las fichas y del tablero: estilo clásico, estilo Pokemon, estilo Manga, etc. Utilice el patrón *Fábrica abstracta* para construir las diferentes versiones del aspecto visual del juego.

2.02 Builder (constructor)

El patrón Builder se utiliza para crear objetos complejos. Podemos replantear el mismo ejemplo utilizado con la Fábrica Abstracta para utilizar el patrón Builder. Del enunciado se deduce que depende de la *Cuenta* que le creamos al cliente los tipos de

tarjetas y regalos que se creen. Podemos representar esto mediante un *Builder* de la siguiente manera:

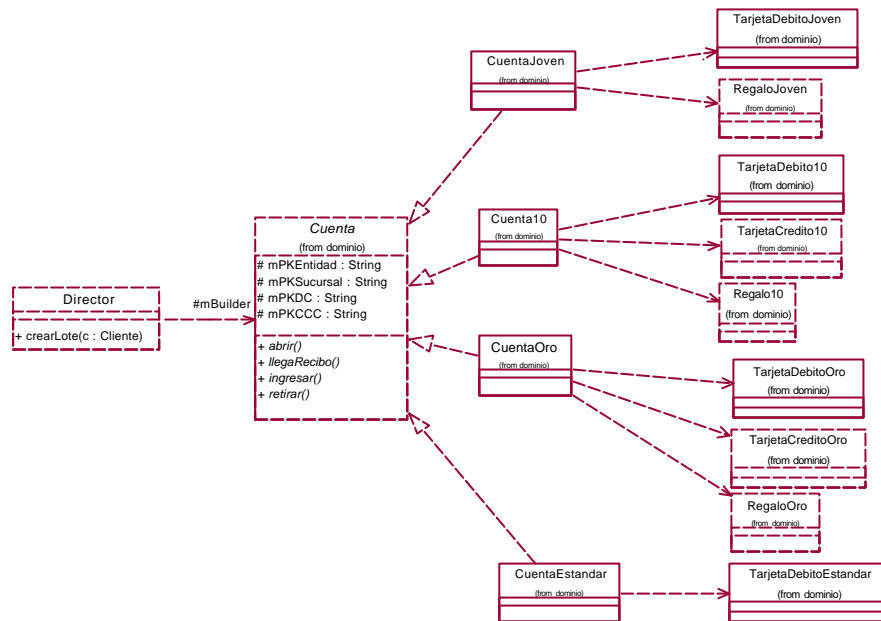


Figura 126. Un Builder para crear la familia de productos bancarios

En la figura anterior, la clase *Cuenta* hace las veces de constructor abstracto (de *Builder* abstracto). Obsérvese también que hemos añadido una nueva clase (*Director*), que es la encargada de llamar al constructor, una vez instanciada.

Con esta estructura, la creación de la familia de productos podría residir en el siguiente método:

```

public class Director {
    Cuenta mBuilder;
    ...
    void crearLote(Cliente c) {
        if (c.getTipo()==CLIENTE_JOVEN)
            mBuilder=new CuentaJoven(c);
        else if (c.getTipo()==CLIENTE_10)
            mBuilder =new Cuenta10();
        else if (c.getTipo()==CLIENTE_ORO)
            mBuilder =new CuentaOro(c);
        else
            mBuilder =new CuentaEstandar(c);
        mBuilder.abrir();
    }
}
  
```

Cada constructor concreto (clases *CuentaJoven*, *Cuenta10*, *CuentaOro* y *CuentaEstandar*) es el responsable de crear los objetos correspondientes, lo cual se hace en el

método *abrir()*, que recibe diferentes implementaciones en cada clase, como por ejemplo:

```
public class CuentaJoven extends Cuenta {
    ...
    public void abrir() throws Exception {
        TarjetaDebitoJoven t=new TarjetaDebitoJoven(c);
        RegaloJoven r=new RegaloJoven(r);
    }
}

public class CuentaOro extends Cuenta {
    ...
    public void abrir() throws Exception {
        TarjetaDebitoOro t=new TarjetaDebitoOro(c);
        TarjetaCreditoOro t=new TarjetaCreditoOro(c);
        RegaloOro r=new RegaloOro(r);
    }
}
```

2.02.1 Ejercicio: parchís

Plantee el ejercicio del parchís propuesto en la página 140 con el patrón *Builder*.

2.02.2 Ejercicio: alternativas de diseño e implementación

Si tuviera que diseñar el sistema bancario que venimos describiendo, ¿con cuál de estos dos diseños se quedaría? ¿Por qué? ¿Cuál piensa que daría más problemas de mantenimiento?

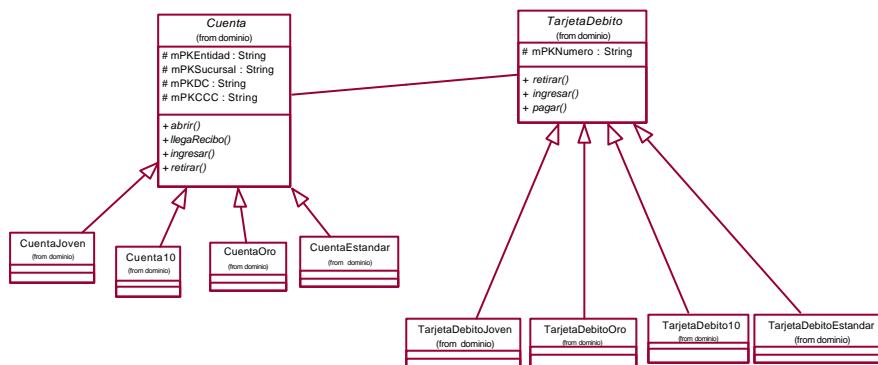


Figura 127. Alternativa A.

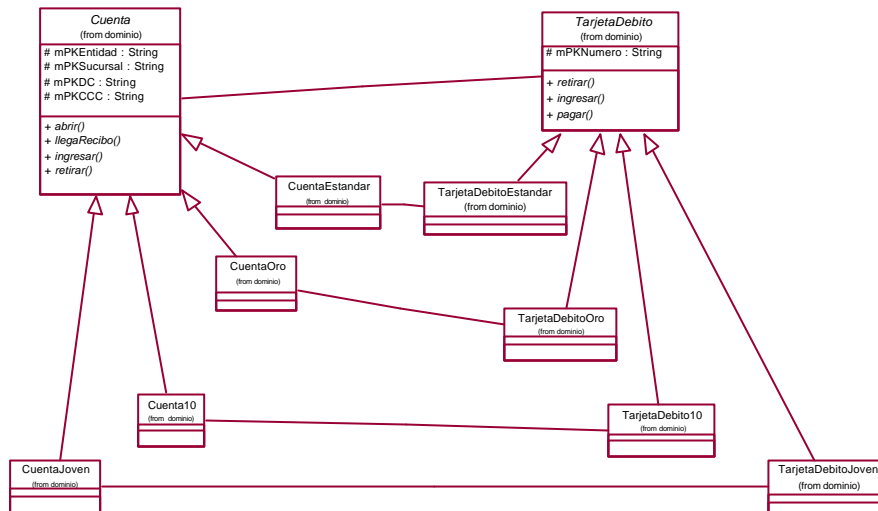


Figura 128. Alternativa B

3. Patrones estructurales

3.01 Patrón *Adapter* (Adaptador)

Cuando una clase *A* necesita utilizar los servicios prestados por otra clase *B*, pero la interfaz ofrecida por *B* no es la esperada por *A*, se puede utilizar el patrón *Adaptador* para adaptar la interfaz de *B* a lo que espera *A*.

Supongamos, en nuestro sistema bancario, que la clase *Tarjeta* posee un método *transferir(float importe, String cuentaDestino)* que se utiliza para realizar transferencias. Supongamos también que el banco ha adquirido una partida de un modelo novedoso de cajeros automáticos, entre cuyas posibilidades se encuentra esta posibilidad. Sin embargo, la interfaz esperada por el cajero para realizar esta operación es *order(String target, float amount)*. Para que nuestras instancias de *Tarjeta* respondan adecuadamente a los nuevos cajeros, podemos o bien modificar la clase *Tarjeta* añadiéndole el nuevo método, o bien utilizar el patrón *Adaptador*.

Con la primera solución haríamos, más o menos, lo siguiente:

```
public abstract class Tarjeta {
    ...
    protected Cuenta mCuentaAsociada;

    // Este método ya existía
    public void transferir(float importe, String cuentaDestino)
        throws Exception {
        if (mCuenta.getSaldo() < importe)
            throw new Exception("Saldo insuficiente");
        if (!mCuenta.getBloqueada())
```

```

        throw new Exception("La cuenta está bloqueada");
        mCuenta.transferir(importe, cuentaDestino);
    }

    // Añadimos este otro para que los cajeros puedan comunicar con
    // nuestras tarjetas
    public void order(String target, float amount) throws Exception {
        transferir(amount, target);
    }
}

```

La solución utilizando el patrón *Adaptador* es mucho más elegante y no precisa que toquemos el código fuente de la clase de dominio (*Tarjeta*, en este caso); en efecto, construiremos una clase nueva que traducirá las llamadas de una forma a otra. En su forma más simple, la solución podría ser la que se muestra en estas dos figuras:

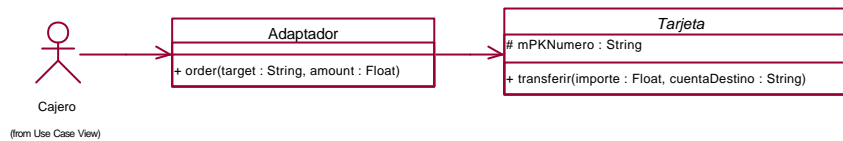


Figura 129. Estructura del *Adaptador*, muy simple, para convertir llamadas

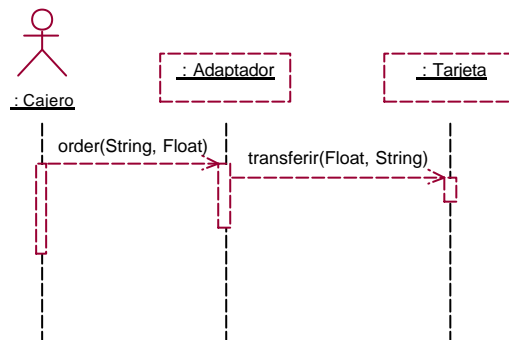


Figura 130. Funcionamiento, muy sencillo, de la conversión

Supongamos ahora que el problema ocurra también con la clase *Cuenta*: es decir, la *Cuenta* posee el mismo método *transferir(float importe, String cuentaDestino)*, pero el cajero también ejecuta la operación *order(String target, float amount)*. En este caso, el *Adaptador* podría ser una clase abstracta, que sería instanciada a un *AdaptadorDeTarjeta* o *AdaptadorDeCuenta* según el caso:

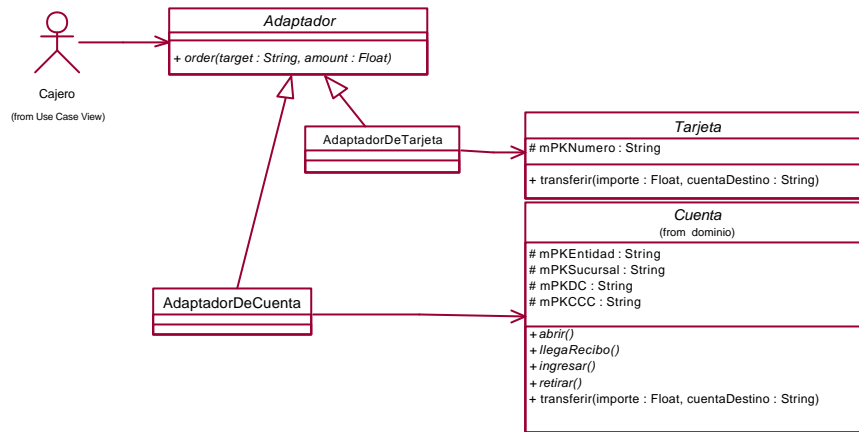


Figura 131. Estructura del *Adaptador*, para adaptar la llamada a dos clases

3.02 Patrón *Composite* (Compuesto)

En ocasiones, un objeto está formado por objetos que, a su vez, poseen en su interior objetos que son de la clase del objeto original. En el procesador de textos Microsoft Word, por ejemplo, es posible agrupar varios objetos de dibujo en un “grupo”; este grupo se maneja como un único objeto, y puede ser también agrupado junto a otros objetos, de forma que un grupo puede contener, mezclados, objetos simples y grupos. La Figura 132(a) muestra cuatro objetos sencillos dibujados con Word; en el lado izquierdo de (b) creamos un grupo a partir de dos objetos, lo que se muestra en el lado derecho; en el lado izquierdo de (c), creamos un nuevo grupo formado por el grupo creado anteriormente y un nuevo objeto simple, cuyo resultado se muestra a la derecha.

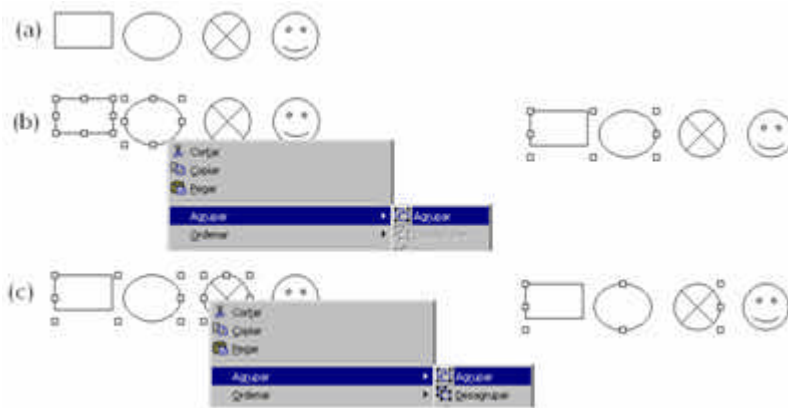


Figura 132. Ejemplo claro del patrón *Composite*

La solución general para este tipo de problemas toma la siguiente forma:

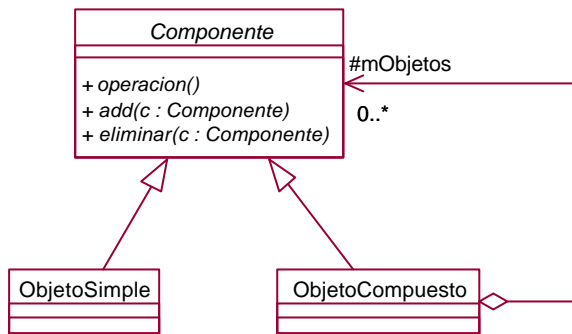


Figura 133. Solución general para el Composite

Las dos subclases implementan las operaciones abstractas definidas en *Componente*; en *ObjetoCompuesto*, la ejecución de *operacion()* llama a *operacion()* de cada uno de sus componentes:

```

public class ObjetoCompuesto extends Componente {
    ...
    protected Vector mObjetos;
    public void operacion() {
        for (int i=0; i<mObjetos.size(); i++) {
            Componente c=(Componente) mObjetos.elementAt(i);
            c.operacion();
        }
    }
}

```

En el ejemplo de los objetos de dibujo, la clase *ObjetoSimple* podría ser a su vez una clase abstracta de la que heredaran otras clases como *Rectángulo*, *Círculo*, *Cara-Sonriente*, etc.

3.03 Patrón *Facade* (Fachada)

Este patrón proporciona un mecanismo común de acceso a las clases incluidas en un subsistema.

Supongamos que, en el sistema bancario, deseamos dotar a los desarrolladores de la capa de presentación de un mecanismo unificado de acceso a la capa de dominio. Podríamos colocar una *Fachada* como punto de acceso a las clases de la capa de dominio; la interfaz de la fachada (al decir ahora “interfaz” nos estamos refiriendo a su conjunto de operaciones públicas) incluiría un conjunto de operaciones que llamarán a las operaciones adecuadas de los objetos de la capa de dominio:

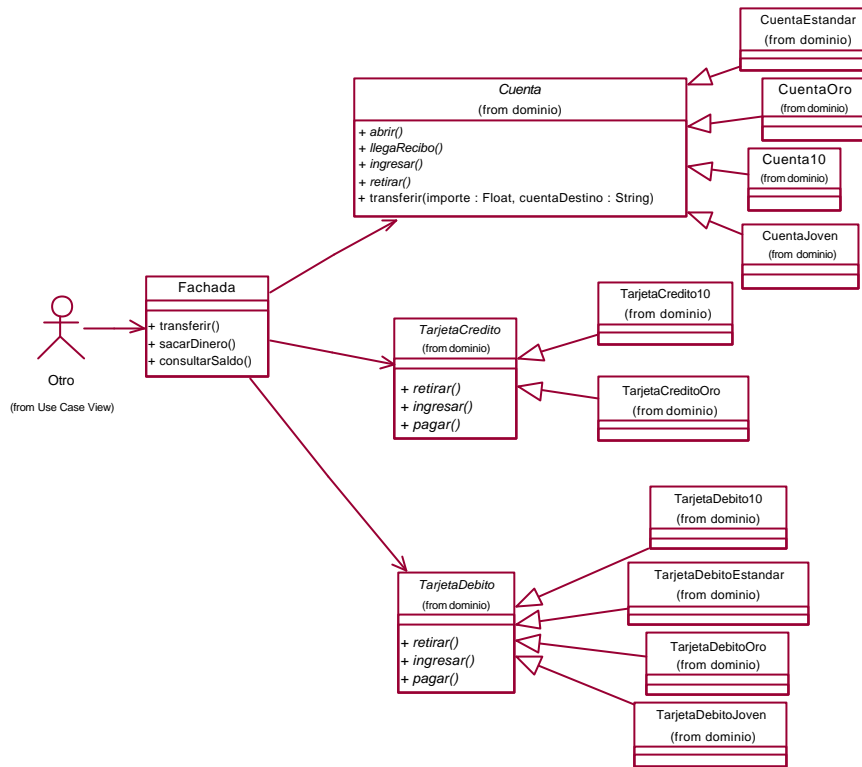


Figura 134. Colocación de una fachada para dar un único punto de acceso a un subsistema

La *Fachada*, por tanto, define la interfaz de acceso al subsistema, ayudando a garantizar la encapsulación de éste. En la figura anterior, tal vez no haya operaciones públicas en las clases situadas a la derecha de *Fachada* (puede restringirse la visibilidad máxima al nivel de subsistema), por lo que las clases de fuera de esta capa no tendrían acceso a estas operaciones. La *Fachada* es el mecanismo idóneo para ofrecer al exterior los servicios que resulten de interés.

3.04 Patrón *Flyweight* (Peso mosca)

En muchas ocasiones, existen en un sistema orientado a objetos muchas instancias de clases que son “demasiado pequeñas” (por ejemplo, puede considerarse que cada carácter de un texto es un objeto de clase *Carácter* del procesador de textos). Un número elevado de instancias hará que caiga el rendimiento del sistema, por lo que sería bueno pensar en la utilización de una alternativa.

Esta alternativa puede pasar por la utilización del patrón *Peso mosca*, mediante el cual, en lugar de crearse tantos objetos como aparentemente se requieren (en el ejemplo del procesador de textos, tantos como número de caracteres haya en el texto), se crean tantos objetos como objetos “esencialmente distintos” hay (en el mismo ejemplo, se crearían tantos objetos como tipos de caracteres haya en el texto: a, b, c... A, B, C...). Cada *peso mosca* posee dos estados, el *intrínseco* y el *extrínseco*:

- El estado *intrínseco* es común a todas las instancias de la clase.
- El estado *extrínseco* depende del contexto en el que se ubica la instancia, por lo que no es compartido con el resto de instancias.

3.04.1 Ejemplo: las fichas del ajedrez

En el juego del ajedrez, cada jugador tiene ocho peones, dos torres, dos caballos, dos alfiles, una reina y el rey. Una posible forma de representarlo, en la que omitimos la mayoría de los atributos y las operaciones, es la siguiente:

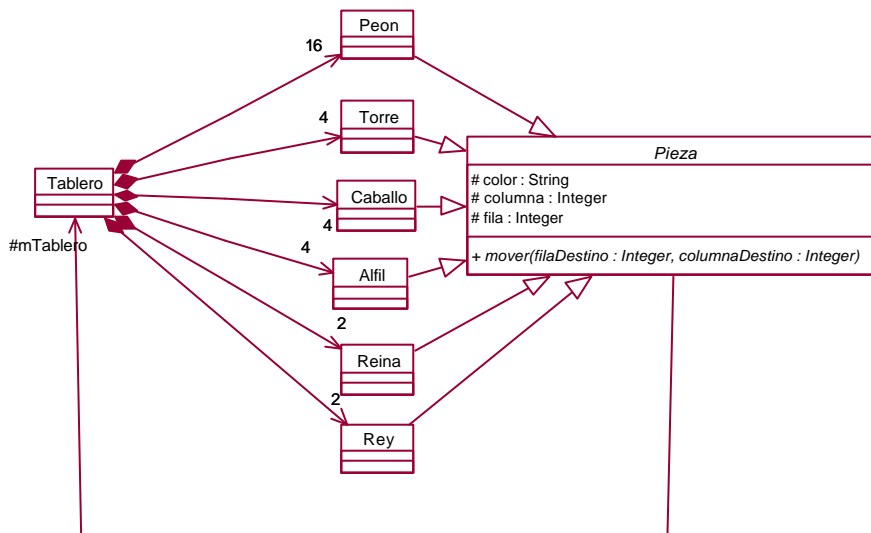


Figura 135. Un posible diseño del tablero de ajedrez

Con este diseño, la operación *mover* toma únicamente las coordenadas de la casilla de destino; puesto que cada pieza conoce el *Tablero* sobre la que se encuentra y las coordenadas en las que se encuentra, la pieza tiene información suficiente para realizar el movimiento.

Obsérvese, como apunte adicional, que hemos representado mediante *composiciones* (véase página 38) las relaciones del tablero con los diferentes tipos de piezas. Esta

solución no está mal, pero supone la creación de 32 instancias de piezas por partida; si esto corriera en un servidor que permitiera jugar a múltiples jugadores múltiples partidas (como es el caso del portal www.ajedrez21.com), el número de objetos en memoria sería demasiado grande.

Si utilizamos el patrón *Peso mosca*, no tendremos 32 instancias por partida, sino sólo seis (una por tipo de pieza) o, incluso, seis instancias para todas las partidas. Con el *Peso mosca*, se dota a una clase de la responsabilidad de conocer el estado *extrínseco* de cada instancia; el estado *intrínseco* vendría dado por el tipo de pieza. La idea del estado *extrínseco* es la eliminación de aquella información que puede calcularse desde otro lado: en nuestro ejemplo, lo interesante de cada tipo de pieza es la forma en que se mueve, incluyendo el color de la pieza, las coordenadas origen y destino y el estado del *Tablero*. Esta información puede ser mantenida en una clase aparte, que bien puede ser el propio *Tablero*.

En la siguiente figura, el *Tablero* conoce a una única instancia de cada tipo de pieza (cada tipo de pieza también conoce al tablero). La signatura de la operación *mover(...)* ha cambiado con respecto a la Figura 135: ahora, pasamos como argumentos no sólo las coordenadas de destino, sino también las de origen y el color de la pieza. Estos tres parámetros adicionales constituyen el mencionado estado *extrínseco*, que es justo lo que necesita la pieza para completar su movimiento. Por otro lado, como estado *intrínseco* queda únicamente la pertenencia de la instancia a una determinada subclase de *Pieza*, y el tablero sobre el que está situada la pieza en cuestión.

