



**Escuela Superior de Informática
Universidad de Castilla-La Mancha**

MANUAL DE INGENIERÍA DEL SOFTWARE II

Macario Polo Usaola

ÍNDICE

Primera parte	15
Capítulo 1. Conceptos fundamentales del paradigma orientado a objetos	17
1. Objeto y clase	17
2. Herencia y polimorfismo	19
3. Relaciones entre clases	21
4. Ocultamiento y visibilidad.....	22
5. Representación de los conceptos anteriores en UML	23
5.1. Representación de campos.....	24
5.2. Representación de operaciones	24
5.3. Herencia	25
5.4. Relaciones entre clases.....	26
6. Representación de los conceptos anteriores en lenguajes de programación.....	28
7. Conceptos sobre diseño arquitectónico.....	30
7.1. Políticas básicas de gestión de la persistencia	31
7.2. Diseño de la capa de dominio	40
7.3. Diseño de la capa de presentación.....	41
7.4. Organización en subsistemas	47
8. Testing y verificación	47
9. Herramientas CASE para desarrollo de software orientado a objetos.....	48
10. Lecturas recomendadas	48
Capítulo 2. Metodologías de desarrollo de software orientado a objetos.....	49
1. OMT: Object Modelling Technique	49
2. Desarrollo dirigido por las pruebas (Test-Driven Development)	51
3. El Proceso Unificado de Desarrollo.....	51
3.1. Fases del Proceso Unificado de Desarrollo	52
3.2. Algunos modelos importantes	56
4. Lecturas recomendadas	67
Capítulo 3. Descripción de un problema.....	69
1. Enunciado	69
2. Adaptación del problema al Proceso Unificado.....	70
3. Lecturas recomendadas	71
Segunda parte	73
Capítulo 4. Aplicación de administración: fase de comienzo.....	75
1. Construcción del diagrama de casos de uso	75
2. Priorización de casos de uso	79
3. Descripción textual de los casos de uso	81
3.1. Descripción textual de casos de uso	83
4. Verificación y validación	91
5. Lecturas recomendadas	92

Capítulo 5. Aplicación de administración: fase de elaboración (I)	93
1. Desarrollo del caso de uso <i>Identificación</i>	93
1.1. Diagrama de secuencia “de análisis” para el flujo de eventos normal.....	94
1.2. Obtención de un modelo de diseño	96
1.3. Obtención de casos de prueba a partir de un flujo de eventos o diagrama de secuencia	105
2. Desarrollo del caso de uso <i>Crear edición</i>	106
3. Desarrollo del caso de uso <i>Crear casillas vacías</i> (primera parte)	108
4. Creación de la base de datos.....	109
4.1. Patrón <i>una clase, una tabla</i>	110
4.2. Patrón <i>un árbol de herencia, una tabla</i>	113
4.3. Patrón <i>un camino de herencia, una tabla</i>	115
4.4. Otras influencias del patrón seleccionado en el código de utilización de la base de datos	116
5. Desarrollo del caso de uso <i>Crear casillas vacías</i> (segunda parte)	117
6. Desarrollo del caso de uso <i>Asignar tipo a casilla</i>	120
7. Desarrollo del caso de uso <i>Crear parking</i>	123
8. El patrón <i>Singleton</i>	130
9. Pruebas del código	131
9.1. Un proceso de pruebas combinando caja negra con caja blanca	132
9.2. Pruebas funcionales con <i>junit</i>	133
9.3. Pruebas de caja blanca	139
9.4. Pruebas del caso de uso <i>Crear parking</i>	143
10. Lecturas recomendadas	151
Capítulo 6. Aplicación de administración: fase de elaboración (II)	153
1. Desarrollo del caso de uso <i>Crear salida</i>	153
2. Desarrollo del caso de uso <i>Crear impuestos</i>	155
3. Introducción a OCL.....	158
3.1. Tipos predefinidos de OCL.....	158
3.2. Colecciones.....	160
3.3. La operación <i>iterate</i>	163
3.4. Navegación entre asociaciones	163
3.5. Ejemplos y notación	164
3.6. Escritura de aserciones en el programa	171
3.7. Utilización de las aserciones para derivar casos de prueba	172
3.8. Interpretación de las aserciones como contratos	173
4. Desarrollo del caso de uso <i>Crear calle</i>	173
4.1. Anotación del diagrama de clases con OCL	175
4.2. Creación de las tablas <i>Calle y Barrio</i>	177
5. Lecturas recomendadas	178
Capítulo 7. Aplicación de administración: fase de elaboración (III)	179

1. Resto de actividades de la fase de Elaboración	179
2. Análisis de algunos casos de uso	180
3. Consideraciones generales sobre el diseño detallado	180
3.1. Diseño de la base de datos	180
3.2. Métodos de acceso a la base de datos	182
3.3. Utilización de herencia	182
3.4. Decisiones sobre el desarrollo de otros casos de uso	185
3.5. Modificación del plan de iteraciones	186
4. Descripción arquitectónica del sistema	186
4.1. Vista funcional de la arquitectura	186
4.2. Vista de subsistemas con diagramas de paquetes	187
5. Lecturas recomendadas	187
Capítulo 8. Aplicación de administración: fase de construcción	189
1. Estado actual de los requisitos	189
2. Desarrollo del caso de uso <i>Crear barrio</i>	189
3. Desarrollo del caso de uso <i>Crear tarjeta</i>	190
3.1. Escenario correspondiente a la inserción de una tarjeta	195
3.2. Escenario correspondiente a la visualización de las tarjetas	199
3.3. Escenario correspondiente a la eliminación de las tarjetas	201
4. Posibles refactorizaciones	201
5. Pruebas de clases abstractas con JUnit	203
6. Lecturas recomendadas	204
Capítulo 9. Aplicación de administración: fase de transición	205
Capítulo 10. Resumen	207
1. XXX	207
Tercera parte	209
Capítulo 11. Aplicación servidora (I)	211
1. Recolección de requisitos	211
2. El patrón <i>Fachada</i>	212
3. Elección del middleware	213
4. Desarrollo del caso de uso <i>Identificarse</i>	218
4.1. Pruebas de <i>identificar</i>	221
5. Desarrollo del caso de uso <i>Crear partida</i>	223
5.1. Pruebas del código con varios clientes	226
6. Desarrollo del caso de uso <i>Consultar partidas abiertas</i>	228
7. Desarrollo del caso de uso <i>Unirse a partida</i>	229
8. El patrón <i>Intérprete</i>	230
9. Aplicación del patrón <i>Intérprete</i> al cliente de pruebas	231
10. Desarrollo de los casos de uso <i>Comenzar partida</i> y <i>Tirar dados</i>	233
10.1. Pruebas del caso de uso	238
10.2. Los valores “interesantes” en la fase de pruebas	241

11. Lecturas recomendadas	244
Capítulo 12. Aplicación servidora (II)	245
1. Presentación.....	245
2. Preparación del tablero	245
3. Reparto de dinero.....	247
4. Lanzamiento de los dados en el estado <i>EnJuego</i>	247
5. Compra directa de una calle	248
6. Compra directa de una estación	250
7. Caída en una casilla poseída por otro jugador	251
7.1. Escenario 1: pago directo	251
7.2. Escenario 2: negociación entre jugadores.....	255
7.3. Control de la tirada de dobles	258
7.4. Tres dobles seguidos.....	259
Capítulo 13. Aplicación servidora (III)	263
1. Máquinas de estados	263
1.1. Ejemplos.....	266
1.2. Derivación de especificaciones OCL a partir de máquinas de estados	268
2. Descripción de clases de la aplicación servidora con máquinas de estados	269
3. Lecturas recomendadas	269
Capítulo 14. Aplicación servidora (IV): construcción de un monitor de seguimiento... 270	
1. Análisis del problema	270
1.1. Carga del tablero en <i>JPTablero</i> con una fábrica simple.....	275
1.2. Carga del tablero en <i>JPTablero</i> con un <i>Builder</i> (Constructor)	276
1.3. Carga del tablero en <i>JPTablero</i> con una <i>Abstract factory</i> (Fábrica abstracta) 278	
1.4. Carga del tablero en <i>JPTablero</i> con una fábrica introspectiva	279
1.5. Resultado visual.....	282
2. Seguimiento del juego	282
Cuarta parte	285
Capítulo 15. Aplicación cliente.....	287
1. Introducción.....	287
Capítulo 16. Parámetros de calidad en software orientado a objetos	289
1. Auditoría	289
2. Medición de código	290
2.1. Métricas para sistemas orientados a objeto	290
2.2. Clasificación y utilidad de las métricas anteriores.....	292
3. Modificación del código y nuevos patrones	294
3.1. Patrón Fabricación Pura	295
3.2. Patrón Plantilla de métodos.....	296
3.3. Ventajas y desventajas de la utilización de patrones	298
4. Lecturas recomendadas	298

Apéndice I. Notación de UML.....	301
----------------------------------	-----

Índice de tablas

Tabla 1. Descripción resumida de algunas características del Proceso Unificado.....	56
Tabla 2. Descripción de algunos términos	57
Tabla 3. Notación para representar clases de análisis.....	62
Tabla 4. Tipos de casillas en el juego del Monopoly.....	77
Tabla 5. Posibles acciones ordenadas por las tarjetas de Suerte y de Caja de Comunidad.....	77
Tabla 6. Plan de iteraciones	81
Tabla 7. Descripción textual del caso de uso <i>Identificación</i>	84
Tabla 8. Descripción textual del caso de uso <i>Crear edición</i>	86
Tabla 9. Descripción textual de <i>Crear casillas vacías</i>	87
Tabla 10. Descripción textual del caso de uso <i>Asignar tipo a casilla</i>	90
Tabla 11. Descripción textual del caso de uso <i>Crear parking</i>	91
Tabla 12. Lista de comprobación para casos de uso	92
Tabla 13. Lista de comprobación para diagramas de secuencia	105
Tabla 14. Caso de prueba 1, “en positivo”	105
Tabla 15. Caso de prueba 2, “en negativo”	106
Tabla 16. Caso de prueba 3, “en negativo”	106
Tabla 17. Algunos operadores de mutación	140
Tabla 18. Descripción textual del caso de uso <i>Crear salida</i>	153
Tabla 19. Descripción textual del caso de uso <i>Crear impuestos</i>	155
Tabla 20. Tipos básicos escalares y sus operaciones	159
Tabla 21. Atributos de <i>OclType</i>	159
Tabla 22. Atributos y operaciones de <i>OclAny</i>	160
Tabla 23. El único miembro de <i>OclExpression</i>	160
Tabla 24. Operaciones de <i>Collection</i>	161
Tabla 25. Operaciones de <i>Set(T)</i> , donde <i>T</i> es el tipo de los elementos del conjunto.....	161
Tabla 26. Operaciones de <i>Bag(T)</i> , en donde <i>T</i> es el tipo de los elementos de la bolsa	162
Tabla 27. Operaciones de <i>Sequence(T)</i> , siendo <i>T</i> el tipo de los elementos de la secuencia	162
Tabla 28. Sintaxis de la restricciones sobre operaciones (pre y postcondiciones)	167
Tabla 29. Valores obtenidos para probar el supuesto método <i>solicitar</i> del Cuadro 53.....	173
Tabla 30. Descripción textual del caso de uso <i>Crear calle</i>	175
Tabla 31. Algunos casos de uso cuyo análisis no se ha abordado	180
Tabla 32. Nueva versión del plan de iteraciones	186
Tabla 33. Estado actual de los requisitos	189
Tabla 34. Tipos de estados	265
Tabla 35. Métricas de acoplamiento para objetos	293
Tabla 36. Umbrales de algunas métricas recomendados por la NASA	293
Tabla 37. Fragmento de la tabla con algunas métricas de la Aplicación para investigadores ...	294

Índice de figuras

Figura 1. Representación de la clase <i>Persona</i> en UML (diagrama realizado con el entorno de desarrollo Oracle JDeveloper 10g)	24
Figura 2. Relaciones de herencia de la superclase <i>Persona</i> con tres subclases	25
Figura 3. Diagrama con herencia, una asociación y una composición	26
Figura 4. Algunas posibles relaciones entre clases	28
Figura 5. Adición de operaciones CRUD a la clase <i>Empleado</i>	34
Figura 6. Delegación de las responsabilidades de persistencia a una clase asociada	37
Figura 7. Diseño alternativo al de la Figura 6	37
Figura 8. Otro diseño alternativo	38
Figura 9. Creación de una superclase <i>Persistente</i>	40
Figura 10. Una posible ventana para subir los sueldos	42
Figura 11. Diagrama de clases en el que la ventana conoce a un <i>Empleado</i> , al que envía mensajes	42
Figura 12. Diseño alternativo al de la Figura 11, en donde hay una mutua relación de conocimiento	44
Figura 13. Adición de un observador para un diseño alternativo a los anteriores	45
Figura 14. Adición de una interfaz para lograr un mayor desacoplamiento	47
Figura 15. Representación en UML de un paquete	47
Figura 16 (adaptada de Larman, 2001). El PUD es iterativo e incremental (los círculos representan el tamaño del sistema)	52
Figura 17. Estructura general del Proceso Unificado de Desarrollo	53
Figura 18. Flujos de trabajo del Proceso Unificado	53
Figura 19. Diagrama de casos de uso para una biblioteca	58
Figura 20. El mismo diagrama que en la figura anterior, pero delimitando el sistema	58
Figura 21. El mismo diagrama de casos de uso, dibujado con Rational Rose	59
Figura 22. El diagrama de casos de uso de la biblioteca, con nuevos casos de uso y relaciones entre ellos	60
Figura 23. El mismo diagrama, con dos relaciones de herencia entre actores y entre casos de uso	61
Figura 24. Relación entre el caso de uso Mantenimiento de préstamos y las clases que le darán servicio	62
Figura 25. Adición de nuevas clases a la descripción del caso de uso	63
Figura 26. La Figura 25, pero usando estereotipos en lugar de iconos	64
Figura 27. Modelo de clases de análisis procedente del caso de uso Mantenimiento de préstamos	64
Figura 28. Derivación de clases de diseño a partir de clases de análisis	65
Figura 29. Descripción muy general de las relaciones entre actores y clases de diseño, y entre clases de diseño y clases de diseño	66
Figura 31. Vista funcional (I) de la aplicación de administración	76

Figura 32. Vista funcional (II) de la aplicación de administración	79
Figura 33. Plantilla de descripción de casos de uso del plugin de Eclipse	82
Figura 34. Clases de análisis involucradas en el caso de uso <i>Identificación</i>	83
Figura 35. Clases de análisis seleccionadas	84
Figura 36. Documento procedente de las entrevistas con el cliente.....	85
Figura 37. Clases de análisis involucradas en el caso de uso <i>Crear edición</i>	86
Figura 38. Clases de análisis para <i>Crear casillas vacías</i>	87
Figura 39. Boceto de la ventana para el caso de uso <i>Asignar tipo a casilla</i>	88
Figura 40. Clases de análisis identificadas para <i>Asignar tipo a casilla</i>	89
Figura 41. Diagrama de secuencia para el escenario correspondiente al flujo de eventos normal	95
Figura 42. Diseño de la clase <i>JFIdentificacion</i> utilizando el plugin <i>Visual Editor</i> del entorno de desarrollo <i>Eclipse</i>	97
Figura 43. Diagrama de secuencia “de diseño”	98
Figura 44. Tabla <i>Usuarios</i> de la base de datos	99
Figura 45. Diagrama de secuencia modificado	100
Figura 46. Diagrama de colaboración procedente de la transformación automática del diagrama de secuencia de la Figura 45.....	101
Figura 47. Diagrama de secuencia correspondiente al escenario de error.....	102
Figura 48. Parte del código procedente del diagrama de secuencia.....	103
Figura 49. Diagrama de clases procedente de los diagramas de secuencia anteriores	104
Figura 50. Escenario normal del caso de uso <i>Crear edición</i>	107
Figura 51. Diagrama de clases con las nuevas adiciones.....	107
Figura 52. Nueva versión del diagrama de clases.	109
Figura 53. Diagrama de clases de una biblioteca.....	110
Figura 54. Diagrama resultante de aplicar el patrón una clase, una tabla al diagrama de clases de la Figura 53.....	111
Figura 55. La aplicación estricta del patrón a veces no es recomendable	111
Figura 56. Adición de una columna que no procede de ningún campo	112
Figura 57. Diagrama resultante de aplicar el patrón una árbol de herencia, una tabla al diagrama de clases de la Figura 53	114
Figura 58. Diagrama resultante (versión 1) de aplicar el patrón un árbol de herencia, una tabla	115
Figura 59. Separación de los distintos préstamos	116
Figura 60. En esta alternativa, cada Edición está compuesta de muchas instancias de <i>Casilla</i> (que es abstracta), que posee varias especializaciones	118
Figura 61. En esta otra, <i>Casilla</i> es concreta, y posee un <i>Tipo</i> abstracto, especializado al subtipo correspondiente a la casilla	118
Figura 62. Vista parcial del posible diseño de la base de datos	119
Figura 63. Diseño alternativo de la base de datos	120
Figura 64. Prototipo de la ventana para configurar las casillas del tablero	121

Figura 65. Creación de un subtipo de <i>JButton</i> para manipular casillas	121
Figura 66. Acciones realizadas al pulsar el botón correspondiente a una casilla	122
Figura 67. Cada objeto <i>JBCasilla</i> conoce el tablero sobre el que está situado	122
Figura 68. La interfaz de usuario considerando la casilla del <i>Párking</i>	124
Figura 69. Comportamiento deseado para la casilla del <i>Párking</i>	125
Figura 70. Hacemos que los diferentes <i>JPanels</i> implementen una interfaz	126
Figura 71. Adición de un evento a un componente visual en Eclipse	126
Figura 72. Implementación del método <i>seleccionarSolapa</i> , que se ejecuta al seleccionar una solapa en <i>JFTablero</i>	127
Figura 73. Implementación de <i>setCasilla</i> en <i>JPParking</i>	127
Figura 74. Fragmento del diseño de clases	129
Figura 75. La clase <i>Cuenta</i> que deseamos probar	133
Figura 76. Resultado de ejecutar la clase de prueba del Cuadro 30	135
Figura 77. Registro con información de un usuario en la tabla <i>Usuarios</i>	138
Figura 78. Mutantes generados para la clase <i>Sesion</i>	141
Figura 79. Resultados de la ejecución de los casos de prueba	143
Figura 80. Para esta prueba, requerimos estos datos en la tabla <i>Casilla</i>	144
Figura 81. Superación del caso de prueba	147
Figura 82. El segundo caso de prueba encuentra un error	148
Figura 83. Un procedimiento almacenado	150
Figura 84. Llamada al procedimiento almacenado de la Figura 83	151
Figura 85. Se crea la clase <i>Salida</i> como especialización de <i>Tipo</i>	154
Figura 86. Un buen entorno de desarrollo facilita la escritura del código	156
Figura 87. Adición de una nueva tabla a la base de datos	156
Figura 88. Diseño de la solapa <i>JPImpuestos</i> y parte de su código	157
Figura 89. Dos ejemplos para ilustrar la navegación entre asociaciones	164
Figura 90. Un diagrama de clases para anotar con OCL	164
Figura 91. Jerarquía en Java de los objetos <i>Throwables</i>	171
Figura 92. Descripción de <i>Crear calle</i> con una máquina de estados	174
Figura 93. Fragmento del diagrama de clases, con <i>Calle</i> como subclase de <i>Tipo</i>	176
Figura 94. Adición de las tablas a la base de datos	177
Figura 95. Diseño de la base de datos	181
Figura 96. Reestructuración de las solapas para aprovechar la herencia	184
Figura 97. Fragmento del nuevo diseño de la capa de presentación	185
Figura 98. Diagrama de paquetes de la Aplicación para investigadores	187
Figura 99. Diálogo para la asignación de precios de adquisición de casas a los barrios	190
Figura 100. Estructura del subsistema de tarjetas	191
Figura 101. La tabla <i>Tarjeta</i>	192
Figura 102. Diseño de la ventana de creación de tarjetas (clase <i>JDTarjetas</i>)	193
Figura 103. Las solapas tienen un elemento común (un <i>JPComun</i>)	195
Figura 104. Las solapas conocen a un <i>JPComun</i>	195

Figura 105. Diagrama de colaboración correspondiente al inserción de una <i>Tarjeta</i> de tipo <i>IrA</i>	195
Figura 106. Cada tipo de solapa conoce a un determinado tipo de tarjeta	196
Figura 107. Esquema de la alternativa 1	197
Figura 108. Esquema de la alternativa 2	198
Figura 109. El diálogo <i>JDTarjetas</i> , mostrando información de una tarjeta	199
Figura 110. Descripción del escenario de visualización de la información de una tarjeta de tipo <i>IrA</i>	200
Figura 111. Descripción del escenario de eliminación de una tarjeta de tipo <i>IrA</i>	201
Figura 112. Clases contenidas en el paquete de dominio	202
Figura 113. Asistente para la reubicación de clases de Eclipse	202
Figura 114. Estructura del paquete de dominio después de refactorizar	203
Figura 115. Funcionalidades principales de la aplicación servidora	212
Figura 116. Un subsistema genérico con una fachada	213
Figura 117. El patrón Fachada, adaptado a nuestro problema	213
Figura 118. Exposición de los servicios de <i>Servidor</i> con <i>RMI</i>	215
Figura 119. Representación en UML del proxy de acceso al servicio web de <i>Solicitud</i>	218
Figura 120. Identificación por <i>RMI</i>	219
Figura 121. Conexión entre el dominio y la presentación	220
Figura 122. Escenario de arranque del <i>Servidor</i> y parte de la implementación	220
Figura 123. Escenario de identificación con notificación a la ventana	221
Figura 124. Comando de compilación con <i>rmic</i> (izquierda), y contenidos de la carpeta <i>dominio</i> antes (centro) y después (derecha) de la compilación	222
Figura 125. Al fondo, la instancia de <i>JFMain</i> ; en primer plano, <i>JUnit</i>	223
Figura 126. Adición de relaciones entre <i>Partida</i> y <i>Jugador</i>	224
Figura 127. Diseño del cliente de prueba, basado también en <i>RMI</i>	224
Figura 128. En el servidor, la clase <i>Servidor</i> conoce ahora a muchos clientes remotos	225
Figura 129. Código que vamos escribiendo en <i>crearPartida</i> (en <i>Servidor</i>)	226
Figura 130. El <i>workspace</i> consta de tres proyectos	226
Figura 131. Aspecto de la ventana del cliente de prueba	227
Figura 132. Escenario de conexión de dos clientes (al fondo, la ventana del servidor)	228
Figura 133. Ejemplo del patrón Intérprete	231
Figura 134. Estructura inicial de clases para el subsistema <i>cliente.parser</i>	232
Figura 135. <i>Expresion</i> conoce a <i>Cliente</i>	232
Figura 136. Paso de mensajes para procesar el comando de conexión al <i>Servidor</i>	233
Figura 137. Escenario de conexión de dos jugadores y de comienzo de una partida	236
Figura 138. Aplicación del patrón <i>Estado</i> a la <i>Partida</i> , que delega su estado a <i>Momento</i>	237
Figura 139. Dos jugadores decidiendo el turno	238
Figura 140. Mensajes enviados desde el servidor a los clientes al ejecutar el caso de prueba anterior	239
Figura 141. Modificación del caso de prueba y resultados obtenidos en la consola	240

Figura 142. Diagrama de flujo de <i>tirarDados</i> (del Cuadro 96)	242
Figura 143. Tablero que utilizaremos para continuar el desarrollo.....	245
Figura 144. Estructura de clases.....	246
Figura 145. Escenario normal correspondiente a la compra de una calle.....	249
Figura 146. Paso de mensajes para realizar un pago directo	253
Figura 147. Paso de mensajes al tirar los dados durante el juego.....	254
Figura 148. Mensajes enviados y recibidos por los clientes.....	256
Figura 149. Superación de la última versión del caso de prueba.....	258
Figura 150. Turno normal (izquierda) y turno en este escenario (derecha)	259
Figura 151. Sintaxis abstracta de las máquinas de estados. © OMG.....	264
Figura 152. Máquina de estados para una <i>Cuenta</i> (ejemplo 1)	267
Figura 153. Máquina de estados para una <i>Cuenta</i> (ejemplo 2)	268
Figura 154. Estructura formada por las principales clases del sistema.....	270
Figura 155. Escenario de la carga del panel de la partida	271
Figura 156. La <i>Partida</i> conoce a un <i>IVentanaTablero</i> , que puede instanciarse a un <i>JPTablero</i>	272
Figura 157. Resultado visual obtenido al crear una <i>Partida</i>	274
Figura 158. Jerarquía parcial de casillas “visuales” y una fábrica que se encarga de su construcción	275
Figura 159. Dos árboles de herencia en paralelo	276
Figura 160. Con el <i>Builder</i> , se asigna a uno de los objetos la responsabilidad de crear el resto del lote (izquierda). A la derecha, posible implementación de la operación en una de las especializaciones	277
Figura 161. Adición del <i>Director</i> al esquema de la Figura 160. A la derecha, <i>construirLote</i> en <i>Director</i>	277
Figura 162. Esquema de la solución con la Fábrica abstracta.....	278
Figura 163. Algunas de las clases Java que permiten la introspección	280
Figura 164. Casillas en el dominio y en la presentación.....	281
Figura 165. Aspecto de <i>JFSeguimiento</i> con el <i>JPTablero</i> correspondiente a una partida.....	282
Figura 166. Relación entre <i>Casilla</i> y su representación visual	284
Figura 167. Resultado de la auditoría de la <i>Aplicación para investigadores</i>	290
Figura 168. Operaciones estáticas, algunas de las cuales pueden ser llevadas a clases auxiliares	294
Figura 169. Refactorización para crear una fabricación pura para <i>Proyecto</i>	296

Primera parte

Capítulo 1. CONCEPTOS FUNDAMENTALES DEL PARADIGMA ORIENTADO A OBJETOS

En este capítulo se presentan los conceptos más importantes del paradigma orientado a objetos, como *objeto*, *clase*, *herencia*, etc. También se realiza una introducción breve al lenguaje UML, que se utiliza para representar y documentar sistemas orientados a objeto; se describe cómo hacer corresponder la representación del sistema con su implementación, y se realiza una introducción al diseño arquitectónico.

1. Objeto y clase

Podemos definir un sistema orientado a objetos como una comunidad de objetos que, en ejecución, colaboran entre sí y con otros sistemas para resolver uno o más problemas determinados. Un objeto es una instancia de una clase, y una clase es una plantilla mediante la que se describe la estructura y comportamiento de una familia de objetos.

La estructura se describe mediante una enumeración de campos, cada uno de los cuales tendrá, al menos, un nombre y un tipo de dato. El comportamiento se describe enumerando e implementando las operaciones que pueden actuar sobre las instancias de la clase. Normalmente se distinguen dos tipos de operaciones:

- a) Comandos, que alteran o pueden alterar los valores de los campos de una instancia (es decir, modifican su “estado”).
- b) Consultas, que no alteran el valor de los campos, y que se utilizan para preguntar por el estado de la instancia.

Las operaciones llevan parámetros en muchas ocasiones. Éstos se utilizan en la implementación de la operación para refinar su comportamiento. Del mismo modo, la ejecución de una operación puede devolver algún resultado, que será de un tipo de dato: las consultas, por ejemplo, devuelven un resultado.

Ejemplo 1. Todas las personas son objetos o instancias de una clase que podríamos llamar Persona. La clase Persona debe definir la estructura y comportamiento de sus objetos. En cuanto a estructura, podemos describir una Persona haciendo una enumeración de sus campos, que pueden ser la estatura (si se almacena en centímetros, lo definiríamos como un campo de tipo entero; si en metros, lo definiríamos como un número real), el peso (entero, por ejemplo), así como otros datos importantes tales como el nombre, los apellidos, el número de identificación fiscal (estos tres podrían ser cadenas de caracteres) o la edad (de tipo entero). Cada instancia de Persona tendrá sus propios valores de sus campos.

Un posible comando para esta clase se correspondería con la operación “cumplirAños”, que incrementaría en uno el valor del campo “edad”. Otro comando podría corresponderse con “expedirDNI” que, de alguna manera, asignaría un valor al campo que almacena el número de identificación fiscal, que podría haber tenido valor nulo hasta la ejecución de esta operación.

Una posible consulta podría ser “getEdad¹”, que devolvería el valor del campo “edad” de la instancia sobre la que se ejecuta la operación. Otra podría ser “getNombreCompleto”, que podría devolver una cadena de caracteres con los valores concatenados de los campos “nombre” y “apellidos”.

Los valores de los campos de una clase pueden tener valores libres, valores predeterminados, restricciones sobre sus valores o ser calculables en función de otros campos.

Ejemplo 2. Si utilizamos la clase Persona descrita en el Ejemplo 1 para almacenar información de los empleados de una empresa, podríamos imponer la restricción de que el campo “Edad” debe ser mayor o igual a la edad mínima que requiera la legislación para trabajar.

Si, en lugar de la edad, tuviéramos un campo para almacenar la fecha de nacimiento, podríamos describir el campo edad como un valor calculado, dependiente de la fecha actual y del valor de la fecha de nacimiento de cada persona. En este caso, en lugar de crear un campo específico para almacenar la edad, podríamos haber creado una operación de consulta “getEdad” que devolviera el resultado de esa operación. Esta es la solución habitual para los campos calculados; no obstante, y por razones de rendimiento, en ocasiones resulta interesante almacenar valores calculables en campos específicos.

Igualmente, existen campos y operaciones “estáticos”: un campo estático es un campo cuyo valor es compartido por todas las instancias de una clase; una operación estática no requiere de la existencia de una instancia para ser ejecutada. Por lo general, las operaciones estáticas actúan sobre campos estáticos.

¹ Castellanizando el nombre de la operación, podríamos haberla llamado *obtenerEdad*; no obstante, se encuentra bastante extendido el uso del prefijo *get* para nombrar a las operaciones de tipo consulta. Del mismo modo, suele utilizarse el prefijo *set* para nombrar a operaciones cuya principal función es la asignación de valores.

Ejemplo 3. Siguiendo con el Ejemplo 2, en el que utilizamos la clase Persona para almacenar la información de todos los empleados de una empresa, podríamos crear en la clase Persona un campo estático llamado Empresa, de tipo cadena de caracteres, en el que guardaríamos el nombre de la empresa. El campo sería estático porque todos los empleados pertenecen a la misma empresa.

Una posible operación estática podría ser “cambiarNombreDeEmpresa”, que cambiaría el valor del campo estático “Empresa”.

Las operaciones estáticas se utilizan en muchas ocasiones para crear bibliotecas de operaciones: por ejemplo, podríamos disponer de una clase Trigonometría que contuviera las operaciones trigonométricas definidas como estáticas. Para calcular, por ejemplo, el coseno de Π , no sería preciso crear una instancia de un número real y, sobre ésta, ejecutar la operación “coseno”: bastaría con llamar a la función de biblioteca “coseno” pasándole como parámetro el valor de P .

Mediante el término “miembro” hacemos referencia a todos los campos y operaciones de una clase. Por otro lado, el término “método” es sinónimo en muchas ocasiones de “operación”, aunque en algunos casos se limita su aplicación al de “operación ya implementada”.

2. Herencia y polimorfismo

Existen instancias que tienen parte de su estructura o de su comportamiento común, pero que sin embargo difieren en otra porción de su estructura o comportamiento. En estas ocasiones puede resultar conveniente utilizar herencia para describir, por un lado, la estructura o el comportamiento comunes (lo cual se realiza en la denominada “superclase”) y, por otro, la estructura y comportamiento particulares de subgrupos de instancias (que se realiza en las “subclases”). Las subclases pueden entenderse como tipos especiales de la superclase: de hecho, en muchas ocasiones se utiliza el término “especialización” como sinónimo de subclase. Del mismo modo, utilizaremos en ocasiones “generalización” para referirnos a la superclase.

Ejemplo 4. Volviendo al Ejemplo 1, la descripción que hemos dado de la clase Persona nos sirve para describir tipos especiales de persona, como estudiantes, empleados o jubilados: todos tendrán nombre, apellidos, edad, estatura, peso y, probablemente, número de seguridad social. sin embargo, cada uno de ellos tendrá una serie de particularidades que los hacen distintos unos de otros:

De un estudiante almacenaremos quizá el centro de estudios en el que se encuentra matriculado, las calificaciones que ha obtenido, etc. Igualmente, incluiremos en la definición de la clase Estudiante operaciones para matricularse en una asignatura, aprobarla, suspenderla, etc.

De un empleado, por ejemplo, convendrá almacenar su salario, su fecha de contratación, etc., y describir operaciones como “contratar”, “subirSueldo” o “jubilarse”.

De un jubilado nos interesará conocer la fecha en que se jubiló, datos relativos a su pensión, etc., así como describir aquellas operaciones que puedan resultar de interés para gestionar las instancias de esta clase.

La herencia, en este caso, pasaría por la descripción de la superclase Persona y de las subclases Estudiante, Empleado y Jubilado. En principio, las subclases heredan todo lo que se encuentre definido en la superclase. Cada subclase, además, aporta algo particular a la definición que recibe de la superclase.

Existen las denominadas clases abstractas, que son un tipo especial de clases de las cuales no se pueden crear instancias directamente.

Ejemplo 5. Si en el sistema que venimos describiendo en este ejemplo se van a manipular instancias exclusivamente de Estudiante, Empleado y Jubilado, pero no personas como tales, podemos definir la clase Persona como abstracta. Al hacerlo, impediremos la creación de instancias de clase Persona, pero permitiremos la creación de instancias de sus subclases.

Las clases abstractas pueden contener operaciones abstractas, que son operaciones de las cuales se describe su cabecera, pero a las que no se les da implementación. Las operaciones abstractas se utilizan para describir el protocolo de uso de las subclases: todas las instancias de las subclases responderán a la operación abstracta definida en la superclase, pero cada una lo hará, probablemente, de una manera distinta. Esta idea está relacionada también con el concepto de “polimorfismo”, mediante el cual, una clase puede tener varias operaciones diferentes pero con el mismo nombre (o, desde otro punto de vista, varias implementaciones distintas de la misma operación).

Ejemplo 6. Podemos añadir a la superclase Persona una operación denominada, por ejemplo, “calcularPagoDeImpuestos”. Asumiendo que para calcular los impuestos que deben pagar los estudiantes, los empleados y los jubilados se utilicen algoritmos diferentes, podríamos declarar la operación como abstracta en Persona, debiendo redefinirla en las subclases.

La herencia es una de las grandes contribuciones que el paradigma orientado a objetos ha hecho al desarrollo de software. Gracias a ella se alcanzan niveles importantes de reutilización. Del mismo modo, la combinación de herencia y polimorfismo permiten ejecutar operaciones sobre instancias sin saber y sin importarnos cuál es exactamente el tipo de la instancia. De manera general, las subclases deben redefinir la operación abstracta con exactamente la misma cabecera que se encuentra descrita en la superclase.

Ejemplo 7. Supongamos que disponemos de un array que almacena instancias de estudiantes, empleados y jubilados. Además de ser objetos de las clases “Estudiante”, “Empleado” y “Jubilado”, son también instancias de la clase “Persona”. Si queremos calcular el total de impuestos que recaudaremos con la colección de personas almacenadas en el array, podemos recorrerlo y ejecutar, sobre cada una de las instancias contenidas, la operación “calcularPagoDeImpuestos” descrita en el Ejemplo 6.

Cada instancia ejecutará su versión de la operación, pero al programador no le interesa cómo esté implementada la operación en cada subclase: lo único que sabe es que todas las instancias responden a esa operación, a la cual llaman utilizando la cabecera definida en la superclase.

Como se observa, la operación abstracta describe, en cierto modo, parte del protocolo de utilización de las instancias: todas, sean del tipo que sean, responden adecuadamente a la operación abstracta.

3. Relaciones entre clases

Como se dijo al principio de este capítulo, un sistema orientado a objetos es una comunidad de objetos que, en ejecución, resuelve algún problema determinado. Podríamos ampliar la definición especificando que el problema se resuelve mediante colaboraciones mutuas entre los objetos y que, para que se den tales colaboraciones, es preciso que los objetos que colaboran se conozcan. Para que dos objetos se conozcan, es necesario que ese conocimiento se haya hecho explícito en la definición de la clase correspondiente.

Ejemplo 8. Como apuntábamos en el Ejemplo 4, de cada empleado nos interesa conocer su salario. Éste puede estar compuesto de un sueldo base, una serie de complementos, dietas, etc. Mensualmente, la empresa retribuirá al empleado una cantidad en concepto de impuestos y de cuotas para la seguridad social, la cual dependerá de los ingresos totales del empleado. Podría ser conveniente, por tanto, definir una clase Salario que contuviera diferentes campos, uno por cada tipo de percepción cobrada por el empleado.

Cada instancia de Empleado conocerá a una instancia de Salario, que se corresponderá con el sueldo mensual que le corresponde según su categoría, puesto desempeñado, antigüedad en la empresa, etc. Para ejecutar, por ejemplo, la operación “calcularPagoDeImpuestos” antes mencionada, es muy posible que la instancia de Empleado deba consultar a la instancia de Salario a la cual conoce.

Además de las relaciones de herencia, se distinguen normalmente cuatro tipos de relaciones entre clases:

- a) **Agregación:** se da cuando dentro de la instancia de la clase existe una instancia de otra clase. Por ejemplo, dentro de una instancia de clase Persona, existe una instancia de clase Cadena de caracteres que se corresponde con el nombre de la persona.
- b) **Composición:** se trata de una relación de agregación “fuerte”, en el sentido de que la instancia contenida nace después de la instancia continente y muere antes.

- c) Asociación: ocurre cuando una instancia conoce a otra instancia de la misma o de otra clase, pero ambas instancias pueden también vivir separadamente y sin conocerse.
- d) Dependencia: bajo esta denominación se encuentran las relaciones temporales entre instancias de clases, que se dan, por ejemplo, cuando en una operación de una clase se toma un parámetro de otra clase. La relación entre las dos clases se limita al tiempo que dura la ejecución de la operación.

Las asociaciones, agregaciones y composiciones pueden ser navegables en uno o en dos sentidos. En una relación binaria, y si la relación es navegable en dos sentidos, entonces las instancias de cualquiera de las dos clases pueden tener conocimiento de las instancias de la otra clase. Si es navegable en un solo sentido, sólo las instancias de la clase origen de la relación tendrán conocimiento de las instancias correspondientes a la clase destino.

4. Ocultamiento y visibilidad

Los objetos se comunican entre sí mediante “paso de mensajes”, que no es sino la invocación de operaciones definidas en la clase de un objeto desde otro objeto.

Para ello, el objeto receptor del mensaje ofrece al objeto emisor una serie de operaciones, que son las que éste puede ejecutar y que serían sus operaciones “visibles”. Probablemente, el objeto receptor dispondrá de otras operaciones que no ofrece al emisor, las cuales “oculta”. Del mismo modo, los objetos ocultan normalmente su estado (es decir, sus atributos), aunque pueden hacerlo accesible mediante un conjunto de operaciones.

El “ocultamiento”, por tanto, es la capacidad que tienen los objetos para ocultar su estado o su comportamiento. Normalmente, es deseable que los campos de un objeto sean manipulables únicamente por el propio objeto, de modo que se limite su acceso al resto de objetos del sistema mediante un conjunto de operaciones controladas, que se exponen públicamente al resto de objetos. Al limitar el acceso a los campos de un objeto mediante operaciones públicas, se garantiza que el objeto sólo cambiará de estado cuando se den las condiciones necesarias para modificarlo.

Ejemplo 9. Si la clase Persona no ocultara al resto de objetos el campo en el que almacena la edad, otro objeto podría acceder directamente al campo y darle, por ejemplo, un valor negativo. Si se restringe el acceso al campo mediante una operación “setEdad”, puede controlarse en esta operación que no se den valores inadecuados al campo que modifica.

Existen diferentes niveles de visibilidad:

- a) Visibilidad pública: el miembro está accesible desde cualquier objeto del sistema.

- b) Visibilidad privada: el miembro es accesible sólo desde las instancias de la clase en la que se encuentra definido.
- c) Visibilidad protegida: el miembro es accesible a las instancias de la clase en la que está definida y a las instancias de sus especializaciones.

Al conjunto de operaciones públicas de una clase se lo conoce también como “servicios”, pues puede entenderse que representan, en efecto, el conjunto de servicios ofrecidos al resto de clases del sistema.

5. Representación de los conceptos anteriores en UML

UML (“Unified Modeling Language” o Lenguaje Unificado de Modelado) es un lenguaje estandarizado por el Object Management Group (OMG) que se utiliza para diseñar, construir y documentar sistemas software mediante el paradigma orientado a objetos. UML incluye un conjunto muy amplio de diagramas que se utilizan para representar la estructura y el comportamiento de sistemas.

El diagrama de clases es uno de los diagramas más conocidos y utilizados de UML. Un diagrama de clases muestra la estructura de un sistema o subsistema desde el punto de vista de las clases que lo componen, así como de las relaciones que existen entre tales clases.

Una clase se representa mediante un rectángulo que tiene normalmente tres compartimentos: en el superior se escribe el nombre de la clase; en el central se escribe la lista de campos de la clase; en el inferior se escribe la lista de operaciones de la clase. Pueden añadirse compartimentos adicionales con otros propósitos; además, los tres compartimentos mencionados pueden incluir otras informaciones.

La Figura 1 representa en UML la clase *Persona*:

- a) Las instancias de esta clase tienen siete campos (apellidos, empresa, estatura...), de los tipos indicados en la figura. El campo *empresa* es estático, lo cual se representa subrayando el nombre y el tipo del campo. Todos los campos, además, son privados, lo cual se denota con el signo menos (-) que tienen delante.
- b) La clase tiene cuatro operaciones, una de ellas (*cambiarEmpresa*) estática. Las operaciones devuelven resultados de diferentes tipos de datos. Todas las operaciones mostradas son públicas, lo que se denota con el signo más (+) que tienen delante.

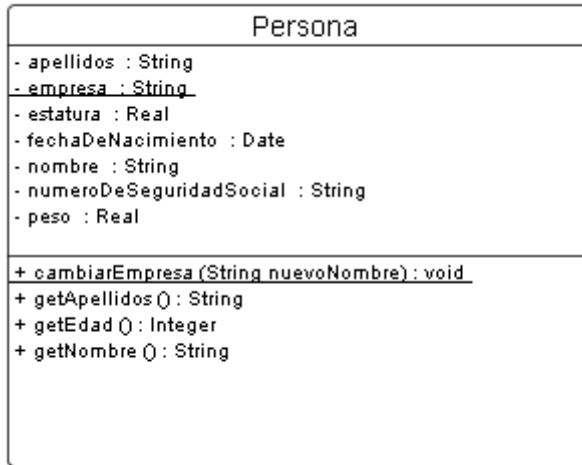


Figura 1. Representación de la clase *Persona* en UML (diagrama realizado con el entorno de desarrollo Oracle JDeveloper 10g)

5.1. Representación de campos

En UML, la notación que se utiliza para representar los campos en los diagramas de clases es la siguiente:

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

La visibilidad se corresponderá con algunos de los tipos de visibilidad definidos en UML, y se codifica utilizando + para visibilidad pública, # para visibilidad protegida y – para visibilidad privada.

La *multiplicidad* hace referencia al número de valores que puede almacenar el campo (por defecto, se asume que es uno). El *valorInicial* representa el valor que toma ese campo al construir instancias de la clase. Las *propiedades*, por último, pueden tomar los valores *changeable*, *addOnly* o *frozen*.

Recuérdese, además, que los campos estáticos deben subrayarse.

5.2. Representación de operaciones

La notación que se utiliza en los diagramas de clases de UML para representar operaciones es la siguiente:

visibilidad nombre (listaDeParámetros) : tipoDeRetorno {propiedades}

La *listaDeParámetros* representa la lista de argumentos que toma la operación. Cada argumento se representa con la notación:

clase nombre : tipo = valorPorDefecto

El valor de *clase* puede ser *in*, *out* o *inout*, según el parámetro sea, respectivamente, de entrada, de salida o de entrada y salida. Con *tipo* representamos el tipo de dato del parámetro. El *tipoDeRetorno* denota el tipo del resultado devuelto por la operación. Las *propiedades* se utilizan para repre-

sentar el tipo de operación de que se trate: si es una consulta, puede utilizarse la propiedad *{consulta}* o *{query}*. Las operaciones abstractas suelen representarse escribiéndolas en cursiva, aunque podrían dejarse en letra normal y ser adornadas con la propiedad *{abstracta}* o *{abstract}*.

Igual que ocurre con los campos, las operaciones estáticas deben subrayarse.

5.3. Herencia

La herencia se representa con una línea dirigida, orientada desde la subclase hasta la superclase. La línea debe terminar en una flecha triangular.

En la Figura 2 se muestra un diagrama de clases con herencia: existe una superclase *Persona* que es abstracta (obsérvese que el nombre de la clase se encuentra en cursiva) de la que heredan tres especializaciones. Los campos de la superclase son los mismos que en la Figura 1, sólo que, en esta ocasión, se han hecho protegidos para que sean accesibles a las subclases.

Cada subclase aporta campos y operaciones nuevas a los campos y operaciones que ya hereda.

La operación *calcularPagoDeImpuestos* que aparece en *Persona* es abstracta; sin embargo, la herramienta con la que hemos dibujado la figura no muestra esta característica de ninguna manera específica.

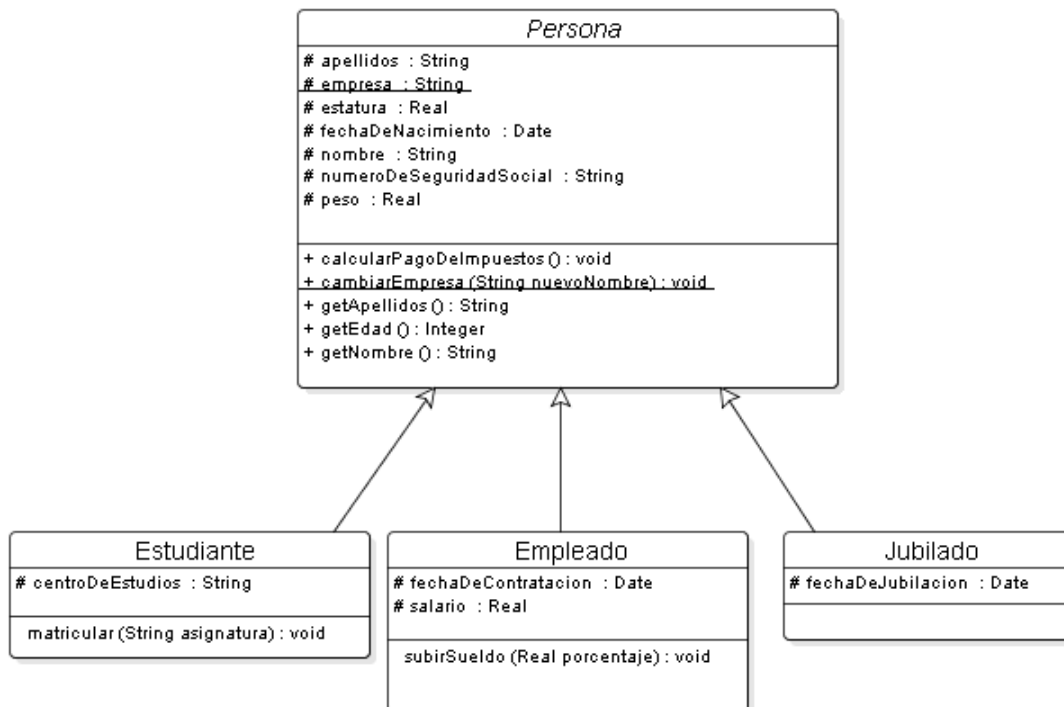


Figura 2. Relaciones de herencia de la superclase *Persona* con tres subclases

5.4. Relaciones entre clases

En la Figura 3 se representa el hecho de que cada instancia de *Empleado* conoce a una instancia de clase *Salario* que, a su vez, contiene una serie de complementos, cada uno de los cuales tiene un *nombre* y un *importe*.

La relación desde *Salario* hacia *Complemento* es una composición (se denota con el rombo relleno de color) navegable en un solo sentido (el indicado por la flecha: desde *Salario* hasta *Complemento*). Además, la multiplicidad de esta relación es 1 en *Salario* y cero a muchos (*) en *Complemento*, lo que significa que cada instancia de *Salario* puede tener muchos complementos. Posiblemente un mismo complemento esté también presente en varios salarios; sin embargo, al no ser navegable la relación desde *Complemento* hasta *Salario*, las instancias de *Complemento* no tendrán conocimiento de las instancias de *Salario*, por lo que resulta poco relevante y poco importante anotar la multiplicidad real en el lado de *Salario*.

La relación entre *Empleado* y *Salario* es una asociación (se trata de una línea sin rombos ni triángulos) navegable en los dos sentidos (la línea que las une no tiene puntas de flecha). Cada empleado tiene un único salario, aunque el mismo salario puede ser compartido por varios empleados. Puesto que la relación es navegable en los dos sentidos, en este caso sí que es interesante recoger su multiplicidad en ambos extremos.

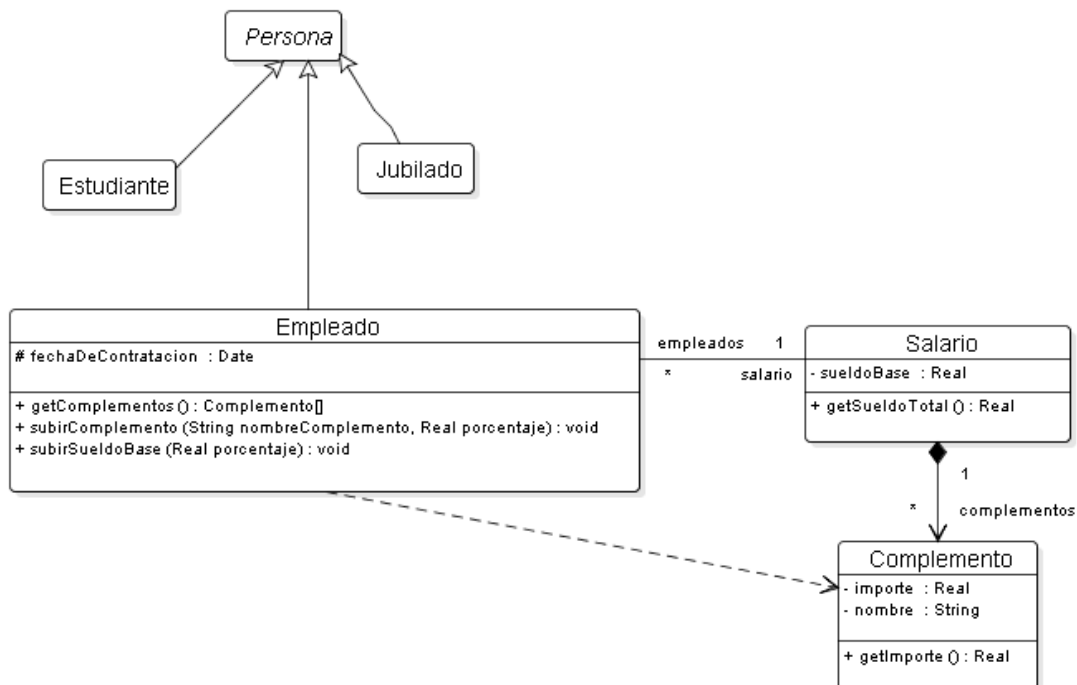


Figura 3. Diagrama con herencia, una asociación y una composición

La relación entre *Empleado* y *Complemento* es una dependencia. Las dependencias representan relaciones temporales entre instancias de clases.

Pueden darse, por ejemplo, porque en la implementación de una operación se cree una instancia de otra clase, porque una operación tome un parámetro de la otra clase o, como en la figura, porque una operación devuelva como resultado uno o más objetos de la clase de la que se depende: como se observa, la operación “getComplementos” de la clase *Empleado* devuelve un array de objetos de clase *Complemento*, por lo que el conocimiento de *Complemento* desde *Empleado* se limita al tiempo que dura la ejecución de esa operación. Como, además, no existe una relación más duradera entre ambas clases (asociación, agregación o composición), se representa la dependencia entre las dos clases con la línea punteada y terminada en flecha, que indica que la clase origen depende de la clase destino.

Los textos que etiquetan algunos extremos de algunas relaciones son importantes, ya que identifican el nombre de los objetos relacionados. Por ejemplo, el término *salario* en el extremo derecho de la asociación entre *Empleado* y *Salario* indica que un empleado hará referencia a su respectiva instancia de *Salario* mediante el término *salario*. Del mismo modo, los trabajadores que cobran un mismo salario serán identificados desde *Salario* con el término *empleados*. En el caso de la relación entre *Salario* y *Complemento*, puesto que desde ésta no se va a navegar hacia aquella, es poco relevante asignar un nombre en el extremo en el que se encuentra el rombo. A estos textos se los llama “nombres de rol”, ya que representan el rol que las instancias representan en la definición de las clases que las conocen.

Otro aspecto reseñable de la figura anterior es el hecho de que se han ocultado ciertos detalles de algunas clases (*Persona*, *Estudiante* y *Jubilado*), ya que nos interesaba centrar la atención en una porción determinada del diagrama. Esta ocultación de elementos en un diagrama es completamente legal en UML.

La siguiente figura muestra algunas posibilidades de relaciones de conocimiento entre clases:

- a) Entre A y B hay una asociación de conocimiento mutuo, en la que cada instancia de A conoce a una instancia de B, y viceversa.
- b) Entre C y D hay también una asociación bidireccional, en la que cada instancia de D conoce a una o más de C, y donde cada instancia de C puede conocer a una o ninguna instancias de D.
- c) La relación entre E y F es una relación de agregación, en la que cada instancia de E “tiene dentro” una instancia de F, a la cual conoce; sin embargo, la instancia de F no conoce a la instancia de E en la cual está contenida.

- d) G puede contener muchas instancias de H, a las que además conoce con el nombre “h”; a su vez, las instancias de H conocen a la instancia de G dentro de la cual están contenidas.

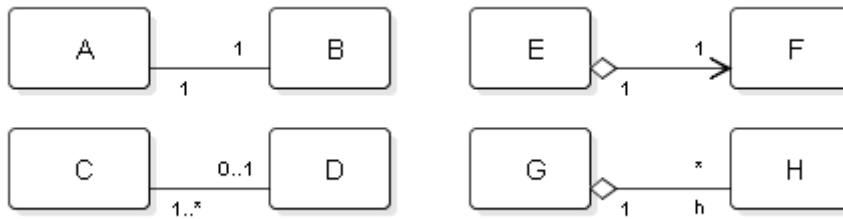


Figura 4. Algunas posibles relaciones entre clases

En general, se utilizará una agregación cuando el elemento que posee el rombo “tiene en su interior” elementos del otro extremo. En muchas ocasiones, la distinción entre una asociación y una agregación es puramente conceptual, ya que al traducir la relación a un lenguaje de programación se hace, en muchas ocasiones, exactamente de la misma manera.

6. Representación de los conceptos anteriores en lenguajes de programación

Todos los conceptos mencionados en las secciones anteriores pueden ser directamente representados utilizando lenguajes de programación orientados a objetos convencionales.