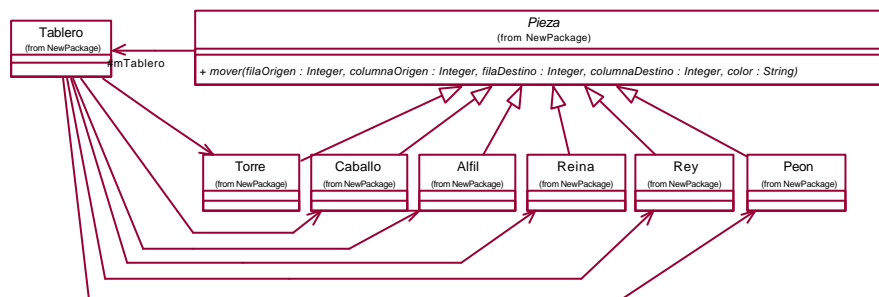


Figura 136. El diseño del tablero de ajedrez, utilizando un *Peso mosca*

Haciendo una correspondencia con la estructura original del patrón *Flyweight*, la clase *Piezas* corresponde con la clase *PesoMosca*, el *Tablero* con el *Cliente* y cada una de las subclases de *Pieza*, con los *PesoMoscaConcreto*. En el patrón original existe una clase auxiliar que se encarga de crear las instancias de *PesoMosca*: en rigor, *Tablero*

conocería a una clase *FabricaDePesosMosca* que crearía los pesos mosca al principio, y que devolvería al *Cliente* aquéllos que éste le fuera solicitando. En la siguiente figura, “alguien” (un actor, una ventana u otro objeto que, en este contexto, no nos interesa) solicita al *Tablero* que mueva un peón blanco desde la posición (2,5) a la (4,5). El *Tablero* pide a la fábrica que le devuelva la instancia de *Peón* (ponemos “la” en cursiva porque existe sólo una instancia de esta clase); cuando el *Tablero* posee por fin la referencia, le pide al *Peón* que se mueva pasándole la información de su estado extrínseco.

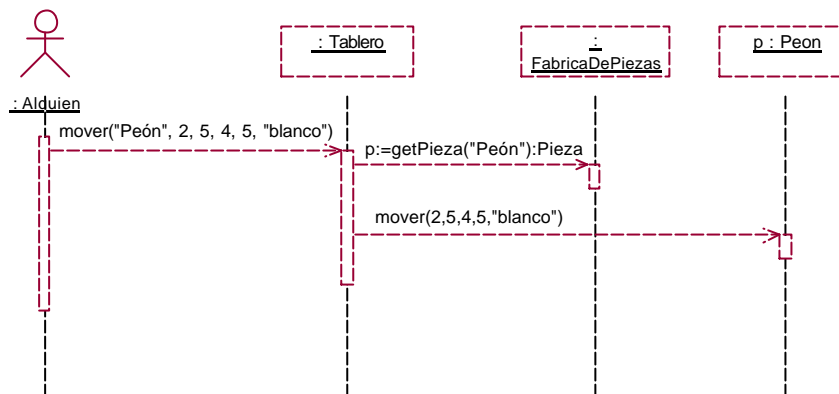


Figura 137. El *Tablero* solicita a la fábrica una instancia de *Peón*

Evidentemente, la utilización de pesos mosca supone un importante ahorro de espacio de memoria, pero conlleva un incremento de los costes de cálculo, debidos fundamentalmente al procesamiento del estado extrínseco. Como en todo, el diseñador deberá saber valorar las ventajas y desventajas de cada alternativa, que en este caso son dos:

- ¿Más memoria y menos lentitud (Figura 135)?
- ¿Más rapidez y menos memoria (Figura 136)?

3.05 Patrón *Proxy*

Un *Proxy* es una clase que sustituye a otro objeto, de manera que controla el acceso a éste. El sustituido puede no ser realmente un objeto, sino un dispositivo físico, un sistema externo, etc. El *Agente de base de datos* (véase página 54) constituye realmente un proxy que viene a sustituir al sistema de acceso al gestor de bases de datos.

4. Patrones de comportamiento

4.01 Patrón *Chain of responsibility* (cadena de responsabilidad)

Este patrón se utiliza para desacoplar al emisor de un mensaje del receptor de éste: el emisor envía el mensaje a un receptor, que lo va pasando a otros hasta que alguien se hace cargo del mensaje.

La estructura básica de este patrón se muestra en la siguiente figura: en ella, se define una clase abstracta *Gestor* en la que se incluye una operación *solicitud()* que heredan todas sus especializaciones. La idea es que alguien pueda ejecutar la operación en uno cualquiera de los objetos de estas clases: si el objeto es capaz de hacerse cargo de la operación, la ejecuta; en otro caso, la pasa a su *sucesor*, asociación definida en la superclase y que es también heredada por las subclases.

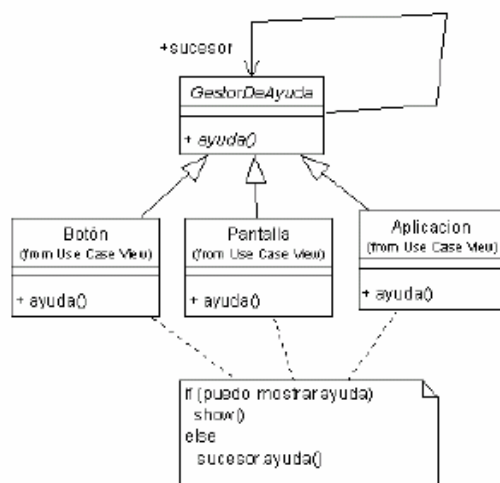


Figura 138. Ejemplo del patrón *Chain of responsibility*

4.02 Patrón *Mediator* (Mediador)

El Mediador es una clase que puede recibir mensajes de múltiples clases, y que los comunica al resto de objetos. Viene a ser, por tanto, un observador que comunica a múltiples instancias de múltiples clases.

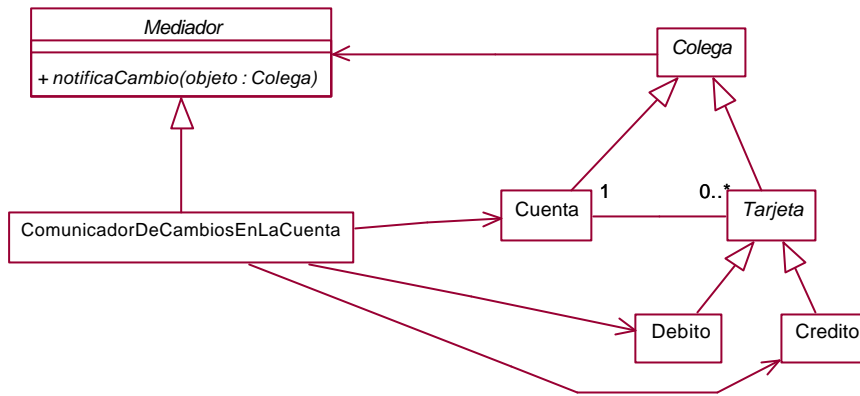


Figura 139. Ejemplo del patrón *Mediator*

4.03 Patrón *Interpreter* (Intérprete)

El patrón Intérprete se utiliza en situaciones en las que debe procesarse algún tipo de lenguaje sencillo. La solución propuesta consiste en disponer de una clase abstracta que representa una expresión genérica del lenguaje. A partir de esta clase *Expresión*, iremos instanciando clases que procesan los diferentes elementos del lenguaje. *Expresión* incluye la operación abstracta *interpretar(Contexto)*, que es redefinida en cada una de sus especializaciones.

De forma general, se usa una clase por cada regla de la gramática. Los símbolos del lado derecho se representan como propiedades de cada una de las clases.

Supongamos que, en el sistema bancario, es posible realizar algunas operaciones sobre las cuentas con un lenguaje de comandos sencillo, que viene dado por la siguiente gramática:

```

operación      : retirada | ingreso | transferencia
retirada       : 'retirar' CANTIDAD CUENTA
ingreso        : 'ingresar' CANTIDAD CUENTA
transferencia  : 'transferir' CANTIDAD CUENTA CUENTA CONCEPTO
  
```

Aplicando el patrón Intérprete y un poco de heurística, podemos obtener la siguiente jerarquía de clases:

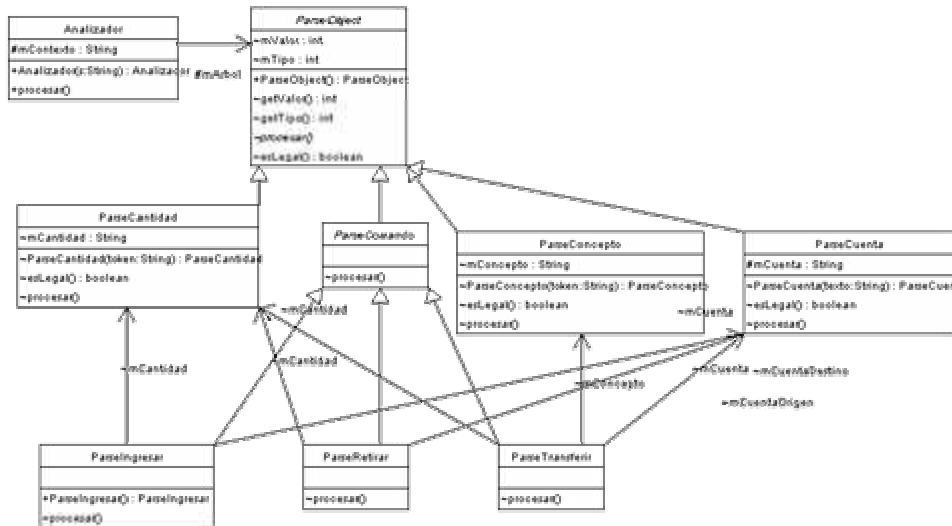


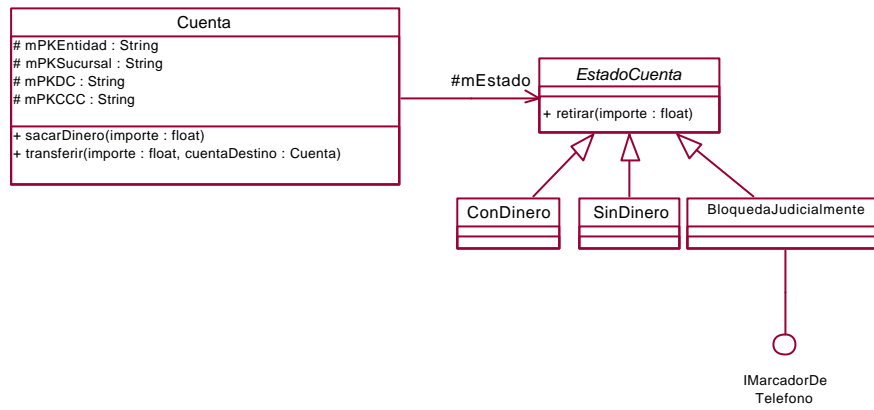
Figura 140. Ejemplo del patrón Intérprete

Como se observa, se ha añadido al diagrama una clase auxiliar, el *Analizador*, que se encarga de la construcción del árbol sintáctico que representa el texto que se quiere analizar, y de llamar posteriormente al método *procesar* de cada expresión.

4.04 Patrón State(Estado)

Mediante el patrón Estado, delegamos parte del estado de un objeto a un objeto auxiliar, de un tipo abstracto *Estado*, que se instancia al estado concreto del objeto. Este patrón se utiliza cuando hay una parte importante del comportamiento de un objeto que depende de su estado.

En el sistema bancario, podemos delegar el comportamiento de la operación *retirar* de la clase *Cuenta* a una clase asociada *EstadoCuenta* con tres especializaciones: *ConDinero*, *SinDinero* y *BloqueadaJudicialmente* (Figura 141). Cuando se ejecuta la operación *retirar* en una instancia de *Cuenta*, se llama a *retirar* en el objeto de clase *EstadoCuenta* asociado. Además, tanto el cambio de estados como la adición de nuevos estados son muy sencillas, y no conllevan normalmente una modificación del código de la clase *Cuenta*.

**Figura 141.** Ejemplo del patrón Estado

CAPÍTULO 11. PRUEBAS DE SISTEMAS ORIENTADOS A OBJETO

1. Introducción

La fase de pruebas es una de las etapas más costosas del ciclo de vida software. Algunos estudios indican que las pruebas consumen más del 50% de los costes totales de la vida de un producto software.

En este capítulo veremos dos enfoques diferentes para la realización de pruebas en sistemas orientados a objetos: pruebas mediante mutación y la estrategia de pruebas de eXtreme Programming (Programación Extrema o, abreviadamente, XP).

2. Pruebas mediante mutación

En el contexto de las pruebas del software, un mutante es una copia del programa que se está probando (programa “original”) al que se le ha introducido un único y pequeño cambio sintáctico (por ejemplo, cambiar un signo + por un *). Supongamos que ejecutamos un mismo caso de prueba en el programa original y en el mutante, y supongamos además que la salida del programa original es correcta, y que la salida del mutante difiere en algo de la salida del original. Entonces, podremos asegurar que, para ese caso de prueba, la instrucción mutada es correcta en el programa original. Si a ambos programas les pasamos una batería de casos de prueba y siempre obtuviéramos resultados correctos en el original e incorrectos en el mutante, podríamos afirmar con poco temor a equivocarnos que la instrucción mutada de nuestro programa es correcta.

Desde luego, puede en muchos casos demostrarse la corrección de algoritmos mediante métodos matemáticos y formales, pero es un proceso ciertamente costoso que las pruebas mediante mutación (también muy costosas, pero muy automatizables) vienen en cierto modo a suplir.

2.01 Terminología

Sean P un programa y M un mutante de P , y sean $f(P, c)$, $f(M, c)$ las salidas de los programas P y M con el caso de prueba c : se dice que M está *vivo* si $f(M, c) = f(P, c)$; en otro caso, se dice que M está *muerto*. En este caso, se dice que P *mata* a M con c .

Evidentemente, dado un programa P y una familia $\{M_1, M_2, \dots, M_n\}$ de mutantes de P , cuantos más mutantes mate P , más seguros estaremos de la corrección de P . Un mutante M_i puede quedar vivo por dos razones:

1) Porque M_i sea *funcionalmente equivalente* a P : es decir, porque ambos programas producen siempre las mismas salidas. Sería el caso de mutar el operador $*$ por $/$ en la instrucción $x=y*1$.

2) Porque, aunque ambos programas no sean funcionalmente equivalentes, no se ha encontrado ningún caso de prueba c que mate a M_i .

Dado que, para un programa P se generan muchos mutantes, es muy difícil asegurarse de que un programa y uno de sus mutantes son funcionalmente equivalentes, por lo que se asume que lo son cuando ofrecen las mismas salidas con un conjunto amplio de casos de prueba.

2.02 Generación de mutantes

La generación de mutantes se consigue aplicando *operadores de mutación* al código fuente del programa que queremos probar.

Algunos operadores de mutación clásicos son los siguientes:

Operador	Descripción
ABS	Sustituir una variable por el valor absoluto de dicha variable
ACR	Sustituir una referencia variable a un array por una constante
AOR	Reemplazamiento de un operador aritmético
CRP	Reemplazamiento del valor de una constante
ROR	Reemplazamiento de un operador relacional
RSR	Reemplazamiento de la instrucción Return
SDL	Eliminación de una sentencia
UOI	Inserción de operador unario (p.ej.: en lugar de x , poner $-x$)

Tabla 4. Algunos operadores de mutación clásicos

Como puede uno imaginarse, el número de mutantes que puede generarse a partir de un programa sencillo, de pocas líneas de código, es muy grande. Por ello, se han realizado algunos estudios que han evidenciado que, aplicando pocos operadores, se pueden conseguirse los mismos resultados que si se aplicaran muchos. Los operadores más efectivos son el *ABS*, *0* (sustitución de una variable por el valor 0), *<0* (sustitución de una variable por un valor menor que 0), *>0* (sustitución de una variable por un valor mayor que 0), *AOR*, *ROR* y *UOI*.

Además, también se han propuesto operadores específicos para programas escritos en lenguajes orientados a objeto, algunos de los cuales se muestran en la Tabla 5.

Operador	Descripción
AMC (Access Modifier Change)	Reemplazo del modificador de acceso (por ejemplo: ponemos <i>private</i> en lugar de <i>public</i>)
AOC (Argument Order Change)	Cambio del orden de los argumentos pasados en la llamada a un método (p.ej.: en lugar de <i>Persona p=new Persona("Paco", "Pil")</i> poner <i>Persona p=new Persona("Pil", "Paco")</i>)
CRT (Compatible Reference Type Replacement)	Sustituir una referencia a una instancia de una clase por una referencia a una instancia de una clase compatible (p.ej.: en vez de poner <i>Persona p=new Empleado()</i> , poner <i>Persona p=new Estudiante()</i>).
EHC (Exception Handling Change)	Cambiar una instrucción de manejo de excepciones (try...catch) por un sentencia que propague la excepción (throw), y viceversa
EHR (Exception Handling Removal)	Eliminación de una instrucción de manejo de excepciones
HFA (Hiding Field variable Addition)	Añadir en la subclase una variable con el mismo nombre que una variable de su superclase
MIR (Method Invocation Replacement)	Reemplazar una llamada a un método por una llamada a otra versión del mismo método
OMR (Overriding Method Removal)	Eliminar en la subclase la redefinición de un método definido en una superclase
POC (Parameter Order Change)	Cambiar el orden de los parámetros en la declaración de un método (p.ej.: poner <i>Persona(String apellidos, String nombre)</i> en vez de <i>Persona(String nombre, String apellidos)</i>)
SMC (Static Modifier Change)	Añadir o eliminar el modificador <i>static</i>

Tabla 5. Algunos operadores de mutación para orientación a objetos

Los operadores de mutación deben ser aplicados según el contexto del lugar del código fuente en el que se vaya a realizar la mutación. Si esto no se hiciera así, podrían generarse multitud de mutantes funcionalmente equivalente al programa original, otros que ni siquiera compilarían, etc. A modo de ejemplo, obsérvese el siguiente fragmento de código:

```
import java.util.*;

public class Persona {
    protected String mPKNombre, mPKApellidos;
    protected float mSalario;

    /** Construye una instancia de clase persona con el nombre y apellidos
     *  pasados como parámetro
     */
    public Persona(String n, String a) {
        mPKNombre=n;
        mPKApellidos=a;
    }

    public String getNombreCompleto() {
        return mPKNombre+" "+mPKApellidos;
    }
}
```

Si le aplicamos, por ejemplo, el operador AOR, podremos obtener mutantes en los que se han cambiado los asteriscos de la sentencia `import` o de los comentarios por un signo `+`: el primero no compilaría, y los restantes serían funcionalmente equivalentes al

programa original. Lo mismo ocurriría con los signos + del método `getNombreCompleto()`, que se utilizan en el programa original como operadores de concatenación pero que podrían ser malinterpretados como operadores aritméticos por el generador de mutantes.

Para implementar un generador de mutantes, una solución pasa por el uso de un patrón Intérprete (página 152) que mute o no dependiendo del contexto. Aunque el Intérprete está recomendado sólo para el procesamiento de pequeños lenguajes, en este caso podría ser utilizado porque prácticamente sólo se necesita información del contexto para realizar o no la mutación (dependiendo del operador de mutación, puede ser necesario otra información, como el tipo de las variables que intervienen en la expresión que se muta).

2.03 Pruebas de caja negra mediante mutación de interfaces

Las técnicas de mutación también pueden utilizarse para realizar pruebas de caja negra de clases o componentes. La idea consiste en mutar en el cliente las llamadas a los servicios ofrecidos (el conjunto de métodos públicos) por la clase o componente que se está probando. No obstante, puede también mutarse en la clase que se prueba, si bien en este caso tendríamos que disponer de su código fuente.

Algunos operadores de mutación aplicables a pruebas de caja negra son los siguientes:

Operador	Descripción
Swap	Intercambiar los parámetros de tipos compatibles en la llamada al método.
Twiddle	Reemplazar una variable x de tipo numérico o carácter por $succ(x)$ o $pred(x)$
Set	Prefijar el valor de un parámetro. Estos valores pueden ser escogidos apoyándonos en técnicas como los valores límite, conjetura de errores, etc.
Nullify	Anular el valor de algún parámetro

Tabla 6. Operadores para mutar interfaces

En este tipo de mutación, se pasa el caso de prueba a la clase “buena” y a los mutantes; para detectar los mutantes vivos y muertos es necesario poder observar la salida del componente o interrogar a éste acerca de su propio estado.

3. La estrategia de pruebas de Programación Extrema

Programación Extrema aboga por que se escriban pruebas antes de codificar, y se aboga además por que las escriban los propios programadores. En particular, éstos deben escribirse pruebas cuando se dé alguna de las siguientes circunstancias:

- a) Cuando la interfaz de un método no esté clara, debe escribirse una prueba antes de escribir el método.
- b) Cuando, aunque la interfaz de un método esté clara, no lo está su implementación o se sospecha que ésta puede ser algo complicada, se escribirá una prueba antes de escribir el método.
- c) Cuando pensemos que alguna circunstancia extraña puede afectar al funcionamiento del código, se escribirá una prueba simulando esa circunstancia.
- d) Cuando se encuentra un problema en un código ya escrito, se escribirá una prueba que reproduzca el problema.
- e) Cuando vayamos a modificar un código ya existente y no estemos seguros de cómo va a comportarse, escribiremos un caso de prueba ex profeso para esto (puede ocurrir, no obstante, que tal caso de prueba ya existiera y podamos encontrarlo en un repositorio).

Como se verá en un próximo capítulo, los clientes deben también describir pruebas de tipo funcional para cada escenario de la aplicación.

3.01 Pruebas utilizando *junit*

junit es un “framework” que puede descargarse gratuitamente de www.junit.org. En este framework se define un conjunto amplio de clases que automatizan parcialmente la ejecución de pruebas unitarias.

Siendo por ejemplo *A* la clase que se va a probar, se construye una clase de nombre *testA* en la que se implementa un conjunto de métodos dirigidos a realizar pruebas. En cada uno de estos métodos se crea manualmente un objeto de clase *A*, que se corresponderá con el resultado esperado de la prueba; también en cada método se construye un objeto de clase *A* ejecutando las operaciones definidas en esta clase, que se corresponde con el resultado obtenido. Si el objeto construido manualmente es igual que el construido automáticamente, entonces la prueba ha sido superada.