El fragmento de código siguiente muestra el resultado de generar el código Java correspondiente a la clase *Persona* con la herramienta con la que se ha dibujado la Figura 3. Existen muchas herramientas capaces de generar fragmentos de código importantes a partir de diagramas de clase. Obsérvense, en este caso, los modificadores *abstract* en la cabecera de la clase, *protected* en la declaración de los campos, *public* en la cabecera de las operaciones, *static* en el campo *empresa* y en la operación *cambiarEmpresa*. A pesar de que la herramienta no señalaba de ninguna manera que la operación *calcularPagoDeImpuestos* es abstracta, a la hora de generar el código sí que se añade el modificador *abstract* a la operación. Además, se observan algunas implementaciones por defecto de las consultas: *return null*, *return 0*, etc.


```

public abstract class Persona
{
    protected float peso;
    protected float estatura;
    protected String numeroDeSeguridadSocial;
    protected String nombre;
    protected static String empresa;
    protected String apellidos;
    protected Date fechaDeNacimiento;

    public String getNombre()
    {
        return null;
    }
    public String getApellidos()
    {
        return null;
    }
    public int getEdad()
    {
        return 0;
    }
    public static void cambiarEmpresa(String nuevoNombre)
    {
    }
    public abstract void calcularPagoDeImpuestos();
}

```

Cuadro 1. Código Java de la clase Persona de la Figura 3

El código de la clase *Empleado* se muestra en el cuadro siguiente: obsérvese que se ha añadido *extends Persona* a la cabecera de la clase para denotar que *Empleado* es una especialización de *Persona*. La asociación desde *Empleado* a *Salario* se ha traducido a un campo *salario* de tipo *Salario*, que es el nombre de rol de la clase *Salario* en la relación entre ambas clases en la Figura 3.

```

public class Empleado extends Persona
{
    protected Date fechaDeContratacion;
    protected Salario salario;

    public Complemento[] getComplementos()
    {
        return null;
    }

    public void subirSueldoBase(float porcentaje)
    {
    }

    public void subirComplemento(String nombreComplemento, float porcentaje)
    {
    }
}

```

Cuadro 2. Código Java de la clase Empleado de la Figura 3

Por último, se reproduce en el Cuadro 3 el código de la clase *Salario*. Obsérvense los comentarios que la herramienta generadora de código añade delante de los campos *complementos[]* y *empleados[]*: con ellos, se representa el hecho de que las dos relaciones que se están modelando con estos campos son, respectivamente, una asociación y una agregación. Muchas herramientas utilizan marcas de este tipo cuando generan código, y las utilizan para mantener adecuadamente la correspondencia entre código y diagramas.

```
public class Salario
{
    private float sueldoBase;

    /**
     * Comment here
     * @link aggregationByValue
     * @label UMLAssociation2
     * @associates <{Complemento}>
     */
    protected Complemento complementos[];

    /**
     * Comment here
     * @label Salarioempleado0
     * @associates <{Empleado}>
     */
    protected Empleado empleados[];

    public float getSueloTotal()
    {
        return 0;
    }
}
```

Cuadro 3. Código Java de la clase *Salario*

Existe un tipo especial de operaciones llamadas “constructores”, que se utilizan para crear instancias de clases. Normalmente, los constructores son operaciones cuyo nombre es igual que el de la clase, pero que no tienen tipo de retorno.

7. Conceptos sobre diseño arquitectónico

Es habitual que las aplicaciones se manejen mediante algún tipo de interfaz de usuario “amigable”, en el sentido de que está compuesto por una o más ventanas que facilitan su utilización por parte de los usuarios. Cada vez más, se construyen aplicaciones que pueden ser manejadas por tipos diferentes de ventanas: piénsese, por ejemplo, en las páginas web mediante las que los clientes de un banco pueden realizar ciertas operaciones sobre sus cuentas, tarjetas, etc. Realmente, estas páginas web actúan como meros transmisores de mensajes de los deseos del usuario hacia la lógica de la aplicación, que reside en un nivel algo más profundo. De hecho, es deseable que dicha lógica pueda ser ejercitada desde tipos diferentes de ventanas, de modo que los empleados del banco puedan ejecutar la misma lógica utilizando una interfaz de usuario que no sea la web. Esto se consigue dotando a las aplicaciones de un adecuado diseño arquitectónico, por el que la aplicación se estructura en una serie de capas: una de ellas puede contener la lógica de la aplicación, en la que residen todas las clases, con sus campos y operaciones, destinadas a resolver el problema de que se trate; puede haber otra capa de interfaz de usuario que actúe, como hemos dicho, como transmisor de mensajes desde el usuario hacia la lógica de la aplicación y viceversa; puede haber capas adicionales, destinadas por ejemplo a la gestión de la persistencia de los objetos en bases de datos, a las comunicaciones con otros sistemas, etc.

En situaciones como la descrita se habla de arquitectura multicapa, en la que, por lo general, se distinguen como mínimo tres capas:

- a) Capa de presentación, en la que se incluyen las ventanas que el usuario utilizará para hacer uso de las diferentes funcionalidades de la aplicación.
- b) Capa de dominio o de negocio, en la que reside la lógica de la aplicación y donde, en principio, se encuentra la mayor parte de la complejidad del sistema. Las principales funcionalidades de la aplicación se encontrarán implementadas en esta capa. Puesto que es la más compleja, a esta capa es a la que los ingenieros de software dedicarán su mayor interés.
- c) Capa de persistencia o de datos, que gestiona la persistencia de las instancias de la capa de dominio. Con *persistencia* nos referimos al hecho de que una instancia pueda ser guardada en algún mecanismo de almacenamiento, de manera que pueda ser utilizada posteriormente, incluso entre apagados y encendidos del sistema.

El diseño arquitectónico cuidadoso de la aplicación permitirá ofrecer distintas interfaces de usuario de una misma aplicación, quizá cada una de ellas destinada a tipos diferentes de usuario. La idea es reutilizar la capa de dominio, la más compleja, desarrollándola una sola vez, pero siendo capaces de ofrecer diferentes vistas de ella.

7.1. Políticas básicas de gestión de la persistencia

Las clases cuyas instancias se guardan en algún sistema de almacenamiento persistente se denominan “clases persistentes”. Por lo general, lo mínimo que se hará con las clases persistentes de la capa de dominio será: (1) crear instancias nuevas y guardarlas en disco; (2) leer instancias del disco; (3) modificar el estado de las instancias y actualizarlo en disco; y (4) eliminar instancias del disco. Este conjunto mínimo de operaciones se conoce como “operaciones CRUD”, de las siglas de *Create*, *Read*, *Update* y *Delete*:

- a) **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* de SQL, y podría estar implementada en un método *insert* de la clase persistente.
- b) **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 *Se-*

lect de SQL, y podría estar implementada en un constructor materializador, que añadiríamos a la clase persistente.

- c) **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, y podría implementarse en un método *update* de la clase persistente.
- d) **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* de SQL, y podríamos implementarla en un método *delete* de la clase persistente.

Existen múltiples formas de gestionar la persistencia. Dos de las posibles son las siguientes:

- Podemos dar implementación a las operaciones CRUD en las propias clases persistentes, que suelen ser las clases de dominio. Esto tiene la desventaja de que hacemos a estas clases (que son las que, como hemos dicho, tienen la mayor parte de la complejidad del sistema), dependientes del mecanismo de almacenamiento de las instancias. Supongamos que se ha optado por una base de datos relacional para almacenar en tablas las instancias de las clases persistentes: implementaremos las operaciones CRUD mediante instrucciones SQL que, como es sabido, es un lenguaje estandarizado por ISO y que, teóricamente, debe ser común a todos los gestores relacionales del mercado. Ocurre, sin embargo, que ninguno de los fabricantes se adapta completamente al estándar SQL, y que además cada fabricante dota a sus gestores de bases de datos de características que los diferencian del resto: en una base de datos de Microsoft Access, por ejemplo, almacenaríamos los valores de tipo booleano en columna de tipo Sí/No y utilizaríamos las palabras *true* y *false* para dar valor; en SQL Server, del mismo fabricante, deberíamos utilizar el tipo *bit* y los valores 1 y 0; en Oracle deberíamos usar columnas de tipo *Numeric...* en definitiva, si embebemos el código SQL en el cuerpo de los métodos CRUD, haremos a la capa de dominio excesivamente dependiente del sistema de almacenamiento.
- Podemos dar implementación a las operaciones CRUD en clases asociadas que se encarguen únicamente de gestionar la persistencia de las instancias de las clases de dominio. De este modo, las clases de dominio se olvidan de las responsabilidades de persistencia y se centran en implementar los meca-

nismos de resolución del problema, que es para lo que han sido diseñadas. Las clases en las que se ha incorporado la responsabilidad de gestionar la persistencia se dedican sólo a eso, de manera que un cambio en el gestor de base de datos afectaría únicamente a un conjunto reducido de clases.

Además, los dos mecanismos anteriores pueden implementarse de muchas formas; asumiendo la utilización de una base de datos relacional, se puede: (1) construir completamente las sentencias SQL en las operaciones CRUD y mandarlas a ejecutar; (2) utilizar sentencias preparadas y pasar los valores por parámetros; (3) llamar a procedimientos almacenados, que residen en el gestor de base de datos.

En la Figura 5 se han añadido las operaciones CRUD básicas a la clase Empleado: además de un constructor sin parámetros, se le ha añadido un constructor materializador, que toma como parámetro el valor del campo que identifica de manera única a los empleados; se han añadido también las operaciones *insert*, *delete* y *update* para, respectivamente, insertar, actualizar y borrar instancias de la base de datos.

En la implementación del constructor materializador, se buscará en la base de datos el empleado cuyo número de seguridad social coincida con el valor del parámetro pasado: si se encuentra, se asignará a los campos de la instancia los valores de las columnas correspondientes; en caso negativo, puede notificarse lanzando una excepción.

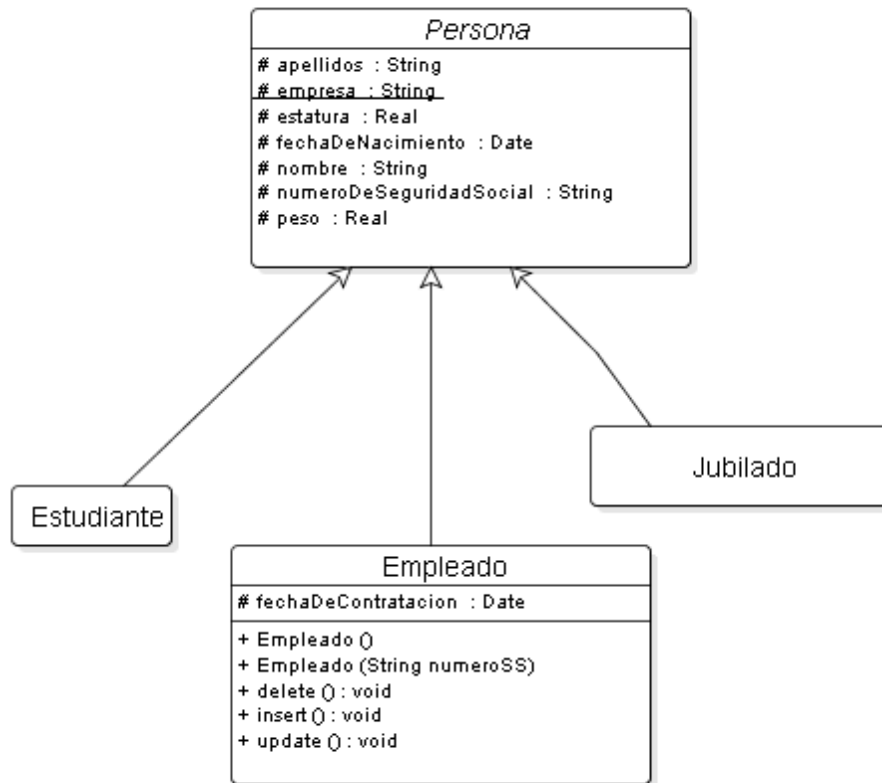


Figura 5. Adición de operaciones CRUD a la clase *Empleado*

En el Cuadro 4 se muestra una de las posibilidades de implementación del constructor materializador de *Empleado* en lenguaje Java, que es conforme al diagrama de clases mostrado en la figura anterior:

- 1) En primer lugar, se construye una cadena de caracteres en la que colocamos la instrucción SQL que queremos lanzar a la base de datos.
- 2) Creamos *bd*, una instancia de *Connection*, que es una interfaz² del paquete *java.sql* que nos permitirá el acceso a la base de datos. La implementación de esta interfaz es dada por cada fabricante de gestores de bases de datos. Al principio la iniciamos a *null*, pendientes de conectarla a la base de datos que resulte de nuestro interés.
- 3) En el bloque *try* colocamos aquellas instrucciones que pueden lanzar una excepción: en general, todas aquellas que acceden a la base de datos pueden lanzarla (por ejemplo, porque no haya conexión con la

² El significado del término “interfaz” se discute en la sección 7.2, página 40.

base de datos, porque no exista la tabla Empleados, porque nos hayamos equivocado al escribir el nombre de alguna columna, porque el usuario que se encuentra conectado no tenga permiso de lectura en la tabla, etc.).

- 4) La primera instrucción del *try* se conecta a una base de datos llamada *BDPersonas*.
- 5) A continuación, creamos un *ResultSet* (una interfaz definida en el paquete *java.sql* que sirve para representar conjuntos de registros leídos de la base de datos) con el que podremos manipular los datos que vayamos leyendo. *ResultSet* tiene la operación *next()*, que devuelve *true* si quedan datos por leer o, por el contrario, hemos llegado al final del conjunto de registros.
- 6) Dentro del *if*, a cuya rama *true* pasamos si hay registros para leer, vamos leyendo datos del conjunto de registros leído y los vamos asignando a los campos correspondientes de la clase Empleado. *ResultSet* dispone de diferentes operaciones *get*, que permiten leer datos de tipos diversos.
- 7) Si la llamada a *next* nos hubiera devuelto *false*, significaría que no hay registros en la tabla Empleados que cumplieran la condición de la instrucción SQL, por lo que lanzaríamos explícitamente una excepción con el mensaje “No se encuentra el registro”.
- 8) Si se lanza la excepción indicada en el punto anterior, o falla cualquiera de las instrucciones anteriores que se encuentran incluidas en el bloque *try*, el control del programa pasaría al bloque *finally*, que cerraría la conexión a la base de datos (de acuerdo con la documentación de Java, no se produce error al cerrar una conexión cerrada, por lo que el manejo de la excepción es correcto aunque fallara la primera instrucción del *try*). Tras ejecutar el *finally*, el programa captura la excepción en bloque *catch* y la relanza (merced a que dispone en su cabecera de la cláusula *throws Exception*) hacia el punto del programa que haya llamado al constructor que estamos describiendo.

- 9) El bloque *finally* también se ejecuta aunque no haya habido problemas en la ejecución de este método: de hecho, se pone en un bloque *finally* todo aquello que debe ejecutarse tras el *try*, haya o no fallo.

```
public Empleado(String numeroSS) throws Exception
{
    String SQL="Select * from Empleados where NumeroDeSeguridadSocial='" +
                                                    numeroSS + "'";
    Connection bd=null;
    try {
        bd=DriverManager.getConnection("BDPersonas");
        ResultSet r=bd.createStatement().executeQuery(SQL);
        if (r.next())
        {
            apellidos=r.getString("Apellidos");
            empresa=r.getString("Empresa");
            estatura=r.getFloat("Estatura");
            fechaDeNacimiento=r.getDate("FechaDeNacimiento");
            nombre=r.getString("Nombre");
            numeroDeSeguridadSocial=r.getString("NumeroDeSeguridadSocial");
            peso=r.getFloat("Peso");
            fechaDeContratacion=r.getDate("FechaDeContratacion");
        } else throw new Exception("No se encuentra el registro");
    }
    catch (Exception ex)
    {
        throw new Exception("Error al leer el registro " + ex.getMessage());
    }
    finally
    {
        bd.close();
    }
}
```

Cuadro 4. Una implementación del constructor *Empleado(String)*

En la Figura 6, se delegan las responsabilidades de persistencia a una clase auxiliar³. En esta implementación, las operaciones en *PersistEmpleado* son todas estáticas, de manera que cuando un empleado desea insertarse, actualizarse, etc., llama al método correspondiente de la clase auxiliar que le corresponde. Así, el código de la operación *insert* en *Empleado* podría ser el que se muestra en el siguiente cuadro:

```
public void insert() throws Exception
{
    PersistEmpleado.insert(this);
}
```

Cuadro 5. Implementación de *insert* en *Empleado*, que delega la inserción a su clase asociada

En la Figura 6, hay dos dependencias: una desde *Empleado* hacia *PersistEmpleado*, y otra en sentido contrario: la primera denota el hecho de que *Empleado* utiliza ocasionalmente los servicios ofrecidos por *PersistEmpleado*; la segunda representa que en las operaciones de *PersistEmpleado* se toman instancias de *Empleado* como parámetro.

³ La creación de una clase auxiliar, a la que la clase principal delega las operaciones que no son de negocio, es la aplicación del patrón Fabricación Pura.

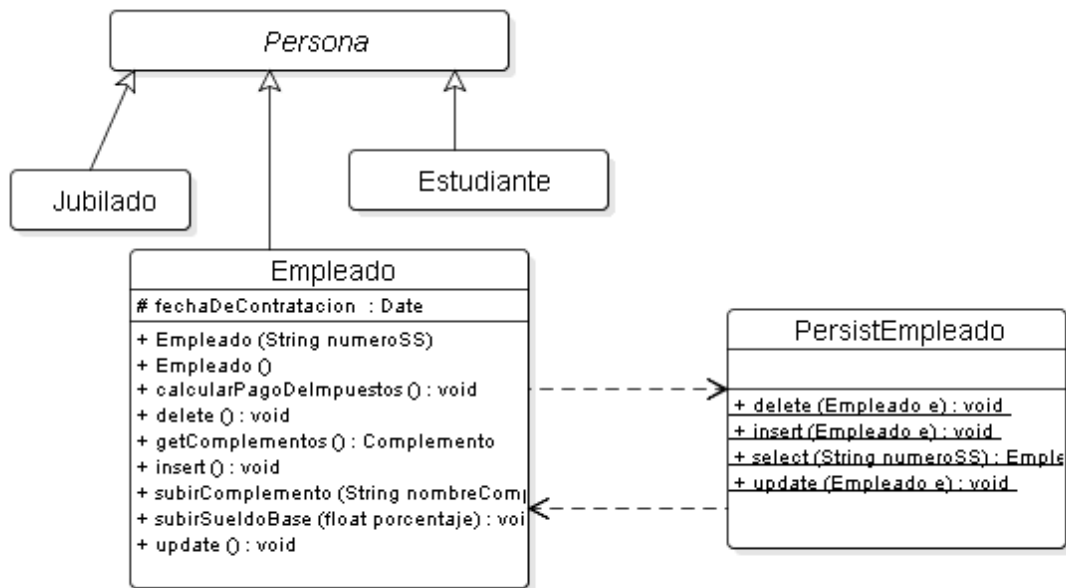


Figura 6. Delegación de las responsabilidades de persistencia a una clase asociada

Un diseño alternativo al de la figura anterior pasa por crear una asociación entre *Empleado* y *PersistEmpleado*, de manera que toda instancia de *Empleado* conozca, en todo momento, una instancia de *PersistEmpleado* a la que delegar las operaciones de persistencia. En este caso, las operaciones de la clase auxiliar pueden dejar de ser estáticas, ya que se utilizarán a través de la instancia *mGestorDePersistencia* (Figura 7):

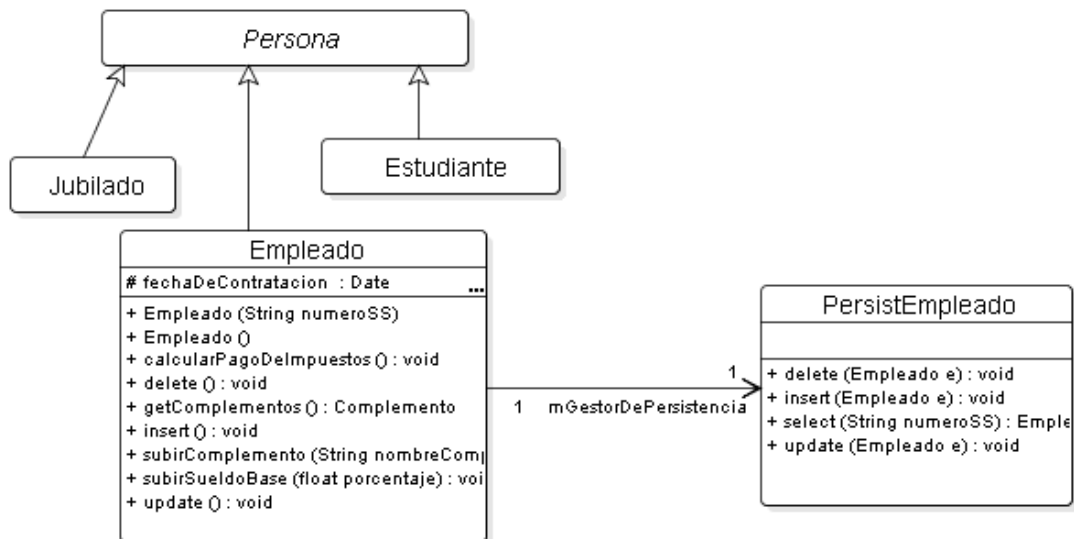


Figura 7. Diseño alternativo al de la Figura 6

```

public void insert() throws Exception
{
    this.mGestorDePersistencia.insert(this);
}
  
```

Cuadro 6. Método *insert* en *Empleado* para la Figura 7

Como comentábamos, existen muchos diseños alternativos, y ofrecemos a continuación otro: en el de la Figura 8, la clase auxiliar conoce a la instancia de dominio cuya persistencia manipula; por eso, sus operaciones no toman como parámetro ni devuelven instancias de *Empleado*.

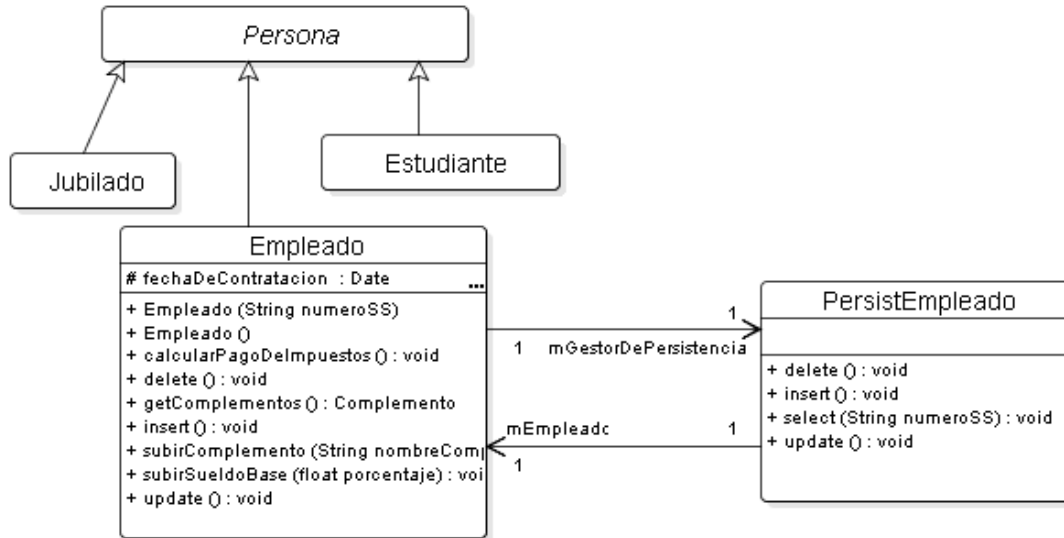


Figura 8. Otro diseño alternativo

Con el diseño de la figura anterior, las operaciones *insert* en *Empleado* y en *PersistEmpleado* podrían tener, en Java, las implementaciones mostradas en el Cuadro 7: obsérvese que, en *Empleado*, se accede a la operación de la clase auxiliar mediante *mGestorDePersistencia* (véase el rol en el diagrama de clases de la Figura 8), y que en *PersistEmpleado* se accede a la instancia de clase persistente mediante su campo *mEmpleadoc* (véase también el rol en la figura anterior).

En la implementación del segundo *insert* hemos introducido otra novedad, ya que se usa un objeto de clase *PreparedStatement* para ejecutar la operación sobre la base de datos: en la instrucción SQL que queremos ejecutar, en lugar de concatenar manualmente los valores de los parámetros, escribimos interrogaciones, una por parámetro; a continuación, en el bloque *try*, asignamos mediante las instrucciones *setString*, *setFloat*, *setDate*... los valores correspondientes a cada uno de los ocho parámetros de que consta la instrucción.

```

public void insert() throws Exception {
    this.mGestorDePersistencia.insert();
}

public void insert() throws SQLException {
    String SQL="Insert into Empleados (Apellidos, Empresa, Estatura, " +
        "FechaDeNacimiento, Nombre, NumeroDeSeguridadSocial, Peso, FechaDeContratacion) " +
        "values (?, ?, ?, ?, ?, ?, ?, ?)";
    Connection bd=null;
    try {
        bd=DriverManager.getConnection("BDPersonas");
        PreparedStatement p=bd.prepareStatement(SQL);
        p.setString(1, mEmpleado.getApellidos());
        p.setString(2, mEmpleado.getEmpresa());
        p.setFloat(3, mEmpleado.getEstatura());
        p.setDate(4, new java.sql.Date(mEmpleado.getFechaDeNacimiento().getTime()));
        p.setString(5, mEmpleado.getNombre());
        p.setString(6, mEmpleado.getNumeroDeSeguridadSocial());
        p.setFloat(7, mEmpleado.getPeso());
        p.setDate(8, new java.sql.Date(mEmpleado.getFechaDeContratacion().getTime()));
        p.executeQuery();
    }
    catch (SQLException ex) {
        throw new SQLException("Error al insertar: " + ex.getMessage());
    }
    finally {
        bd.close();
    }
}

```

Cuadro 7. Implementaciones de *insert* en *Empleado* y en *PersistEmpleado*

En el problema que venimos describiendo, todas las clases de dominio son persistentes, ya que en la base de datos almacenamos instancias de *Empleado*, *Jubilado*, *Estudiante* y, aunque sea abstracta, también de *Persona*. Podríamos construir, por tanto, una clase abstracta *Persistente* de la que heredaran las clases persistentes. En ésta se encontrarían, sin implementación, las cabeceras de las operaciones *CRUD*, y podría haber otras operaciones que sí podríamos implementar.

De las operaciones mostradas en la clase *Persistente* de la Figura 9, sólo tienen implementación *abrirConexion* y *cerrar*, siendo abstractas las otras cuatro (si bien deberían aparecer en cursiva para denotar que son abstractas, la herramienta utilizada para dibujar el diagrama no tiene esta funcionalidad).

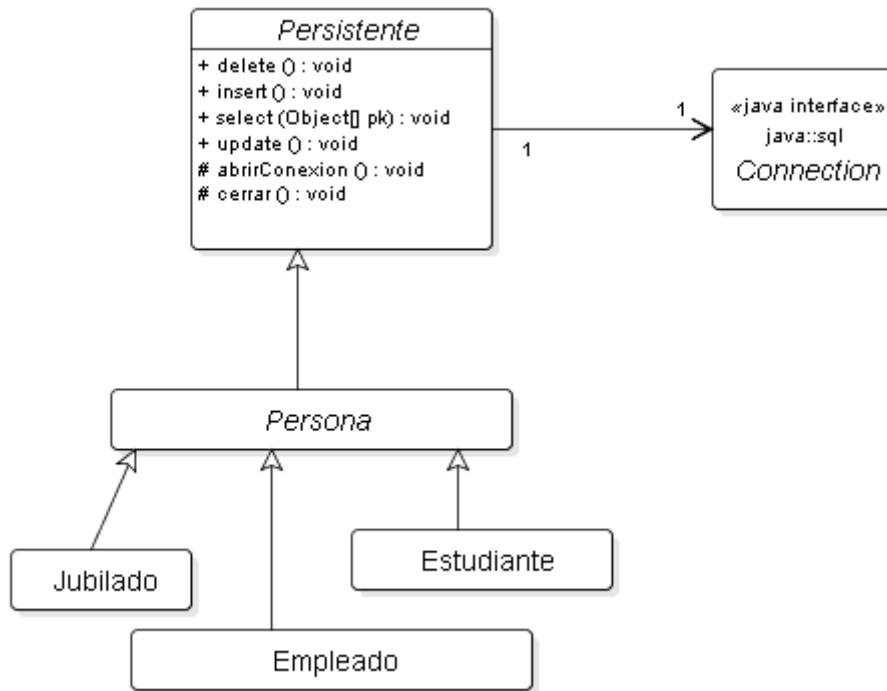


Figura 9. Creación de una superclase *Persistente*

7.2. Diseño de la capa de dominio

Como se ha dicho, en la capa de dominio residen las clases que realmente resuelven el problema para el que se ha construido el producto software, por lo que debe prestarse al diseño de esta capa la máxima atención posible. En general, los dos principios fundamentales que deben regir al ingeniero de software en el diseño de cualquier sistema son el mantenimiento del “bajo acoplamiento” y la “alta cohesión”:

- Por *acoplamiento* entendemos el grado en que un elemento está relacionado con otros elementos: aplicado a clases, una clase está acoplada a todas aquellas otras clases de las cuales depende. Cuanto mayor sea el acoplamiento de una clase, mayor sensibilidad tendrá a cambios en el diseño del problema.
- La *cohesión* vendría a ser el grado de relación que tienen entre sí los elementos de un conjunto. En un sistema orientado a objetos, una clase es altamente cohesiva cuando las operaciones definidas en la clase tienen mucha relación entre sí, o cuando tienen mucha relación con el propósito de la clase.

Por lo general, los objetivos de alta cohesión y bajo acoplamiento son incompatibles, en el sentido de que incrementar la cohesión suele conllevar un aumento (indeseable) del acoplamiento, y viceversa. Retrocediendo a los ejemplos de las figuras anteriores, la introducción de las clases auxiliares a

las que delegábamos las operaciones de persistencia aumentaban la cohesión de las clases de dominio (porque dejaban de hacer cosas que realmente no les correspondían, como ocuparse de su persistencia), pero incrementaban el acoplamiento, ya que cada clase de dominio pasaba a depender de su correspondiente clase auxiliar, y la clase auxiliar de la de dominio. Si optábamos por asignar las responsabilidades de persistencia a las propias clases de dominio eliminando las clases auxiliares, disminuíamos el acoplamiento, pero también la cohesión. Por ello, a la hora de diseñar sistemas orientados a objeto es preciso buscar un nivel de compromiso entre acoplamiento y cohesión.

En el acoplamiento de una clase influyen todas las relaciones de conocimiento que tiene respecto de otras: asociaciones, agregaciones, composiciones, dependencias... pero también las relaciones de herencia, ya que un cambio en la superclase tendrá, posiblemente, influencia en sus especializaciones.

Una posible forma de disminuir el acoplamiento (o de conseguir un acoplamiento “menos malo”) es hacer diseños orientados a interfaces. En principio, baste saber que una interfaz es un subconjunto del conjunto de operaciones públicas ofrecido por una clase, pero en donde las operaciones carecen de implementación. En la Figura 9 se creó una asociación entre *Persistente* y la interfaz *java.sql.Connection*, que nos sirve como mecanismo de acceso a la base de datos, sin importarnos en principio cómo estén implementadas las operaciones de esa interfaz. De hecho, quien realmente da implementación a tales operaciones es el fabricante del gestor de bases de datos: lo que a nosotros nos interesa, como ingenieros de software, es conocer el conjunto de operaciones ofrecido por el fabricante del gestor para poder hacer uso de ellas. En nuestro ejemplo, *java.sql.Connection* es una interfaz estandarizada, que es implementada por todos los fabricantes de gestores de bases de datos: económicamente, a los fabricantes les resulta interesante implementarla, ya que así garantizan que los desarrolladores Java utilizarán sus bases de datos.

7.3. Diseño de la capa de presentación

La capa de presentación es un mero transmisor de mensajes entre el usuario y la capa de dominio, y viceversa. De manera general, habrá una ventana en la capa de presentación por cada clase en la capa de dominio, que constituye la “vista” de las instancias de tal clase. En estas ventanas se podrá escribir código que haga una validación de los datos que van a ser enviados a la de dominio.

En muchas ocasiones ocurren cambios de estado en las instancias de la capa de dominio que deben ser comunicados a las ventanas de la capa de

presentación que las están observando. Como es deseable que la capa de dominio tenga el menor acoplamiento posible respecto de otras capas (incluyendo la de presentación), se recurre a argucias de diversos tipos para lograr pasar mensajes de dominio a presentación, pero sin que se produzca un acoplamiento real. A continuación veremos algunos posibles diseños.

7.3.1 Actualización de la ventana por consulta del estado

Supongamos que los sueldos de los empleados se actualizan a través de una ventana como la mostrada en la siguiente figura:

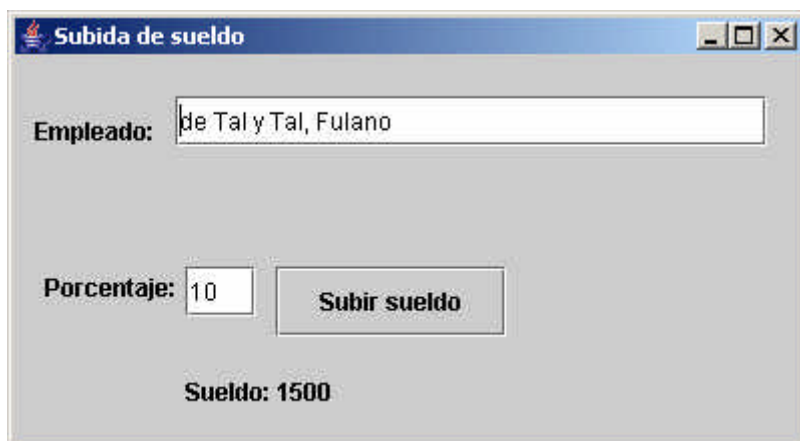


Figura 10. Una posible ventana para subir los sueldos

Cuando se pulsa el botón etiquetado “Subir sueldo”, se ejecuta el método *subirSueldoBase(float porcentaje)* sobre *mEmpleado* (la instancia de *Empleado* a la cual conoce esta ventana) a la que pasa como parámetro el porcentaje que se haya escrito en la caja de texto. La siguiente figura muestra un posible diagrama de clases coherente con esta descripción:

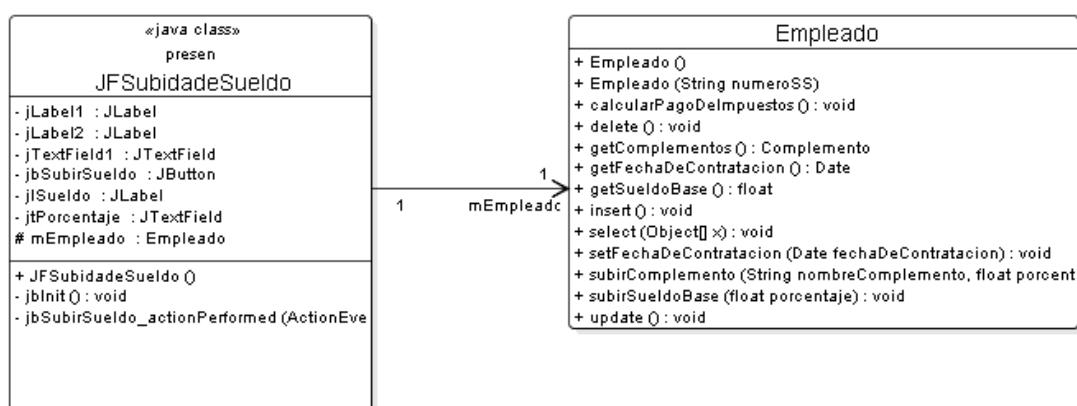


Figura 11. Diagrama de clases en el que la ventana conoce a un *Empleado*, al que envía mensajes

Cuando *mEmpleado* ejecuta el método, actualiza su campo *mSueldoBase*; la ventana, entonces, consulta el valor del sueldo llamando al método

getSueldoBase() de *mEmpleado*, y muestra el nuevo sueldo en el lugar adecuado de la ventana que mostrábamos en la Figura 10. El código de estas operaciones se muestra en el Cuadro 8:

- En el lado izquierdo aparece el código que se ejecutará cuando el usuario pulse el botón de subida de sueldo: dentro del bloque *try* se lee y transforma a *float* el porcentaje de subida que desea aplicarse; si la instrucción fallara (porque lo escrito contuviera por ejemplo letras y no pudiera transformarse a *float*) el control del programa saltaría al bloque *catch*, que mostraría un mensaje de error; si la transformación a *float* no fallara, se ejecutaría el método sobre *mEmpleado* (véase el rol en la Figura 11), que podría a su vez fallar. Si todo va bien, en la etiqueta *jlSueldo* de la ventana se coloca el nuevo valor del sueldo.
- El código del lado derecho es el correspondiente a la clase *Empleado*: recibe como parámetro el porcentaje que desea subirse el sueldo, recalcula el valor del campo y, por último, actualiza el estado de la instancia sobre la base de datos. La instrucción de actualización *update* puede fallar y lanzar una excepción. En este caso, en lugar de tratarla y procesarla con un bloque *try/catch*, la lanzamos “hacia atrás”, hacia el método que ha invocado a este método en el que nos encontramos. Esto lo estamos indicando con la cláusula *throws Exception* que hemos añadido a la cabecera del método.

<pre>private void jbSubirSueldo_actionPerformed(ActionEvent e) { try { float porcentaje=Float.parseFloat (this.jtPorcentaje.getText()); mEmpleado.subirSueldoBase(porcentaje); jlSueldo.setText("Sueldo: " + mEmpleado.getSueldoBase()); } catch (Exception ex) { JOptionPane.showMessageDialog(this, "Error al subir el sueldo: " + ex.getMessage()); } }</pre>	<pre>public void subirSueldoBase(float porcentaje) throws Exception { this.mSueldoBase*=(1+porcentaje/100); this.update(); }</pre>
--	--

Cuadro 8. Fragmentos de código en la ventana (izquierda) y en la clase de dominio (derecha)

La actualización de la ventana, en este caso, se ha producido porque la propia ventana ha estado pendiente de consultar el estado de la instancia de dominio a la cual conoce: es decir, ejecuta sobre ésta un servicio y, a continuación, le pregunta por su estado para actualizar su vista. Una desventaja importante de este método de actualización ocurre cuando el estado de la instancia de dominio puede cambiar sin intervención previa de la ventana que la está apuntando: piénsese en una aplicación multiusuario en la que dos

usuarios están trabajando simultáneamente con la misma instancia de dominio; si uno de ellos actualiza el sueldo del empleado, es deseable que este nuevo estado se refleje no sólo en la ventana del usuario que ha lanzado la actualización, sino en ambas. Para ello, es preciso dotar a la clase de dominio de algún tipo de relación con las clases de la capa de presentación.

7.3.2 Conexión directa de dominio con presentación

En el diagrama de clases de la Figura 12, la clase de dominio conoce a la clase de presentación. Puede conocer, además, a muchas instancias, de manera que la instancia de dominio puede comunicar sus cambios de estado a muchas instancias de la ventana, ya que puede haber más de una que la esté observando.

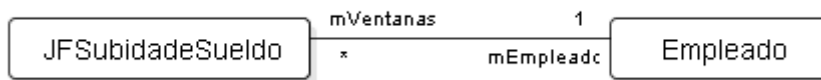


Figura 12. Diseño alternativo al de la Figura 11, en donde hay una mutua relación de conocimiento

Empleado poseerá una colección de instancias de tipo *JFSubidadeSueldo*, que representa las ventanas a las que notifica sus cambios de estado. Tendrá también alguna operación que permita a una ventana decirle a *Empleado* que desea ser notificada; tendrá otra operación para eliminarse de la lista de notificables; por último, dispondrá de un método para notificar. Estos cuatro fragmentos de código se muestran en el siguiente cuadro:

```

public class Empleado extends Persona
{
    ...
    protected Vector mVentanas=new Vector(); // Colección de ventanas a las que notificar
    ...

    public void suscribir(JFSubidadeSueldo v)    // Método para añadirse a la lista de
    {                                             // notificables
        mVentanas.add(v);
    }

    public void borrar(JFSubidadeSueldo v)      // Método para dejar de ser notificable
    {
        mVentanas.remove(v);
    }

    public void notificarSueldo()                // Método para actualizar las ventanas
    {
        for (int i=0; i<mVentanas.size(); i++)
        {
            JFSubidadeSueldo v=(JFSubidadeSueldo) mVentanas.get(i);
            v.muestraSueldo(this.getSueldoBase());
        }
    }
}
    
```

Cuadro 9. Fragmentos de código para conectar dominio con presentación

El diseño tiene sin embargo una desventaja importante, y es que acopla completamente *Empleado* (una clase de dominio, compleja) con la clase *JFSubidadeSueldo*, de manera que la clase de dominio necesitará de la existencia de objetos de clase *JFSubidadeSueldo* para funcionar. Si añadimos un nuevo tipo de ventana para manipular empleados, deberemos modificar el código en *Empleado*.

7.3.3 Desacoplamiento de dominio respecto de presentación

En el diseño de la sección anterior, se daba a *Empleado* la responsabilidad de hacer las notificaciones, directamente a instancias de *JFSubidadeSueldo*, lo cual: (1) Crea un acoplamiento indeseable y (2) disminuye la cohesión de *Empleado*, que ya no se ocupa sólo del dominio, sino también de temas de gestión de la presentación.

Una primera solución pasa por delegar a una clase auxiliar las responsabilidades correspondientes a la actualización de la capa de presentación. De este modo, cuando el *Empleado* cambie su estado, lo notifica a esta nueva clase (a la que llamaremos *Observador*) que, a su vez, lo notifica a todas las instancias de *JFSubidadeSueldo* que estén interesadas en el *Empleado*.

El diagrama de clases de esta solución se muestra en la siguiente figura:

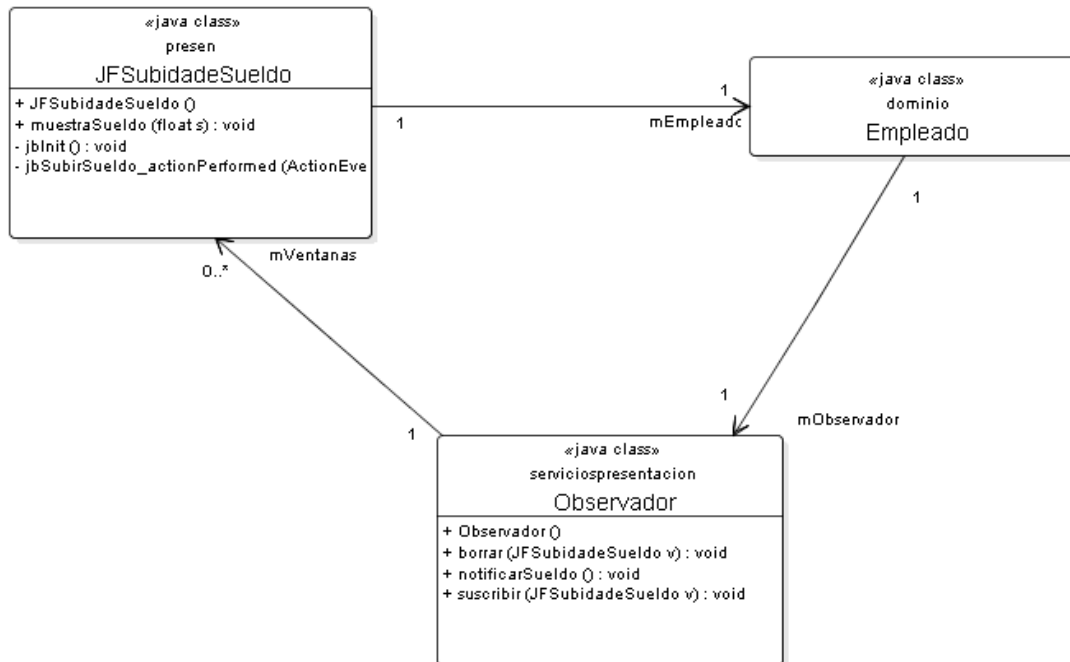


Figura 13. Adición de un observador para un diseño alternativo a los anteriores

En este caso, cuando el empleado cambia su estado, lo notifica al *Observador*, el cual es el responsable de notificar a todas las ventanas. De

hecho, el código que mostrábamos para la clase *Empleado* en el Cuadro 9 podríamos trasladarlo directamente a la clase *Observador*.

El acoplamiento, en esta solución, es “menos malo” que en el ejemplo anterior, ya que la clase de dominio deja de depender (al menos, de forma directa) de la presentación; el “acoplamiento malo” se ha delegado a una clase asociada, el *Observador*, que proporciona además más cohesión al conjunto. Sigue ocurriendo que, si añadimos un nuevo tipo de ventana para ver empleados, deberemos cambiar y recompilar el código de *Observador*.

7.3.4 Adición de interfaces para disminuir el acoplamiento

Si nos fijamos en el código del Cuadro 9, que ya hemos dicho que podría haberse cortado y pegado en la clase *Observador* de la Figura 13, lo único que interesa al *Observador* de las ventanas es que éstas respondan a la operación *muestraSueldo(float)*. Como se recordará de la breve discusión tenida en la página 41, una “interfaz” es un subconjunto del conjunto de operaciones públicas ofrecido por una clase, pero en donde las operaciones carecen de implementación. La interfaz, además, se “puede sacar fuera”, en el sentido de que puede haber múltiples clases que implementen la misma interfaz (recuérdese, en el caso de la interfaz *java.sql.Connection*, que cada fabricante de gestores de bases de datos la implementaba a su manera, sin importarnos cómo). En nuestro caso, podemos crear un interfaz en la que se encuentre la operación *muestraSueldo(float)*, de manera que ésta sea implementada por la ventana *JFSubidadeSueldo* o por cualquier otra clase interesada en mostrar el estado de los empleados.

La siguiente figura muestra esta posible solución: cuando el *Empleado* cambia de estado, lo notifica a su *mObservador*, que a su vez lo notifica a todos los objetos almacenados en *mVentanas*. En *mVentanas* no se almacenan instancias de *JFSubidadeSueldo*, sino cualquier tipo de objeto que implemente la operación *muestraSueldo(float)* o, más propiamente, que implemente la interfaz *IVentanaEmpleado*. Así, es posible crear otros tipos de ventanas para manipular empleados, que podrán ser notificadas del cambio de estado de la instancia a la cual apuntan, siempre y cuando cumplan el requisito de implementar la interfaz. De este modo, las interfaces nos permiten referirnos a una variedad de clases de manera unificada.

En UML, se utiliza la flecha discontinua terminada en triángulo para denotar las relaciones de implementación, que va orientada desde la clase hacia la interfaz.

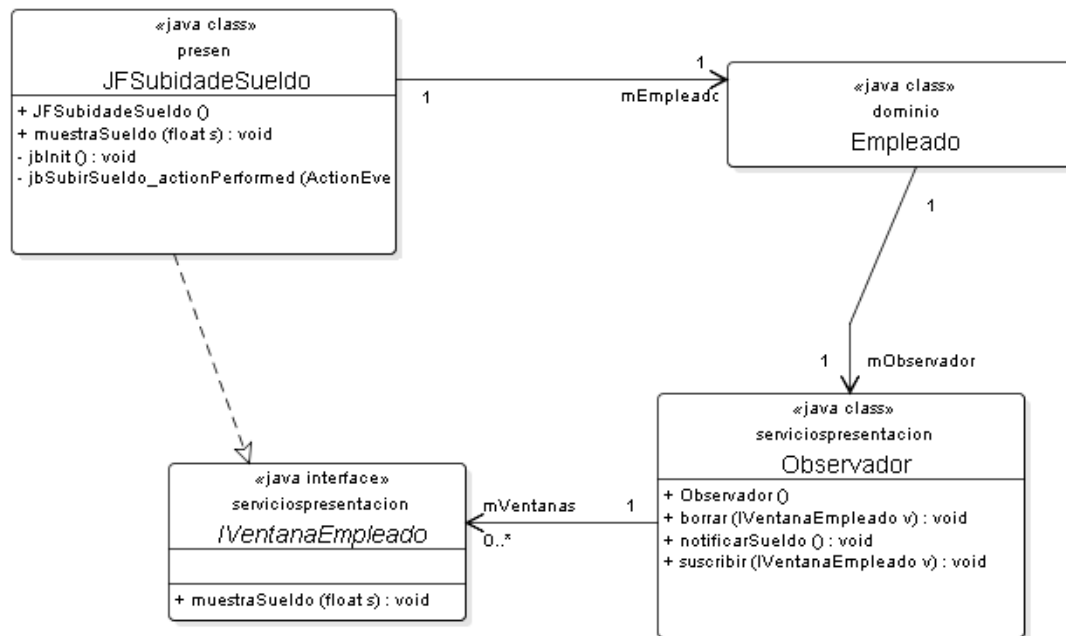


Figura 14. Adición de una interfaz para lograr un mayor desacoplamiento

7.4. Organización en subsistemas

Las diferentes capas de una aplicación pueden estructurarse en subsistemas (se denominan “paquetes” en Java y “espacios de nombres” en .NET). En la Figura 14, los elementos del diagrama aparecen etiquetados con los nombres de los subsistemas de los que forman parte: *presen* en la clase *JFSubidadeSueldo*; *dominio* en *Empleado*; *serviciospresentacion* en *Observador* e *IVentanaEmpleado*.

En UML, también a los subsistemas se los llama “paquetes”, y se representan con una figura como la siguiente:

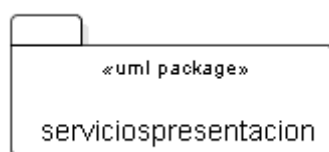


Figura 15. Representación en UML de un paquete

8. Testing y verificación

//TODO

9. Herramientas CASE para desarrollo de software orientado a objetos

Existen en el mercado multitud de herramientas para automatizar el proceso de desarrollo de software orientado a objetos. Entre las herramientas más conocidas se encuentran Rational Rose (de Rational, compañía adquirida en 2004 por IBM), Together (de Together Inc.), Poseidon (de Gentleware), Eclipse (que surgió del proyecto Eclipse, iniciado por IBM) o Microsoft Visio (de Microsoft). Además, algunos entornos de desarrollo incluyen plugins para mantener la coherencia entre el código que se está escribiendo y los diagramas que lo representan. Tal es el caso de entornos como JDeveloper (de Oracle), Eclipse o Visual Studio .NET (de Microsoft). La ya mencionada herramienta Eclipse es un caso algo especial, ya que se trata de un entorno de ingeniería de software asistida por computador completamente adaptable y ampliable mediante el desarrollo de plugins, para lo cual la propia herramienta incorpora un asistente.

Todas estas herramientas permiten dibujar todos o prácticamente todos los tipos de diagramas de UML, si bien cada una tiene sus particularidades y características especiales, que las dotan de una mayor o menor capacidad para automatizar ciertas etapas del proceso, que hacen que la herramienta se ajuste más o menos a la notación estandarizada de UML o que sea más o menos aplicable para el desarrollo de software utilizando ciertas metodologías.

10. Lecturas recomendadas

En la página web de Object Architects (<http://www.objectarchitects.de/>) aparecen multitud de artículos, ejemplos y referencias en los que se discute la correspondencia entre el mundo de objetos y el mundo relacional.

En el capítulo séptimo del libro “Análisis y diseño de aplicaciones informáticas de gestión”, de Piattini y otros, se presenta una excelente discusión sobre los conceptos de acoplamiento y cohesión.

El capítulo 19 del mismo libro se dedica a la tecnología CASE. Presenta los principales elementos de las herramientas CASE, tipos de herramientas, etc.

En diferentes capítulos del libro de Craig Larman (“UML y Patrones”) se presentan y explican conceptos importantes sobre la arquitectura multicapa.

Capítulo 2. METODOLOGÍAS 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 expresión y comunicación, y de metodologías de trabajo adecuadas.

Para lo primero, existen diferentes 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.

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 de Desarrollo (UDP). Recientemente han surgido las denominadas metodologías ágiles, a la cabeza de las cuales se encuentra la Programación Extrema (eXtreme Programming o, abreviadamente, XP), que proponen formas de comunicación y desarrollo ciertamente rompedores.

En este capítulo se describen con brevedad algunas características de varias metodologías de desarrollo de software orientado a objetos. El capítulo nos sirve, además, para introducir algunos elementos y conceptos de UML que no se han visto en el capítulo anterior.

1. OMT: Object Modelling Technique

OMT combina una metodología y un conjunto de modelos para representar y desarrollar software orientado a objetos. Los modelos que se utilizan en OMT para representar el software son los tres siguientes:

- Modelo de objetos: describe la estructura estática del sistema, mostrando las clases y objetos que lo forman y las relaciones entre ellos. Se representa utilizando diagramas de clases y diagramas de objetos.
- Modelo dinámico: describe el comportamiento del sistema, mostrando los eventos externos que afectan al sistema, las secuencias de operaciones, las interacciones entre objetos y los estados de los objetos. Se representa utilizando diagramas de estado.
- Modelo funcional. describe la forma en que el sistema procesa los datos. Se representa utilizando diagramas de flujos de datos.

Debido a la aparición y estandarización de UML como lenguaje de modelado, y a la existencia de otras metodologías de desarrollo, OMT se encuentra en claro desuso. No obstante, OMT ha aportado parte de su cuerpo de conocimientos a otras metodologías, resultando especialmente interesante su fase de análisis aplicada a la construcción del modelo de objetos. Los pasos que se siguen en esta etapa son los siguientes:

1. *Identificación de clases.* Se extraen todos los sustantivos de la descripción del problema, convirtiendo cada sustantivo en una clase candidata.
2. *Selección de clases.* Se seleccionan las clases importantes para el análisis, eliminando aquellas que no tengan sentido en el área de aplicación, las clases redundantes, las clases que serán realmente atributos, operaciones o roles, 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.

2. Desarrollo dirigido por las pruebas (Test-Driven Development)

En el TDD, los casos de prueba se escriben antes que el propio código, de manera que van guiando la codificación del sistema.

Supongamos que se desea implementar una clase que represente una cuenta corriente. Cuando se ingresa, se espera que el saldo de la cuenta sea el que tuviera antes más el importe ingresado. El lado izquierdo del Cuadro 10 muestra el código del posible caso de prueba para probar esta funcionalidad: obsérvese que se declara un objeto de clase *Cuenta*, que se crea y que sobre ella se ejecuta la operación *ingresar* pasándole 1000 euros como parámetro. A continuación, se escribe una aserción para comprobar que el saldo de la cuenta es de 1000 euros.

Del pequeño caso de prueba debemos escribir código que satisfaga las aserciones descritas en el caso. El lado derecho muestra el posible código obtenido que, de acuerdo al TDD, se escribe con posterioridad al caso.