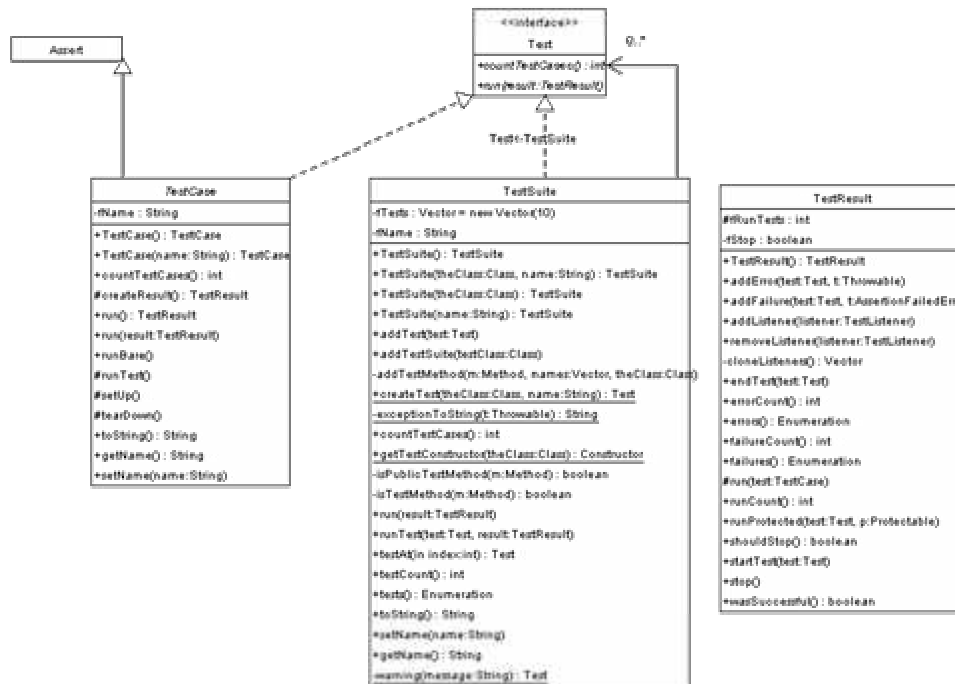


Figura 142. Algunas de las clases incluidas en *junit*

La clase *testA* que hemos mencionado debe ser una especialización de *TestCase* (Figura 142). Como se observa, *TestCase* hereda a su vez de la clase *Assert*, cuyos miembros se muestran en la siguiente figura:

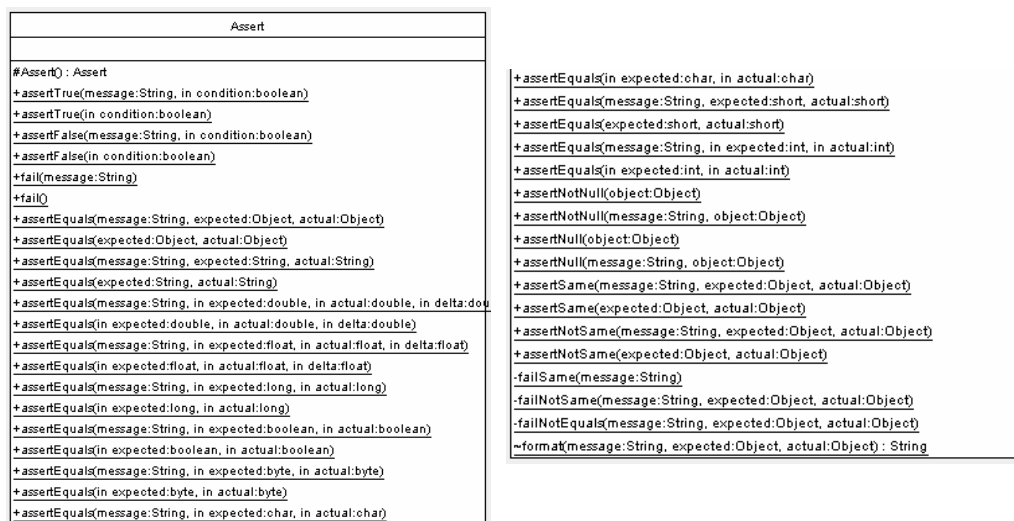


Figura 143. Estructura de la clase Assert

Assert incluye una larga lista de operaciones utilizables para comparar objetos de varios tipos y de varias formas.

3.02 Ejemplo

Supongamos que disponemos de la clase `Lista` cuyo código aparece a continuación:

```
package dominio;

import java.util.Vector;

public class Lista extends Vector {
    public Lista() {
        super();
    }

    public Lista(String[] elementos) {
        this();
        if (elementos==null) return;
        for (int i=0; i<elementos.length; i++) addElement(elementos[i]);
    }

    public Lista ordenar() {
        if (size()==0)
            return this;
        ordenar(0, size()-1);
        return this;
    }

    protected void ordenar(int iz, int de) {
        ...
    }
}
```

Esta clase representa una lista de cadenas que puede ser ordenada mediante la ejecución de su operación pública *ordenar* (que llama, a su vez, a la operación protegida del mismo nombre). Para probar el funcionamiento de esta clase construimos una nueva clase *TestLista* con un conjunto de métodos *testXXX*, cada uno de los cuales prueba una cierta funcionalidad.

TestLista consta de una serie de atributos (*listaTodosIguales*, *listaOrdenada*, etc.) que se utilizan posteriormente para comparar los resultados esperados con los obtenidos. En cada método *testXXX* se construye manualmente una instancia de la clase que se está probando y que representa el resultado esperado (el objeto llamado *expected*); igualmente, construimos un objeto que es el resultado obtenido llamando a métodos de la clase que se está probando (por ejemplo: en el método *testOrdenarTodosIguales* el resultado realmente obtenido es el que obtiene al ejecutar *listaTodosIguales.ordenar*). Cuando ya tenemos ambos objetos, los comparamos utilizando alguno de los métodos *assert* que *TestLista* hereda de *Assert* a través de *TestCase* (Figura 144).

```

package dominio;

import junit.framework.*;

public class TestLista extends TestCase {
    Lista listaTodosIguales;
    Lista listaOrdenada;
    Lista listaAlReves;
    Lista listaNulal;
    Lista listaNula2;
    Lista listaVacía;

    public TestLista(String nombre) {
        super(nombre);
    }

    public void setUp() {
        String[] e1={"a", "a", "a", "a", "a"};
        listaTodosIguales=new Lista(e1);
        String[] e2={"a", "b", "c", "d", "e"};
        listaOrdenada=new Lista(e2);
        String[] e3={"e", "d", "c", "b", "a"};
        listaAlReves=new Lista(e3);
        listaNulal=null;
        String[] e4=null;
        listaNula2=new Lista(e4);
        String[] e5={};
        listaVacía=new Lista(e5);
    }

    public void testOrdenarTodosIguales() {
        String[] ex={"a", "a", "a", "a", "a"};
        Lista expected=new Lista(ex);
        this.assertEquals(expected, listaTodosIguales.ordenar());
    }

    public void testOrdenarReves() {
        String[] ex={"a", "b", "c", "d", "e"};
        Lista expected=new Lista(ex);
        this.assertEquals(expected, listaAlReves.ordenar());
    }

    public void testOrdenarNulal() {
        this.assertNull(listaNulal);
    }

    public void testOrdenarNula2() {
        String[] ex=null;
        Lista expected=new Lista(ex);
        this.assertEquals(expected, listaNula2.ordenar());
    }

    public void testOrdenarListaVacía() {
        String[] ex={};
        Lista expected=new Lista(ex);
        this.assertEquals(expected, listaVacía.ordenar());
    }
}

```

La estructura que, en general, utilizaremos siempre que deseemos realizar pruebas con *junit* se muestra en la siguiente figura:

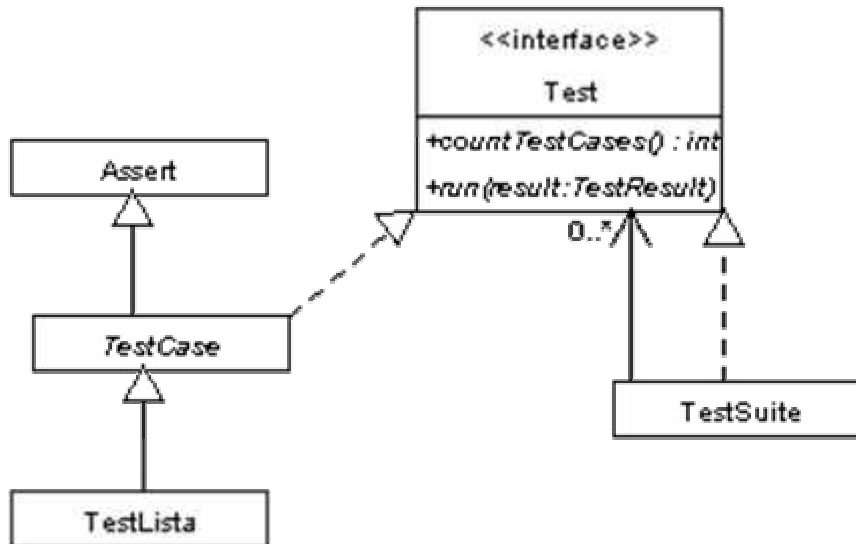


Figura 144. Relación de nuestro caso de prueba con el resto de clases importantes de *junit*

La clase *TestSuite* nos sirve para crear grupos de casos de prueba: un *TestSuite* tiene un conjunto de *Tests* (nótese la asociación), y tanto los *TestCase* como los propios *TestSuite* son *Tests*. Esta estructura responde a un patrón *Composite* (véase página 145). El siguiente código, añadido a la clase *TestLista*, construye varios grupos de casos de prueba:

```

public static TestSuite suite() {
    TestSuite raiz=new TestSuite("raíz");
    TestSuite suite1=new TestSuite("Iguales");
    suite1.addTest(new TestLista("testOrdenarTodosIguales"));
    TestSuite suite2=new TestSuite("Al revés");
    suite2.addTest(new TestLista("testOrdenarReves"));
    TestSuite suite3=new TestSuite("Nulas o vacías");
    suite3.addTest(new TestLista("testOrdenarNula1"));
    suite3.addTest(new TestLista("testOrdenarNula2"));
    suite3.addTest(new TestLista("testOrdenarListaVacía"));
    raiz.addTest(suite1);
    raiz.addTest(suite2);
    raiz.addTest(suite3);
    return raiz;
}

```

Una vez construido el grupo de casos de prueba (*TestSuite*), puede ser ejecutado utilizando una herramienta gráfica que incorpora *junit*. Una forma de llamar a esta herramienta es la siguiente:

```

public static void main(String[] args) {
    junit.swingui.TestRunner.run(TestLista.class);
}

```

La siguiente figura muestra el aspecto de esta herramienta: en la parte izquierda, los cinco casos de prueba se han ejecutado satisfactoriamente; en la figura de la derecha se han producido errores en alguno de los casos de prueba.

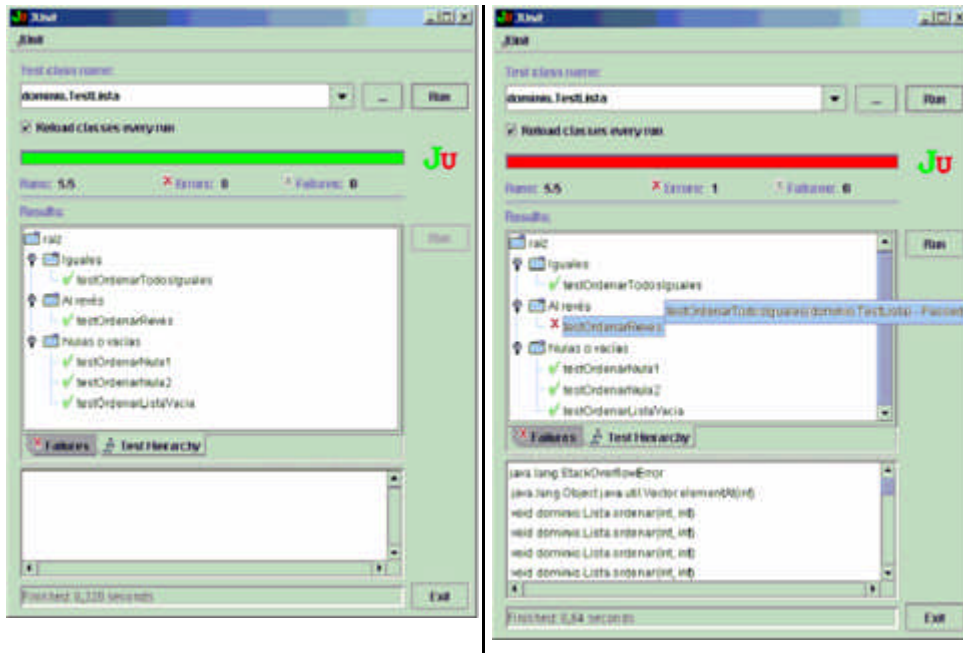


Figura 145. Aspecto de l *TestRunner*, herramienta que se utiliza para ejecutar casos de prueba

Los casos de prueba de cada clase quedan almacenados en la correspondiente clase *Test*; cuando se realiza un cambio en la clase, los casos de prueba deben ser ejecutados de nuevo. Puesto que disponemos de la herramienta mostrada en la figura anterior, que los ejecuta automáticamente, no tenemos excusa para dejar de realizar pruebas en nuestros programas.

3.03 Detalles de implementación de *junit*

Es importante que el nombre de la clase que extiende a *TestCase* comience por *Test*, y que el nombre de los métodos que prueban empiece por *test* (en minúscula). Esto es así porque *junit* utiliza Reflexión (véase página 60) para conocer cuáles son los métodos que debe ejecutar.

El siguiente código es uno de los constructores de objetos de clase *TestSuite*. Como se observa, se le pasa una instancia de clase *Class* como parámetro. Esta clase es inspeccionada utilizando Reflexión: se comprueba, por ejemplo, que la clase sea pública (instrucción `!Modifier.isPublic(theClass.getModifiers())`), que la clase sea un *Test* (instrucción `Test.class.isAssignableFrom(superClass)`), etc.


```

public TestSuite(final Class theClass) {
    fName= theClass.getName();
    try {
        getTestConstructor(theClass);
    } catch (NoSuchMethodException e) {
        addTest(warning("Class "+theClass.getName()+
            " has no public constructor TestCase(String name) or TestCase()"));
        return;
    }

    if (!Modifier.isPublic(theClass.getModifiers())) {
        addTest(warning("Class "+theClass.getName()+" is not public"));
        return;
    }
    Class superClass= theClass;
    Vector names= new Vector();
    while (Test.class.isAssignableFrom(superClass)) {
        Method[] methods= superClass.getDeclaredMethods();
        for (int i= 0; i < methods.length; i++) {
            addTestMethod(methods[i], names, theClass);
        }
        superClass= superClass.getSuperclass();
    }
    if (fTests.size() == 0)
        addTest(warning("No tests found in "+theClass.getName()));
}

```


CAPÍTULO 12. NOTACIONES FORMALES. LENGUAJE OCL.

1. Introducción

El lenguaje OCL (*Object Constraint Language* o *Lenguaje de Restricciones sobre Objetos*) se utiliza como complemento a UML. OCL permite describir de manera más precisa y sin ambigüedades los sistemas descritos con UML.

OCL se utiliza para describir los siguientes tres tipos de restricciones (ya se mencionaron en la página 75 al hablar de los Contratos):

- Precondiciones: para una operación de una cierta clase, una *precondición* especifica una condición que debe ser cierta antes de ejecutar la operación.
- Postcondiciones: para una operación de una cierta clase, una *postcondición* representa una condición que debe ser cierta después de ejecutar la operación.
- Invariantes: una *invariante* para una clase, tipo o interfaz representa una condición que siempre debe ser satisfecha por todas las instancias de tal clase, tipo o interfaz. Por tanto, siempre deben ser ciertas.

Las restricciones se escriben en OCL utilizando una notación específica, en la que intervienen tipos (en el sentido de *clases*), cada uno con sus operaciones y atributos. En la siguiente sección se presentan los tipos predefinidos en OCL; después, se ilustran la notación añadiendo algunas diferentes tipos de restricciones a un ejemplo.

2. Tipos predefinidos de OCL

2.01 Tipos básicos no escalares

En OCL existe un conjunto de tipos básicos, que son independientes del modelo de objetos que se esté desarrollando, como *Real*, *Integer*, *Boolean* y *String*. Además, todos los tipos tanto de UML como de OCL tienen un tipo, que es una instancia de un tipo especial de OCL llamado *OclType*. Todo *OclType* tiene los atributos y operaciones que se muestran en la Tabla 7:

Atributos de <i>OclType</i>		
Atributo u operación	Tipo	Descripción
name	String	nombre del tipo (en la clase <i>Persona</i> , este atributo vale "Persona")
attributes	Set(String)	conjunto formado por los nombres de los atributos del tipo (por ejemplo, si el tipo es la clase <i>Persona</i> , en sus <i>attributes</i> tendríamos <i>mNombre</i> , <i>mApellidos</i> , <i>mNIF</i> , etc.)
associationEnds	Set(String)	conjunto formado por los nombres de los tipos a los que se puede navegar mediante asociaciones y agregaciones desde el tipo en el que nos encontramos (en la Figura 44, página 50, desde <i>Tarjeta</i> podemos navegar hasta <i>Cuenta</i>)
operations	Set(String)	conjunto formado por todas las operaciones de este tipo
supertypes	Set(<i>OclType</i>)	conjunto formado por los supertipos directos ("padres" de este tipo)
allSupertypes	Set(<i>OclType</i>)	conjunto formado por todos los supertipos de este tipo (padres, abuelos, etc.)
allInstances	Set(type)	conjunto de todas las instancias de <i>type</i> y de sus supertipos

Tabla 7. Atributos de *OclType*

Además, todo tipo de OCL es un subtipo de *OclAny*, que es el supertipo de todos los tipos de un modelo de objetos. Por tanto, todos los miembros definidos en *OclAny* los tienen también todos los tipos que vayamos definiendo:

Atributos y operaciones de <i>OclAny</i>		
Atributo u operación	Tipo	Descripción
objeto = (x : <i>OclAny</i>)	Boolean	Operador de comparación, que devuelve <i>true</i> si los dos objetos que comparamos son iguales y <i>false</i> en caso contrario. A todos los tipos de nuestros modelos podemos aplicarle el operador =
objeto <> (x : <i>OclAny</i>)	Boolean	Operador de comparación, que devuelve <i>true</i> si los dos objetos que comparamos son distintos y <i>false</i> si son iguales. A todos los tipos de nuestros modelos podemos aplicarle el operador <>
objeto.oclType	<i>OclType</i>	Tipo de este objeto
objeto.oclIsKindOf(type: <i>OclType</i>)	Boolean	Devuelve <i>true</i> si <i>type</i> es supertipo del tipo de <i>objeto</i> o el mismo tipo
objeto.oclIsTypeOf(type: <i>OclType</i>)	Boolean	Devuelve <i>true</i> si el tipo del objeto es <i>type</i>

Tabla 8. Atributos y operaciones de *OclAny*

Por último, OCL define también el tipo *OclExpression*, que representa el tipo de cualquier expresión de OCL (o sea: puesto que toda expresión tiene un tipo, toda expresión es un objeto de clase/tipo *OclExpression*). El único miembro del tipo *OclExpression* se muestra en la siguiente tabla:

El único miembro de <i>OclExpression</i>		
Atributo u operación	Tipo	Descripción
expr.evaluationType	<i>OclType</i>	Tipo del objeto obtenido al evaluar <i>expr</i>

Tabla 9. El único miembro de *OclExpression*

2.02 Tipos básicos escalares

Respecto de los tipos básicos “escalares”, éstos y sus operaciones se muestran en la siguiente tabla:

Tipo	Operaciones	
Integer	+(Integer):Integer; *(Integer):Integer; abs():Integer; mod(Integer):Integer; min(Integer):Integer; Además: < > <= >= = <>	-(Integer):Integer; /(Integer):Integer div(Integer):Integer max(Integer)
Real	+(Real):Real /(Real):Real round():Real Además: < > <= >= = <>	-(Real):Real abs():Real max(Real):Real min(Real):Real floor():Real
Boolean	and or xor not implies if-then-else-endif = <>	
String	concat(String):String substring(Integer, Integer):String toReal():Real toLower():String Además: = <>	size():Integer toInteger():Integer toUpper():String

Tabla 10. Tipos básicos de OCL y sus operaciones

2.03 Colecciones

Existe también la posibilidad de manipular colecciones de elementos, para lo que se definen los tipos *Collection*, *Set*, *Bag* y *Sequence*, siendo estos tres últimos especializaciones de *Collection* (que, por otra parte, es un tipo abstracto):

- Un *Set* (conjunto) representa un conjunto de elementos en el que puede haber sólo un ejemplar de cada elemento.
- Una *Bag* (bolsa) representa un conjunto de elementos en el que cada elemento puede aparecer más de una vez.
- Una *Sequence* (secuencia) es una *bag* cuyos elementos están ordenados, en el sentido de que los elementos tienen un número de orden que podemos utilizar para acceder a un elemento dado.

Las operaciones definidas para *collection* y heredadas por sus tres especializaciones se muestran en la siguiente tabla:

Operaciones de <i>Collection</i>	
Operación	Tipo del resultado
size	Integer
includes(x : OclAny)	Boolean
count(x : OclAny)	Integer
includesAll(c : Collection)	Boolean
isEmpty	Boolean
notEmpty	Boolean
sum	Integer o Real
exists(expr:OclExpression)	Boolean
forAll(expr : OclExpression)	Boolean
iterate(expr : OclExpression)	expr.evaluationType

Tabla 11. Operaciones del tipo *Collection*, también disponibles en *Set*, *Bag* y *Sequence*

2.03.1 Set (conjunto)

Además de las operaciones anteriores, *Set* incluye las siguientes:

Operaciones de <i>Set(T)</i> (<i>T</i> es el tipo de los elementos del conjunto)		
Operación	Tipo devuelto	
union(s : Set(T))	Set(T)	
union(b : Bag(T))	Bag(T)	
= (s : Set(T))	Boolean	
intersection(s : Set(T))	Set(T)	
intersection(b : Bag(T))	Bag(T)	
- (s : Set(T))	Set(T)	Devuelve los elementos que están en un conjunto pero no en ambos
including(x : T)	Set(T)	añade <i>x</i> al conjunto
excluding(x : T)	Set(T)	elimina <i>x</i> del conjunto
symmetricDifference(s : Set(T))	Set(T)	conjunto de elementos que están en uno o en otro conjunto, pero no en ambos
select(expr : OclExpression)	Set(expr.type)	subconjunto de los elementos de <i>self</i> para los que <i>expr</i> es cierta
reject(expr : OclExpression)	Set(expr.type)	subconjunto de los elementos de <i>self</i> para los que <i>expr</i> es falsa
collect(expr : OclExpression)	Bag(expr.oclType)	<i>bag</i> resultante de aplicar <i>expr</i> a todos los elementos de <i>self</i>
count(x : T)	Integer	Nº de apariciones de <i>x</i> en <i>self</i> . Como es un <i>Set</i> , el resultado es 0 o 1
asSequence	Sequence(T)	
asBag	Bag(T)	

Tabla 12. Operaciones definidas para el tipo *Set*

2.03.2 Bag (bolsa)

En la siguiente tabla aparecen las operaciones definidas para el tipo *Bag*:

Operaciones de <i>Bag</i> (<i>T</i>) (<i>T</i> es el tipo de los elementos de la bolsa)		
Operación	Tipo devuelto	
union(<i>s</i> : Set(<i>T</i>))	Bag(<i>T</i>)	Como puede observarse, unir un <i>Set</i> y un <i>Bag</i> produce siempre un <i>Bag</i>
union(<i>b</i> : Bag(<i>T</i>))	Bag(<i>T</i>)	
= (<i>s</i> : Bag(<i>T</i>))	Boolean	
intersection(<i>s</i> : Set(<i>T</i>))	Set(<i>T</i>)	
intersection(<i>b</i> : Bag(<i>T</i>))	Bag(<i>T</i>)	
including(<i>x</i> : <i>T</i>)	Bag(<i>T</i>)	añade <i>x</i> a la bolsa
excluding(<i>x</i> : <i>T</i>)	Bag(<i>T</i>)	elimina todas las apariciones de <i>x</i> de la bolsa
select(<i>expr</i> : OclExpression)	Bag(<i>T</i>)	“subbolsa” de los elementos de <i>self</i> para los que <i>expr</i> es cierta
reject(<i>expr</i> : OclExpression)	Set(<i>T</i>)	“subbolsa” de los elementos de <i>self</i> para los que <i>expr</i> es falsa
collect(<i>expr</i> : OclExpression)	Bag(<i>expr.oclType</i>)	<i>bag</i> resultante de aplicar <i>expr</i> a todos los elementos de <i>self</i>
count(<i>x</i> : <i>T</i>)	Integer	Nº de apariciones de <i>x</i> en <i>self</i> . Como es una <i>Bag</i> , el resultado es 0 o mayor que cero.
asSequence	Sequence(<i>T</i>)	
asSet	Set(<i>T</i>)	

Tabla 13. Operaciones definidas para el tipo *Bag*

2.03.3 Sequence (secuencia)

Para terminar con las colecciones, enumeramos en la siguiente tabla las operaciones del tipo *Sequence*:

Operaciones de <i>Sequence</i> (<i>T</i>) (<i>T</i> es el tipo de los elementos de la secuencia)		
Operación	Tipo devuelto	
count(<i>x</i> : <i>T</i>)	Integer	Nº de apariciones de <i>x</i> en <i>self</i> .
= (<i>s</i> : Sequence(<i>T</i>))	Boolean	
union(<i>s</i> : Sequence(<i>T</i>))	Sequence(<i>T</i>)	
append(<i>x</i> : <i>T</i>)	Sequence(<i>T</i>)	Añade el elemento al final
prepend(<i>x</i> : <i>T</i>)	Sequence(<i>T</i>)	
subsequence(<i>inf</i> :Integer, <i>sup</i> :Integer)	Sequence(<i>T</i>)	
at(<i>pos</i> : Integer)	<i>T</i>	
first	<i>T</i>	
last	<i>T</i>	
including(<i>x</i> : <i>T</i>)	Sequence(<i>T</i>)	Añade el elemento al final (como append)
excluding(<i>x</i> : <i>T</i>)	Sequence(<i>T</i>)	elimina todas las apariciones de <i>x</i>
select(<i>expr</i> : OclExpression)	Sequence(<i>T</i>)	
reject(<i>expr</i> : OclExpression)	Sequence(<i>T</i>)	
collect(<i>expr</i> : OclExpression)	Sequence(<i>expr.oclType</i>)	
asBag	Bag(<i>T</i>)	
asSet	Set(<i>T</i>)	
iterate(<i>expr</i> : OclExpression)	<i>expr.evaluationType</i>	

Tabla 14. Operaciones definidas para el tipo *Sequence*

2.03.4 Significado de algunas operaciones de las colecciones

No todas las operaciones se comportan igual en todos los tipos de colecciones. Algunas de estas particularidades son las siguientes:

Operación	Significado en...		
	Set	Bag	Sequence
=	Todos los elementos de ambos conjuntos son iguales	Todos los elementos son iguales y, además, cada uno aparece el mismo número de veces en ambos conjuntos	Como en Bag, pero además todos los elementos están ordenados de igual forma
including	Añade un elemento al conjunto si es que no estaba ya presente	Añade el elemento a la bolsa, aunque ya esté presente	Añade el elemento al final de la secuencia
excluding	Elimina el elemento del conjunto	Elimina de la bolsa todas las apariciones del elemento	Elimina de la secuencia todas las apariciones del elemento
intersection	Puede aplicarse para obtener la intersección de dos conjuntos (devuelve un conjunto), dos bolsas (devuelve una bolsa) o un conjunto y una bolsa (devuelve un conjunto)		No aplicable

Tabla 15. Significado de algunas operaciones en las colecciones. La tabla no incluye el tipo *Collection* porque es abstracto

La operación *forall* equivale al cuantificador universal (\forall), y se utiliza para denotar que todos los elementos de una colección deben cumplir cierta condición. Del mismo modo, la operación *exists* representa el cuantificador existencial (\exists), y se usa cuando en una colección debe existir al menos un elemento que cumpla la condición.