<pre>public void testIngresar() { Cuenta c=new Cuenta(); c.ingresar(1000.0); assertEquals(c.getSaldo(),1000.0); }</pre>	<pre>public class Cuenta { protected double saldo; public Cuenta() { saldo=0; } public void ingresar(double importe) { saldo+=importe; } public double getSaldo() { return saldo; } }</pre>
---	---

Cuadro 10. Un caso de prueba (izquierda), escrito antes que el código que lo supera (derecha)

3. El Proceso Unificado de Desarrollo

El Proceso Unificado de Desarrollo (PUD) es una metodología de desarrollo de software que se autodenomina “dirigida por casos de uso, iterativa e incremental”.

Dirigido por casos de uso Un caso de uso representa un requisito funcional del sistema que se va a desarrollar. Como sabemos, un requisito funcional es una funcionalidad que se espera que sirva el sistema. El término “dirigido por casos de uso” significa que el software se construye de acuerdo a una cierta ordenación que se realiza con los requisitos funcionales deseados por el cliente.

Iterativo El hecho de que el PUD sea “iterativo” indica que 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, tras la iteración, se obtiene un sistema ejecutable probado e integrado.

Incremental Por lo general, en cada iteración se selecciona un pequeño conjunto de requisitos funcionales (es decir, de casos de uso), los cuales se diseñan, se implementan y se prueban. Esto supone un refinamiento progresivo del sistema, que va además aumentando de tamaño (de aquí el término “incremental”), aunque alguna iteración puede dedicarse a mejorar software desarrollado con anterioridad.

La Figura 16 resume gráficamente las ideas anteriores.

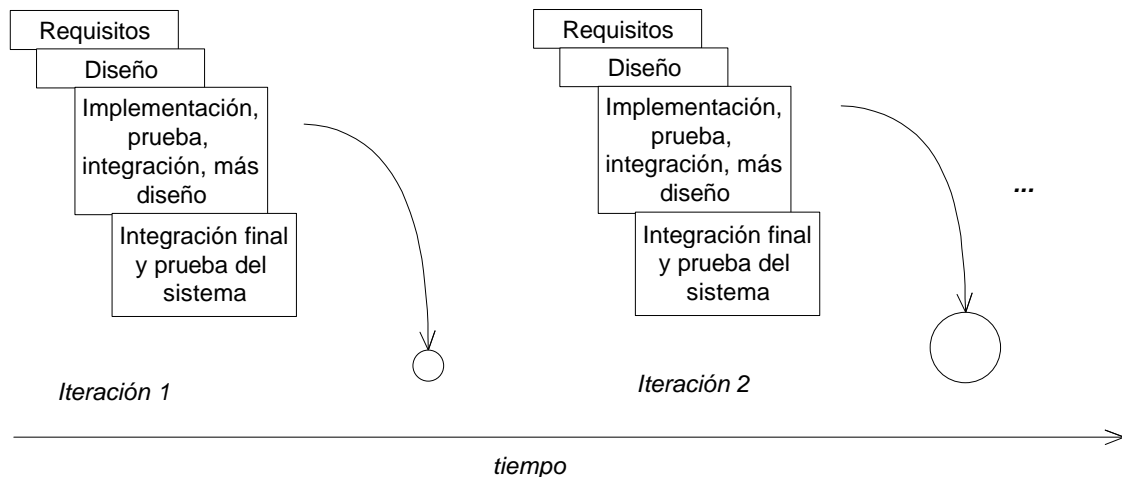


Figura 16 (adaptada de Larman, 2001). El PUD es iterativo e incremental (los círculos representan el tamaño del sistema)

El desarrollo del producto software guiado por 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. De hecho, el Proceso Unificado pone un énfasis muy especial en la gestión de los riesgos durante el desarrollo.

3.1. Fases del Proceso Unificado de Desarrollo

Ciclos y versiones La aplicación del Proceso Unificado al desarrollo de un proyecto supone la realización de una serie de *ciclos*, los cuales representan la vida del sistema. Cada ciclo concluye con una *versión* nueva del producto, susceptible de ser entregada al cliente. Cada versión incluye código fuente, manuales de usuario y otros posibles artefactos.

Fases

Un ciclo se descompone en las *fases* de Comienzo, Elaboración, Construcción y Transición, y cada fase consta de varias iteraciones. Una de las posibles vistas gráficas del proceso se muestra en la Figura 17.

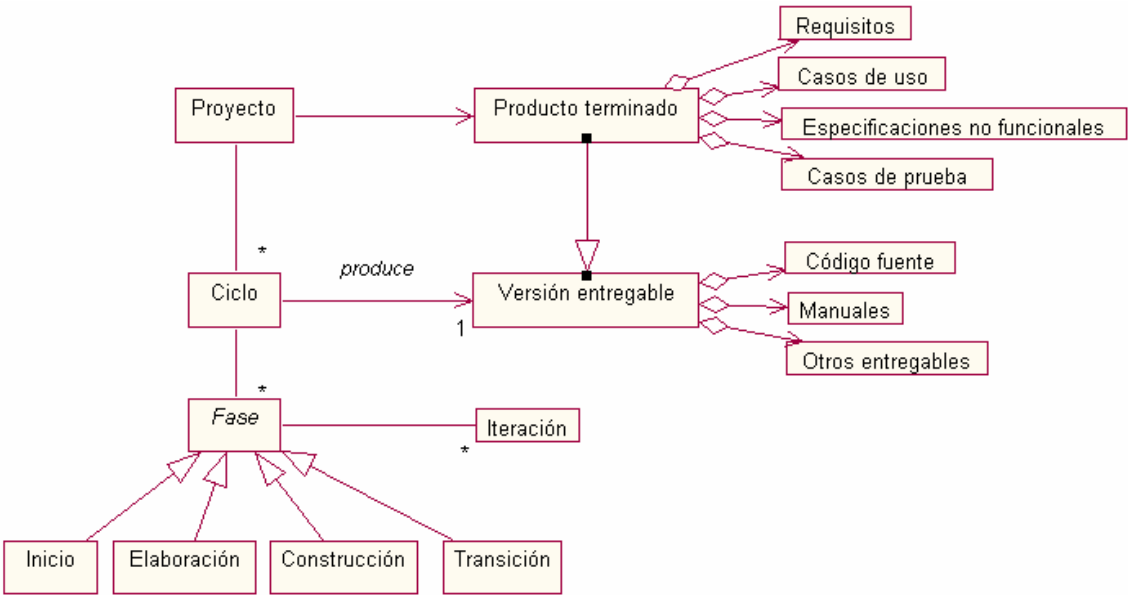


Figura 17. Estructura general del Proceso Unificado de Desarrollo

**Flujos de
trabajo en el
Proceso
Unificado**

Para desarrollar un proyecto software conforme al PUD es preciso ejecutar una serie de *flujos de trabajo* durante cada fase. Los flujos de trabajo fundamentales se muestran en la Figura 18.

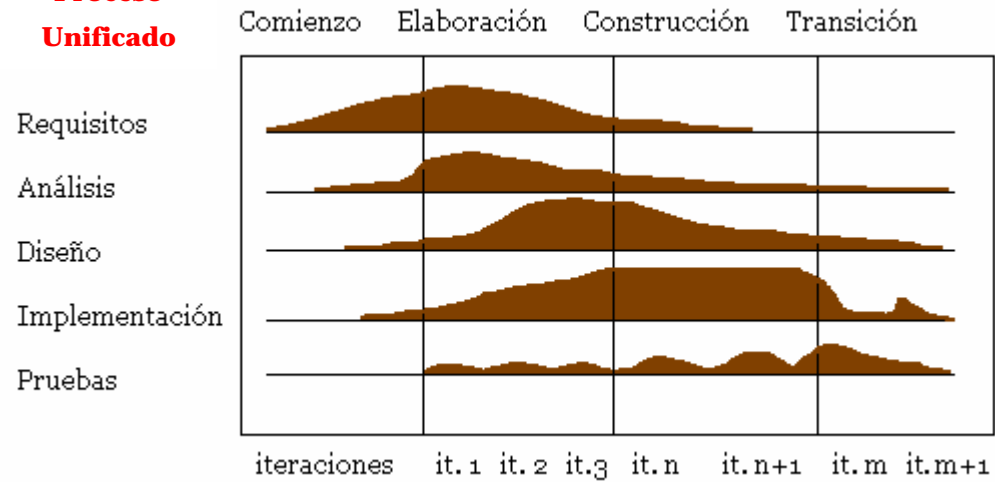


Figura 18. Flujos de trabajo del Proceso Unificado

Como se observa, prácticamente en todas las fases se realizan actividades de cada flujo (en las fases de Comienzo, Elaboración y Construcción, por ejemplo, se realizan normalmente actividades relacionadas con el flujo de *Requisitos*), si bien el esfuerzo dedicado en cada una va cambiando conforme avanza el proyecto.

3.1.1 Fase de comienzo

Vista funcional En esta fase se estudia la viabilidad del proyecto, se planifica su desarrollo y se delimita su alcance (es decir, se especifica qué se va a implementar y qué no). Para ello, se identifican los requisitos en una vista funcional a alto nivel (un diagrama de casos de uso que muestre el contexto del sistema) y se describen los más importantes (en torno al 10-20%).

Modelo de dominio Si el proyecto es viable, se construye un primer modelo de dominio del sistema, en el que se capturan los elementos más importantes, que se obtienen de forma parecida al modo en que, en OMT, se identificaban clases, operaciones, campos, etc.

Lista de riesgos El avance del proyecto estará amenazado por una serie de riesgos (desconocimiento de una cierta tecnología que será preciso utilizar, por ejemplo) que deben identificarse e incluirse en un documento.

Diseño arquitectónico Las guías maestras de diseño del sistema se plasmarán en un boceto del diseño arquitectónico del sistema.

Plan de proyecto Por último, el desarrollo del proyecto se planifica y se plasma en una primera versión del plan de proyecto. El estándar IEEE 1058 (Cuadro 11) puede ser de utilidad para confeccionar este documento.

Página de Título
Prefacio
Tabla de Contenidos, Lista de Figuras y Lista de Tablas
1. Introducción.
1.1. Visión General del proyecto.
1.2. Productos Finales.
1.3. Evolución del Plan de Proyecto.
1.4. Documentos de Referencia.
1.5. Definiciones y Acrónimos.
2. Organización del Proyecto.
2.1. Modelos de Procesos.
2.2. Estructura Organizativa.
2.3. Fronteras e interfaces organizativas.
2.4. Responsabilidades.
3. Procesos de Gestión.
3.1. Objetivos y prioridades de Gestión.
3.2. Suposiciones, dependencias y restricciones.
3.3. Gestión de Riesgos.
3.4. Mecanismos de supervisión y control.
3.5. Plan de Personal.
4. Proceso Técnico
4.1. Metodologías, Técnicas y Herramientas.
4.2. Documentación Software.
4.3. Funciones de Apoyo al proyecto.
5. Plan de Desarrollo.
5.1. Paquetes de Trabajo.
5.2. Dependencias.
5.3. Recursos.
5.4. Presupuesto y distribución de recursos.
5.5. Calendario.
Componentes adicionales
Índice
Apéndices.

Cuadro 11. Estructura del plan del proyecto según el estándar IEEE 1058

3.1.2 Fase de elaboración

En esta fase se analiza de forma detallada la práctica totalidad de los requisitos, lo cual llevará a terminar de establecer la base arquitectónica del producto software y a tener más base de conocimiento para concluir el plan del proyecto, que dirija especialmente a las dos próximas fases.

El análisis detallado y casi completo de los requisitos nos dará pie para la identificación completa de los elementos que compondrán el modelo de dominio del sistema, lo que a su vez permitirá establecer un diseño arquitectónico sólido y no desechable, por el que se guiarán los miembros del equipo durante el desarrollo.

Como se observa en la Figura 18, en esta fase se realizan actividades de todos los flujos, desde captura de requisitos y su análisis, hasta codificación y pruebas.

3.1.3 Fase de construcción

El propósito de esta fase es obtener un producto software en su versión operativa inicial, de forma que tenga la calidad adecuada para el objetivo para el que ha sido construido y cumpla con los requisitos. Al finalizar esta fase, el producto software estará tan solo a falta de ser sometido a las pruebas de aceptación y a su preparación para la instalación en la plataforma final. Probablemente se haya redactado una primera versión del manual del usuario.

Se redactará entonces el plan de proyecto para la fase de transición y se dispondrá ya de una versión ejecutable del sistema.

Trazabilidad del proceso

De manera ideal, debería ser posible realizar un seguimiento desde un artefacto correspondiente a cualquier nivel de abstracción hacia otro, por lo que uno de los entregables de esta fase es el conjunto de modelos UML debidamente actualizado.

3.1.4 Fase de transición

Esta fase tiene como objetivo principal la entrega e instalación del software en la organización cliente. Para ello, será quizá preciso crear procedimientos específicos de instalación y redactar por completo el manual de usuario (incluyendo la propia ayuda de la herramienta). También se realizarán las pruebas de aceptación y se llevará a cabo la revisión de los modelos elaborados durante el desarrollo.

3.1.5 Resumen de las fases

En la Tabla 1 se muestran algunas de las actividades principales de cada fase, así como el número típico de iteraciones que componen cada fase.

Fase	Iteraciones (nº medio)	Actividades principales	Entregables
Comienzo	1	Estudio de viabilidad Identificación y descripción general de requisitos Análisis detallado de los requisitos más importantes (10-20%)	Primera versión del modelo de dominio que describe el contexto del sistema Vista funcional a alto nivel Borrador del diseño arquitectónico Lista inicial de riesgos Primera versión del plan del proyecto
Elaboración	3-5	Diseño e implementación de la arquitectura base del sistema (arquitectura, además, no desechable) Análisis detallado de casi la totalidad de requisitos Planificación	Modelo de dominio completo Nueva versión de los modelos anteriores UML (casos de uso, análisis, etc.) Diseño arquitectónico no desechable Lista de riesgos actualizada Plan del proyecto para las dos próximas fases
Construcción	Depende del tamaño del proyecto	Análisis detallado del resto de requisitos Desarrollo y prueba del software	Plan del proyecto para la fase de Transición Versión ejecutable del sistema Todos los modelos UML Descripción de la arquitectura Esbozo del manual de usuario
Transición	1-2	Redacción de manuales de usuario Instalación	Versión ejecutable del sistema, incluyendo el sistema de instalación Documentación legal (contratos, licencias, etc.) Descripción de la arquitectura actualizada Manuales de usuario Otra documentación, incluyendo la <i>on-line</i>

Tabla 1. Descripción resumida de algunas características del Proceso Unificado

3.2. Algunos modelos importantes

Durante el desarrollo del producto se construyen muchos tipos de *artefactos*, 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. Se puede considerar la *vista funcional* del sistema.
2. El diagrama de casos de uso se *especifica* mediante un *modelo de análisis*.
3. El diagrama de casos de uso se *realiza* mediante un *modelo de diseño*.
4. El sistema se *despliega* o *distribuye* (es decir, 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 resaltar 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 con la notación vista en el Capítulo 1.
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 2. Descripción de algunos términos

3.2.1 Diagramas de casos de uso

Un diagrama de casos de uso representa la vista funcional del sistema que debemos desarrollar. Cada caso de uso representa un requisito del sistema⁴. El diagrama muestra las relaciones entre los actores, los casos de uso y, tal vez, las interfaces de sistemas externos con los que se comunica el sistema que estamos desarrollando.

Los actores representan todos aquellos elementos que se comunican con nuestro sistema pero que no forman parte de él. En esta categoría suelen caer tres tipos de elementos: personas (usuarios del sistema, que actúan sobre él desencadenando la ejecución de operaciones, o que reciben información de él), otros sistemas software o hardware y el tiempo (entendido como un reloj que, cada cierto tiempo, provoca la ejecución de alguna operación en nuestro sistema). En ocasiones, los otros sistemas se representan con la interfaz que ofrecen a nuestro sistema para que éste se comuniquen con él.

Supongamos que deseamos automatizar el proceso de préstamos de libros en una biblioteca. Las funcionalidades que podríamos requerir del sistema son el mantenimiento de socios (creación, actualización de datos, eliminación), de libros y la gestión de los préstamos (lo que incluiría el control del número de libros prestados, de los plazos de préstamos, de las posibles sanciones a los socios que se retrasen en exceso, etc.). Por hacer más completo el diagrama, supongamos que existe la posibilidad de buscar en otras bi-

⁴ Normalmente, se asimila un caso de uso con un requisito funcional. En este libro se utilizan indistintamente estos dos términos.

bibliotecas aquellos ejemplares que los socios soliciten y que no estén disponibles en la nuestra. Un posible diagrama de casos de uso para representar esta vista funcional sería el siguiente:



Figura 19. Diagrama de casos de uso para una biblioteca

Los casos de uso se representan con los óvalos, en cuyo interior se escribe el nombre de la funcionalidad correspondiente. En el diagrama hay dos actores (el bibliotecario y la biblioteca externa), que se representan con el mismo símbolo (el monigote), independientemente de su naturaleza. Las líneas que unen los actores con los casos de uso representan relaciones de comunicación.

En ocasiones, los casos de uso del sistema que se está desarrollando se enmarcan en un cuadrado, que representa los límites del sistema, como en la siguiente figura:



Figura 20. El mismo diagrama que en la figura anterior, pero delimitando el sistema

Tanto la Figura 19 como la Figura 20 se han dibujado con uno de los múltiples plugins para Eclipse que permiten el dibujo de diagramas UML. Como comentábamos en la sección 8 del Capítulo 1 (página 47), cada herramienta tiene características que las hacen mejores o peores, más o menos adecuadas. En este sencillo ejemplo, el plugin utilizado no permite indicar el sentido de las líneas; sin embargo, la especificación de OMG para UML2 no permite indicar quién desencadena la comunicación en una relación entre un actor y un caso de uso usando flechas. La herramienta Rational Rose sí que

permite esta especificación (aunque, por ejemplo, no permite enmarcar los casos de uso del sistema dentro de un cuadrado). Del mismo modo, el plugin utilizado no permite añadir interfaces a los diagramas de casos de uso, cosa que sí permite Rational Rose. La Figura 21 muestra el mismo diagrama de casos de uso dibujado con esta herramienta: obsérvese la dirección de las flechas (indican qué actor o caso de uso inicia la comunicación, pero sin prohibir que, con posterioridad, haya comunicaciones en el otro sentido) y la presencia de la interfaz “Biblioteca externa”.

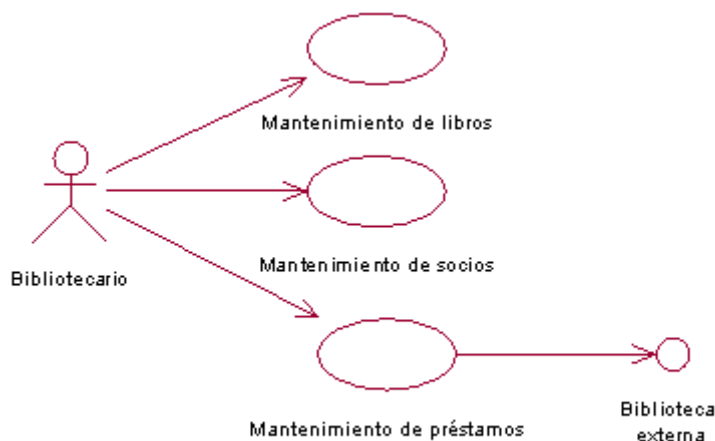


Figura 21. El mismo diagrama de casos de uso, dibujado con Rational Rose

**Relaciones
de inclusión
y extensión
en diagra-
mas de casos
de uso**

Entre los casos de uso pueden existir también relaciones que, por lo general, serán relaciones de inclusión o relaciones de extensión. Un caso de uso A incluye a otro B cuando en el comportamiento de A está incluido el comportamiento de B; por otro lado, A extiende a B cuando A añade su comportamiento al comportamiento de B. En el ejemplo de la biblioteca, podríamos representar el hecho de que en el mantenimiento de los socios se encuentran incluidas las funcionalidades de altas, bajas y modificaciones; del mismo modo, la gestión de las devoluciones y la creación de préstamos son funcionalidades incluidas en el mantenimiento de préstamos. Además, cuando hay retraso en la devolución de un libro prestado, es preciso gestionar la sanción. Las relaciones entre estos casos de uso se representan en la Figura 22.

**Los casos de
uso no de-
ben ser ex-
cesivamente
simples**

Un caso de uso debe entenderse como una funcionalidad de una cierta envergadura ofrecida por el sistema; de este modo, probablemente los casos de uso Altas, Bajas y Modificaciones podrían no tener la categoría de caso de uso y podrían ser eliminados del diagrama.

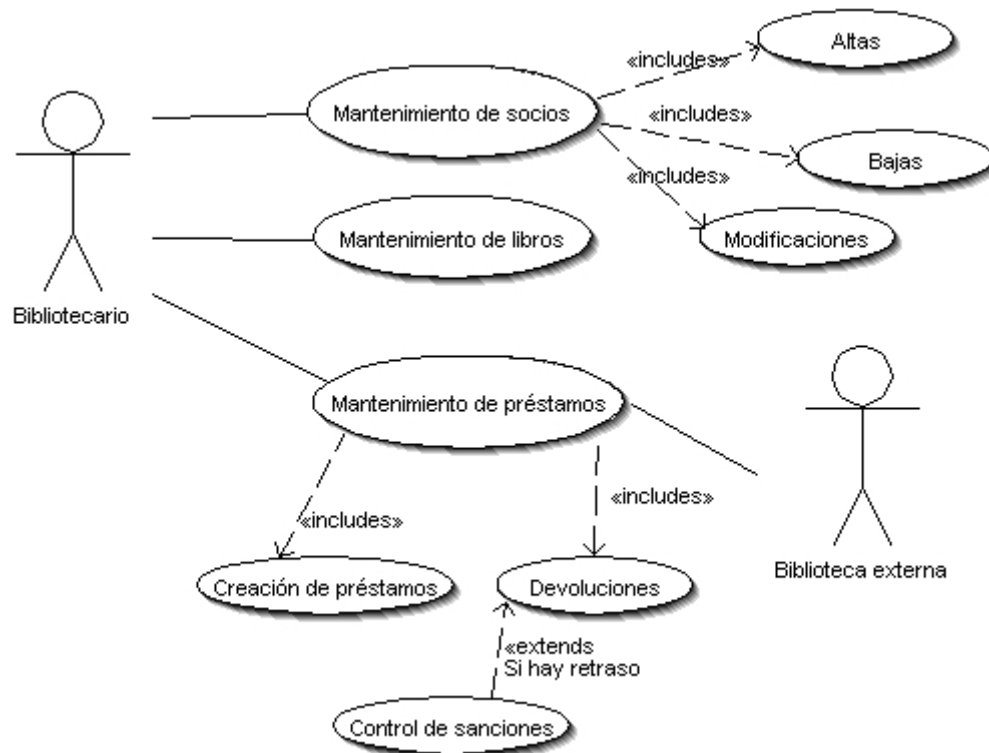


Figura 22. El diagrama de casos de uso de la biblioteca, con nuevos casos de uso y relaciones entre ellos

Relaciones de herencia en diagramas de casos de uso

Además de las relaciones indicadas, existen también relaciones de herencia entre actores o entre casos de uso. Es posible que la biblioteca tenga un actor distinguido que realice préstamos de material especial (archivos históricos, por ejemplo); en este caso, podríamos identificar un actor especializado de Bibliotecario y un caso de uso que especialice a Creación de préstamos, como se muestra en la Figura 23.

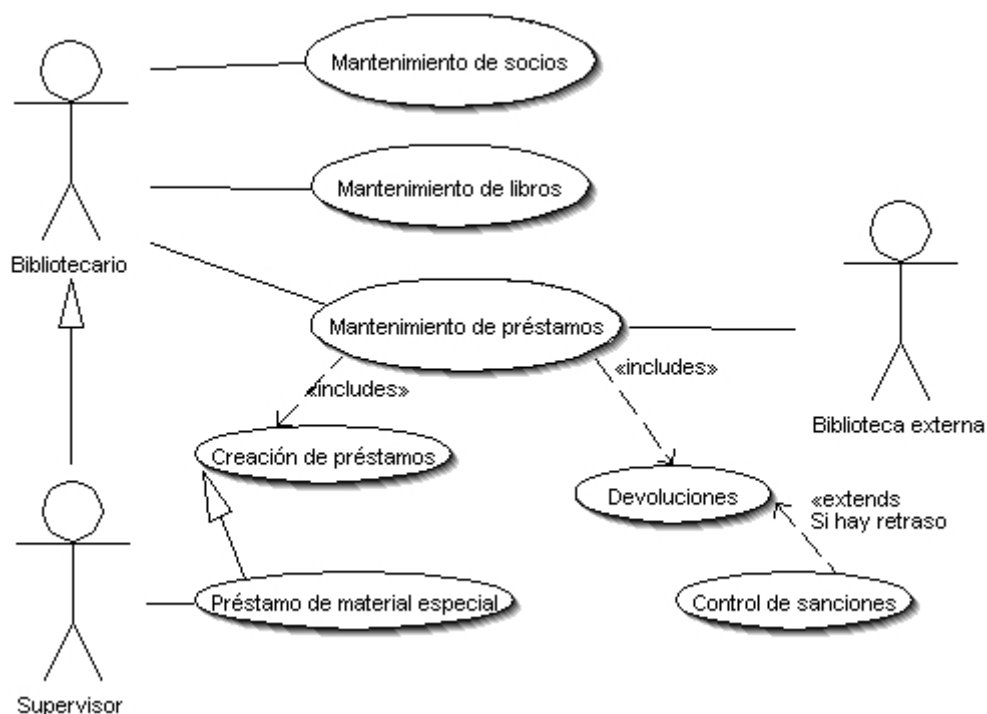


Figura 23. El mismo diagrama, con dos relaciones de herencia entre actores y entre casos de uso

3.2.2 El modelo de análisis

El modelo de análisis se construye a partir del diagrama de casos de uso: para cada caso de uso hacemos una identificación inicial de las posibles clases van a intervenir para ejecutar la funcionalidad que describe el caso de uso. Desde luego, tal identificación es muy preliminar, y podremos hacerlo de forma parecida a cómo en OMT se realiza la identificación de las clases que intervienen en la resolución del problema (véase la sección 1 de este mismo capítulo, página 49).

A partir del diagrama de casos de uso la Figura 21, podríamos asumir que, en la ejecución del caso de uso “Mantenimiento de préstamos” van a intervenir las clases Libro, Socio y Préstamo. La Figura 24 muestra, en UML, la relación entre un caso de uso y algunas de las clases que, al menos de forma preliminar, se ha identificado que le darán servicio.

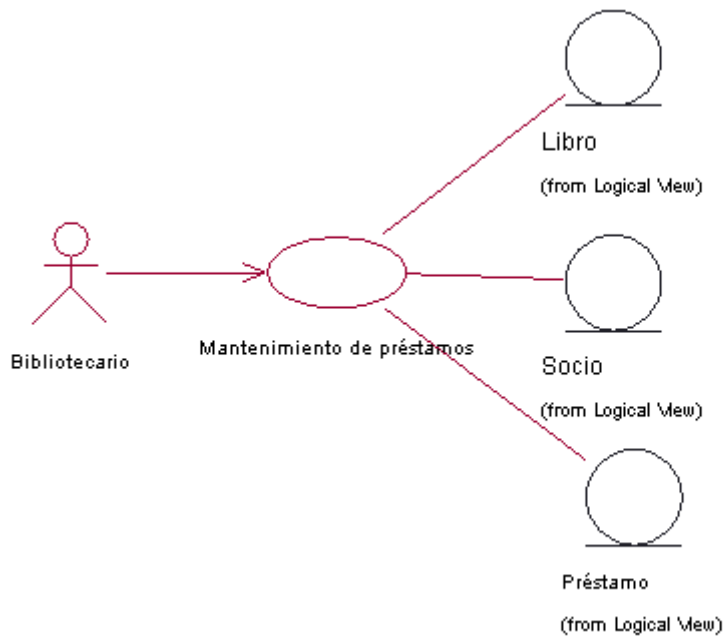


Figura 24. Relación entre el caso de uso Mantenimiento de préstamos y las clases que le darán servicio

Se observa la notación algo especial con que se están representando esas tres clases. UML dispone de la siguiente notación gráfica para representar clases de análisis.

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. Las ventanas que utilizan los usuarios para manejar la aplicación caen dentro de esta categoría.
	<i>Entidad</i> : denotan objetos que perviven. Suelen ser los objetos persistentes de la capa de dominio. Los libros, los préstamos, los socios... son clases de tipo Entidad porque se almacenarán en una base de datos.
	<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. Una clase que se ocupe de comprobar si el socio está sancionado o si el libro devuelto tiene retraso es una clase de control.

Tabla 3. Notación para representar clases de análisis

En la Figura 25 se han añadido nuevas clases de análisis a la descripción del caso de uso “Mantenimiento de préstamos”: además de las clases Li-

bro, Socio y Préstamo, indicamos que intervendrán también una “Ventana de préstamos” y un “Gestor de préstamos”.

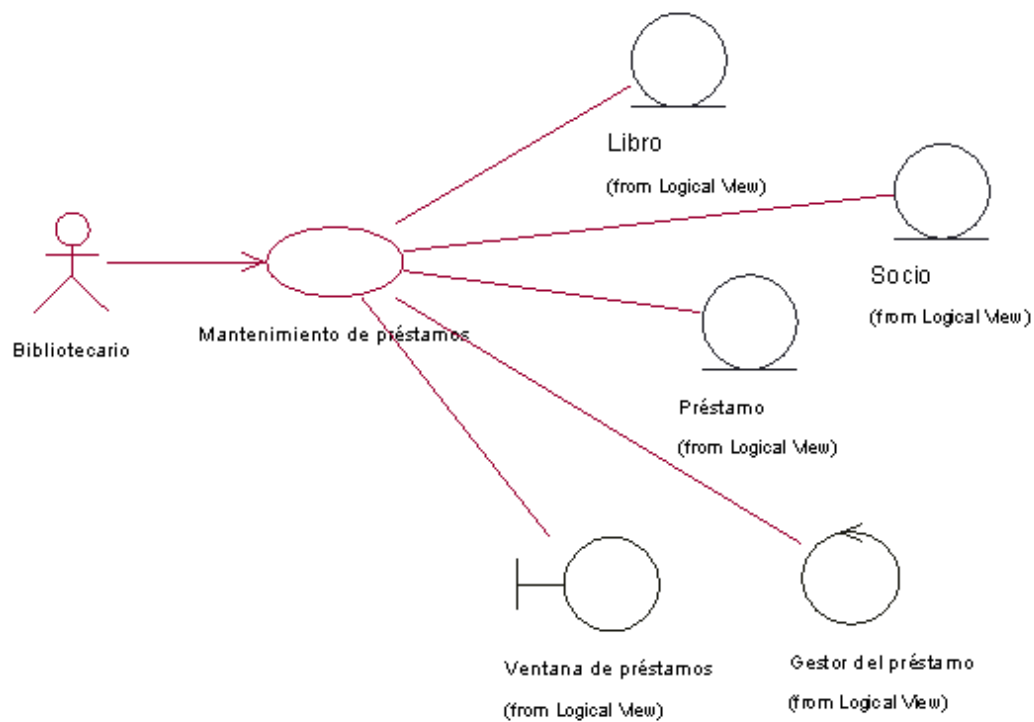


Figura 25. Adición de nuevas clases a la descripción del caso de uso

En UML, la figura anterior se puede también dibujar utilizando estereotipos en lugar de iconos. Un estereotipo es un texto que anota un elemento de UML, que va entre los símbolos « y », y que denota un cierto tipo de elemento:

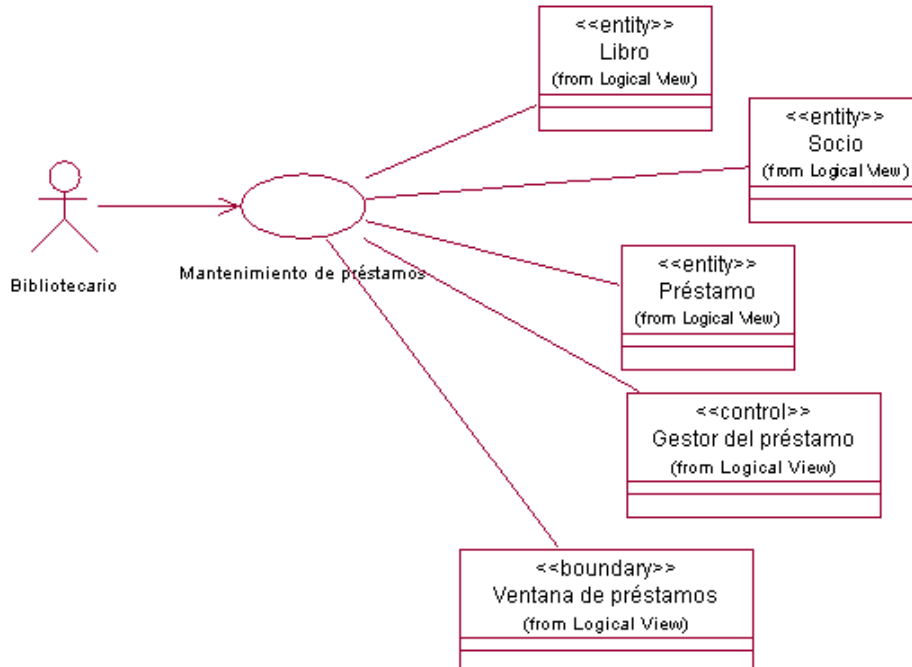


Figura 26. La Figura 25, pero usando estereotipos en lugar de iconos

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 ventana y que ésta utiliza el Gestor del préstamo, que a su vez hace uso de Libro, Socio y Préstamo.

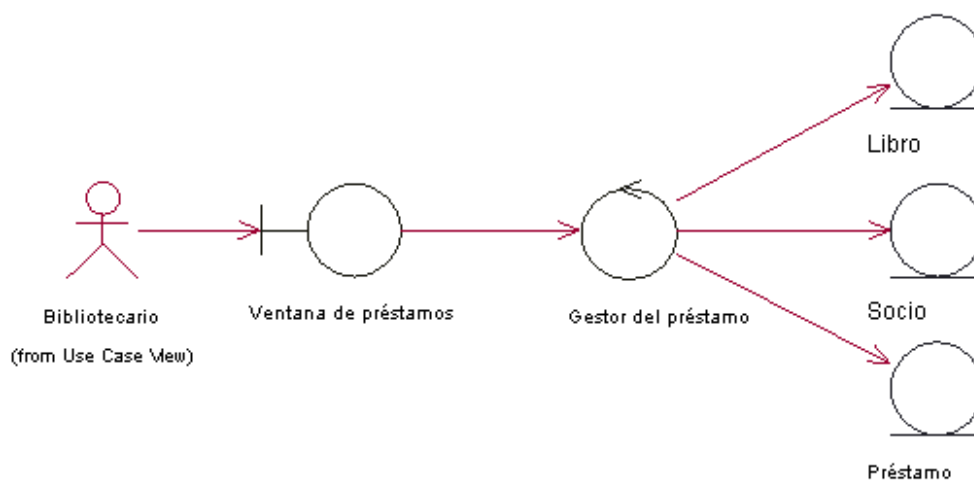


Figura 27. Modelo de clases de análisis procedente del caso de uso Mantenimiento de préstamos

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 otro tipo de diagrama que muestre, de forma más detallada, cómo intervienen las clases indicadas en la ejecución de la funcionalidad que se está describiendo. En particular, para describir el comportamiento del sistema en las diferentes situaciones que pueden ocurrir en un caso uso, suelen utilizarse diagramas de interacción y máquinas de estados.

3.2.3 Modelo de diseño

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

La construcción del modelo de diseño debe abordarse considerando ya las posibles restricciones que nos vaya a imponer el entorno de instalación y ejecución del sistema final.

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 28. En ella se representa el hecho de que, de una sola clase de análisis, se han obtenido dos clases de diseño. Hemos estereotipado las relaciones con el estereotipo *traza* para denotar qué clases de diseño proceden de qué clases de análisis.

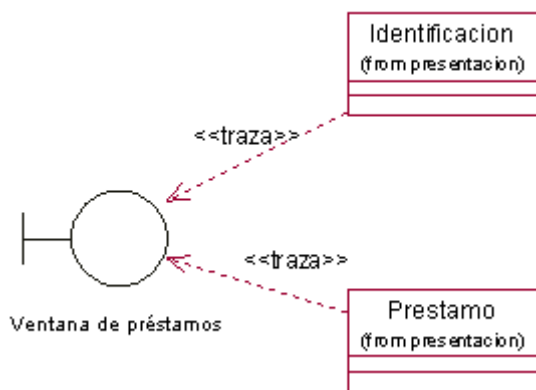


Figura 28. Derivación de clases de diseño a partir de clases de análisis

También podemos ir construyendo un diagrama que muestre cómo interactúan los actores no ya con las clases de análisis (como hacíamos en la Figura 27), sino con las clases de diseño que proceden de aquéllas. Es importante mantener la coherencia entre todos los artefactos que se van construyendo durante el desarrollo del producto software: por ello, si en la Figura 27 el *Bibliotecario* interactuaba con *Ventana de préstamos* y, en la Figura 28, *Identificación* y *Prestamo* procedían de *Ventana de préstamos*, entonces lo

coherente es que *Bibliotecario* interactúe con las clases de diseño que proceden de la clase de análisis *Ventana de préstamos*, tal y como se muestra en la Figura 29. Por otro lado, y como ya debemos tener en cuenta los detalles de la plataforma de ejecución, quitamos los acentos a los nombres de las clases (*Préstamo* ha pasado a ser *Prestamo*, por ejemplo).

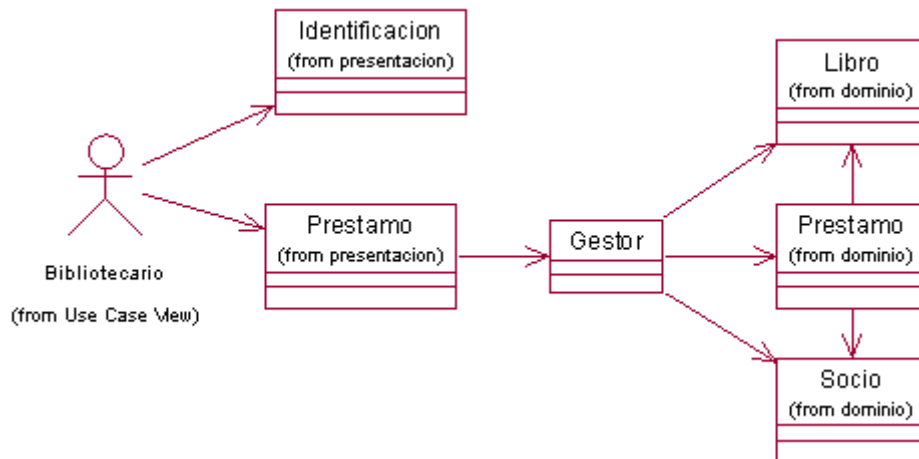


Figura 29. Descripción muy general de las relaciones entre actores y clases de diseño, y entre clases de diseño y clases de diseño

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 17), dispondremos de una *versión entregable*.

3.2.4 Diagrama de componentes

Un componente 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.

3.2.5 Diagrama de despliegue

Representa el conjunto de nodos (ordenadores u otros sistemas hardware) en los que se ejecutará la aplicación, las relaciones entre estos nodos y las relaciones entre los diferentes componentes y los nodos.

3.2.6 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.

4. Lecturas recomendadas

Puede saberse más sobre OMT consultando el libro de Rumbaugh de 1991. En español, puede conseguirse en Internet el libro de apuntes de Alfredo Weitzenfeld (<http://cannes.itam.mx/Alfredo/>).

La sección 6.16 del libro de Craig Larman (edición española del año 2001).

En la página web de Macario Polo Usaola hay un tutorial sobre JUnit, con apuntes adicionales y ejemplos prácticos (<http://www.inf-cr.uclm.es/www/mpolo>). Además, puede descargarse información adicional de <http://www.junit.org>

Capítulo 3. DESCRIPCIÓN DE UN PROBLEMA

En este capítulo se presenta el enunciado de un problema que nos servirá como ejemplo de aplicación de los diferentes métodos y técnicas que se presentarán a lo largo del libro. Nos servirá para, progresivamente, ir presentando los contenidos de la materia de Ingeniería del Software.

1. Enunciado

Se desea implementar una aplicación que permita a varios usuarios jugar, a través de la red, al conocido juego del Monopoly. El sistema constará de un servidor que gestionará el desarrollo de las diferentes partidas que puedan estar jugándose. Es decir, que podrá haber varias partidas en juego simultáneamente, teniendo cada una entre 2 y 6 jugadores.

El servidor estará implementado en Java, mientras que los clientes podrán estar implementados en Java o en C# (un lenguaje estándar del ECMA, incluido en la plataforma Microsoft .NET).

El escenario de ejecución habitual comenzará por que un usuario se conecta a la web del servidor que, además de información de diverso tipo, le permite registrarse introduciendo en un formulario cierta información personal. Una vez registrado, el usuario puede descargarse la aplicación para jugar, que ya no se trata de una aplicación web.

Cuando el usuario ejecuta la aplicación que se ha descargado, se identifica con un nombre de usuario y una contraseña. En respuesta, el servidor le envía la lista de partidas pendientes de comenzar. El usuario puede optar por: (1) unirse a una partida pendiente de comenzar; (2) crear él mismo una partida, que quedará pendiente de comenzar hasta que se cumpla una de estas dos condiciones: (1) que se apunte al menos un jugador más y el jugador que la ha creado decida comenzarla; (2) que el número de jugadores anotados a la partida sea seis, caso en el cual la partida comienza automáticamente.

Una vez comenzada la partida, el desarrollo del juego se rige por las reglas del Monopoly: cada jugador tira un dado y avanza su ficha desde la casilla en la que se encuentre hasta la casilla destino. Al llegar a la casilla destino, y en función del estado del jugador y del estado de la casilla, se ejecuta una acción.

El servidor mantendrá en una base de datos un registro con la evolución de las diferentes partidas. Cada partida podrá jugarse con una edición

distinta del Monopoly: es decir, podrá haber partidas con las calles de Madrid, con las de Barcelona, etc. El administrador, previa identificación, podrá: (1) crear las ediciones del Monopoly que desee (que se almacenarán en la base de datos); (2) realizar cualquier tipo de operación con los usuarios registrados; (3) arrancar y parar el servidor del juego.

El cliente nos impone que la base de datos esté implementada en SQL Server.

2. Adaptación del problema al Proceso Unificado

A partir del enunciado del problema podemos entender que debemos desarrollar hasta cinco aplicaciones diferentes, si bien todas ellas interrelacionadas:

1. Una aplicación web que permita el registro de usuarios. El cliente nos impone desarrollarlo en Java. Esta aplicación correrá en el servidor.
2. Una aplicación para el administrador del sistema, que le permitirá crear ediciones del Monopoly y gestionar usuarios. El cliente también nos impone desarrollarlo en Java.
3. Un cliente que permita a los usuarios jugar remotamente, implementado en Java.
4. Un cliente que permita a los usuarios jugar remotamente, implementado en C#.

De las aplicaciones 3 y 4 podemos deducir la existencia de una quinta, que juegue el papel de servidor para gestionar las diferentes partidas.

El entorno tecnológico nos restringe las posibilidades de desarrollo: en efecto, no existiría dificultad en la construcción de una sola capa de dominio si todas las posibles interfaces de la aplicación compartieran plataforma de ejecución. Al imponernos Java como plataforma de desarrollo de la aplicación web, debemos optar por la utilización de páginas JSP (Java Server Pages), servlets, etc. Para la instalación del sistema, optaremos por un servidor compatible con Java, como Tomcat.

Respecto de la aplicación 2, podemos optar por integrar sus funcionalidades en la primera aplicación, de modo que, en función de los permisos del usuario conectado, se permita simplemente el registro de usuarios, o también la gestión de éstos.

Las aplicaciones 3 y 4 compartirán gran parte de la documentación de análisis y diseño, si bien las diferentes características de los lenguajes de programación y de los entornos de desarrollo harán que comiencen a diferenciarse en alguna etapa concreta.

Desde este punto, y hasta el capítulo XXX, nos centraremos en el desarrollo de la aplicación de administración.

3. Lecturas recomendadas

El libro de Steve McConnell, “Desarrollo y Gestión de Proyectos Informáticos”, editado por McGraw-Hill, incluye algunos capítulos dedicados a la planificación del ciclo de vida, estimación de tiempos y control de proyectos informáticos.

Segunda parte

Capítulo 4. APLICACIÓN DE ADMINISTRACIÓN:

FASE DE COMIENZO

Una vez que conocemos bien el enunciado del problema y que hemos delimitado el alcance de la Aplicación de administración, en este capítulo procederemos a la construcción de la vista funcional del sistema, que implementaremos en la forma de un diagrama de casos de uso. A partir de éstos, redactaremos la primera versión de un plan de iteraciones que nos guíe en las fases sucesivas.

Identificaremos algunas clases de análisis y, a partir de ellas, construiremos clases de diseño e identificaremos los escenarios más relevantes de cada caso de uso.

1. Construcción del diagrama de casos de uso

El diagrama de casos de uso muestra, entre otras cosas, las relaciones entre los actores y los casos de uso. Asimilando un caso de uso con una funcionalidad, el diagrama mostrará, además, las relaciones existentes entre casos de uso.

Del enunciado del problema, las relaciones entre actores y casos de uso vienen descritas en frases como las siguientes:

1. “El administrador, previa identificación, podrá crear las ediciones del Monopoly que desee”.
2. El administrador, previa identificación, podrá “realizar cualquier tipo de operación con los usuarios registrados”.
3. El administrador, previa identificación, podrá “arrancar y parar el servidor de juego”.

Podemos claramente deducir la existencia del actor *Administrador*, que es la persona encargada de la administración del sistema. Además, y puesto que el sistema almacena la información de las partidas, ediciones y usuarios en una base de datos, podemos identificar también el actor *Base de datos*.

Respecto de los casos de uso, éstos se encuentran dispersos, normalmente en forma de verbos, entre el texto de las frases anteriores: *crear edición, realizar cualquier tipo de operación, arrancar y parar...* y también, obviamente, la *identificación*.

El diagrama de casos de uso debe mostrar las funcionalidades de la aplicación indicando qué hace el sistema y qué actores está involucrados en cada funcionalidad, pero sin preocuparse de los detalles del cómo lo hace. En la Figura 30 aparece una versión inicial del diagrama de casos de uso del sistema. Como se observa, de momento no es más que una lista de actores y funcionalidades, sólo que puesta de forma gráfica: en la figura, se indica que el *Administrador* se identifica ante el sistema, crea ediciones y gestiona usuarios, funcionalidades éstas para las que utiliza la base de datos; además, arranca y para el sistema.

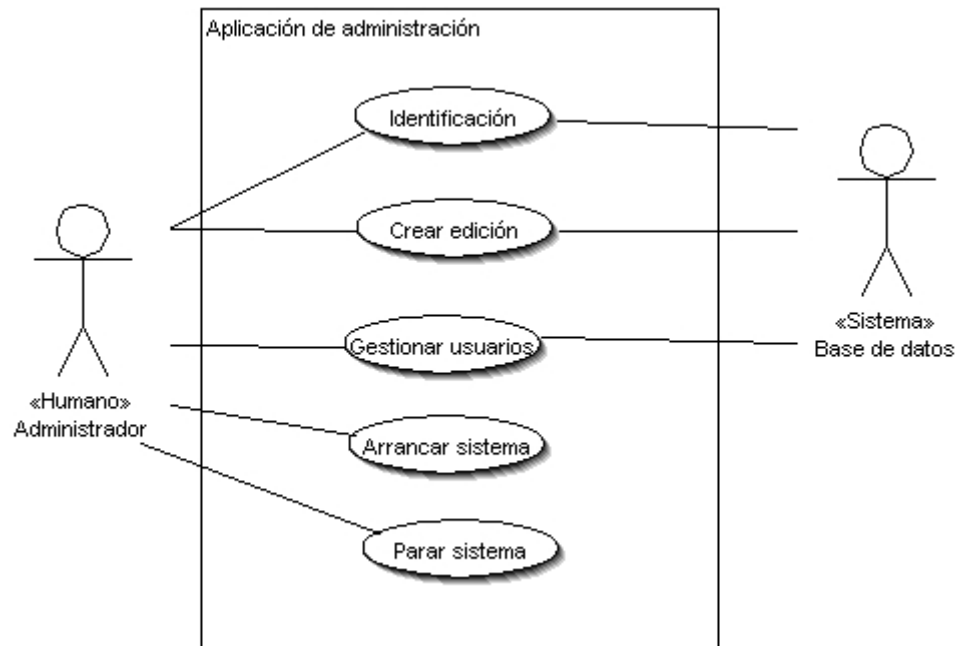


Figura 30. Vista funcional (I) de la aplicación de administración

Cada caso de uso puede suponer la ejecución de otros casos de uso. La creación de ediciones supone, por ejemplo, la creación de las diferentes casillas del tablero y de las tarjetas de Suerte y de Caja de comunidad.

Los tipos de casillas que hay en el Monopoly se explican en la Tabla 4.

Tipo de casilla	Descripción
Calles	Hay 22 y están agrupadas en barrios, que se identifican con colores. Tienen nombre, precio de adquisición, valor hipotecario, precio de adquisición de casas y hoteles y varios precios de alquiler (según la calle esté sin edificar, con una casa, con dos casas, con tres casas, con cuatro casas y con hotel). Se puede edificar en una calle cuando el jugador propietario tiene todas las calles de un barrio.
Estaciones	Hay cuatro. Tienen nombre, precio de adquisición, valor hipotecario y, en función del número de estaciones que tenga el jugador propietario, varios precios de alquiler.
Compañías de servicios públicos	Hay dos: la compañía de la luz y la del agua). Tienen nombre, precio de adquisición y valor hipotecario. Tienen dos precios de alquiler, según el propietario de la casilla tenga una o dos compañías.
Impuestos	Hay dos. Tienen un nombre. Cuando un jugador cae en ellas, paga a la banca una cantidad determinada.
Salida	Hay una. La banca le paga al jugador una cierta cantidad cada vez que pasa por ella.
Casillas de suerte	Hay 3. El jugador coge una tarjeta de suerte y realiza la acción indicada en la tarjeta.
Casillas de Caja de comunidad	Hay 3. El jugador coge una tarjeta de caja de comunidad y realiza la acción indicada en la tarjeta.
Parking gratuito	No se realiza ninguna acción.
Vaya a la cárcel.	Hay una. Si un jugador cae en esta casilla, va a la casilla de cárcel. El jugador puede optar por pagar una determinada multa y seguir jugando, o bien por quedarse tres turnos sin tirar. El jugador también va a la cárcel si se lo manda una casilla de Suerte o de Caja de comunidad.
Cárcel	Si un jugador cae en esta casilla, no realiza ninguna acción.

Tabla 4. Tipos de casillas en el juego del Monopoly

Las tarjetas de Suerte y de Caja de comunidad son muy similares y se cogen del montón cada vez que un jugador cae en una casilla de Suerte o de Caja de Comunidad. La tarjeta ordena la realización de una acción, que se corresponden con las acciones indicadas en la :

Tipo de acción
Ir a una casilla determinada
Avanzar o retroceder un cierto número de casillas
Pagar una cantidad fija a la banca
Cobrar una cantidad fija de la banca
Pagar una cantidad a la banca según el número de casas y hoteles que tenga el jugador
Cobrar una cantidad fija de los otros jugadores

Tabla 5. Posibles acciones ordenadas por las tarjetas de Suerte y de Caja de Comunidad

Para dar por terminada la creación de una *Edición* es preciso crear el conjunto de casillas y tarjetas del juego. Toda edición tendrá exactamente cuarenta casillas y un número variable de tarjetas de *Suerte* y de *Caja de comunidad*. Al crear la edición y guardarla en la base de datos, podremos también almacenar la información de las cuarenta casillas sin asignarles tipo. La creación de las tarjetas será posterior, ya que el número de éstas es variable.

Con estas consideraciones, podemos modificar el diagrama de casos de uso anterior para indicar que la ejecución del caso de uso *Crear edición* supondrá la ejecución de un nuevo caso de uso *Crear casillas vacías*, que creará las cuarenta casillas del tablero sin asignación de tipo.

Posteriormente el usuario asignará el tipo a las cuarenta casillas y crearán las tarjetas del juego. Estas funcionalidades se representan en la Figura 31: el caso de uso *Crear edición* incluye la ejecución de la funcionalidad *Crear casillas vacías*. Por otro lado, el actor *Administrador* puede crear las tarjetas y asignar tipos a las casillas del tablero.

Obsérvese que el diagrama de casos de uso no es una máquina de estados (es decir, en ningún caso pretenden mostrar los diferentes estados por los que puede pasar el juego del Monopoly) ni indica multiplicidades de ningún tipo (no se indica cuántas veces se ejecuta cada funcionalidad). Nótese que la relación desde *Crear edición* hacia *Crear casillas vacías* está anotada con el estereotipo *includes*. *includes* se utiliza para distinguir, en el diagrama de casos de uso, una funcionalidad que, por alguna razón, se desea representar de forma separada: puede ser una funcionalidad reutilizable (usada por varios casos de uso), una funcionalidad con envergadura suficiente como para marcarla aparte (como ocurre en este caso)... Obsérvese también que la flecha se traza desde el caso de uso principal hasta el caso de uso incluido: es decir, que siguiendo el sentido de la flecha podemos construir una frase en la forma de sujeto-verbo-predicado: *Crear edición incluye a Crear casillas vacías*. En cierto modo, las relaciones de inclusión pueden servir para representar relaciones de orden entre casos de uso, aunque no siempre es así: en la figura, *Crear casillas vacías* se ejecuta después de *Crear edición*; sin embargo, podría haber una relación directa entre *Administrador* y el caso de uso incluido, con lo que la relación de orden ya no sería válida.

Véanse también dos nuevos casos de uso cuyo nombre se ha escrito en letra cursiva, lo que denota que se tratan de casos de uso abstractos: *Crear tarjeta* y *Asignar tipo a casilla*.