La operación más compleja pero también más potente para trabajar con colecciones es la operación *iterate*, que se utiliza para recorrer una colección y hacer operaciones con sus elementos. La sintaxis general de *iterate* es la siguiente:

```
colección->iterate(elemento : Tipo1 ;
                  resultado : Tipo2 = expresión
                  | operación(elemento, resultado))
```

Un pseudocódigo en “pseudoJava” que describe el funcionamiento de esta operación es el siguiente:

```
resultado = asignación inicial
for (int i=0; i<colección.size(); i++) {
    elemento=colección.elementAt(i);
    resultado=operación(elemento, resultado);
}
return resul;
```

3. Ejemplos

A continuación, introduciremos la notación de OCL escribiendo algunas restricciones para el diagrama de clases de la siguiente figura:

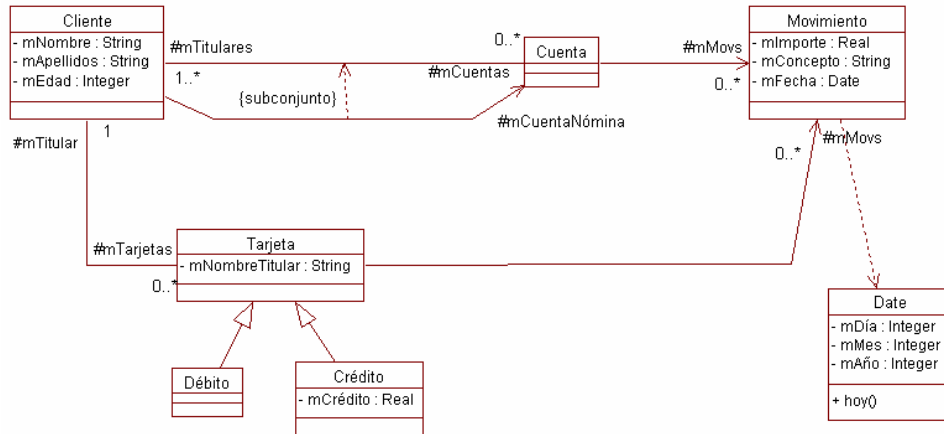


Figura 146. Un diagrama de clases

Queremos representar las siguientes restricciones:

- 1) La edad de cualquier cliente es siempre mayor o igual a cero.
- 2) En toda cuenta debe haber al menos un titular de 18 años o más.
- 3) El saldo de toda cuenta debe ser siempre mayor o igual a cero (el saldo se calcula como la suma del importe de sus movimientos).
- 4) El titular de toda tarjeta de crédito debe tener la nómina domiciliada en la misma cuenta que está asociada a la tarjeta.
- 5) El nombre del titular que figura en toda tarjeta es la concatenación del nombre y apellidos del titular.
- 6) La cantidad gastada en cierto mes con una tarjeta de crédito es menor o igual al crédito de la tarjeta.

Todas estas restricciones son invariantes, ya que son condiciones que deben ser satisfechas en todo momento por las cuentas y tarjetas de crédito. La clase dentro de la cual especificamos la invariante es lo que se llama su *contexto*. Así, la primera invariante, que afecta al atributo *mEdad* de la clase *Cliente*, se expresa en OCL como sigue:

```
context Cliente inv:
  mEdad >= 0
```

Restricción 1

La segunda restricción se refiere a los titulares de las cuentas; el contexto, en este caso, es *Cuenta*:

```
context Cuenta inv:
  self.mTitulares->exists(c : Cliente | c.mEdad >= 18)
```

Restricción 2

En este segundo caso:

- *self* se refiere a la instancia de *Cuenta* (entiéndase *self* como el *this* en Java o C++);
- *self.mTitulares* representa la colección (*collection*) de titulares de esta *Cuenta*: obsérvese que *mTitulares* es el nombre de rol de la clase *Cliente* en la asociación desde *Cuenta* a *Cliente*; además, esta expresión devuelve una colección porque la multiplicidad de este lado de la asociación es *.
- La flecha se utiliza para representar una operación que se realiza sobre una colección. Como podemos ver en la Tabla 11, *exists* es una operación definida para el tipo *collection*.
- A *exists* le estamos pasando como parámetro una *OclExpression* con la que decimos que, en la colección de titulares de esta cuenta, existe un cliente cuya edad es mayor o igual que 18 años.

Aprovecharemos la tercera restricción para definir una variable en la que guardamos el saldo de la cuenta (línea que empieza con *let*); luego, escribimos la invariante propiamente dicha imponiendo que tal variable debe ser mayor o igual a cero:

```
context Cuenta inv:
  let saldo : real = self.mMovs.mImporte->sum in
  saldo >= 0
```

Restricción 3

En la declaración de la variable *saldo* hemos incluido su tipo (*: real*) y le asignamos valor (*=self.mMovs.mImporte->sum*). Supongamos que la clase *Cuenta* tuviera un atributo *mSaldo* derivado (es decir, su valor es calculable a partir de otros elementos); en este caso, la restricción sería igual a la anterior, pero no declararíamos *mSaldo*:

```
context Cuenta inv:
  mSaldo : real = self.mMovs.mImporte->sum
  and
  mSaldo >= 0
  -- En este caso no declaramos mSaldo porque
  -- es un atributo de Cuenta.
  -- Estas tres líneas son comentarios.
```

Restricción 4

La cuarta restricción es más compleja, ya que involucra a *Crédito*, *Cliente* y *Cuenta*. Sin embargo, puede describirse de manera muy sencilla de este modo:

```
context Crédito inv:
  self.mCuentaAsociada.mTitulares->count(self.mTitular) > 0
  and
  self.mTitular.mCuentaNomina = self.mCuentaAsociada
```

Restricción 5

En la primera parte de esta última restricción estamos diciendo que el número de apariciones del titular de esta tarjeta (*self.mTitular*) en la colección de titulares de la cuenta asociada a esta tarjeta (*self.mCuentaAsociada.mTitulares*) tiene que ser mayor que cero; en la segunda parte decimos que la cuenta nómina del titular de la tarjeta es la misma cuenta asociada a esta tarjeta.

Otra característica interesante de este último ejemplo es que estamos accediendo, desde el contexto *Crédito*, a *Cuenta* y a *Cliente*, que *Crédito* hereda de *Tarjeta*.

La quinta restricción es muy sencilla:

```
context Tarjeta inv:
  self.mNombreTitular=
    mTitular.mNombre.concat(" ").concat(mTitular.mApellidos)
```

Restricción 6

Para describir la sexta restricción debemos seleccionar aquellos movimientos de la tarjeta de crédito que hayan tenido lugar este mes y sumarlos:

```
context Crédito inv:
  self.mCrédito>=self.mMovs-> select(m:Movimiento|m.mFecha.mes=Date.hoy())
```

Restricción 7

Todas las restricciones que afectan a colecciones pueden expresarse mediante la operación *iterate*. Así, la segunda restricción de la Sección 3 (al menos uno de los titulares de las cuentas debe tener 18 años o más), por ejemplo, puede expresarse así:

```
context Cuenta inv:
  let r : integer = self.mTitulares->iterate(
    p : Titular ;
    mayores : integer =0 |
    if p.mEdad>=18 then
      mayores=mayores@pre+1
    else
      mayores=mayores@pre+0
    endif
  ) in
  r>0
```

Figura 147. La Restricción 2, representada mediante la operación *iterate*

El ejemplo de la Figura 147 está representando el siguiente pseudocódigo (véase página 172):

```
int resultado=0;
for (int i=0; i<this.mTitulares.size(); i++) {
  Titular p=(Titular) mTitulares.elementAt(i);
  if (p.mEdad>=18)
    mayores=mayores+1;
}
return resul;
```

Figura 148. Pseudocódigo correspondiente a la figura anterior

Una restricción adicional que podemos representar con respecto de la Figura 146 es el hecho de que la cuenta con la nómina de un cliente es una de las cuentas de las que

dicho cliente es titular. Esto ya está realmente especificado en el diagrama UML, pero puede también describirse en OCL de esta manera:

```
context Cliente inv:
  self.mCuentas->includes(self.mCuentaNómina)
```

Restricción 8. La instancia representada por un rol está incluida en la colección denotada por otro rol

Si la multiplicidad de *mCuentaNómina* fuera *, la restricción sería muy parecida, pero utilizaríamos la operación *includesAll*:

```
context Cliente inv:
  self.mCuentas->includesAll(self.mCuentaNómina)
```

Restricción 9. Las instancias representadas por un rol están incluidas en la colección denotada por otro rol

4. Tipos del modelo

Como se observa, en la Restricción 7 se accede al atributo *mes* de *mFecha* (que, en el diagrama de la Figura 146, es de clase *Date*); también se accede a la operación *hoy()* de *Date*. Esto significa que, dentro de una especificación OCL, podemos utilizar cualquier tipo definido en el modelo UML. En este ejemplo concreto, además, ocurre que la operación *hoy()* es una operación de clase (estática), y hacemos referencia a ella poniendo el nombre de la clase delante del nombre de la operación (como hacemos en Java al ejecutar un método estático).

Por tanto, podemos utilizar tanto los atributos como las operaciones del tipo, y utilizarlos para escribir las restricciones.

5. Precondiciones y postcondiciones

Así como el contexto de una invariante es la clase, tipo o interfaz para cuyas instancias se define la invariante, el contexto de las precondiciones y postcondiciones es la operación cuya descripción están completando.

De forma general, la sintaxis para la escritura de pre y postcondiciones es la siguiente:

```
context Tipo :: operación(parámetro1 : Tipo1, ...) : TipoDevuelto
  pre : ...
  post : ...
```

Restricción 10. Sintaxis general de precondiciones y postcondiciones

El *Tipo* que se especifica tras el *context* es la clase, interfaz o tipo al que pertenece la *operación*.

Para denotar cuál debe ser el resultado devuelto por la operación que se está describiendo se utiliza la palabra clave *result*. En las postcondiciones se utiliza *@pre* para denotar el valor de una variable antes de la operación. Así, por ejemplo, una posible descripción de la operación *retirar(importe : real) : real* de la clase *Cuenta* (que toma como parámetro y devuelve la cantidad que se retira) podría ser la siguiente:

```
context Cuenta :: retirar(importe : real) : real
pre : importe > 0
    and getSaldo() >= importe
post : result = importe
```

Restricción 11. Descripción de la operación de retirada mediante una pre y una postcondición

En la restricción anterior estamos utilizando la operación *getSaldo()* que, evidentemente, no es una operación estándar de OCL; para poder utilizar esta operación en una expresión OCL, debe encontrarse adecuadamente descrita también en OCL. Podemos basarnos en la Restricción 4 (página 174), en la que describíamos una invariante para un supuesto atributo *mSaldo*, pero adaptándola a la operación *getSaldo*:

```
context Cuenta :: getSaldo() : real :
post : result = self.mMovs.mImporte->sum
```

Restricción 12. Descripción de la operación *getSaldo*

6. Representación de diagramas de estados

La siguiente figura muestra un posible diagrama de estados que representa parte del comportamiento de una Cuenta.

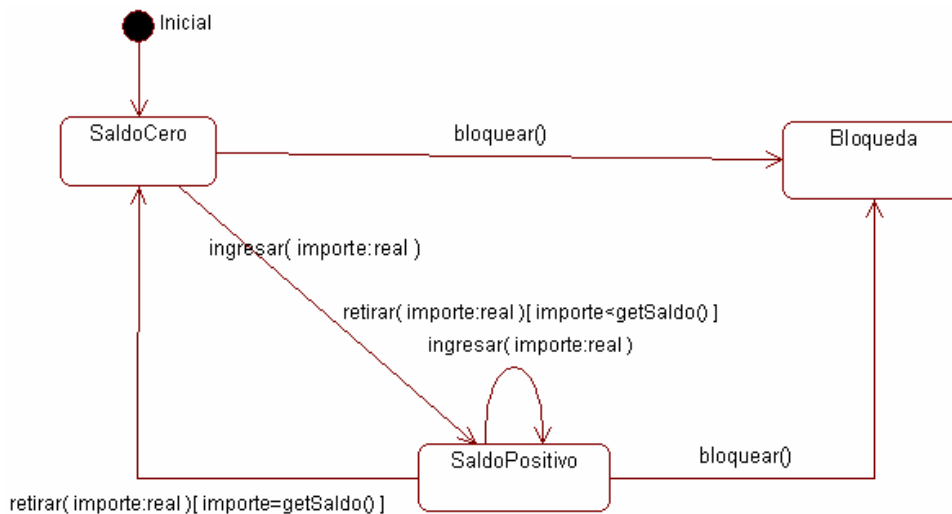


Figura 149. Un diagrama de estados

De forma general, de cada transición podemos considerar que su estado origen y su posible condición (“guarda”) son precondiciones de la operación (que corresponde al evento que etiqueta la transición), y que el estado destino es una postcondición. En este ejemplo, la única precondición para que ocurra la operación *ingresar* es que la cuenta tenga o saldo positivo o saldo cero: o, lo que es lo mismo, que no esté bloqueada. Tras la ejecución de la operación, la cuenta tendrá siempre saldo positivo. En otras palabras:

```
context Cuenta :: ingresar(importe : real)
pre  : not Bloqueada
post : SaldoPositivo
```

Restricción 13. Representación de parte del diagrama de estados de la Figura 149

La operación *retirar* es ligerísimamente más compleja:

```
context Cuenta :: retirar(importe : real)
pre  : SaldoPositivo and importe<=getSaldo()
post : (SaldoPositivo and getSaldo(>0)
or
(SaldoCero and getSaldo(=0)
```

Restricción 14. Representación de parte del diagrama de estados de la Figura 149

7. Enumeraciones

Como se observa en las dos restricciones anteriores, consideramos los estados como atributos de tipo booleano. También puede considerarse que cada clase tiene un atributo *state* de tipo *Enumeration*. En OCL, se hace referencia a los posibles valores de una enumeración anteponiendo el carácter # a dicha enumeración. De este modo, las restricciones anteriores pueden representarse así:

```
context Cuenta :: ingresar(importe : real)
pre  : state<> #Bloqueada
post : state=#SaldoPositivo
```

Restricción 15. La Restricción 13, considerando ahora un atributo *state* dentro de *Cuenta*

```
context Cuenta :: retirar(importe : real)
pre  : state=#SaldoPositivo
post : (state=#SaldoPositivo and getSaldo(>0)
or
(state=#SaldoCero and getSaldo(=0)
```

Restricción 16. La Restricción 16, considerando ahora un atributo *state* dentro de *Cuenta*

No obstante, si describimos la *Cuenta* y su estado como hacíamos en la Figura 141, (página 154) en la que utilizábamos el patrón *Estado*, las restricciones se escribirían haciendo referencia al atributo *mEstado* y utilizando, por ejemplo, la operación *oclIsTypeOf* (véase Tabla 8) para obligar a establecer el tipo de *mEstado* a uno de los tipos especializados de *EstadoCuenta*:

```
context Cuenta inv :
  mEstado.oclIsTypeOf(ConDinero) xor
  mEstado.oclIsTypeOf(SinDinero) xor
  mEstado.oclIsTypeOf(Bloqueada)
```

Restricción 17. Una restricción para los estados de la Figura 141

8. Mensajes

En OCL también existe la notación para representar mensajes. Para representar un mensaje enviado desde una instancia a otra se utiliza el operador \wedge . Así, por ejemplo, para denotar que, tras sacar dinero con una tarjeta de débito se efectúa una retirada en la cuenta asociada (véase Figura 146), podemos escribir lo siguiente:

```
context Débito :: sacarDinero(importe : real)
pre : mCuentaAsociada.getSaldo()>=importe
post : mCuentaAsociada^retirar(importe)
```

Restricción 18. Un paso de mensajes

En esta última restricción no sería preciso representar el estado de *mCuentaAsociada* tras la ejecución de la operación, ya que la operación *retirar* en *Cuenta* se encuentra totalmente descrita en OCL.

Por otro lado, en ocasiones se envía una lista de mensajes a un grupo de objetos. Puede hacerse referencia a la lista de mensajes mediante el operador $\wedge\wedge$, que devuelve una secuencia de elementos de tipo *OclMessage*. Supongamos que, al eliminar una *Cuenta*, se da valor *false* a un atributo *mBorrado* de todos sus movimientos y que, además, se han enviado tantos mensajes como movimientos haya:

```
context Cuenta :: borrar()
let mensajesDeBorrar : Sequence(OclMessage) = mMovs^^borrar()
post : mensajesDeBorrar.size = mMovs.size
```

Restricción 19

En ocasiones no se conocen los valores de los parámetros que se envían con el mensaje, utilizándose en este caso interrogaciones en lugar de nombres de los parámetros.

El valor devuelto por un mensaje enviado es accesible a través de la operación *result()*, cuyo tipo es el mismo que devuelve el mensaje. Supongamos que la operación *sacarDinero* de *Débito* llama a *retirar* de su cuenta asociada, y que devuelve además el saldo que queda en la cuenta tras la retirada:

```
context Cuenta :: retirar(importe : real) : boolean
  pre : importe >= sel.getSaldo()
  post :
    let mensaje : OclMessage = mCuentaAsociada.retirar(importe) in
    mensaje.hasReturned()
    and
    mensaje.result() = mCuentaAsociada.getSaldo()
```

Restricción 20

CAPÍTULO 13.

PROGRAMACIÓN EXTREMA

1. Introducción

Programación Extrema (*eXtreme Programming* o *XP*) es uno de los denominados “métodos ágiles” para desarrollo de software. Los métodos ágiles pretenden el desarrollo de aplicaciones de manera que se satisfagan los requisitos de usuario de forma rápida. En XP se consigue este objetivo mediante una serie de reglas y prácticas, como la programación por pares (*pair programming*), “historias del usuario” (*user stories*) y refactorización (*refactoring*).

Una de las características de XP que más llama la atención, desde el punto de vista de la Ingeniería del Software clásica, y que es objeto de una gran parte de las críticas que recibe es que toda la documentación de un proyecto desarrollado mediante XP está compuesta del código fuente y las historias del usuario. También se le critica que se hable de que XP es un método de desarrollo nuevo, cuando realmente es el producto de haber unido un conjunto de técnicas de toda la vida.

2. Modelo de proceso

El desarrollo de software mediante XP es “dirigido por las pruebas” (*test-driven*), llegándose a construir los casos de prueba incluso antes de tener el código que se va a probar. Por ello, en un equipo de trabajo de un proyecto XP no hay solamente informáticos (se recomienda que haya entre 2 y 12 analistas/programadores o desarrolladores), sino que también se requiere la participación constante del usuario (que, entre otras cosas, prepara pruebas de aceptación) y gestores del proyecto.

2.01 Planificación

2.01.1 Recepción de las historias del usuario

El trabajo comienza cuando el equipo de desarrollo recibe un conjunto de historias del usuario. Una historia del usuario es una breve nota (la brevedad se acota físicamente al limitar el tamaño del papel en el que deben ser descritas a 10x15 cms, como medio

A4) en la que el representante del usuario explica en unas pocas frases carentes de terminología técnica alguna característica o funcionalidad de la aplicación.

Las historias del usuario, entonces, sustituyen al clásico documento de especificación de requisitos. Como puede además deducirse del pequeño tamaño de una historia de usuario, su nivel de detalle no es muy grande, debiendo contener la información suficiente para realizar una estimación aproximada del tiempo que llevará la implementación de la historia.

2.01.2 Plan de entregas (*release plan*)

Se recomienda que cada historia sea implementable en un *tiempo de desarrollo ideal* de entre 1 y 3 semanas. El tiempo de desarrollo ideal es el tiempo de desarrollo si no hubiera distracciones de ninguna clase. Si se estima un tiempo mayor, la historia debe replantearse y ser dividida en dos o más historias; si se estima un tiempo de pocos días, deberían agruparse varias historias. Realizada la planificación, el equipo de desarrollo solicitará al cliente (que, como se ha dicho, está a tiempo completo junto a los desarrolladores) información más detallada para llevar a cabo la implementación. La implementación de una historia se corresponde aproximadamente con una iteración, y cada iteración se descompone en tareas de entre 1 y 3 días.

Con el conjunto de historias se prepara un “plan de entregas”, que especifica qué historias del usuario van a implementarse y en qué fechas van a estar implementadas y operativas, a la manera de un plan de iteraciones.