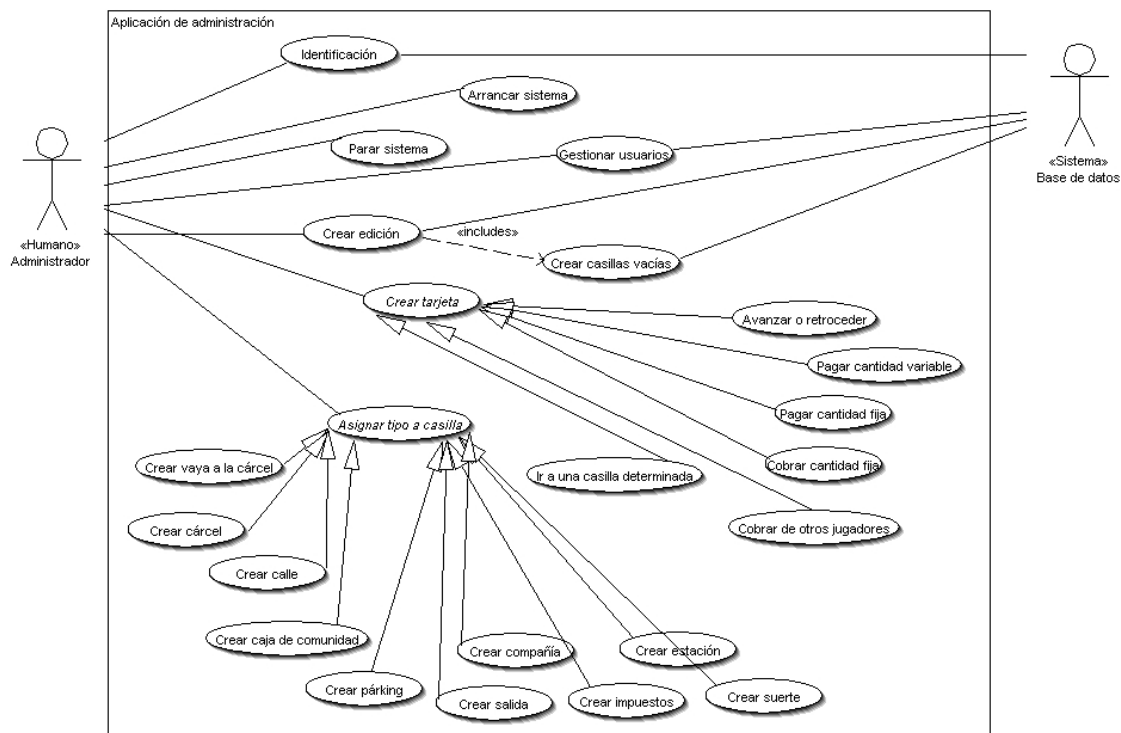


Figura 31. Vista funcional (II) de la aplicación de administración

2. Priorización de casos de uso

El plan de iteraciones es un documento que guiará al grupo de ingenieros de software durante el desarrollo del proyecto. En él se indica el orden en que se irán implementando los casos de uso. Para establecer este orden, debe realizarse una priorización de casos de uso que atienda a criterios como complejidad, necesidad de un caso de uso para implementar otro, etc.

Los casos de uso que teníamos en la Figura 31 son los siguientes: Identificación, Arrancar sistema, Parar sistema, Gestionar usuarios, Crear edición, Crear casillas vacías, Asignar tipo a casilla, Crear vaya a la cárcel, Crear cárcel, Crear calle, Crear caja de comunidad, Crear parking, Crear salida, Crear compañía, Crear impuestos, Crear estación, Crear suerte, Crear tarjeta, Avanzar o retroceder, Ir a una casilla determinada, Cobrar de otros jugadores, Cobrar cantidad fija, Pagar cantidad fija y Pagar cantidad variable.

Para realizar cualquier operación, el usuario del sistema (actor *Administrador*) debe identificarse. Para parar el sistema, primero habrá sido preciso arrancarlo. Además, como quizás no tenga mucha sentido arrancar el sistema sin que el administrador tenga la posibilidad de gestionar usuarios con al menos una edición creada, podemos dejar los casos de uso *Arrancar sistema* y *Parar sistema* para el final. Respecto del orden de desarrollo de los casos de uso correspondientes a la creación de los diferentes tipos de casillas y tarjetas, comenzaremos por la creación de los diferentes tipos de casillas y, dentro de éstos, por aquellos cuyas casillas tengan menos complicación.

Con estas consideraciones, podemos plantear la siguiente ordenación:

1. Identificación
2. Crear edición
3. Crear casillas vacías
4. Asignar tipo a casilla
5. Crear parking
6. Crear salida
7. Crear impuestos
8. Crear cárcel
9. Crear compañía
10. Crear estación
11. Crear calle
12. Crear vaya a la cárcel
13. Crear caja de comunidad
14. Crear suerte
15. Crear tarjeta
16. Ir a una casilla determinada
17. Avanzar o retroceder
18. Cobrar cantidad fija
19. Pagar cantidad fija
20. Pagar cantidad variable
21. Cobrar de otros jugadores
22. Gestionar usuarios
23. Arrancar sistema
24. Parar sistema

Como se recordará (Tabla 1, página 56), en la fase de Comienzo (en la que nos encontramos) se analizan y detallan aquellos casos de uso que el equipo de desarrollo considere de más importancia. De estas tareas puede decidirse la viabilidad o inviabilidad del proyecto con las condiciones pactadas con el cliente (tiempo, coste, calidad deseada y alcance del proyecto).

Una vez realizada la ordenación de los casos de uso, debemos planificar un conjunto de iteraciones, y ubicar en cada una de ellas un conjunto de casos de uso. Tendremos en cuenta que cada iteración conlleva un poco de análisis, diseño, codificación y pruebas. Así, en cada iteración intentaremos colocar casos de uso más o menos similares, de manera que, concluidos y probados los desarrollos de la iteración, podemos asegurar que el fragmento de sistema que llevamos construido es funcionalmente correcto.

En la iteración 1, por ejemplo, además de *Identificación*, desarrollaremos *Crear edición*, *Crear casillas vacías*, *Asignar tipo a casilla* y *Crear parking*. El Parking gratuito es probablemente la más sencilla de todas las casillas, ya que el jugador que llega a ella no ejecuta ninguna acción. A la vis-

ta de esto, parece conveniente terminar la iteración con el desarrollo de ese caso de uso y con la prueba de todos los anteriores.

Con estas consideraciones, una primera versión del plan de iteraciones podría ser la mostrada en la siguiente tabla:

Fase	Iteración	Caso de uso	Flujos de trabajo				
			R	A	D	I	P
Comienzo	1	Identificación	X	X			
		Crear edición	X	X			
		Crear casillas vacías	X	X			
		Asignar tipo a casilla	X	X			
		Crear parking	X	X			
Elaboración	2	Identificación		X	X	X	X
		Crear edición		X	X	X	X
		Crear casillas vacías		X	X	X	X
		Asignar tipo a casilla		X	X	X	X
		Crear parking		X	X	X	X
		Crear salida		X	X	X	X
		Crear impuestos		X	X	X	X
		Crear cárcel		X			
		Crear compañía		X			
		Crear estación		X			
		Crear calle		X			
		Crear vaya a la cárcel		X			
Construcción	3	Crear cárcel		X	X	X	X
		Crear compañía		X	X	X	X
		Crear estación		X	X	X	X
		Crear calle		X	X	X	X
		Crear vaya a la cárcel		X	X	X	X
	4	Crear caja de comunidad		X	X	X	X
		Crear suerte		X	X	X	X
	5	Crear tarjeta		X	X	X	X
		Ir a una casilla determinada		X	X	X	X
		Avanzar o retroceder		X	X	X	X
	6	Cobrar cantidad fija		X	X	X	X
		Pagar cantidad fija		X	X	X	X
		Pagar cantidad variable		X	X	X	X
		Cobrar de otros jugadores		X	X	X	X
	7	Gestionar usuarios	X	X	X	X	X
	8	Arrancar sistema	X	X	X	X	X
		Parar sistema	X	X	X	X	X
Transición							

Tabla 6. Plan de iteraciones

3. Descripción textual de los casos de uso

Además de gráficamente, los casos de uso deben especificarse también textualmente, en documentos que expliquen de manera clara el requisito funcional que están representando. Existen multitud de plantillas diseñadas para contener la información significativa de cada caso de uso. La plantilla incluida en el plugin de la herramienta Eclipse que estamos utilizando es bastante completa (Figura 32). Además del nombre del caso de uso y de si es o no abstracto, incluye la siguiente información:

- Precondiciones

- Postcondiciones
- Rango
- Flujo normal de eventos
- Flujos de eventos alternativo
- Descripción

Una parte importante de las actividades de análisis se dedica a la redacción de documentos en lenguaje natural, en los que se explican claramente cada uno de los requisitos del sistema. La especificación de requisitos debe ser correcta, completa y sin inconsistencias. La utilización de plantillas como la mostrada en la figura ayuda a que estos documentos se redacten adecuadamente.

Figura 32. Plantilla de descripción de casos de uso del plugin de Eclipse

Las precondiciones se utilizan para describir el estado del sistema antes de que se ejecute el caso de uso. Por “estado del sistema” puede entenderse no el estado completo de todos los objetos involucrados, sino el de aquellos fragmentos del sistema interesados en el caso de uso.

Las postcondiciones describen el estado en que queda el sistema (o el fragmento del sistema que resulte relevante) tras la ejecución de la funcionalidad representada por el caso de uso.

El campo *rank* (rango, categoría) denota el grado de prioridad que daremos al caso de uso en el plan de iteraciones.

El flujo normal de eventos representa la secuencia habitual de operaciones que ejecuta el sistema al servir la funcionalidad que se está describiendo. Habrá situaciones anormales (como aquellas en las que se produzca un error, o no se pueda conectar con un dispositivo externo) que se describen en las secciones de los flujos de eventos alternativos.

Por último, se completa la descripción del caso de uso con un texto explicativo que se redacta en formato libre y que puede incluir figuras, algoritmos, etc.

3.1. Descripción textual de casos de uso

Para esta fase se ha decidido especificar detalladamente los casos de uso *Identificación*, *Crear edición*, *Crear casillas vacías*, *Asignar tipo a casilla* y *Crear parking*.

3.1.1 Descripción del caso de uso *Identificación*

La Figura 33 muestra las clases de análisis identificadas a primera vista para *Identificación*. Puesto que, de acuerdo con el diagrama de casos de uso (Figura 31), *Identificación* se relaciona con dos actores distintos, creamos dos clases *boundary*: una ventana, adecuada al actor humano, y un agente de base de datos, adecuado para manejar la relación del caso de uso con la base de datos. Además, inicialmente identificamos una clase *controller* que controle la ejecución del caso de uso, y una clase *entity* a la que asignaremos las responsabilidades de dominio.

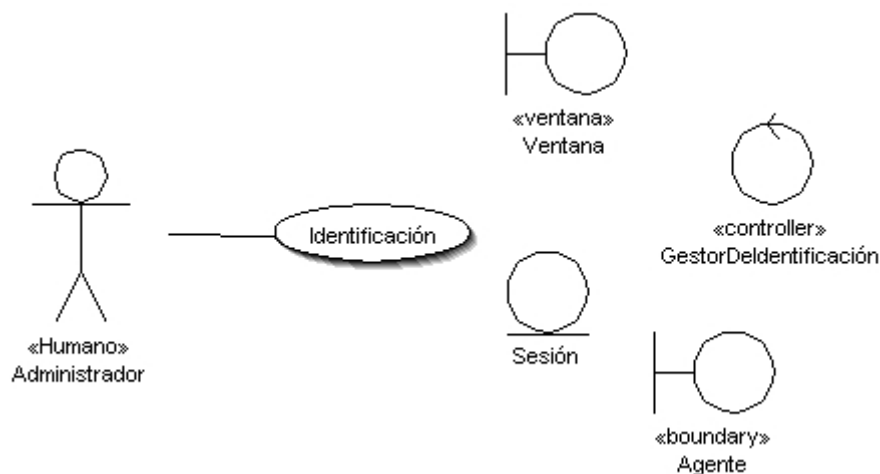


Figura 33. Clases de análisis involucradas en el caso de uso *Identificación*

De todos modos, el caso de uso es tan sencillo que podemos asignar a la ventana las responsabilidades del controlador, eliminando de este modo una de las clases de análisis:

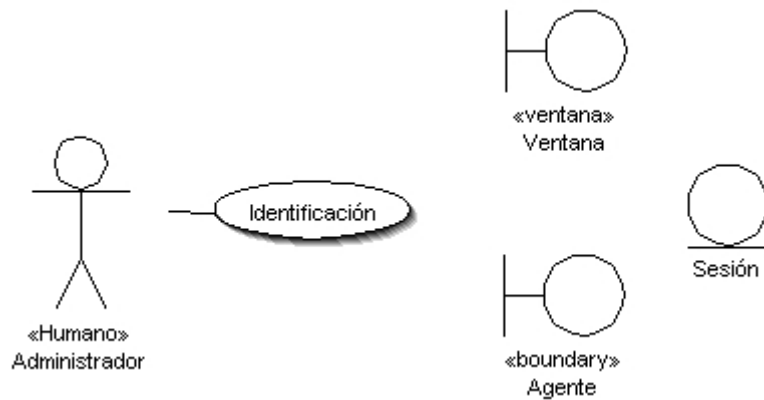


Figura 34. Clases de análisis seleccionadas

La plantilla con la descripción textual del caso de uso la redactaremos teniendo en cuenta las clases de análisis que se han identificado en la figura anterior. A continuación se muestra la descripción que hemos introducido en el plugin de Eclipse.

Nombre: Identificación
Abstracto: no
Precondiciones: 1. El usuario no debe ya estar identificado ante el sistema
Postcondiciones:
Rango: 1
Flujo normal: 1. El usuario escribe el login y la contraseña en una ventana 2. La ventana crea una sesión con el login y la contraseña 3. La sesión comprueba que el usuario no esté ya identificado 4. La sesión valida sus parámetros con la base de datos a través del agente, que son correctos 5. Se añade la sesión a la lista de sesiones 6. El usuario queda identificado ante el sistema 7. La ventana de identificación muestra una ventana con el menú de opciones
Flujo alternativo 1: 1. El usuario escribe el login y la contraseña en una ventana 2. La ventana crea una sesión con el login y la contraseña 3. La sesión comprueba que el usuario ya estaba identificado previamente 4. Se muestra un mensaje de error en la ventana del usuario
Flujo alternativo 2: 1. El usuario escribe el login y la contraseña en una ventana 2. La ventana crea una sesión con el login y la contraseña 3. La sesión comprueba que el usuario no esté identificado previamente 4. La sesión valida sus parámetros con la base de datos, que son incorrectos 5. Se muestra un mensaje de error en la ventana del usuario
Descripción: Este caso de uso se ejecuta cuando el usuario administrador desea conectarse al sistema para administrarlo.

Tabla 7. Descripción textual del caso de uso Identificación

De la descripción de este caso de uso podemos resaltar algunas cuestiones:

1. La precondición denota que, si el usuario ya está conectado al sistema, éste no debe permitirle volver a conectarse.
2. No hay postcondición porque, como se ha dicho, éstas afectan a todos los escenarios posibles del caso de uso. Por tanto, sería incorrecto indicar como postcondición que “El usuario queda

identificado ante el sistema”, ya que esta restricción no sería válida para ninguno de los flujos alternativos.

3. La mayoría de los eventos de los distintos flujos están redactados en la forma sujeto-verbo-predicado. Esta redacción nos ayudará posteriormente a identificar más claramente los objetos y sus clases, y a representar los pasos de mensajes en términos de objetos reales.

3.1.2 Descripción del caso de uso *Crear edición*

Los documentos preparados en las tareas de recolección de requisitos resultan de suma utilidad para la descripción textual del caso de uso. Durante las entrevistas con el cliente, pueden prepararse en papel bocetos de la interfaz de usuario. La Figura 35 muestra el posible diseño propuesto para la ventana de creación de ediciones. De acuerdo con los textos manuscritos de la figura, y con el diagrama de la Figura 31, este caso de uso tiene relación con *Crear casillas vacías*.

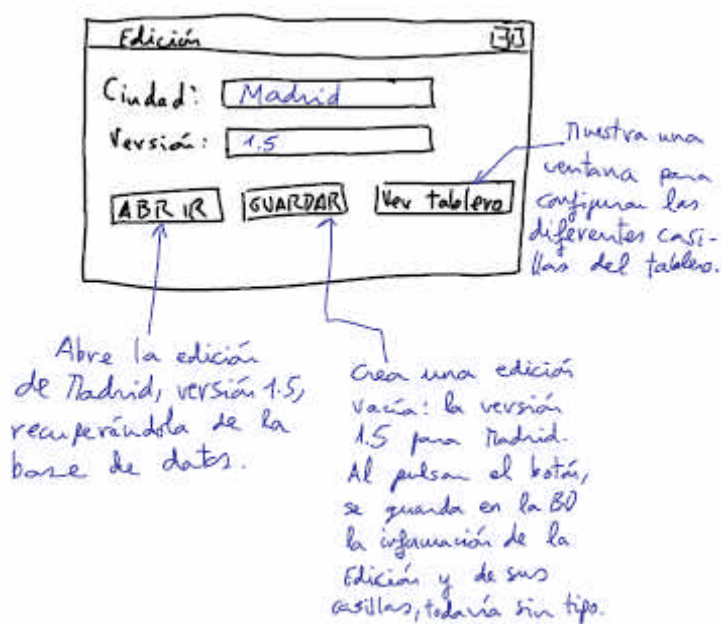


Figura 35. Documento procedente de las entrevistas con el cliente

En este caso de uso, entonces, entran en juego las clases de análisis identificadas en la Figura 36.

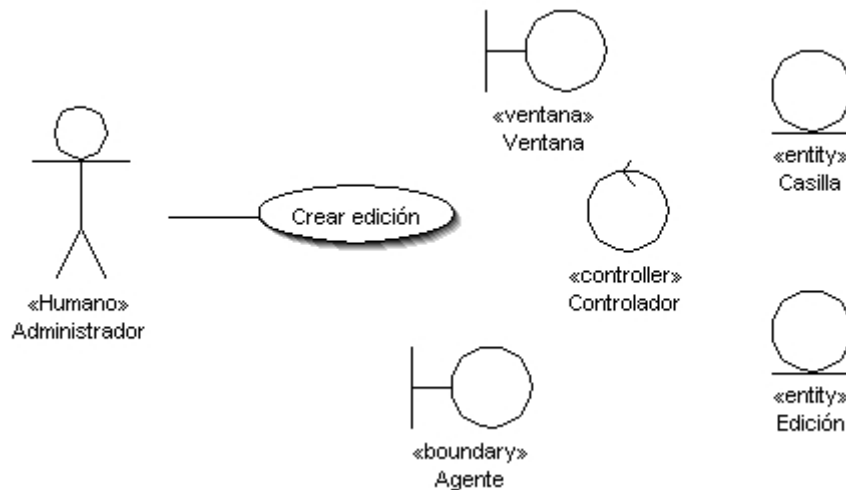


Figura 36. Clases de análisis involucradas en el caso de uso *Crear edición*

La descripción textual de este caso de uso se muestra en la Tabla 8. Obsérvese que no ponemos flujos alternativos, aunque podría haberlos: por ejemplo, el sistema no debe permitir que se inserten dos ediciones con el mismo número de versión para la misma ciudad. Del mismo modo, podríamos recoger como un flujo alternativo el hecho de que no haya conexión con la base de datos... Sin embargo, son escenarios tan sencillos y tan obvios, que no merece la pena considerarlos como tales.

Nombre: Crear edición
Abstracto: no
Precondiciones: 1. El usuario se encuentra identificado como administrador
Postcondiciones: 1. Se crea una nueva edición del juego, que se almacena en la base de datos, incluyendo 40 casillas sin especificación del tipo
Rango: 2
Flujo normal: 1. El usuario selecciona en una ventana la opción de crear una edición nueva 2. Se muestra una ventana adecuada para crear una edición nueva del juego 3. El usuario escribe el nombre de la ciudad que se corresponde con esta edición, y le asigna también un número de versión 4. La ventana envía los datos introducidos al controlador del caso de uso, que crea una nueva Edición, a la que asigna los valores pasados tecleados por el usuario 5. El controlador le dice a la Edición que se inserte en la base de datos 6. La Edición se inserta a sí misma a través del Agente 7. Se ejecuta el caso de uso <i>Crear casillas vacías</i> , que crea las cuarenta casillas del tablero vacías (es decir, sin especificación del tipo)
Descripción:

Tabla 8. Descripción textual del caso de uso *Crear edición*

3.1.3 Descripción del caso de uso *Crear casillas vacías*

Este caso de uso es llamado desde *Crear edición*: de hecho, el séptimo mensaje de la descripción textual dada en la Tabla 8 indica explícitamente que se ejecuta esta funcionalidad. En la vista funcional del sistema (Figura 31, página 79), se observa que este caso de uso no tiene relación directa con el actor *Administrador*, por lo que no se creará clase *boundary* para gestio-

nar las relaciones del caso de uso con este actor, aunque sí con la base de datos. Las clases involucradas, por tanto, son las siguientes:

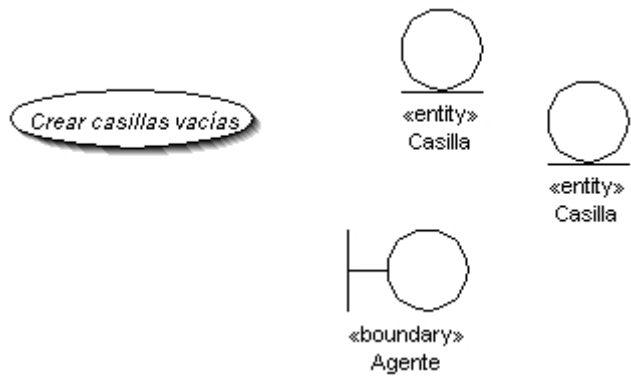


Figura 37. Clases de análisis para *Crear casillas vacías*

La descripción textual del caso de uso sería la siguiente:

Nombre: Crear casillas vacías
Abstracto: no
Precondiciones: 1. Hay una edición creada, que es la responsable de crear sus 40 casillas
Postcondiciones: 1. Se crean las 40 casillas del juego sin especificar su tipo, que quedan almacenadas en la base de datos
Rango: 3
Flujo normal: 1. La <i>Edición</i> crea 40 instancias de <i>Casilla</i> , y las va insertando una a una en la base de datos
Descripción:

Tabla 9. Descripción textual de *Crear casillas vacías*

3.1.4 Descripción del caso de uso *Asignar tipo a casilla*

La Figura 38 es un boceto de la interfaz de usuario, dibujado durante las entrevistas con el cliente. Éste desea crear las casillas en una ventana más o menos como la mostrada en la figura.

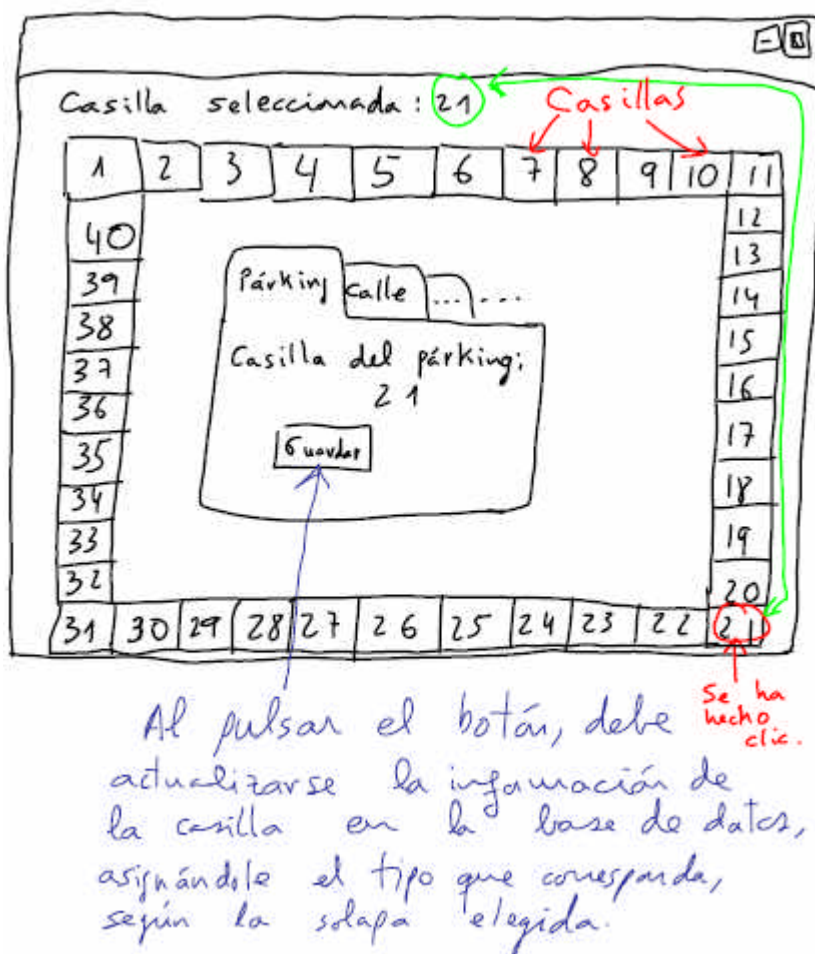


Figura 38. Boceto de la ventana para el caso de uso Asignar tipo a casilla

De acuerdo con las especificaciones del cliente, el *Administrador* seleccionará una de las casillas mostradas en la ventana y, a continuación, elegirá el tipo que le corresponde seleccionando una de las solapas. Un escenario típico de ejecución sería el siguiente:

1. Utilizando la ventana cuyo boceto se mostraba en la Figura 35 (página 85), el usuario ha creado una *Edición* nueva, con lo que ésta dispone de 40 casillas sin tipo. Acto seguido, ha pulsado el botón etiquetado "Ver tablero" y se muestra la ventana de la Figura 38.
2. El usuario hace click en una de las casillas (por ejemplo, en la nº 21), que no tiene tipo asignado.
3. En la zona central de la ventana, el usuario elige una solapa correspondiente a un tipo. Elige, por ejemplo, la solapa etiquetada *Páking*, lo cual denota que la casilla será la de "Páking gratuito".

4. El usuario pulsa el botón *Guardar*, con lo que el tipo de la casilla 21 se actualiza en la base de datos, almacenándose como la del Párking gratuito.

Otro posible escenario de ejecución se da cuando la *Edición* ya dispone del número de casillas de cierto tipo (por ejemplo, dispone ya de una Salida, Cárcel, Párking gratuito o Vaya a la cárcel, o de cuatro estaciones, o de dos casillas de impuestos), pero el usuario decide crear una nueva casilla de ese tipo. Por ejemplo: supongamos que el usuario decide cambiar la ubicación del Párking gratuito de la casilla 21 a la casilla 30. La descripción del escenario podría ser la siguiente:

1. Utilizando la ventana cuyo boceto se mostraba en la Figura 35 (página 85), el usuario ha asignado a la casilla nº 21 el tipo Párking gratuito.
2. El usuario hace click en la casilla nº 30, que no tiene tipo asignado.
3. En la zona central de la ventana, el usuario elige la solapa etiquetada Párking, lo cual denota que la casilla 30 será la de "Párking gratuito".
4. El usuario pulsa el botón Guardar, con lo que: (1) el tipo de la casilla 21 se elimina de la base de datos; (2) el tipo de la casilla 30 se almacena como la del Párking gratuito.

De acuerdo con estos escenarios, los elementos más importantes que intervendrán en la ejecución de este caso de uso serán la *Ventana*, que contendrá las diferentes *solapas* y los *botones* correspondientes a las casillas, la *Edición* y cada una de las *Casillas*. Nótese que se debe hacer una distinción clara entre los elementos que actúan de mero interfaz de usuario (como los *botones correspondientes a casillas*) y los elementos que resuelven el problema (las *Casillas*).

El diagrama de clases de análisis podría ser el siguiente:

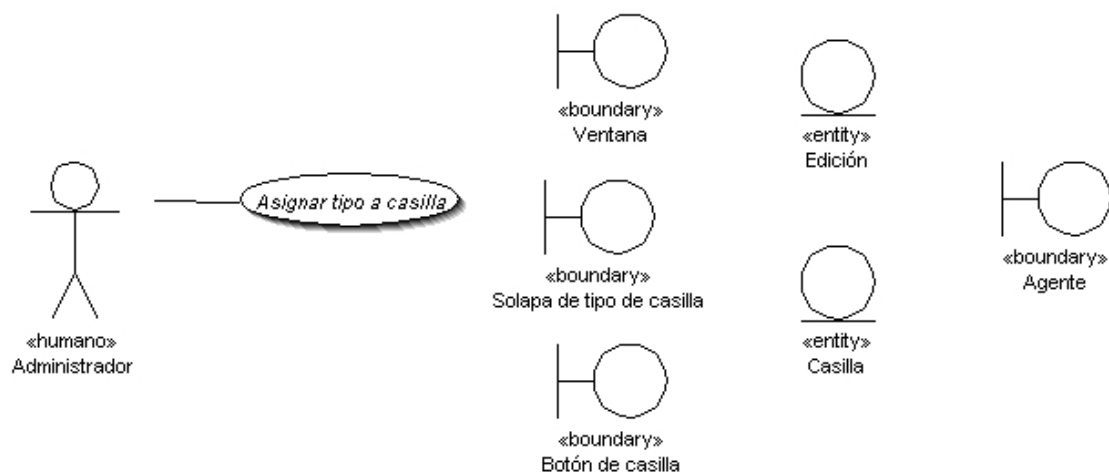


Figura 39. Clases de análisis identificadas para *Asignar tipo a casilla*

A partir de estas clases de análisis y de la descripción de los dos escenarios anteriores, puede redactarse la descripción textual de la Tabla 10.