Durante la implementación de la historia, se construyen también uno o varios casos de prueba de caja negra a partir de la información contenida en la propia historia, que serán utilizados como pruebas de aceptación. En éstas, el cliente especifica los escenarios que debe superar la implementación de la historia de usuario.

2.02 Diseño

Los desarrolladores trabajan de dos en dos en la misma máquina en el diseño de cada tarea, usando como fuente de trabajo la información de la tarea (que procede de la descomposición de la iteración). Esta información se va representando directamente en código fuente, que debe ser tan simple como sea posible. No deben añadirse funcionalidades que no formen parte de esta tarea.

2.03 Codificación

En XP se aboga por codificar los casos de prueba de unidad antes de la propia unidad. Los potenciales beneficios de esta filosofía son la sencillez en la posterior codificación de la unidad, la autolimitación en los contenidos de ésta (el programador no escribirá más código del necesario), una inmediata realimentación mientras se trabaja, un mejor diseño (más orientado hacia la satisfacción de los requisitos del cliente)... De lo que ya sabemos por el Capítulo 11, cada caso de prueba suele incluir la comparación de un objeto esperado con otro construido usando los métodos de la unidad; esto permite que posteriores desarrolladores vean los casos de prueba como ejemplos de uso del código, lo que puede por tanto facilitar su comprensión.

Una vez programado un caso de prueba, se pasa a codificar un pequeño fragmento de la unidad que sea capaz de superar el caso de prueba; superado éste, se programa un segundo caso de prueba y otro nuevo fragmento *ex profeso* para este caso, etc. Como se observa, al programar la unidad no se añade nada que no sea estrictamente necesario para los casos de prueba que se llevan hasta el momento.

2.04 Integración continua

Cada vez que se disponga de una nueva clase probada unitariamente, ésta es integrada con el resto del sistema. Si el sistema funciona correctamente (pasa todos los tests), los desarrolladores dan por terminada la tarea; en caso contrario, son responsables de dejarlo funcionando bien. Si, tras un cierto tiempo, no descubren los errores, desecharán su código y desarrollarán de nuevo su clase.

2.05 40 horas semanales

Los desarrolladores deben trabajar 8 horas al día durante 5 días a la semana.

3. Las cuatro variables de los proyectos de desarrollo de software

Programación Extrema define cuatro variables para los proyectos software, que son el *coste*, el *tiempo*, la *calidad* y el *alcance* (*scope*). Tres de ellas son controlables por el cliente o el jefe del proyecto, mientras que el valor de la variable que queda libre será establecido por el equipo de desarrollo dependiendo de los valores de las otras tres.

Así, por ejemplo, incrementar la *calidad* del producto puede implicar un mayor *tiempo* para desarrollarlo, ya que se requerirán pruebas más exhaustivas (no obstante, la estrategia de pruebas semiautomáticas de XP puede hacer que esta relación no se cum-

pla). Respecto del *alcance*, ésta es la variable cuyo control suele dejarse al equipo de desarrollo, que irá determinando en cada momento qué requisitos se irán implementando en cada iteración.

4. Programación dirigida por las pruebas (*test-driven development*)

En esta sección realizaremos un ejemplo en el que se ilustra uno de los principios básicos de XP, que es la *programación dirigida por las pruebas*. Como se ha dicho, consiste en diseñar los casos de prueba antes de tener el propio código. Si el lector no lo ha hecho ya, le conviene releer la Sección 3 (página 159) del capítulo dedicado a Pruebas.

El mecanismo de desarrollo que seguiremos será, resumidamente, el siguiente:

1) Escribir un caso de prueba como si ya tuviéramos la interfaz de la clase que se está probando (en inglés se la llama *Class Under Test* o, abreviadamente en algunos libros, *CUT*). El caso de prueba incluirá todos los elementos que se consideren necesarios para obtener las respuestas correctas.

2) Ejecutar todas las pruebas que se llevan hasta la fecha y ver cuál falla

3) Hacer un pequeño cambio en el código

4) Hacer un pequeño cambio y comprobar que todas tienen éxito

5) Refactorizar para eliminar duplicación

Supongamos que deseamos construir una aplicación de venta electrónica. En este momento estamos centrados en el carrito, en el que el usuario pone los artículos que desea comprar y del que puede quitar aquellos que desee. En cualquier momento el usuario podrá preguntar al carrito por el importe total de los productos que contiene. Supondremos que ya disponemos de la clase *Producto*, que posee la especificación que se muestra en el siguiente cuadro:

```
package dominio;
public class Producto {
    protected String mCodigo, mDescripcion;
    protected double mPrecio, mPorcentajeIVA;

    public Producto() {
    }
    public String getCodigo() {
        return mCodigo;
    }
    public void setCodigo(String newMCodigo) {
        mCodigo = newMCodigo;
    }
    public String getDescripcion() {
        return mDescripcion;
    }
    public void setDescripcion(String newMDescripcion) {
        mDescripcion = newMDescripcion;
    }
}
```

```

    }

    public double getPorcentajeIVA() {
        return mPorcentajeIVA;
    }

    public void setPorcentajeIVA(double newMPorcentajeIVA) {
        mPorcentajeIVA = newMPorcentajeIVA;
    }

    public double getPrecio() {
        return mPrecio;
    }

    public void setPrecio(double newMPrecio) {
        mPrecio = newMPrecio;
    }

    public double getPVP() {
        return mPrecio*(1+mPorcentajeIVA/100);
    }
}

```

Cuadro 1

Al desarrollar dirigidos por las pruebas, supondremos que la clase que estamos probando (en inglés se la llama *class under test* o, abreviadamente, *CUT*) ya dispone de la interfaz adecuada. De este modo, podemos escribir, dentro de una clase *TestCarrito* (especialización de la clase *TestCase* de junit) que usaremos para probar la clase *Carrito*, el método que hemos resaltado en negrilla en el siguiente cuadro:

```

package dominio;

import junit.framework.*;

public class TestCarrito extends TestCase {
    public TestCarrito(String nombre) {
        super(nombre);
    }

    public void testCarritoVacioGetImporte() {
        Carrito c=new Carrito();
        this.assertEquals(0.0, c.getImporte());
    }

    public static void main(String[] args) {
        junit.swingui.TestRunner.run(TestCarrito.class);
    }
}

```

Cuadro 2

Evidentemente, el código del Cuadro 2 no compilará porque no tenemos clase *Carrito*; además, para que el código compile, esta clase necesita un constructor sin argumentos y un método *getImporte()* que devuelva un *double*. El siguiente código es aparentemente suficiente para compilar:

```

package dominio;

public class Carrito {
    public Carrito() {
    }

    public double getImporte() {
        return 0.0;
    }
}

```

Cuadro 3

Sin embargo, falla la compilación de *TestCarrito* con el siguiente mensaje:

```
Error(12,8): method assertEquals(double, double) not found in class
dominio.TestCarrito
```

En efecto, no existe en *TestCase* ni en *Assert* (recuérdese la figura Figura 142, página 160) ningún método *assertEquals* que tome dos parámetros de tipo *double*. Podríamos añadir un método con esta signatura a *TestCarrito*, pero también se puede aprovechar otro de los métodos de *TestCase*, de modo que cambiemos la implementación del método y que quede de esta guisa:

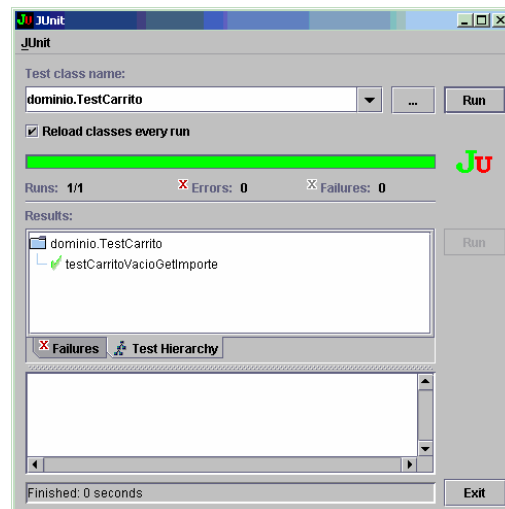
```

public void testCarritoVacioGetImporte() {
    Carrito c=new Carrito();
    this.assertTrue(0.0==c.getImporte());
}

```

Cuadro 4

El código ya no sólo compila, sino que el caso de prueba pasa (como era de esperar) perfectamente:

**Figura 150**

Con el siguiente caso de prueba probaremos que el número de productos que hay en un carrito vacío es cero:

```

    public void testCarritoVacioGetImporte() {
        Carrito c=new Carrito();
        this.assertEquals(0, c.getNumeroDeProductos());
    }

```

Cuadro 5

El código que debemos añadir a *Carrito* para que compile y supere la prueba es el siguiente:

```

    public int getNumeroDeProductos() {
        return 0;
    }

```

Cuadro 6

Ahora escribimos un caso de prueba para comprobar que el número de productos en un carrito vacío en el que hemos puesto un producto es, en efecto, uno:

```

    public void testCarritoConUnProducto() {
        Carrito c=new Carrito();
        Producto p=new Producto("123", "La ruta no natural", 4.21, 7);
        c.add(p);
        this.assertEquals(c.getNumeroDeProductos(), 1);
    }

```

Cuadro 7

Hemos resaltado en negrilla el método *add(Producto)*, que debe ser añadido a *Carrito*. La clase puede ser reescrita como sigue (hemos resaltado en negrilla el código añadido en este último “viaje”):

```

package dominio;

import java.util.Vector;

public class Carrito {
    Vector mProductos;

    public Carrito() {
        mProductos=new Vector();
    }

    public double getImporte() {
        return 0.0;
    }

    public int getNumeroDeProductos() {
        return 0;
    }

    public void add(Producto p) {
        mProductos.addElement(p);
    }
}

```

Cuadro 8

Como acabamos de ver en la página 184, es preciso reejecutar todos los casos de prueba. Si no dispusiéramos de una herramienta que lo automatice, la reejecución sería muy costosa (esta es una de las razones que conducen a que, en Ingeniería del Software,

no se preste a la etapa de pruebas la atención que merece). Mediante junit, y puesto que vamos almacenando todos los casos en *TestCarrito*, la ejecución de todos los casos es inmediata:

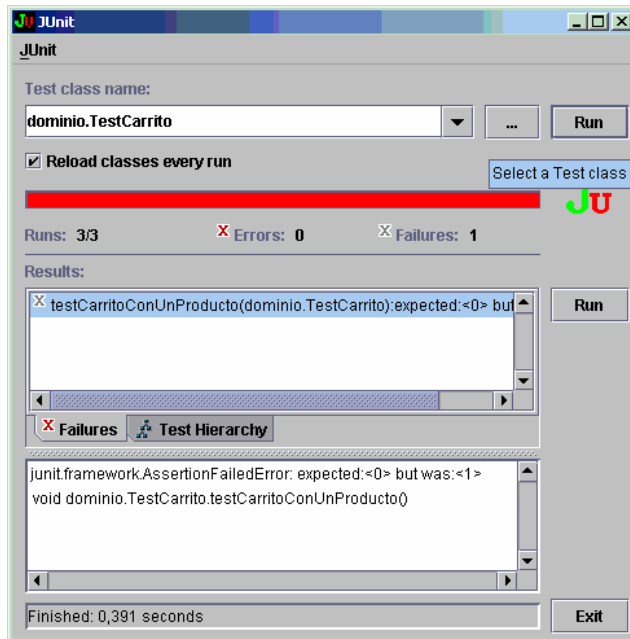


Figura 151

La herramienta nos informa de que se ha producido un fallo en el método *testCarritoConUnProducto*. Debemos revisar el código de *Carrito* para corregir, de la manera más simple posible, la causa del error. Éste estaba, claramente, en *getNumeroDeProductos*, que devolvía siempre cero y que debe devolver el número de elementos en el vector *mProductos*:

```
public int getNumeroDeProductos() {
    return mProductos.size();
}
```

Cuadro 9

Ahora todos los casos de prueba pasan y junit muestra la esperada barra verde que ya vimos en la Figura 150.

Escribamos un caso de prueba para comprobar que el precio de los productos que hay en un carrito coincide con la suma de los precios con IVA:


```

public void testComprobarPrecio() {
    Carrito c=new Carrito();
    Producto libro1=new Producto("123", "La ruta no natural", 100, 7);
    c.add(libro1);
    Producto libro2=new Producto("124", "Tendiendo al equilibrio", 200, 7);
    c.add(libro2);
    this.assertTrue(c.getImporte()==321.0);
}

```

Cuadro 10

Ahora ha sido preciso modificar el código del método *getImporte()* de *Carrito*, que siempre devolvía cero.

Mediante el siguiente caso de prueba comprobaremos que se elimina del carrito el artículo cuyo código pasamos por parámetro:

```

public void testComprobarEliminacion() {
    Carrito c=new Carrito();
    Producto libro1=new Producto("123", "La ruta no natural", 100, 7);
    c.add(libro1);
    Producto libro2=new Producto("124", "Tendiendo al equilibrio", 200, 7);
    c.add(libro2);
    Producto libro3=new Producto("125", "Las aventuras del Dr. Peterson",
200, 7);
    c.add(libro3);
    c.remove("123");
    this.assertEquals(c.toString(), "124#125");
}

```

Cuadro 11

Con la siguiente implementación del método *remove(String)* de *Carrito*, junit nos informa de un error al ejecutar este caso:

```

public void remove(String codigo) {
    for (int i=0; i<mProductos.size(); i++) {
        Producto p=(Producto) mProductos.elementAt(i);
        if (p.getCodigo().equals(codigo)) {
            mProductos.removeElementAt(i);
            return;
        }
    }
}

```

Cuadro 12

El error se debe a que la última instrucción ejecutable del Cuadro 11 ejecuta el método *toString* de *Carrito*, que esperamos nos devuelva una cadena compuesta por el código de cada producto contenido en el *Carrito* y separado del siguiente por una almohadilla. Evidentemente, el *toString()* de *Carrito* está heredado de la clase *Object* y muestra su implementación por defecto; por tanto, antes de considerar el caso de prueba del Cuadro 11 debemos construir otro para el nuevo *toString()* que debemos escribir en *Carrito*:

```

public void testToStringEnVacio() {
    Carrito c=new Carrito();
    this.assertEquals(new String(), c.toString());
}

```

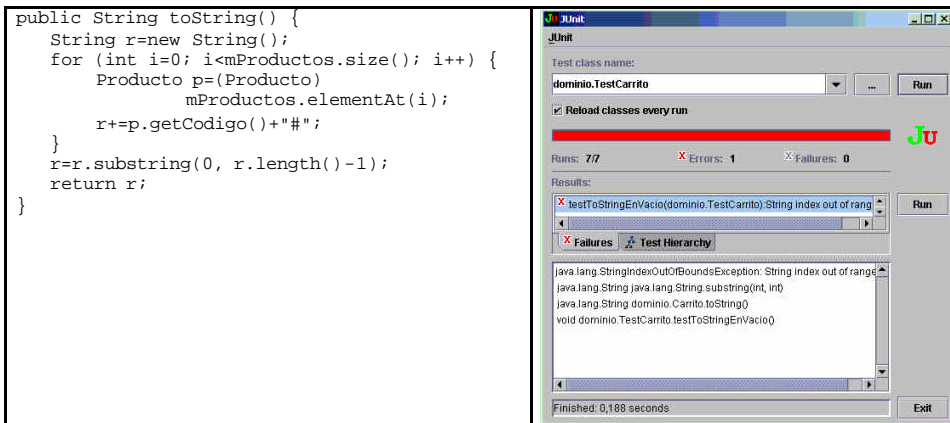
```

public void testToStringEnLleno() {
    Carrito c=new Carrito();
    Producto libro1=new Producto("123", "La ruta no natural", 100, 7);
    c.add(libro1);
    Producto libro2=new Producto("124", "Tendiendo al equilibrio", 200, 7);
    c.add(libro2);
    Producto libro3=new Producto("125", "Las aventuras del Dr. Peterson",
200, 7);
    c.add(libro3);
    this.assertEquals("123#124#125", c.toString());
}

```

Cuadro 13

Si damos a *toString()* la implementación de la izquierda, junit muestra el resultado de la derecha:

**Cuadro 14****Figura 152**

El error se produce al probar con el carrito vacío en la penúltima instrucción (la que recorta la cadena). Lo modificamos como sigue y las pruebas pasan:

```

public String toString() {
    String r=new String();
    for (int i=0; i<mProductos.size(); i++) {
        Producto p=(Producto) mProductos.elementAt(i);
        r+=p.getCodigo()+"#";
    }
    if (r.length()>0)
        r=r.substring(0, r.length()-1);
    return r;
}

```

Cuadro 15

Además, también se supera el caso de prueba de la eliminación, que aparecía en el Cuadro 11.

La siguiente prueba comprobará que dos carritos a los que añadimos los mismos productos son iguales:

```

public void testIgualdad() {
    Carrito c1=new Carrito();
    Producto libro1=new Producto("123", "La ruta no natural", 100, 7);
    c1.add(libro1);
    Producto libro2=new Producto("124", "Tendiendo al equilibrio", 200, 7);
    c1.add(libro2);
    Producto libro3=new Producto("125", "Las aventuras del Dr. Peterson", 200, 7);
    c1.add(libro3);

    Carrito c2=new Carrito();
    Producto libro4=new Producto("123", "La ruta no natural", 100, 7);
    c2.add(libro4);
    Producto libro5=new Producto("124", "Tendiendo al equilibrio", 200, 7);
    c2.add(libro5);
    Producto libro6=new Producto("125", "Las aventuras del Dr. Peterson", 200, 7);
    c2.add(libro6);

    this.assertEquals(c1, c2);
}

```

Cuadro 16

Este caso falla porque *assertEquals(Object, Object)* llama a *assertEquals(String, Object, Object)*, en cuya implementación se utiliza el método *equals* sobre el segundo parámetro; si no se dice otra cosa, este *equals* es el *equals* heredado de *Object*; es decir, que el código que estamos realmente utilizando es el siguiente:

```

static public void assertEquals(String message, Object expected, Object actual) {
    if (expected == null && actual == null)
        return;
    if (expected!=null && expected.equals(actual))
        return;
    failNotEquals(message, expected, actual);
}

```

Cuadro 17. Código de junit (en la clase *Assert*) al que llama *assertEquals(Object, Object)*

Por tanto, debemos redefinir el método *equals* en *Carrito*, de forma que compruebe que todos los productos contenidos en cada uno son iguales:

```

public boolean equals(Object obj) {
    Carrito otro=null;
    try {
        otro=(Carrito) obj;
    }
    catch (Exception e) {
        return false;
    }
    if (mProductos.size()!=otro.mProductos.size())
        return false;
    for (int i=0; i<mProductos.size(); i++) {
        Producto a=(Producto) mProductos.elementAt(i);
        Producto b=(Producto) otro.mProductos.elementAt(i);
        if (!a.equals(b))
            return false;
    }
    return true;
}

```

Cuadro 18

En este último código, sin embargo, estamos utilizando el método *equals* sobre *Producto*, que no está definido y que conduce a error en la ejecución del caso de prueba

del Cuadro 16. Debemos por tanto redefinir el método *equals* en *Producto*. De esta implementación esperamos que funcionen correctamente los siguientes casos de prueba:

```
package dominio;

import junit.framework.*;

public class TestProducto extends TestCase
{
    public TestProducto(String n)
    {
        super(n);
    }

    public void testIgualdad() {
        Producto a=new Producto("123", "La ruta no natural", 100, 7);
        Producto b=new Producto("123", "La ruta no natural", 100, 7);
        this.assertEquals(a, b);
    }

    public void testDesigualdad() {
        Producto a=new Producto("123", "La ruta no natural", 100, 7);
        Producto b=new Producto("123", "La ruta sí natural", 100, 7);
        this.assertNotSame(a, b);
    }

    public static void main(String[] args) {
        junit.swingui.TestRunner.run(TestProducto.class);
    }
}
```

Cuadro 19

Con el siguiente código en *equals*, los casos de prueba de *Producto* escritos en el cuadro anterior son superados:

```
public boolean equals(Object otro) {
    Producto p;
    try {
        p=(Producto) otro;
    }
    catch (Exception e) {
        return false;
    }
    return (mCodigo.equals(p.mCodigo) &&
mDescripcion.equals(p.mDescripcion) &&
        mPrecio==p.mPrecio && mPorcentajeIVA==p.mPorcentajeIVA);
}
```

Cuadro 20

Ahora debemos reejecutar los casos de *Carrito*, teniendo especialmente en cuenta el del Cuadro 16, que también son superados.

5. Patrones para pruebas

5.01 Objetos Mock (patrón *Mock Objects*)

Este patrón nos dice que, cuando para nuestra prueba necesitemos usar recursos que, de alguna manera, resultan costosos (una base de datos, por ejemplo), utilicemos

objetos Mock para realizar las pruebas. Un Mock es un objeto que simula el comportamiento del recurso costoso y que, por lo general, será programado para que devuelva los mismos resultados.

5.02 Cadena de *log* (patrón *Log String*)

En ocasiones necesitamos conocer la secuencia en que se ejecutan los métodos de la clase que se está probando. Una buena forma de registrar la secuencia de mensajes que se ejecutan en una clase X es almacenar en una cadena unos textos que puede generar cada método; luego, comparamos el contenido de esta cadena con el texto esperado. Esta comparación la haremos dentro de un método *test* de la clase *TestX*.

5.03 Pruebas de excepciones (patrones *Crash Test Dummy* y *Exception Test*)

Igual que es necesario comprobar cómo se comporta el programa en situaciones idóneas, es también importante probarlo en situaciones en que se producen errores. De acuerdo con este patrón, lanzaremos de forma explícita la excepción que se debe producir, que atraparemos, dentro del método *test*, en un bloque *try*. Podemos utilizar también el método *fail(String)*, cuya ejecución hace que el caso de prueba se muestre como erróneo y que muestre un mensaje en la ventana de junit. El siguiente método, tomado de la página web de junit, hará que, aunque no se produzca error en tiempo de ejecución, el caso de prueba se de cómo malo y salga la barra roja:

```
public void testIndexOutOfBoundsException() {
    ArrayList emptyList = new ArrayList();
    try {
        Object o = emptyList.get(0);
        fail("Should raise an IndexOutOfBoundsException");
    } catch (IndexOutOfBoundsException success) {}
}
```

5.04 Uso del método *setUp* (patrón *Fixture*)

Como se recuerda del capítulo dedicado a pruebas, las clases *Test* pueden redefinir el método *setUp()*, que se hereda de *Assert*. Cuando valgan los mismos objetos para varios casos de prueba, aquellos pueden ser definidos como variables de instancia de la clase *Test* y, entonces, ser inicializados en el método *setUp*, que redefiniremos en nuestra clase *Test*.

6. Ejercicio

Un famoso ejercicio de *testing*, que nos sirve para hacernos una idea de los costes reales de un proceso de pruebas exhaustivo, es el es el dado por Glenford Myers en *The Art of Software Testing*. Se pide que se construyan casos de prueba para probar el funcionamiento correcto de un programa que admite tres números enteros que se interpretan como las longitudes de los tres lados de un triángulo; el programa debe decir si el triángulo es equilátero, isósceles o escaleno.

La solución puede parecer trivial, pero muy probablemente se haya olvidado de alguno de los siguientes puntos:

1. ¿Ha construido un caso de prueba que represente un triángulo escaleno válido?
2. ¿Ha construido un caso de prueba que represente un triángulo equilátero válido?
3. ¿Ha construido un caso de prueba que represente un triángulo isósceles válido?
4. ¿Ha construido al menos tres casos de prueba que representen triángulos isósceles válidos, de modo que se han considerado las tres permutaciones de dos lados iguales?
5. ¿Ha construido un caso de prueba en el que un lado es cero?
6. ¿Ha construido un caso de prueba en el que un lado es negativo?
7. ¿Ha construido un caso de prueba con tres números enteros positivos tal que la suma de dos es igual al tercero?
8. ¿Ha construido al menos tres casos de prueba del tipo anterior tal que se han probado las tres permutaciones en las que la longitud de un lado es igual a la suma de las longitudes de otros dos lados?
9. ¿Ha construido un caso de prueba con la suma de dos de los números menores que el tercero?
10. ¿Ha construido al menos tres casos del tipo anterior tal que se han intentado todas las permutaciones?
11. ¿Ha construido un caso de prueba con todos los lados de longitud cero?
12. Y otro punto importante: ¿ha especificado el resultado esperado para cada caso de prueba?

Figura 153. Algunos olvidos típicos en el problema de los triángulos

7. Pruebas de aceptación

Como se indicó al principio de este capítulo, con las historias del usuario se preparan pruebas de aceptación. Un sencillo algoritmo para confeccionarlas es el siguiente:

- 1) Identificar todas las acciones que hay en la historia
- 2) Para cada acción, definir dos pruebas: la primera suministrará al programa un conjunto de datos de entrada X cuyo resultado esperado sea satisfactorio; la segunda suministrará un conjunto de datos Y cuyo resultado esperado sea no satisfactorio.
- 3) Ejecutar más casos satisfactorios (variando el valor de X).
- 4) Ejecutar más casos no satisfactorios (variando el valor de Y).

De este modo, el cliente comprobará que su programa se comporta correctamente tanto cuando se obtienen resultados como cuando no.

CAPÍTULO 14. DESARROLLO DE SISTEMAS ESPECÍFICOS

1. Introducción

En los últimos años han aparecido multitud de nuevas plataformas para desarrollar aplicaciones y ponerlas en explotación. En este capítulo echaremos un vistazo a algunas técnicas para desarrollar aplicaciones web, servicios web, sistemas de información geográficos y aplicaciones basadas en componentes distribuidos; concluiremos con una sección en la que veremos algunos patrones específicamente diseñados para su aplicación en el desarrollo de sistemas colaborativos.

2. Introducción al desarrollo de aplicaciones web

Una aplicación web es una aplicación que ofrece un conjunto de funcionalidades que pueden ejecutarse a través de un navegador que utiliza el cliente. Mediante protocolo *http*, el cliente envía peticiones al servidor, que las ejecuta y que devuelve resultados.

Una gran ventaja de las aplicaciones web es la independencia de la plataforma de ejecución (en el cliente), ya que lo único que necesita es un navegador que interprete adecuadamente texto *html* y, en ocasiones, algunas capacidades adicionales que hoy en día son ofrecidas de manera generalizada (ejecución de *applets* o de lenguajes de *script*, por ejemplo).

Las dos siguientes figuras muestran muy esquemáticamente la diferencia entre descargar una página estática de un servidor web y descargar una página dinámica: en el primer caso, el servidor recibe la petición, recupera la página de su lugar de almacenamiento y la entrega tal cual al cliente, que ya sabrá interpretarla y mostrarla; en el segundo caso, el servidor recibe la petición (una *url* quizás con uno o más parámetros) y realiza algún tipo de procesamiento para entregar al cliente los resultados en un formato que éste entienda (normalmente, en forma de código *html*). El código que el servidor interpreta y ejecuta se conoce como “*script* de servidor”.

Eliminado: ar

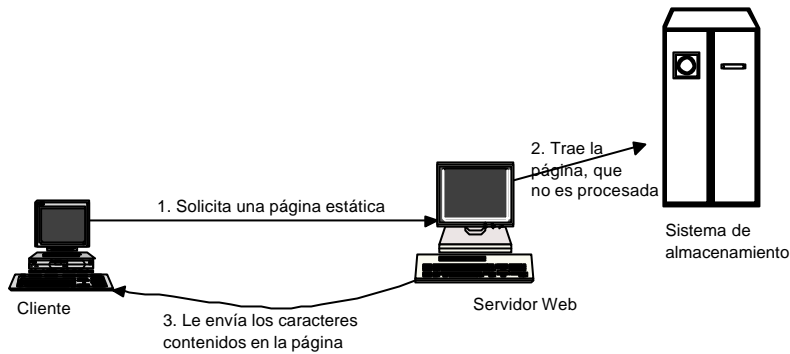


Figura 154. Solicitud de una página sin *scripts* de servidor

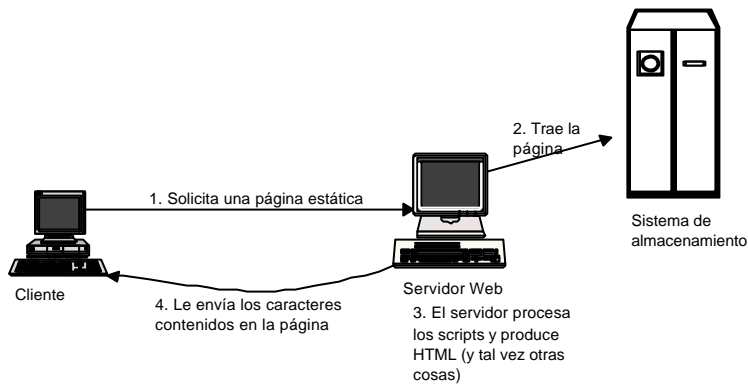


Figura 155. Solicitud de una página con *scripts* de servidor

Existen muchas tecnologías para la programación de “páginas activas” (es decir, con scripts de servidor), entre ellas CGI, PHP, JSP, Servlets, ASP, ASP.NET o CSP, pero básicamente su funcionamiento es siempre el descrito. Existen, por otro lado, lenguajes de script de cliente, como JavaScript o Visual Basic Script, que se ejecutan únicamente en el navegador del cliente y que no deben, en principio, comprometer la seguridad de éste. Evidentemente, y como se ha dicho antes, para que un script de cliente se ejecute en el navegador, éste debe ser capaz de interpretarlo y de ejecutarlo.

2.01 Envío de parámetros a través de una URL

El protocolo http 1.1 (<http://www.w3.org/Protocols/rfc2616/rfc2616.html>) define dos tipos importantes de mensajes, *Request* y *Response*, que respectivamente representan solicitudes que los clientes realizan al servidor y respuestas que los servidores envían a los clientes.

Existen, además, dos tipos destacados de *requests*, las que se realizan siguiendo el método *get* y las que se realizan siguiendo el método *post*:

- Mediante *get*, se solicita del servidor algún tipo de recurso especificando quizás un conjunto de parámetros que viajan con la url, no pudiendo superar ésta los 1024 bytes. Así, la URL `http://161.67.27.108:8080/curri/vercurri.jsp?formato=uclm&id=5450675` representa una solicitud *get* a la máquina 161.67.27.108 en su puerto 8080; se le solicita que nos devuelva la página `curri/vercurri.jsp` y le estamos pasando *uclm* y 5450675 como valores respectivos de los parámetros *formato* e *id*.
- Mediante *post* se solicita al servidor algún tipo de recurso especificando quizás un conjunto de parámetros pero, en este caso, sin limitación de tamaño, ya que el conjunto de parámetros y sus valores viajan en una variable junto a la petición.

La siguiente figura muestra el aspecto visual de un sencillo formulario de identificación y su correspondiente código *html*; en éste, se ha señalado con negrita el fragmento de código que representa la acción que debe ejecutarse en el servidor cuando se pulse el botón etiquetado “Aceptar”, así como el método de envío de los datos: el nombre (en una caja de texto llamada “nombre”) y la password (en una caja especial que muestra asteriscos llamada “pwd”) los recibirá el servidor mediante la url `servlet/identificar`, además, ambos parámetros se enviarán usando el método *post*. Estos dos últimos datos aparecen en la primera línea que describe el formulario.

<p>Intranet de la Escuela Superior de Informática</p> <p>Usuarios de titan2</p> <p>Nombre (p.ej.: <i>fperez</i> y no <i>fulano.perez</i>): <input type="text"/></p> <p>Password: <input type="password"/></p> <p>Aceptar</p>	<pre> <form action=servlet/identificar method=post> <table> <tr> <td colspan="2"> <p align="center">Usuarios de titan2 </td> </tr> <tr> <td> Nombre (p.ej.: <i>fperez</i>
y no <i>fulano.perez</i>): </td> <td> <input type=text name=nombre> </td> </tr> <tr> <td>Password:</td> <td><input type=password name=pwd></td> </tr> <tr> <td> <input type=submit class=boton value=Aceptar> </td> </tr> </table> </form> </pre>
---	---

Figura 156. Un sencillo formulario y su correspondiente código *html*

En este ejemplo, por tanto, los parámetros “nombre” y “pwd” se envían a una *url* que debe encargarse de recogerlos, leer sus valores y procesarlos como corresponda. Puesto que los datos se envían utilizando “post”, la url receptora debe ser capaz de responder a este método de envío. Si se está utilizando un *servlet*, como es este caso, su correspondiente clase (un *servlet* es una clase normal) debe contener la implementación del método *doPost*.

doPost toma dos parámetros: un objeto de clase *HttpServletRequest*, por el que accedemos a los datos que el cliente nos manda, y un objeto de clase *HttpServletResponse*, que no permite entregarle los resultados. La siguiente figura muestra la posible implementación de este método en el *servlet*, contenido en la clase *identificar.java* (el nombre de la clase se corresponde con el último fragmento de la URL).

Un *servlet*, una página JSP o ASP o CSP o PHP o de cualquier otro tipo puede aprovechar por completo todas las funcionalidades de su correspondiente lenguaje de programación. En este caso, el *servlet* va a comprobar que el nombre y contraseña enviados desde el formulario se encuentren almacenados en una base de datos a la que se va acceder por medio de un agente de base de datos (Broker).

Casi lo primero que se hace es crear una instancia del Broker pasándole como argumento el valor del parámetro nombre que se ha recibido desde el formulario (este valor se lee mediante la instrucción *req.getParameter("nombre")*, donde *req* es el objeto de clase *HttpServletRequest*) así como la dirección IP del cliente (se lee mediante *req.getRemoteAddr()*). Lo que pase ahora dependerá evidentemente de la implementación del constructor *Broker(String, String)*, pero podemos suponer que lo que realmente pasa es que se establece la conexión a la base de datos.

A continuación se crea *usu*, un objeto de clase *Usuario* pasando como argumentos el broker que se acaba de crear, el nombre de usuario y la contraseña. Este constructor de *Usuario* comprueba que el nombre y la contraseña existen en la base de datos asociada al Broker que se ha instanciado en la instrucción anterior.

Podemos suponer que si el establecimiento de la conexión o la identificación del usuario falla, el control del programa saltará al bloque *catch* situado más abajo. En caso de que ambas sentencias hayan funcionado, se procede con la siguiente instrucción.

```

public void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException
{
    resp.setContentType("text/html");
    PrintWriter out = new PrintWriter(resp.getOutputStream());
    try {
        Broker bd=new Broker(req.getParameter("nombre"), req.getRemoteAddr());
        Usuario usu=new Usuario(bd, req.getParameter("nombre"), req.getParameter("pwd"));
        ...
    }
    catch (Exception e) {
        req.getSession(true).setAttribute("mensaje", e.getMessage());
        out.println("<html><body onload=\"window.location.href='./error.jsp';\"></body></html>");
    }
    out.close();
}

```

Figura 157. Fragmento del método *doPost* del servlet que recibe el nombre y la contraseña enviados desde el formulario de la Figura 156

2.02 *http* es un protocolo sin estado

http es un protocolo “sin estado”, lo que significa que el servidor no guarda información de la conexión con sus clientes: es decir, que aunque el cliente se conecte e identifique correctamente, si no conservamos de alguna manera la información suya que nos sea relevante, en la segunda página que visitara habríamos perdido sus datos de conexión e identificación y podríamos, por ejemplo, vetarle el acceso, prohibirle la ejecución de operaciones, etc.

Por ello, todas las tecnologías de desarrollo de aplicaciones web incluyen una clase específica que permite almacenar información de la conexión: la sesión (en entornos Java es la *HttpSession*). La sesión consiste realmente en una *cookie* que se guarda en el disco del cliente y que dura lo que dura la conexión. Por eso, si prohibimos totalmente las *cookies* en nuestro navegador de Internet, tendremos serias dificultades para interactuar, por ejemplo, con nuestra oficina bancaria virtual. Por eso es importante también, por motivos de seguridad, usar las opciones de desconexión o de cerrar sesión que habitualmente incluyen las aplicaciones web.

En el ejemplo, nos habíamos quedado con que la conexión a la base de datos se había establecido y con que el usuario se había identificado correctamente. Con el fin de que, por toda su visita al portal, el servidor conozca los datos del cliente que está conectado, debemos entonces guardar alguna información en dicha sesión. La Figura 158 incluye el mismo código que la figura anterior más dos nuevas líneas, marcadas en negrita, en las que se recupera un objeto sesión y se colocan en él dos objetos: el Broker, que da así asignado a este cliente para que realice a través de él todas las operaciones de

base de datos, y el usuario, con su nombre, tal vez la contraseña y otros datos que puedan ser de interés.

En Java, la sesión se recupera mediante una llamada al método *getSession(boolean)* del objeto de tipo *HttpServletRequest* pasada como parámetro al *doPost*. Cada conexión puede tener asociada una sola sesión. Si el valor del parámetro es *true*, se le crea una sesión nueva, destruyéndose sin preguntar más todo lo que hubiera en la sesión antigua; si el valor es *false*, el método devuelve (si la hay) la sesión de la conexión. En el ejemplo de la Figura 158, se le pasa el valor *true* porque el cliente acaba de identificarse, por lo que se supone que acaba de llegar al portal y que no tenía sesión. En las siguientes páginas que visite recuperaremos la sesión pasándole valor *false*, porque se estará utilizando la sesión creada al haber identificado al usuario.

```
public void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException
{
    resp.setContentType("text/html");
    PrintWriter out = new PrintWriter(resp.getOutputStream());
    try {
        Broker bd=new Broker(req.getParameter("nombre"), req.getRemoteAddr());
        Usuario usu=new Usuario(bd, req.getParameter("nombre"), req.getParameter("pwd"));
        HttpSession sesion=req.getSession(true);
        sesion.setAttribute("bd", bd);
        sesion.setAttribute("usuario", usu);
        ...
    }
    catch (Exception e) {
        req.getSession(true).setAttribute("mensaje", e.getMessage());
        out.println("<html><body onload=\"window.location.href='./error.jsp';\"></body></html>");
    }
    out.close();
}
```

Figura 158. Fragmento del método *doPost* del servlet que recibe el nombre y la contraseña enviados desde la Figura 156, continuación de la Figura 157

Por último, si queremos que el navegador del cliente muestre un mensaje cuando el usuario se ha identificado correctamente, el servlet le puede enviar una cadena (normalmente en formato HTML) a través de un objeto de tipo *PrintWriter*, que se habrá recuperado a partir del objeto *HttpServletRequest*:

```

public void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException
{
    resp.setContentType("text/html");
    PrintWriter out = new PrintWriter(resp.getOutputStream());
    try {
        Broker bd=new Broker(req.getParameter("nombre"), req.getRemoteAddr());
        Usuario usu=new Usuario(bd, req.getParameter("nombre"), req.getParameter("pwd"));
        HttpSession sesion=req.getSession(true);
        sesion.setAttribute("bd", bd);
        sesion.setAttribute("usuario", usu);
        String s="<html><head>" +
            "</head>" +
            "<body onload=\"window.location.href= '../menu.jsp';\">" +
            "</body>" +
            "</html>";
        out.println(s);
    }
    catch (Exception e) {
        req.getSession(true).setAttribute("mensaje", e.getMessage());
        out.println("<html><body onload=\"window.location.href='../error.jsp';\"></body></html>");
    }
    out.close();
}

```

Figura 159. Fragmento del método *doPost* del servlet que recibe el nombre y la contraseña enviados desde la Figura 156, continuación de la Figura 158

2.03 Arquitectura de una aplicación web

En principio, una aplicación web puede ofrecer tantas funcionalidades como una “de escritorio”, siendo además deseable que el código de una sea reutilizable para el desarrollo de la otra. Si usamos una arquitectura multicapa, se pondrá especial cuidado en diseñar adecuadamente la capa de dominio, con el fin de que sirva para las dos aplicaciones.

Así, por ejemplo, en el caso de la aplicación bancaria, el siguiente método, que inserta un producto en la base de datos (este código aparece en la página 58), no sería utilizable para una aplicación web del estilo de la que hemos descrito en el epígrafe anterior:

```

public void insert() throws Exception {
    String SQL="Insert into Producto (Codigo, NIFCliente) values (" +
        mPKCodigo + ", '" + mCliente.getNIF() + "'"");
    Broker.getBroker().insert(SQL);
}

```

¿Por qué? Porque, según se ha comentado, cada cliente tiene asociada una sesión en la que se almacena, entre otras informaciones, el Broker que utiliza para realizar las operaciones de persistencia. En el código de arriba, el agente se recupera mediante una llamada al método *getBroker()* de la clase *Broker*, que es un singleton. Que el agente sea un singleton interesa poco para un aplicación web, que va a tener visitas múltiples y

simultáneas. Lo que haremos será pasar el Broker como parámetro al método *insert*. La aplicación de escritorio deberá tener esto en cuenta si queremos que una sola capa de dominio sirva para el desarrollo de las dos aplicaciones:

```
public void insert(Broker bd) throws Exception {
    String SQL="Insert into Producto (Codigo, NIFCliente) values (" +
        mPKCodigo + ", '" + mCliente.getNIF() + "'"");
    bd.insert(SQL);
}
```

Como en la aplicación del escritorio, el usuario usa las funcionalidades de la aplicación mediante la capa de presentación. En el caso de la aplicación web, el usuario interactúa enviando datos a URLs mediante formularios, y estas URLs serán servlets, páginas JSP, ASP o de cualquier otro tipo. La URL receptora de los datos del formulario será la encargada de recuperar el Broker del objeto sesión, actualizar el estado del objeto de dominio que se esté manipulando, y llamar al correspondiente método (un método CRUD, por ejemplo, pero puede ser cualquier otro método de negocio) pasándole si es preciso el Broker como parámetro.

Esta secuencia se resume en los dos siguientes pasos:

1) El usuario rellena los datos de un cliente en una página web y pulsa el botón “Guardar nuevo”:

Ficha de Cliente

Los campos indicados con * son obligatorios

Nif*:

Nombre:

Apellido1:

Apellido2:

Teléfono:

Domicilio:

Código postal:

Localidad:

Provincia:

[Ayuda p](#)

Figura 160

2) Los datos se envían mediante un post a un servlet llamado “gestCliente”. El servlet mira qué botón se ha pulsado en la página anterior para saber la operación que debe realizar. Esto se consigue dando a todos los botones el mismo *name* pero distinto *value* (p.ej.: `<input type=submit name=boton value='Guardar nuevo'>` o `<input type=submit name=boton value='Modificar'>`).

```

...
HttpSession sesion=request.getSession(false);
if (sesion!=null) {
    String BOTON=request.getParameter("BOTON");
    Usuario usu=(Usuario) sesion.getAttribute("usuario");
    Broker bd=(Broker) sesion.getAttribute("bd");

    if (BOTON.equals("Guardar nuevo")) {
        try {
            Cliente o=new Cliente();
            cargaObjeto(request, o, usu, sesion, IConstantesBot ones.SAVE_NEW);
            o.insert(bd);
            out.println("<html><body> .... </body></html>");
        }
        catch (Exception e) {
            sesion.setAttribute("mensaje", e.getMessage());
            out.println("<html><body> .... </body></html>");
        }
    }
}
...

```

Figura 161. Fragmento del método *doPost* en el servlet que recibe los datos del formulario mostrado en la Figura 160

2.04 Ejercicio

El sistema de gestión bancaria permite la manipulación de los datos de su base de datos mediante una aplicación web y una aplicación de escritorio que comparten capa de dominio. Ambas aplicaciones permiten la obtención de listados de, por ejemplo, clientes, como se muestra en la siguiente figura:

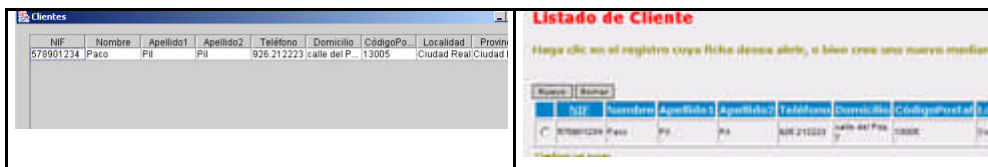


Figura 162

Para la obtención del listado de la derecha, el programador añadió un método *getListado* a la clase *dominio.Cliente*, que devuelve en formato HTML la tabla que se muestra. Explique si esta solución es adecuada o inadecuada y por qué.

2.05 Ejercicio

¿Qué desventajas tiene usar un agente de base de datos singleton en una aplicación web?

3. Introducción a los Servicios web

Mediante los servicios web, un cliente puede ejecutar un método en un equipo remoto, transportando la llamada (hacia el servidor) y el resultado (desde el servidor al

cliente) mediante protocolo *http* (normalmente). La idea es servir la misma funcionalidad que permiten otros sistemas de invocación remota de métodos, como RMI (que vimos en el capítulo 7), pero de un modo más portable.

La portabilidad se consigue gracias a que todo el intercambio de información entre cliente y servidor se realiza en SOAP (*Simple Object Access Protocol*), que un protocolo de mensajería basado en XML: así, la llamada a la operación consiste realmente en la transmisión de un mensaje SOAP, el resultado devuelto también, etc. De este modo, el cliente puede estar construido en Java y el servidor en .NET, pero ambos conseguirán comunicarse gracias a la estructura de los mensajes que intercambian.

Los servidores ofrecen una descripción de sus servicios web en WSDL (*Web Services Description Language*), que es una representación en XML del servicio ofrecido. Así, un cliente puede conocer los métodos ofrecidos por el servidor, sus parámetros con sus tipos, etc., simplemente consultando el correspondiente documento WSDL.

3.01 WSDL

Supongamos que nuestro sistema de gestión bancario utiliza, para validar las operaciones realizadas con tarjeta de crédito, el siguiente método remoto:

```
public boolean validar(String numeroDeTarjeta, double importe)
```

Si este método es accesible como un servicio web, debe estar descrito en WSDL, por ejemplo, del siguiente modo:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <!--
Generated by the Oracle9i JDeveloper Web Services WSDL Generator
-->
- <!--
Date Created: Thu Apr 22 11:00:35 CEST 2004
-->
<definitions name="VisaWS" targetNamespace="http://visa/ dominio/Visa.wsdl "
  xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://visa/ dominio/Visa.wsdl "
  xmlns:ns1="http://visa.dominio/IVisaWS.xsd">
  <types>
    <schema targetNamespace="http://visa.dominio/IVisaWS.xsd"
      xmlns="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-
      ENC="http://schemas.xmlsoap.org/soap/encoding/" />
    <types>
    <message name="validar0Request">
      <part name="numeroDeTarjeta" type="xsd:string" />
      <part name="importe" type="xsd:double" />
    </message>
    <message name="validar0Response">
      <part name="return" type="xsd:boolean" />
    </message>
    <portType name="VisaPortType">
      <operation name="validar">
        <input name="validar0Request" message="tns:validar0Request" />
        <output name="validar0Response" message="tns:validar0Response" />
      </operation>
```

```

</portType>
<binding name="VisaBinding" type="tns:VisaPortType">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
  <operation name="validar">
    <soap:operation soapAction="" style="rpc" />
    <input name="validar0Request">
      <soap:body use="encoded" namespace="VisaWS"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding" />
    </input>
    <output name="validar0Response">
      <soap:body use="encoded" namespace="VisaWS"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding" />
    </output>
  </operation>
</binding>
<service name="VisaWS">
  <port name="VisaPort" binding="tns:VisaBinding">
    <soap:address location="" />
  </port>
</service>
</definitions>

```

Figura 163. Descripción en WSDL del método de validación anterior

De la figura anterior merece la pena destacar algunos elementos:

<pre> <message name="validar0Request"> <part name="numeroDeTarjeta" type="xsd:string" /> <part name="importe" type="xsd:double" /> </message> <message name="validar0Response"> <part name="return" type="xsd:boolean" /> </message> <portType name="VisaPortType"> <operation name="validar"> <input name="validar0Request" message="tns:validar0Request" /> <output name="validar0Response" message="tns:validar0Response" /> </operation> </portType> </pre>	<p>Nombre del método accesible de forma remota, nombres y tipos de los parámetros. El postfijo <i>Request</i> denota el formato en que debe enviarse la solicitud al servidor. Cuando el cliente invoca el servicio, envía un mensaje <i>validar0Request</i>.</p> <p>Tipo del resultado devuelto por el método. El postfijo <i>Response</i> se refiere precisamente a que es el tipo devuelto lo que se está representando.</p> <p>Operaciones que conforman la interfaz del servicio <i>validar</i>, que se corresponden con los dos <i>messages</i> anteriores.</p>
---	---

Figura 164. Significado de algunos elementos del WSDL mostrado en la figura anterior

Los entornos de desarrollo recientes incluyen los *add-ins* necesarios para generar la especificación WSDL de una clase.

3.02 Escritura de un cliente que acceda a un servicio web

El cliente que utiliza el servicio web necesita una clase que actúe como *proxy* entre él mismo y el servicio web ofertado por el servidor. Cuando el *proxy* recibe del cliente una solicitud de llamada al servicio web, el *proxy* la traduce a un mensaje SOAP, que envía al servidor; éste, entonces, lo ejecuta, y devuelve un mensaje SOAP al *proxy*; éste,

entonces, traduce el mensaje a objetos Java, .NET, etc. y entrega el resultado al cliente que efectuó la petición.

La siguiente figura muestra la relación entre el *proxy*, el servicio web y la clase que lo implementa: el servicio web (en el centro) ofrece a los clientes acceso a la operación *validar(String, Double)*; la operación se encuentra realmente implementada en la clase situada abajo (*Visa*), y podría incluir llamadas a otros métodos de otras clases, acceso a una base de datos, acceso a otros servicios web, etc. El elemento de la izquierda (*VisaWSSStub*) es la clase que actúa de proxy entre los clientes y el servicio web. Nótese que esta clase incluye, además de otras, la operación *validar(String, Double)*. El proxy mostrado se ha obtenido de forma automática con un entorno de desarrollo, por lo que sus miembros pueden variar de unos casos a otros. El atributo *_endpoint* representa la URL en la que se encuentra publicado el servicio web.

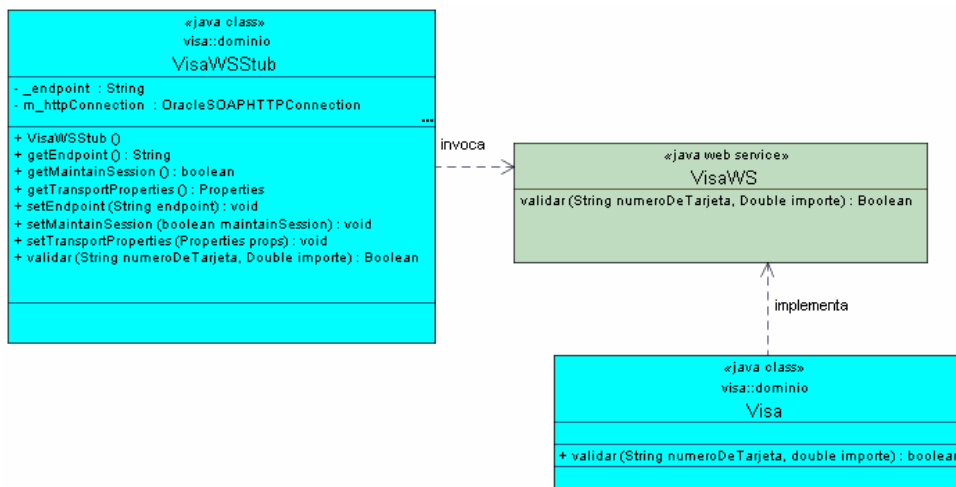


Figura 165. El proxy, el servicio web y la clase que lo implementa

La aplicación cliente hace entonces uso del proxy para acceder al servicio web, por ejemplo con un trozo de código como el que sigue:

```

protected void validarDisponibilidadDeCredito(double importe)
    throws Exception
{
    // Se instancia el proxy
    VisaWSSStub stub = new VisaWSSStub();
    // Se le dice al proxy dónde puede encontrar el servicio web
    stub.setEndpoint("http://161.67.27.108:8988/soap/servlet/soaprouter");
    // Se llama al método ofertado por el proxy
    if (!stub.validar("", new Double(5.0)).booleanValue())
        throw new Exception("Operación no admitida");
}

```

Obsérvese que el tipo del segundo parámetro de la llamada a *validar* es de tipo *Double* (con mayúsculas) y no *double* (con minúsculas), a pesar de que el método toma un *double* en la clase en la que se encuentra implementado. Esto es así porque el generador del proxy ha optado por incluir aquel tipo de dato en lugar de éste.

Cuando el tipo de un parámetro o del resultado es un tipo complejo, es preciso realizar una descripción del tipo (o de la clase) en el documento WSDL. Supongamos que la clase *Visa* ofrece la siguiente operación:

```
public dominio.Tarjeta getTarjetaConMayorCredito()
```

La clase *dominio.Tarjeta* es un tipo complejo y debe especificarse en el documento que describe el servicio web. El entorno de desarrollo es capaz de añadir al WSDL la descripción estandarizada de esta clase:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <!--
Generated by the Oracle9i JDeveloper Web Services WSDL Generator
-->
- <!--
Date Created: Thu Apr 22 18:00:42 CEST 2004
-->
<definitions name="VisaWS" targetNamespace="http://visa/dominio/Visa.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://visa/dominio/Visa.wsdl"
  xmlns:ns1="http://visa.dominio/IVisaWS.xsd">
  <types>
    <schema targetNamespace="http://visa.dominio/IVisaWS.xsd"
      xmlns="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-
      ENC="http://schemas.xmlsoap.org/soap/encoding/">
      <complexType name="dominio_Tarjeta" jdev:packageName="dominio"
        xmlns:jdev="http://xmlns.oracle.com/jdeveloper/webservices">
        <all>
          <element name="numero" type="string" />
          <element name="titular" type="string" />
          <element name="validaDesde" type="string" />
          <element name="validaHasta" type="string" />
        </all>
      </complexType>
    </schema>
  </types>
  <message name="getTarjetaConMayorCreditoRequest" />
  <message name="getTarjetaConMayorCreditoResponse">
    <part name="return" type="ns1:dominio_Tarjeta" />
  </message>
```

Figura 166. El WSDL incluye ahora la descripción del tipo complejo “Tarjeta”

3.03 Ejercicio práctico

Google ofrece un servicio web que permite ejecutar búsquedas en su buscador. Obviamente, el servicio está implementado en las máquinas de Google, pero puede construirse un cliente que utilice el servicio usando su documento WSDL.

Se pide que escriba un pequeño programa que invoque el servicio web de Google, de modo que posibilite la realización de búsquedas y que muestre los resultados en un frame o por la consola. También puede, aprovechando lo visto sobre aplicaciones web al principio de este capítulo, desarrollar una sencilla aplicación web que realice búsquedas en Google utilizando el servicio web en lugar de una llamada directa a la URL.

El documento WSDL está accesible en: <http://api.google.com/GoogleSearch.wsdl>. En <http://api.google.com> hay ejemplos de código y documentación adicional.

3.04 Ejercicio

Rediseñe el sistema mostrado en la Figura 103 utilizando servicios web.

3.05 Ejercicio

Rediseñe el sistema mostrado en la Figura 103 para que se permita el acceso tanto mediante RMI como mediante servicios web.

4. Introducción a las aplicaciones basadas en componentes distribuidos

En terminología UML, un componente es “una parte física reemplazable de un sistema que empaqueta su implementación, y que es conforme a un conjunto de interfaces a las que proporciona su realización” (Rumbaugh et al., 1999). Esto significa, sencillamente, que el componente ofrece una serie de servicios (métodos) a sus usuarios a través de una interfaz; las operaciones incluidas en la interfaz se implementan dentro de la lógica del componente.

Un componente, por tanto, representa una unidad de construcción reutilizable, que puede ensamblarse para formar aplicaciones. Algunos autores consideran que, en un futuro, la mayor parte de las aplicaciones se desarrollarán ensamblando componentes.

Cuando el componente se ejecuta en un servidor y de alguna manera se hace accesible a los clientes (que residen normalmente en distintas máquinas), se habla entonces de “componentes distribuidos”. Los servicios web, que se han visto con brevedad en la anterior sección de este capítulo, pueden considerarse una forma de componentes distribuidos: ofrecen un conjunto de servicios que describen su WSDL, aceptan llamadas remotas vía *http*, ejecutan las operaciones ofrecidas y devuelven resultados.

No obstante, al hablar de componentes distribuidos se piensa más en otras tecnologías, como DCOM (de Microsoft) o EJB (de Sun). En ambas tecnologías existe una

lógica intermedia que captura las llamadas a los servicios que realizan los clientes y las pasan al componente.

4.01 DCOM

En el caso de DCOM (*Distributed Component Object Model*), del lado del cliente existe un middleware que intercepta las llamadas del cliente, las convierte a paquetes DCOM y las envía por la red al servidor; éste reconvierte estos paquetes en llamadas al componente, que ejecuta la operación (Figura 167).

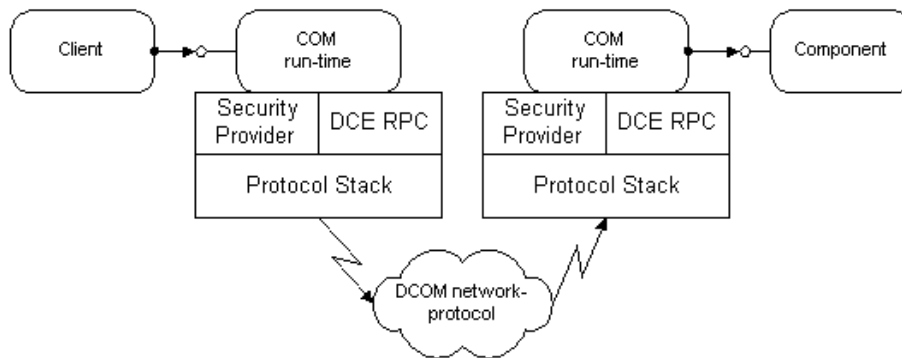


Figura 167. DCOM: componentes COM en diferentes máquinas (tomada de *DCOM Technical Overview*, disponible en msdn.microsoft.com)

4.02 EJB

La estructura de las aplicaciones basadas en EJB (*Enterprise JavaBeans*) es bastante parecida: los clientes acceden a los componentes a través de un contenedor que se ejecuta en el servidor. Lo que el contenedor publica son dos interfaces por cada componente: una interfaz *home*, que ofrece servicios de localización de instancias de componentes, y una interfaz *remota*, que ofrece los métodos de negocio. El componente como tal implementa las operaciones contenidas en ambas interfaces, pero de forma indirecta. La siguiente figura ilustra estas ideas: el contenedor (recuadro grueso del lado derecho) almacena y gestiona el acceso a los componentes (en este caso, el componente *AccountEJB*); a su vez, el contenedor publica una interfaz *AccountHome* con los métodos necesarios para instanciar componentes (incluirá, por ejemplo, un método para instanciar la cuenta corriente número 1234-5678-90-1234567890). Instanciado el componente, el cliente puede ejecutar los servicios ofrecidos en la interfaz *Account*, pudiendo por tanto ejecutar las operaciones de negocio *deposit*, *withdraw* y *open*.

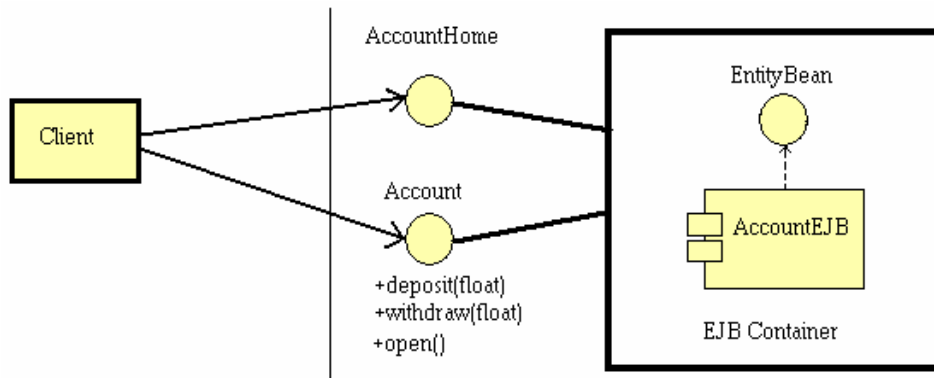


Figura 168. Lados cliente y servidor de una aplicación con EJB

Como se ve en la figura, no existe realmente una relación directa de implementación entre el componente y las interfaces; de hecho, ni siquiera los métodos en las interfaces y en la clase que las implementa se llaman igual, aunque sí que deben cumplir unos criterios de nombrado.

Al igual que ocurre con los servicios web, el cliente no tiene manera de ejecutar de forma directa llamadas a los constructores de la clase que encapsula al componente: lo que se le ofrece son una serie de métodos, cuyo nombre comienza por *findBy*, que, al ejecutarse en el EJB, instancian el componente, devolviendo al cliente una referencia remota. En el EJB, por cada método *findByX* incluido en la interfaz *home* existe un método con la misma signatura, pero cuyo nombre es *ejbFindByX*. Los métodos de negocio, ofertados en la interfaz remota, se implementan en el EJB con el mismo nombre. Esto se ilustra en la Figura 169.

4.02.1 Tipos de EJBs

Existen diferentes tipos de EJBs: *entity beans*, *session beans* y *message driven beans*.

Los primeros (*entity beans*) representan entidades que tienen existencia en algún mecanismo de almacenamiento persistente (ficheros, bases de datos...), y suelen diseñarse considerándolos como objetos CRUD. De hecho, el EJB incorpora estas operaciones en forma de los métodos *ejbLoad* (que instancia el componente a partir de la información contenida en la base de datos, correspondiéndole una instrucción *Select* del SQL o equivalente), *ejbStore* (que actualiza en la base de datos el estado de la instancia del componente, correspondiéndole una instrucción *Update* o similar), *ejbRemove* (que elimina de la base de datos la instancia del componente que ejecuta la operación, corres-

pondiéndole una instrucción *Delete* del SQL) y *ejbCreate* (que inserta en la base de datos una nueva instancia del componente). Como se ha dicho, cualquier llamada a un método del EJB que realice un cliente es interceptada por el contenedor, que es quien decide sobre su ejecución. Habitualmente, cuando el cliente solicita la ejecución de un método de negocio del componente (es decir, un método incluido en la interfaz remota, la interfaz *Persona* en el ejemplo de la Figura 169), el contenedor actualiza primero el estado de la instancia llamando a *ejbLoad*, luego ejecuta el método de negocio y después actualiza el estado en la base de datos llamando a *ejbStore*. Como se ve, el trasiego de bytes entre el contenedor y el disco es muy intenso. El contenedor de componentes, sin embargo, permite modificar la configuración de la ejecución para que las operaciones se ejecuten de otra manera. Hay multitud de fabricantes de contenedores de componentes, como Sun (fabricante del J2EE), WebLogic, Oracle, iPlanet, IBM, Apache (fabricante de Geronimo), Alpha (de Pramati) y otros.

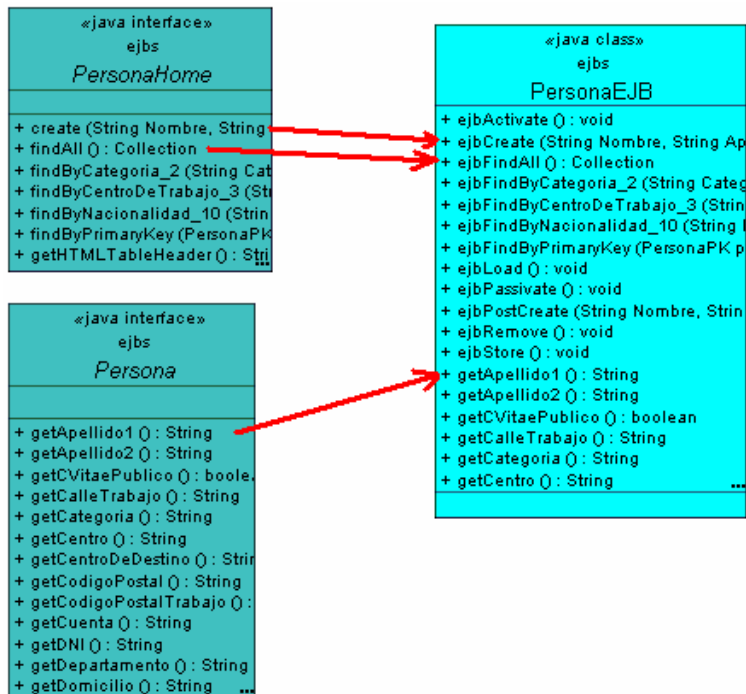


Figura 169. Relaciones entre los nombres de los servicios ofertados en las interfaces y los nombres de los correspondientes métodos en el EJB

Los EJBs del segundo tipo (*session beans*) ejecutan servicios para el cliente, pero sin considerar la gestión de la persistencia (aunque pueden, de hecho, utilizarla). Suelen distinguirse dos subtipos, con estado (*stateful*) y sin estado (*stateless*). Los EJBs sin

estado ejecutan un servicio, devuelven el resultado y se olvidan del cliente. Una calculadora simple sería un buen ejemplo de un EJB sin estado: el EJB recibe una cadena como parámetro, que el EJB interpreta como una expresión matemática que debe calcular: el EJB la calcula, devuelve el resultado al cliente y asunto concluido. Una calculadora con memoria podría ser un ejemplo de un EJB con estado: el cliente realiza una operación, el EJB la calcula y sigue aceptando expresiones para seguir recalculando.

Los EJBs de tipo *message driven* se utilizan para el envío y procesamiento de mensajes asíncronos. A diferencia de los otros dos tipos, éstos no tienen interfaz remota ni interfaz *home*; los clientes localizan el EJB mediante el destino JMS (*Java Messaging Service*), lo que hace las veces de interfaz *home*; la interfaz remota queda reemplazada por el conjunto de mensajes JMS que se envían al componente.

4.02.2 Ciclo de vida de los EJB

Como ya se ha dicho, el cliente no manipula directamente el EJB, sino que es el contenedor el que gestiona las llamadas a los servicios ofrecidos por los componentes.

Por otro lado, y dado que las aplicaciones que utilizan EJBs soportarán normalmente multitud de usuarios, el contenedor dispondrá de alguna política que permita optimizar la gestión de sus recursos. De este modo, el contenedor puede disponer de un *pool* en el que almacenar aquellas instancias de EJBs que no se estén utilizando en algún momento: cuando un cliente solicita la ejecución de un servicio sobre una instancia que se encuentra en el *pool*, el contenedor la recupera, llevando al *pool* (si es preciso) a una instancia distinta.

Los EJBs de tipo *entity* y los de tipo *stateful* poseen, entre otros, los métodos *ejbPassivate* y *ejbActivate*, que se encargan respectivamente de llevar al y de traer del *pool* instancias de componentes. La figura siguiente muestra, en la forma de una máquina de estados, el ciclo de vida de un EJB de tipo *entity*: cada vez que el contenedor crea una instancia del EJB, llama a su método *setEntityContext* de la clase que implementa el componente, colocando la instancia en el *pool*. Desde el *pool*, la instancia del componente puede pasar al estado preparado (*ready*) de dos maneras: o bien invocando a su método *create*, si lo que se quiere es insertar la instancia en el mecanismo persistente, o bien porque el contenedor invoque al método *ejbActivate*. Para regresar desde el estado preparado al estado *pooled*, debe ocurrir o bien que el contenedor ejecute el método *ejbPassivate*, o bien que el cliente invoque al método *remove*.

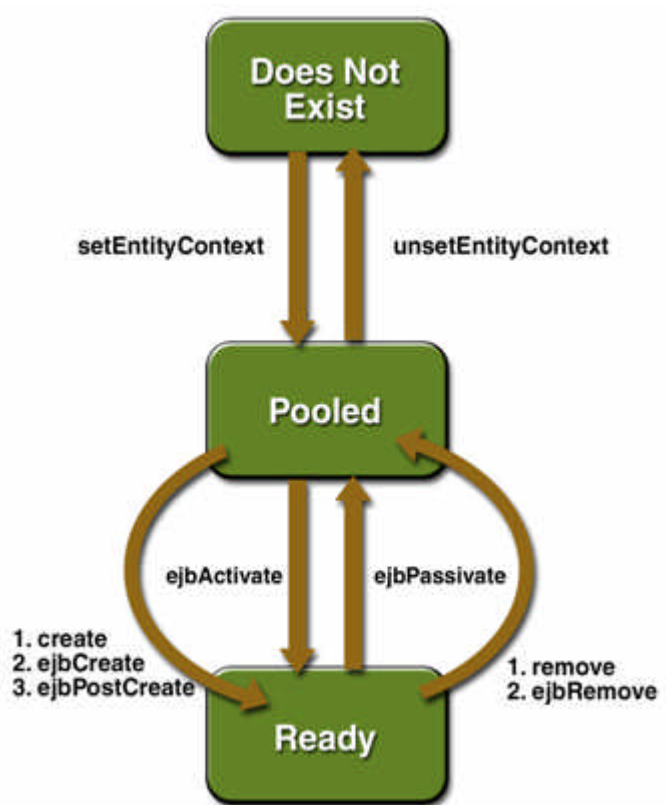


Figura 170. Ciclo de vida de un EJB de tipo entity (tomada del tutorial de Sun)

También podemos ver estos cambios de estado si instrumentamos adecuadamente el código de los métodos en el EJB. Para ilustrarlo, supongamos que añadimos a sus métodos una instrucción para que se muestre en una línea con el nombre del método que se ha ejecutado. En un escenario en el que el cliente obtenga, a través de la interfaz *home*, una referencia a un EJB y luego ejecute sobre ella un método de negocio, la lista de métodos que se ejecuta sería la siguiente:

```

ejbFindByPrimaryKey
ejbActivate
ejbLoad
getNombre (método de negocio)
ejbStore
ejbPassivate
  
```

4.03 Acceso al EJB desde el cliente

El contenedor debe asignar un nombre a cada uno de los componentes que publica. Los clientes recuperan referencias al componente a partir de este nombre. En el siguiente fragmento de código, el cliente recupera una referencia al interfaz *home* del EJB:

```
InitialContext ic = new InitialContext();
Object objRef = ic.lookup("PersonaEJB");
PersonaHome home=(PersonaHome) PortableRemoteObject.narrow(objRef, PersonaHome.class);
```

El cliente debe conocer el nombre con el que se publica el componente (*PersonaEJB* en este ejemplo); entonces, a través de un contexto de ejecución, el cliente obtiene la referencia *home*, de tipo *PersonaHome*. Sobre *home*, el cliente puede ejecutar cualquiera de los métodos de localización de instancias que ofrezca la interfaz *home* correspondiente. El siguiente código podría ser la continuación del fragmento anterior:

```
Collection col=home.findAll();
Iterator it=col.iterator();
while (it.hasNext()) {
    Persona ejb=(Persona) it.next();
    out.print(ejb.toHTMLTableRowString());
}
```

Como se observa, el cliente ejecuta sobre *home* una llamada a su método *findAll* (ofrecido en la interfaz *PersonaHome*, e implementado en *PersonaEJB* con el nombre *ejbFindAll*) para recuperar una colección de instancias de tipo *Persona* (la interfaz remota de *PersonaEJB*). En *PersonaEJB*, el método *ejbFindAll* devuelve una colección de objetos cuyo tipo es la clave principal de las personas: es decir, devuelve una colección de objetos de clase *PersonaPK*. Sin embargo, la colección devuelta por el método *findAll* en la interfaz *PersonaHome* está formada por instancias de *Persona*: de la conversión de uno a otro tipo se encarga el contenedor.

En el código de arriba, el método *toHTMLTableRowString* que se ejecuta sobre la instancia *ejb* de tipo *Persona*, es un método de negocio ofrecido por la interfaz remota (la interfaz *Persona*) e implementado de alguna manera en la clase que encapsula el EJB.

4.04 Algunas ideas sobre Ingeniería del Software Basada en Componentes

La Ingeniería del Software Basada en Componentes (CBSE, por sus siglas en inglés: *Component-Based Software Engineering*) se basa en que muchos de los grandes

sistemas software tienen una base común suficiente que justifica la inversión en el desarrollo de componentes reutilizables.

Asumiendo que el desarrollo tradicional de software comienza por una fase de análisis de requisitos, es muy posible que los ingenieros de software encuentren una y otra vez requisitos iguales o muy parecidos en aplicaciones diferentes. Como sabemos, los requisitos son finalmente satisfechos implementando operaciones en clases diversas, por lo que deberemos detectar el conjunto de clases que satisfacen ese conjunto de requisitos que queremos transformar en un componente software independiente y reutilizable en múltiples aplicaciones (como es fácilmente comprensible, en muchas ocasiones se satisfacen requisitos en fragmentos de clases, lo que llevará a análisis más concienzudos que busquen clases abstractas, clases asociadas, etc.).

Adaptación

Detectado un conjunto de requisitos susceptible de reutilizar, podemos optar por implementarlo como un nuevo componente o por buscarlo en alguna biblioteca de componentes interna o externa (en el mercado). En el caso de que decidamos reutilizar un componente preconstruido, puede ocurrir que éste se adapte total o sólo parcialmente a los requisitos deseados. En este último caso, podrá negociarse o bien una adaptación de los requisitos con el cliente, o bien plantearse la adaptación del componente (habitualmente no se dispone del código fuente de componentes procedentes de bibliotecas externas, por lo que la adaptación puede resultar muy dificultosa); en el primer caso se comprobará que el ahorro en costes de desarrollo sea conveniente con respecto al coste de adquisición y distribución del componente.

Cualificación

En cualquier caso, debemos pensar que la reutilización de componentes nos hace jugar el doble rol de desarrolladores (integraremos y utilizaremos el componente dentro de nuestra aplicación) y de usuarios (utilizaremos, al menos durante el desarrollo, la funcionalidad suministrada por el componente), por lo que haremos un doble análisis del componente: como usuarios, evaluaremos la cualificación del componente para cumplir su cometido; como desarrolladores, evaluaremos características como su interfaz, herramientas de desarrollo e integración necesarias, requisitos hardware (memoria, etc.), requisitos software (necesidad de otros componentes, sistema operativo, etc.), seguridad y forma en que maneja las excepciones.

Composición

En muchos casos ocurre que el componente no es directamente integrable en la aplicación, o que necesita colaborar con otros componentes para satisfacer los requisitos deseados. En estos casos es preciso realizar un esfuerzo adicional de integración mediante, por ejemplo, la escritura de código que conecte ambos componentes (a este código se le denomina *glue-code*), para lo que es conveniente recordar el Patrón Adaptador (página 143).

De manera general, los componentes deben:

- 1) Reflejar conceptos fundamentales del dominio que van a cambiar poco
- 2) Ocultar la forma en que se presenta su estado, pero proporcionando operaciones que permitan manipularlo.
- 3) Ser lo más independientes posible.
- 4) Permitir el manejo de excepciones al programador, ya que su tratamiento dependerá de la aplicación en la que se utilice el componente.

5. Nociones sobre Sistemas de Información Geográfica

[Fuente principal: SOPDE; www.sopde.es]

De acuerdo con el NCGIA (*National Center for Geographic Information and Analysis*, con sede en la Universidad de California), un Sistema de Información Geográfica (SIG o, de las siglas inglesas, GIS), es “un sistema de hardware, software y procedimientos diseñados para soportar la captura, gestión, manipulación, análisis, modelado y visualización de datos espacialmente referenciados para resolver problemas complejos de planeamiento y gestión”. Los SIG están diseñados para recolectar, almacenar y analizar objetos y fenómenos cuya localización geográfica es importante.

La cantidad de información de un SIG puede ser tremendamente grande, y se le pueden requerir niveles de detalle muy pequeños, como se muestra en la Figura 171. Por ello, se requieren mecanismos de almacenamiento y algoritmos de procesamiento eficientes, que trabajaran normalmente con tipos de datos específicos.

5.01 Tipos de datos en SIG

Los SIG almacenan información gráfica y alfanumérica, teniendo cada uno características específicas para garantizar su almacenamiento, procesamiento y representación eficaces.

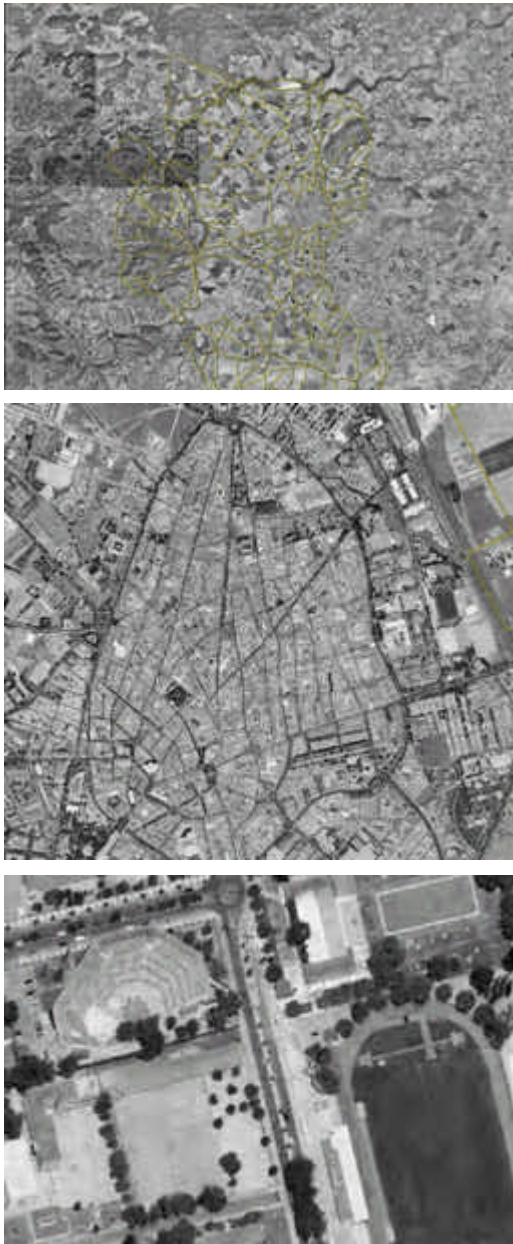


Figura 171. Imagen de los alrededores de Ciudad Real (arriba); la ciudad casi al completo (centro); y detalle del edificio Fermín Caballero (abajo), junto al campo de fútbol del polideportivo (tomadas SIG oleícola del Ministerio de Agricultura, Pesca y Alimentación, en www.mapya.es)

Los datos gráficos incluyen generalmente coordenadas, reglas y símbolos que definen los elementos cartográficos del mapa. Para representar datos gráficos se utilizan normalmente tres tipos de entidades:

(a) Nodos, que representan puntos significativos, como intersecciones o puntos terminales. Constituyen la entidad básica para representar entidades con posición pero sin dimensión (aunque esto depende de la escala).

(b) Líneas, que son elementos de una dimensión definidos por un nodo de inicio y otro de fin.

(c) Polígonos, que son elementos bidimensionales limitados por líneas.

Los datos alfanuméricos se almacenan normalmente en formatos convencionales, aunque últimamente comienzan a utilizarse sistemas de información documental para este tipo de aplicaciones.

Ambos tipos de información (gráfica y alfanumérica) deben estar perfectamente integradas, siendo esta integración una de las principales características de los SIG.

Para representar en el sistema los elementos que componen el mundo real, es preciso abstraer estos elementos a diferentes formas, como puntos, líneas, polígonos, imágenes o etiquetas alfanuméricas, así

como almacenar éstos en algún formato eficiente. Para ello es preciso tener en cuenta las diferentes relaciones existentes entre las entidades:

(a) Las *relaciones topológicas* se refieren a la posición relativa de los elementos contenidos en el SIG.

(b) Mediante las *relaciones de clasificación*, se agrupan en la misma categoría diferentes elementos del mundo real: municipios, calles, ríos, puertos de montaña, etc.

(c) Con las *relaciones de agregación* se representa el hecho de que un elemento del mundo real está compuesto de otros elementos (en el ejemplo de la figura anterior, dentro del elemento “Ciudad Real” está contenido el “Edificio Fermín Caballero”).

Es preciso, además, almacenar de alguna manera las imágenes gráficas. Los formatos más habituales son el *raster* (de cada línea se almacenan todos sus puntos) y el *vectorial* (de cada línea se almacenan únicamente sus puntos de origen y destino).

Cuando se utiliza *raster*, el espacio que se representa se divide en rectángulos cuyo tamaño determina la resolución de la imagen. Así, la precisión en la referencia a cada punto geográfico (*georreferenciación*) viene limitada por la porción de territorio que representa cada rectángulo.

El formato vectorial se basa fundamentalmente en la utilización de nodos, líneas y polígonos. Obviamente, el detalle con que se almacenen las relaciones entre estos elementos supone un compromiso entre volumen de almacenamiento, velocidad de procesamiento y modelado preciso de la realidad. Con respecto al raster, el formato vectorial permite la realización de cálculos de forma muy precisa; por contra, su procesamiento es más complejo.

La información alfanumérica complementa a la gráfica, a la cual describe. Puesto que habitualmente ambas informaciones residen en mecanismos de almacenamiento distintos, deben existir en los dos almacenes identificadores que relacionen la información gráfica con la alfanumérica.

Algunas de las aplicaciones habituales de los SIG son las siguientes:

- (1) Cartografía automatizada.
- (2) Gestión de infraestructuras.
- (3) Gestión territorial.
- (4) Gestión medioambiental.
- (5) Gestión de equipamientos sociales.
- (6) Gestión de recursos geológico-mineros.
- (7) Gestión del tráfico, incluso en tiempo real (Figura 172).

- (8) Demografía.
- (9) Márketing.
- (10) Logística.



Figura 172. Dos vistas distintas en tiempo real del tráfico aéreo en el aeropuerto internacional de Boston, a 90 millas y 10 millas del centro ciudad (<http://www4.passur.com/bos.html>)

CAPÍTULO 15. OTRAS ARQUITECTURAS

6. Introducción

El término “arquitectura software” es probablemente demasiado amplio y demasiado vago y por eso es probable que no tengamos, a estas alturas de los apuntes, una idea clara de lo que es exactamente. Una buena definición por exclusión es aquella que lo define como “la parte del diseño del sistema que no es diseño detallado”.

A lo largo de este documento nos hemos referido repetidas veces a la arquitectura multicapa y a ella hemos dedicado los capítulos tercero y sexto; sin embargo, existen otras arquitecturas posibles más o menos estándares, que vamos a inspeccionar, en algunos casos muy brevemente, en las siguientes páginas.

7. Arquitectura *pipeline*, de “tubería y filtro” o de “flujo de datos”

Estas arquitecturas derivan de sistemas descritos mediante Diagramas de Flujo de Datos: a grandes rasgos (consúltase la documentación de asignaturas anteriores de Ingeniería del Software), estos diagramas describen procesos que aplican alguna transformación a sus datos de entrada produciendo salidas. Idealmente, cada proceso no necesita conocer el modo de trabajo del resto de procesos.

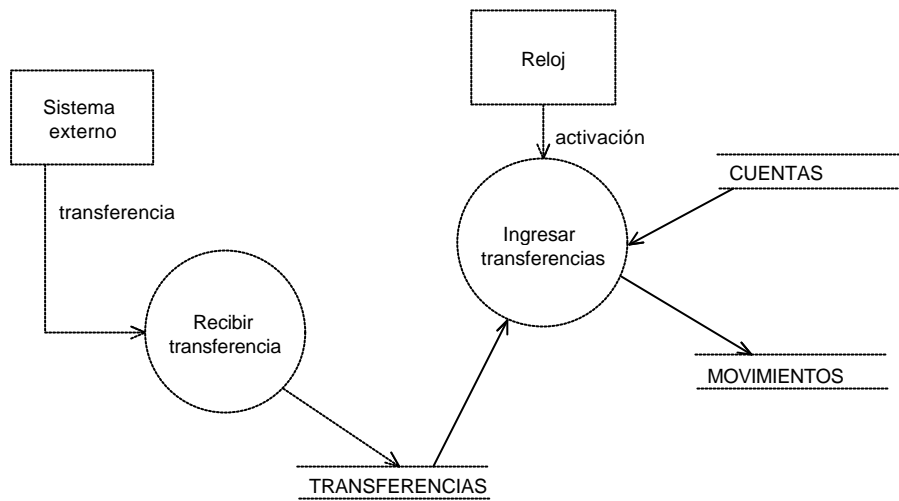
Para que un proceso funcione, necesita únicamente datos de entrada en un cierto formato, y produce datos de salida en otro formato. A cada proceso receptor de información se le denomina “filtro” (puede interpretarse que la transformación que realiza a los datos de entrada es una especie de “filtrado”), y a los flujos que transportan los datos entre procesos diferentes se les llama “tuberías”.

7.01 Ejemplo

Supongamos que nuestro sistema bancario recibe de sistemas externos información de transferencias cuyos destinatarios son clientes de nuestra entidad. Las transferencias son recogidas según van llegando por un proceso de recepción, que las deja guardadas en un almacén intermedio; por la noche, se dispara un proceso de ingreso que lee las

transferencias del almacén intermedio y realiza los ingresos en las cuentas correspondientes.

Mediante un DFD, este modo de procesamiento puede representarse de la siguiente manera:



$\text{transferencia} = \text{CuentaOrigen} + \text{CuentaDestino} + \text{Importe} + \text{Concepto}$

Figura 173. Un pequeño sistema descrito mediante un DFD

Aunque no hay una forma automática y estándar para obtener un diagrama de clases a partir de un DFD, la heurística, nuestra experiencia y el conocimiento que tengamos del sistema puede llevarnos a entender que cada proceso va a transformarse en una operación de alguna clase. Del enunciado del problema podemos suponer que tendremos dos clases: una Receptora y otra Pagadora. Podríamos añadir a aquella la operación *recibir(Transferencia)* y, a ésta, una operación que vaya leyendo todas las transferencias del almacén, que se disparará al llegar la señal del reloj, y otra operación que las vaya ingresando.

8. Arquitectura de componentes

La Ingeniería del Software Basada en Componentes se basa en que muchos de los grandes sistemas software tienen una base común suficiente que justifica la inversión en el desarrollo de componentes reutilizables.

ÍNDICE DE TÉRMINOS

- .NET, 205
- 100%. Véase Regla del 100%
- IAHIT, 51
- ICIT, 53
- ICHIT, 52
- abstracción*, 7
- Abstract factory, 137
- abstracta*, 7
- acoplamiento, 60
- Actividad, 126
- Adapter*, 143
- Agente, 54, 98
- agregación, 13
- Agregaciones, 38
- Almacenamiento*, 16
- alta cohesión*, 60
- aplicaciones web, 196
- Arquitectura de tres capas, 47
- arquitectura multicapa, 16, 47
- artefactos*, 26
- asociación, 13
- Asociaciones, 36
- Asociaciones n-arias, 38
- Assert*, 160
- association class*, 37
- atributos, 33
- Bag*, 169
- bajo acoplamiento*, 60
- base de datos, 50
- Broker*, 54
- Builder*, 140
- Caché, 96
- Cachés, 96
 - Tipos, 100
- Cadena de log, 193
- Capa de Almacenamiento*, 16
- capa de Dominio, 48
- Capa de Dominio*, 16
- Capa de Persistencia, 49
- capa de Presentación, 48
- Capa de Presentación*, 16
- caso de uso
 - descripción con una máquina de estados, 126
- casos de uso*, 27, 71
- Casos de uso
 - Asociaciones, 71
 - Extensiones, 71
- Chain of responsibility*, 151
- Clase, 1, 31
- clase asociativa*, 37
- clase base*, 4
- clase derivada*, 4
- Clases, 31
- clases *amigas*, 2
- Clases anidadas, 41
- Clases parametrizadas, 41
- clases persistentes*, 17
- cohesión, 60
- Collection*, 169
- Compartición de instancias, 98
- compartimentos, 33
- componente, 209
- componentes distribuidos, 209
- composición, 38
- Composite*, 145
- Connection*, 55
- contenedor, 212
- contenedores de componentes, 212
- Contratos, 75
- createRegistry*, 109
- CRUD*, 18, 53, 57
- DCOM, 209
- dependencia, 14
- Dependencias, 40
- diagrama de colaboración*, 12
- diagrama de paquetes, 47
- diagrama de secuencia*, 3, 12
- Diagramas de actividad, 131
- Diagramas de casos de uso, 71
- diagramas de clases, 31
 - Verificación y validación, 43
- diagramas de colaboración, 74
- Diagramas de colaboración, 68
- diagramas de estados (y OCL), 177
- diagramas de interacción*, 12, 65
- diagramas de secuencia, 65, 74
- Dominio*, 16, 48, 57
- EJB, 103, 209
 - entity beans, 211
 - message driven, 213
 - session beans, 212
 - Tipos, 211
- encapsulación, 2
- Enterprise Java Beans*, 103
- entity beans*, 211
- Enumeraciones, 41
- Enumeraciones (en OCL), 178
- enumeration*, 41
- escenario*, 3, 4, 74
 - normal, 74
- especializaciones*, 4
- estereotipos, 2, 32
- estructura estática, 31
- evento del sistema*, 75
- eventos del sistema, 76
- Experto, 57
- eXtreme Programming*, 28, 155
- fábrica abstracta, 137
- Fabricación pura, 58
- Facade*, 146

- Fachada, 146
- Fixture*, 193
- flujo de eventos*, 3
- Flyweight, 147
- focos de control*, 67
- Generalización, 39
- GIS, 217
- guarda, 121
- Herencia, 4, 39
- Herencia múltiple, 6
- http*, 200
- incremento de cohesión, 60
- Ingeniería del Software Basada en Componentes, 215
- Instancia, 2
- integración continua*, 28
- integridad referencial, 54
- Interfaces, 35
- interfaz*, 7
- interfaz remota*, 105
- Interpreter*, 152
- introspección, 60
- Invariantes, 167
- Invocación Remota de Métodos. Véase RMI*
- iteraciones, 26
- java.rmi*, 105
- java.rmi.server*, 105
- JavaScript, 197
- junit*, 159
- Máquinas de estados, 115
- Mediator*, 151
- mensaje, 3
- Mensajes (en OCL), 179
- message driven*, 213
- metodologías, 26
- miembros*, 1
- modelo en V, 29
- Modelo-Vista-Controlador, 91
- multicapa, 16
- mutación, 155
- Object Modeling Technique*, 42
- Objeto, 1
- objeto remoto
 - conexión, 108
- Objetos Mock, 192
- objetos remotos, 106
- Observador*, 91, 95
- OCL, 167
 - Bag, 169
 - Collection, 169
 - diagramas de estados, 177
 - enumeraciones, 178
 - mensajes, 179
 - OclAny, 168
 - OclExpression, 168
 - OclType, 167
 - Sequence, 169
 - Set, 169
- Ocultamiento, 2
- OMT, 42
- operación del sistema*, 75
- operaciones, 34
- operaciones CRUD*, 18, 53, 57
- operaciones del sistema, 76
- páginas activas, 197
- pair programming*, 28
- paquetes, 47
- paso de mensajes*, 3, 65
- Patrón 1C1T, 51
- patrón *Fixture*, 193
- Patrones
 - Agente, 54
 - altacohesión, 60
 - bajo acoplamiento, 60
 - Broker, 54
 - Experto, 57
 - Fabricación pura, 58
 - RCRUD, 60
 - Reflective CRUD, 60
 - Singleton, 56
 - un árbol de herencia, una tabla, 51
 - un camino de herencia, una tabla, 52
 - una clase, una tabla, 51
- Patrones de diseño, 137
- Persistencia, 49
- Plan de iteraciones*, 27
- plantillas, 41
- polimorfismo, 4
- Polimorfismo, 4
- pool*, 111
- postcondiciones, 75, 76, 120
- Postcondiciones, 167
- precondición, 121
- precondiciones, 75, 76
- Precondiciones, 167
- Presentación*, 16, 48
- Proceso Unificado, 26, 74
 - fases, 27
- producto software*, 26
- Productor-Consumidor*, 95
- Programación dirigida por las pruebas, 184
- Programación Extrema, 28
- Programación Extrema, 155
- programación por parejas, 28
- protocolos*, 7
- Proxy*, 150
- pruebas, 155
- Pruebas de excepciones, 193
- pruebas funcionales*, 28
- pruebas unitarias*, 28
- raster*, 219
- RCRUD, 60
- refactoring*, 29
- Reflective CRUD*, 60
- reflexión, 60
- regla "es un"*, 4
- regla del 100%*, 4
- relaciones entre clases, 13
- remota*, 105
- Remote Method Invocation*, 105

Request, 197
Response, 197
reutilización, 16
RMI, 105
script, 197
script de servidor, 196
Sequence, 169
Serialización, 103
serializar, 112
Servicios web, 204
servlet, 199
sesión, 200
session beans, 212
Set, 169
SIG, 217
Singleton, 56
Sistema orientado a objetos, 4
Sistemas de Información Geográfica, 217
sistemas orientados a objetos, 31
SOAP, 205
Sobrecarga, 12
software, 26
State, 153
subcapas, 49
subclase, 4
subestados, 123
subtipo, 4
superclase, 4
supertipo, 4
TablaHash, 97
TestCase, 160
test-driven development, 184
tiempo de desarrollo ideal, 182
Tipos, 41
Tipos de EJBs, 211
tres capas, 47
un camino de herencia, una tabla, 52
una clase, una tabla, 51
UnicastRemoteObject, 105
Utilidades, 40
utilities, 40
variables de los proyectos de desarrollo de software, 183
vectorial, 219
verificación y validación, 29
Verificación y validación
 de contratos, 77
 de diagramas de casos de uso, 73
 de diagramas de clases, 43
 de diagramas de interacción, 70
vinculación dinámica, 9
vinculación postergada, 9
visibilidad, 2
Visual Basic Script, 197
WSDL, 205
XML, 205
XP, 28, 155
 historias del usuario, 181
 Plan de entregas, 182
 Planificación, 181