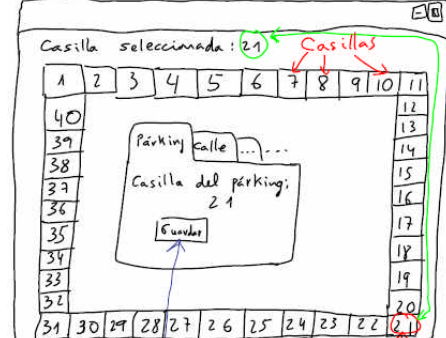
Nombre: Asignar tipo a casilla
Abstracto: sí
Precondiciones: 1. Hay una edición del juego que se está creando
Postcondiciones:
Rango: 4
Flujo normal: 1. En una ventana como la mostrada en la sección de Descripción, el usuario selecciona el número de la casilla a la que desea asignar el tipo. 2. La ventana muestra en la etiqueta situada en arriba a la izquierda el número de la casilla seleccionada. 3. El usuario elige una solapa en la zona central de la ventana, que se corresponde con el tipo que se asignará a la casilla. 4. El usuario da la información necesaria para el tipo de casilla (p.ej.: si es una calle, asigna el nombre, precios, etc.) 5. El usuario pulsa el botón guardar. 6. La ventana, que conoce a la Edición que se está creando, le dice a ésta que el tipo de la casilla seleccionada es el correspondiente a la solapa elegida en el evento 3 7. La Edición comprueba que ella misma admite ese tipo de casilla 8. La Edición asigna a la casilla la información dada por el usuario 9. La Edición asigna a la <i>Casilla</i> el tipo correspondiente 10. La casilla actualiza su tipo en la base de datos a través del Agente
Descripción: La ventana de asignación de tipos tendrá el siguiente aspecto: 

Tabla 10. Descripción textual del caso de uso *Asignar tipo a casilla*

3.1.5 Descripción del caso de uso *Crear parking*

Este caso de uso es una especialización del caso de uso anterior, *Asignar tipo a casilla*. Su descripción textual será igual, pero diferirá en los eventos dependientes del tipo de casilla. Es decir:

Nombre: Crear parking
Abstracto: no
Precondiciones: 1. Hay una edición del juego que se está creando
Postcondiciones:
Rango: 5
Flujo normal: <i>No había casilla de Parking en la edición</i> 1. En una ventana como la mostrada en la sección de Descripción, el usuario selecciona el número de la casilla en la que desea situar el Parking. 2. La ventana muestra en la etiqueta situada en arriba a la izquierda el número de la casilla seleccionada. 3. El usuario elige la solapa Parking 4. El usuario pulsa el botón guardar. 5. La ventana, que conoce a la Edición que se está creando, le dice a ésta que el tipo de la casilla seleccionada es Parking 7. La Edición comprueba que aún no dispone de casilla tipo Parking 8. La Edición asigna a la <i>Casilla</i> el tipo Parking 9. La casilla actualiza su tipo en la base de datos a través del Agente
Flujo alternativo 1: <i>Ya había una casilla de Parking en la edición</i> 1. En una ventana como la mostrada en la sección de Descripción, el usuario selecciona el número de la casilla en la que desea situar el Parking. 2. La ventana muestra en la etiqueta situada en arriba a la izquierda el número de la casilla seleccionada. 3. El usuario elige la solapa Parking 4. El usuario pulsa el botón guardar. 5. La ventana, que conoce a la Edición que se está creando, le dice a ésta que el tipo de la casilla seleccionada es Parking 7. La Edición comprueba que ya dispone de casilla tipo Parking 8. La Edición elimina de la base de datos (a través del Agente) su casilla de Parking 8. La Edición asigna a la <i>Casilla</i> seleccionada en el paso 1 el tipo Parking 9. La casilla actualiza su tipo en la base de datos a través del Agente
Descripción:

Tabla 11. Descripción textual del caso de uso *Crear parking*

4. Verificación y validación

Como se comentó en la sección 8 del Capítulo 1 (página 47), es preciso verificar y validar todos los artefactos que se van construyendo durante el desarrollo. De este modo, evitamos la propagación de errores a fases posteriores: es bien sabido que el coste de reparar un defecto en el código fuente es hasta 100 veces superior al coste de repararlo en la fase de análisis.

Por tanto, deben comprobarse también las descripciones textuales de casos de uso y los propios diagramas de casos de uso.

La siguiente es una lista de comprobación para casos de uso tomada de la página web www.processimpact.com:

Cuestión	Si	No
El caso de uso representa una funcionalidad independiente y bien definida		
El objetivo del caso de uso es claro e inequívoco		
Está claro cuál es el actor o actores involucrados en la ejecución del caso del uso		
El caso de uso carece de detalles de diseño e implementación		
Todos los flujos de eventos alternativos están recogidos y especificados		
Todas las posibles condiciones de excepción están recogidas y documentadas		
No hay flujos de eventos comunes que podrían ser recogidos en casos de uso separados		
Todos los flujos están descritos de forma clara e inequívoca		
Los actores involucrados son los adecuados		
Los flujos de eventos definidos para el caso de uso son factibles		
Los flujos de eventos definidos para el caso de uso son verificables		
Las pre y las postcondiciones delimitan correctamente el caso de uso		

Tabla 12. Lista de comprobación para casos de uso

En la construcción de casos de uso y de sus diagramas, otros errores frecuentes son los siguientes:

- Existencia de casos de uso excesivamente simples
- El diagrama representan “pasos”, similares a un Diagrama de Flujo de Datos o a una máquina de estados
- Los nombres de los casos de uso no representan la funcionalidad que están describiendo
- Creación de un actor que representa el propio sistema que se está describiendo
- Poner las flechas al revés en los diagramas de casos de uso, especialmente en inclusiones y extensiones revés: deben ponerse de forma que el caso de uso del que sale la flecha incluye o extiende al caso de uso al que llega la flecha.

5. Lecturas recomendadas

Sección “Casos de uso en la fase de inicio”, en el capítulo 6 del libro de Craig Larman (2001). En su versión en Español, página 74.

Capítulo 5. APLICACIÓN DE ADMINISTRACIÓN:

FASE DE ELABORACIÓN (I)

En el capítulo anterior hemos abordado la fase de comienzo del proyecto por el que desarrollaremos la Aplicación de administración. Se han identificado y representado en un diagrama los casos de uso, y se ha dado la descripción textual de algunos de ellos. Ahora, en la fase de elaboración, y siguiendo la descripción genérica del Proceso Unificado que mostramos en la Tabla 1 (página 56), realizaremos el análisis detallado de la casi totalidad de los requisitos del sistema (dejaremos para fases posteriores sólo aquellos cuya descripción no tengamos clara, o aquellos cuya implementación tenga una fuerte dependencia existencial de otros requisitos, etc.) y diseñaremos e implementaremos una versión no desechable de la arquitectura del sistema. Igualmente, podremos construir una versión más exacta del plan del proyecto.

Por motivos didácticos, ésta será la fase a la que más espacio dediquemos, ya que se presentarán ideas y conceptos que, posteriormente, deberán volver a ser utilizados en las fases posteriores. En este primer capítulo de los que dedicamos a abordar la fase de elaboración se profundiza, con alto nivel de detalle, en la elaboración completa de tres casos de uso, realizando labores de análisis, diseño de objetos y de base de datos y codificación.

1. Desarrollo del caso de uso *Identificación*

Nos encontramos ante el desarrollo del primer caso de uso de nuestro plan de iteraciones. Realizada su descripción textual (Tabla 7, página 84), podemos pasar a describir sus flujos de eventos en términos de objetos y de mensajes pasados entre dichos objetos. Normalmente se construye un diagrama de secuencia por cada flujo de eventos, representando así los diferentes *escenarios* de ejecución del caso de uso.

Diagramas de interacción

UML posee dos tipos de diagramas que se utilizan para mostrar las interacciones entre objetos: los diagramas de secuencia y los diagramas de colaboración. Ambos tipos de diagramas muestran exactamente lo mismo, pero utilizando notaciones ligeramente distintas.

1.1. Diagrama de secuencia “de análisis” para el flujo de eventos normal

Los eventos con que se describió el flujo normal del caso de uso son los siguientes:

1. El usuario escribe el login y la contraseña en una ventana
2. La ventana crea una sesión con el login y la contraseña
3. La sesión comprueba que el usuario no esté ya identificado
4. La sesión valida sus parámetros con la base de datos a través del agente, que son correctos
5. Se añade la sesión a la lista de sesiones
6. El usuario queda identificado ante el sistema
7. La ventana de identificación muestra una ventana con el menú de opciones

Como hemos redactado la mayor parte de los eventos en términos de sujeto-verbo-predicado, resulta fácil identificar los verbos como mensajes (que luego se traducirán a operaciones de las clases correspondientes), al sujeto como emisor del mensaje y al predicado como receptor. Además, el hecho de que el emisor le diga algo al receptor denota que hay una relación de conocimiento entre las clases de ambos objetos, dato que será útil para la confección posterior del diagrama de clases.

La Figura 40 muestra el diagrama de secuencia correspondiente a este flujo de eventos. La acción comienza cuando el usuario introduce sus datos de identificación en una ventana. Cuando los ha completado, pulsará probablemente un botón para validarse, mensaje que es recibido por la ventana, que crea la sesión y realiza la validación. Como estamos describiendo el flujo correspondiente a una identificación exitosa, se muestra la ventana con el menú de opciones.

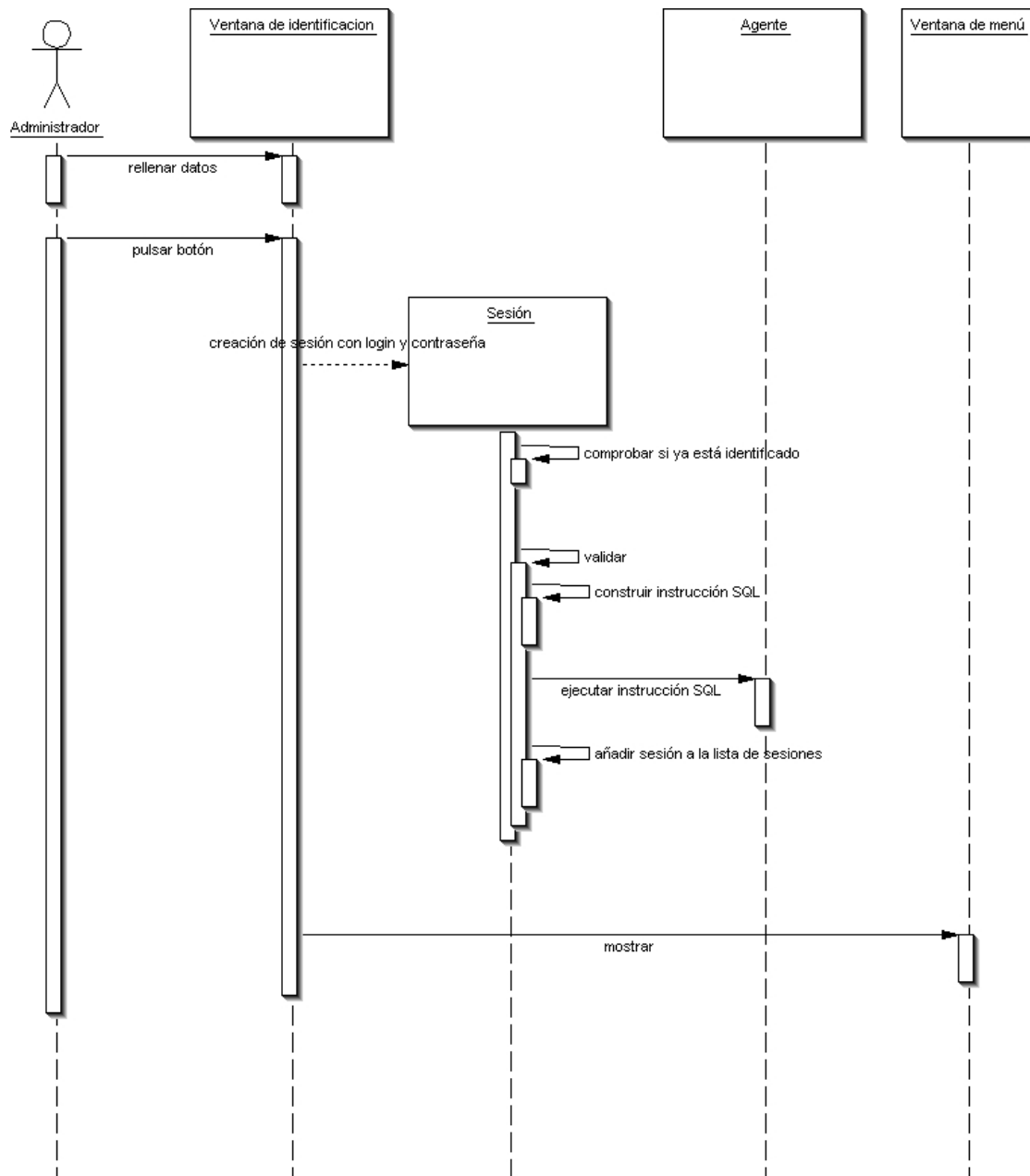


Figura 40. Diagrama de secuencia para el escenario correspondiente al flujo de eventos normal

Como se observa en la figura, en un diagrama de secuencia se muestran, ordenados temporalmente, los eventos que van ejecutándose en un cierto escenario. En este contexto, a los eventos se los llama “mensajes”, y se representan en el diagrama mediante flechas que unen el objeto emisor del evento con el objeto receptor. Si se desea, se puede especificar el fin de un mensaje usando un “foco de control”, que se apoya sobre la “línea de vida” del objeto. Los focos de control sirven para mostrar el anidamiento de los mensajes: en el diagrama, del foco de control en el que termina el mensaje “validar” (en *Sesión*) salen tres mensajes, lo que podría leerse como “Consecuencia de que se ejecute el mensaje *validar* sobre *Sesión* es que se ejecuten

los mensajes *Construir instrucción SQL*, *Ejecutar instrucción SQL* y *Añadir sesión a la lista de sesiones*. Los mensajes se corresponden con operaciones pertenecientes a la clase del objeto situado en el destino de la flecha: es decir, el mensaje “validar” se corresponde con una operación de la clase *Sesión*. Además, los mensajes que empiezan y terminan en la misma “línea de la vida” representan operaciones que ejecuta la propia instancia y que serán, probablemente, privadas o protegidas: este es el caso de los mensajes *comprobar si ya está identificado*, *validar*, *construir instrucción SQL*, etc.

1.2. Obtención de un modelo de diseño

Tanto las clases identificadas en la Figura 33 (página 83) como el diagrama de secuencia dibujado en la figura anterior son elementos de análisis: realmente, ni habrá una clase en el sistema que se llame *Ventana de identificación*, ni los mensajes pasados entre los objetos recibirán nombres como *pulsar botón* o *creación de sesión con login y contraseña*. Un nombre más adecuado para la clase sería *VentanaDeIdentificación*; el mensaje *pulsar botón* se corresponderá con la captura de un evento, y *creación de sesión con login y contraseña* se corresponderá probablemente con una llamada a la construcción de un objeto de clase *Sesión*.

El cambio de nombre de estos elementos, aunque simple, y el hecho de comenzar a traducir estos conceptos de tan alto nivel de abstracción a métodos, tipos, parámetros... ayudan a distinguir la difusa línea que, en ingeniería del software orientado a objetos, separa el análisis del diseño.

Centrándonos en el diseño de la solución, podemos identificar las siguientes clases de diseño:

1. *JFIdentificacion*. Implementará la ventana utilizada por el usuario para identificarse ante el sistema (o sea, lo que hemos llamado *Ventana de identificación*). Seguiremos el convenio de nombrar a las ventanas que implementaremos como *JFrames* de Java utilizando las letras *JF* como prefijo, seguido de un texto que denote claramente la función de la ventana.
2. *Sesión*. Implementará la funcionalidad encargada de validar las credenciales del usuario que intenta identificarse.
3. *Agente*. Implementará un agente de base de datos.

Nuestro objetivo, ahora, es construir una versión del diagrama de secuencia anterior más próxima a la realidad. Dentro del entorno de desarrollo, crearemos un nuevo diagrama de secuencia en el que iremos nombrando objetos de clases reales. Previamente, creamos dichas clases en el entorno de desarrollo que estemos utilizando.

La Figura 41 muestra un fragmento del entorno de desarrollo *Eclipse*: en el lado superior izquierdo aparece un árbol que muestra la estructura del

proyecto *administración*: el primer hijo (etiquetado *gui*) se corresponde con el nombre del paquete en el que ubicaremos la interfaz de usuario. *gui*, a su vez, tiene un hijo, llamado *JFIdentificacion*, que es el nombre de la clase visual que se está diseñando y que aparece en el lado derecho. El árbol inferior izquierdo muestra el conjunto de controles (*widgets*) que se han ido ubicando sobre la ventana.

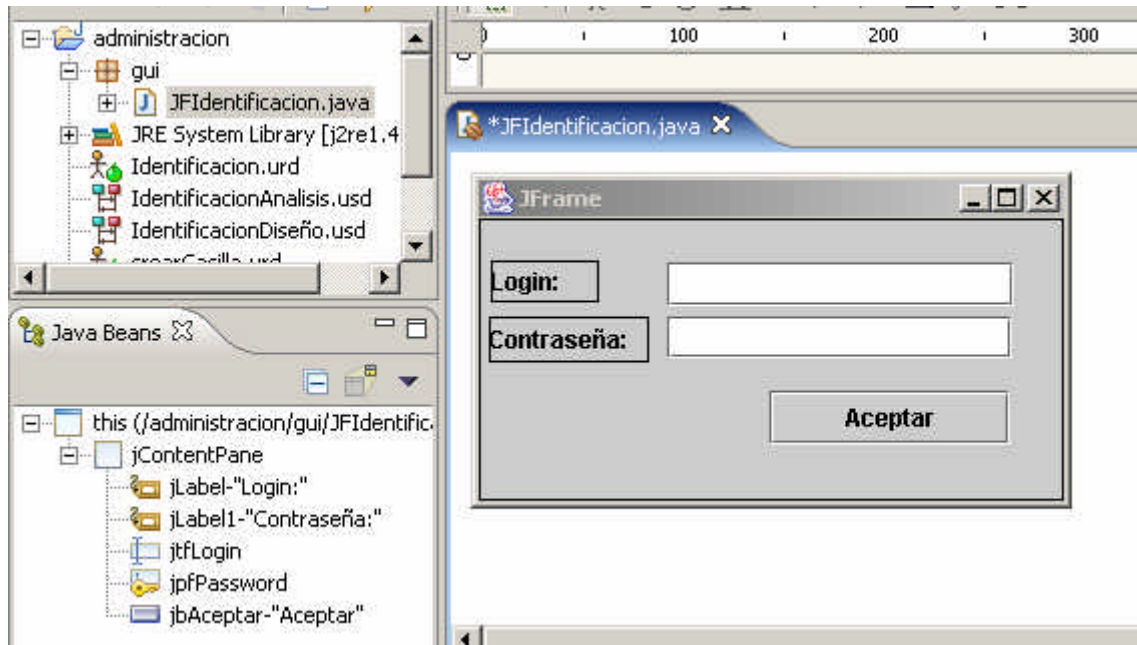


Figura 41. Diseño de la clase *JFIdentificacion* utilizando el plugin *Visual Editor* del entorno de desarrollo *Eclipse*

De los diferentes elementos de análisis vamos obteniendo elementos de diseño. En el diagrama de secuencia de la Figura 42 se han sustituido las clases de análisis anteriores por clases de diseño, y varios mensajes se han traducido a métodos.

Respecto de los objetos de tipos *Connection* y *PreparedStatement*, éstos se tratan de dos interfaces definidas en el paquete *java.sql* que se utilizan para manipular bases de datos desde aplicaciones Java. *Connection* representa un objeto que permite el acceso a la base de datos; *PreparedStatement* representa una sentencia SQL ejecutable sobre la *Connection* asociada.

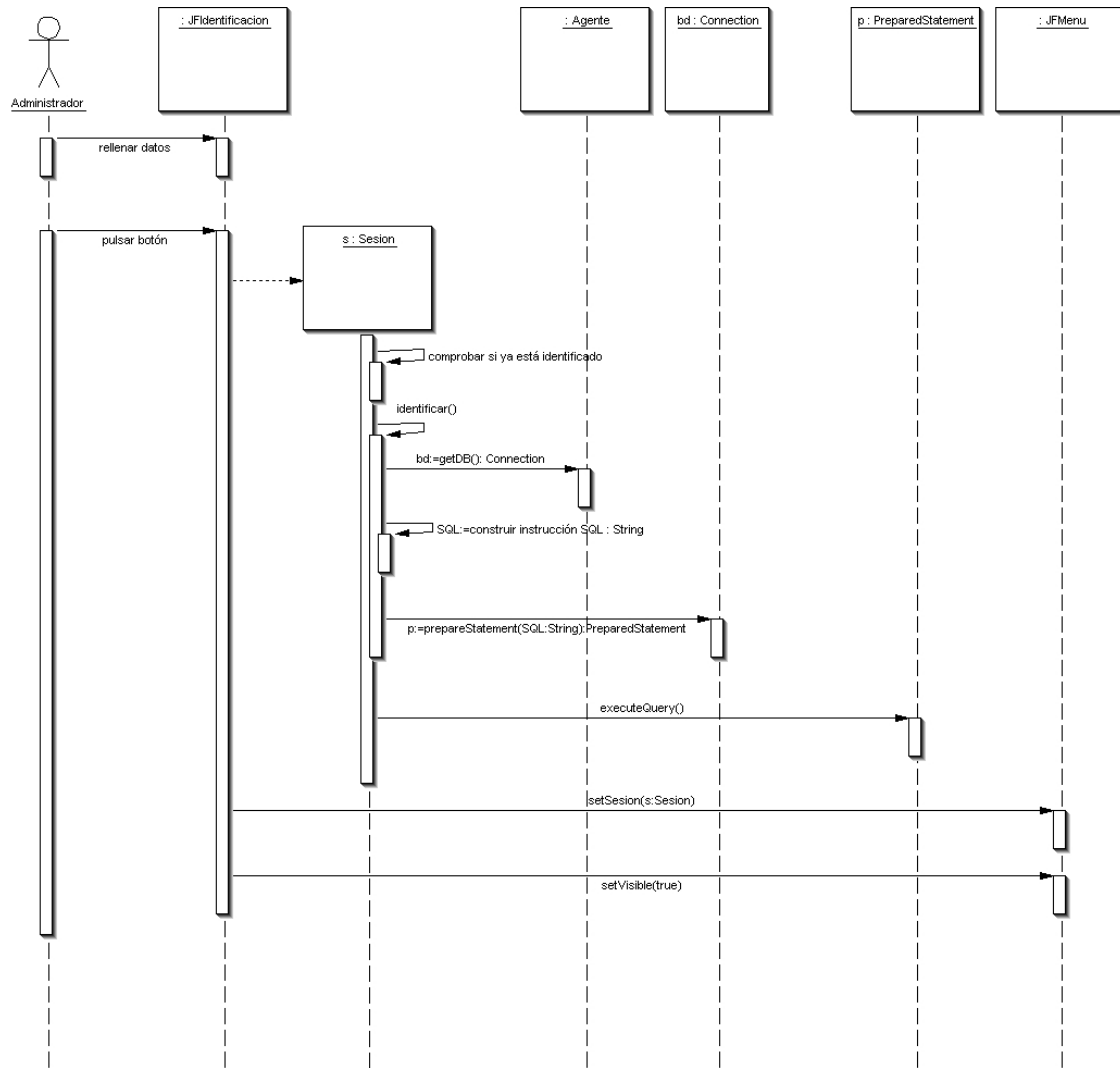


Figura 42. Diagrama de secuencia “de diseño”

1.2.1 Utilización del patrón Agente de base de datos

El Agente de base de datos es un patrón que se utiliza como adaptador entre una aplicación y el gestor de base de datos que se esté utilizando. La idea de su utilización es que todas las clases persistentes accedan a la base de datos a través del agente.

El siguiente cuadro muestra el posible código de la operación *identificar* definida en la clase *Sesión*, que accede a la tabla *Usuarios* de la base de datos a través del *Agente*: la operación recupera una instancia de tipo *Connection* (primera instrucción del bloque *try*), que a continuación utiliza para obtener un *PreparedStatement*, que puede ya cargar con la sentencia SQL embebida en la variable llamada *SQL*.

```

protected boolean identificar() throws SQLException {
    PreparedStatement p=null;
    boolean result=false;
    String SQL="Select count(*) from Usuarios where Nombre=? and Contraseña=?" +
        " and Rol=?";
    try {
        Connection bd=Agente.getAgente().getDB();
        p=bd.prepareStatement(SQL);
        p.setString(1, this.nombre);
        p.setString(2, this.password);
        p.setString(3, "Administrador");
        ResultSet rs=p.executeQuery();
        if (rs.next()) {
            result=(rs.getInt(1)==1);
        }
    }
    catch (SQLException ex) {
        throw ex;
    }
    finally {
        p.close();
    }
    return result;
}

```

Cuadro 12. Código de la operación *Sesion::identificar*

La tabla *Usuarios* tendría el siguiente diseño:

Usuarios				
	Nombre de columna	Tipo de datos	Longitud	Permitir valores nulos
?	Nombre	varchar	20	
	Contraseña	varchar	20	
	Rol	varchar	20	

Figura 43. Tabla *Usuarios* de la base de datos

Los diagramas de secuencia no deben estar muy sobrecargados: deben ser claros y legibles

Si utilizamos el *Agente*, del diagrama de secuencia que dibujábamos en la Figura 42 podríamos quitar los objetos de los tipos *Connection* y *PreparedStatement*, que podremos subsumir en el comportamiento del *Agente*. En cualquier caso, y aunque no utilizáramos este patrón, debemos tener en cuenta que gran cantidad de operaciones de dominio tendrán como consecuencia la actualización del estado de la instancia en la base de datos. Si con todas estas operaciones somos tan detallistas como para especificar en el diagrama de secuencia absolutamente todos los objetos que intervienen en la ejecución del escenario, el diagrama de secuencia dejará de cumplir uno de sus cometidos básicos (el aclaratorio), perdiendo legibilidad. En ocasiones, por tanto, puede ser conveniente dejar de representar mensajes y objetos en determinados diagramas de secuencia.

Aunque no lo hemos considerado en los diagramas de secuencia anteriores, podemos añadir más mensajes al diagrama para denotar que, una vez el usuario se ha identificado, el sistema mostrará una ventana con el menú principal.

Con todas estas consideraciones, el diagrama de secuencia podría quedar como en la Figura 44. Obsérvese que los objetos de tipos *JFIdentificacion* y *JFMenu* son anónimos, pero que el objeto de clase *Sesion* se llama *s*: esto se hace para representar en el diagrama que el parámetro que se pasa en el mensaje *setSesion* desde *JFIdentificacion* hasta *JFMenu* es el mismo objeto que ha creado *JFIdentificacion*.

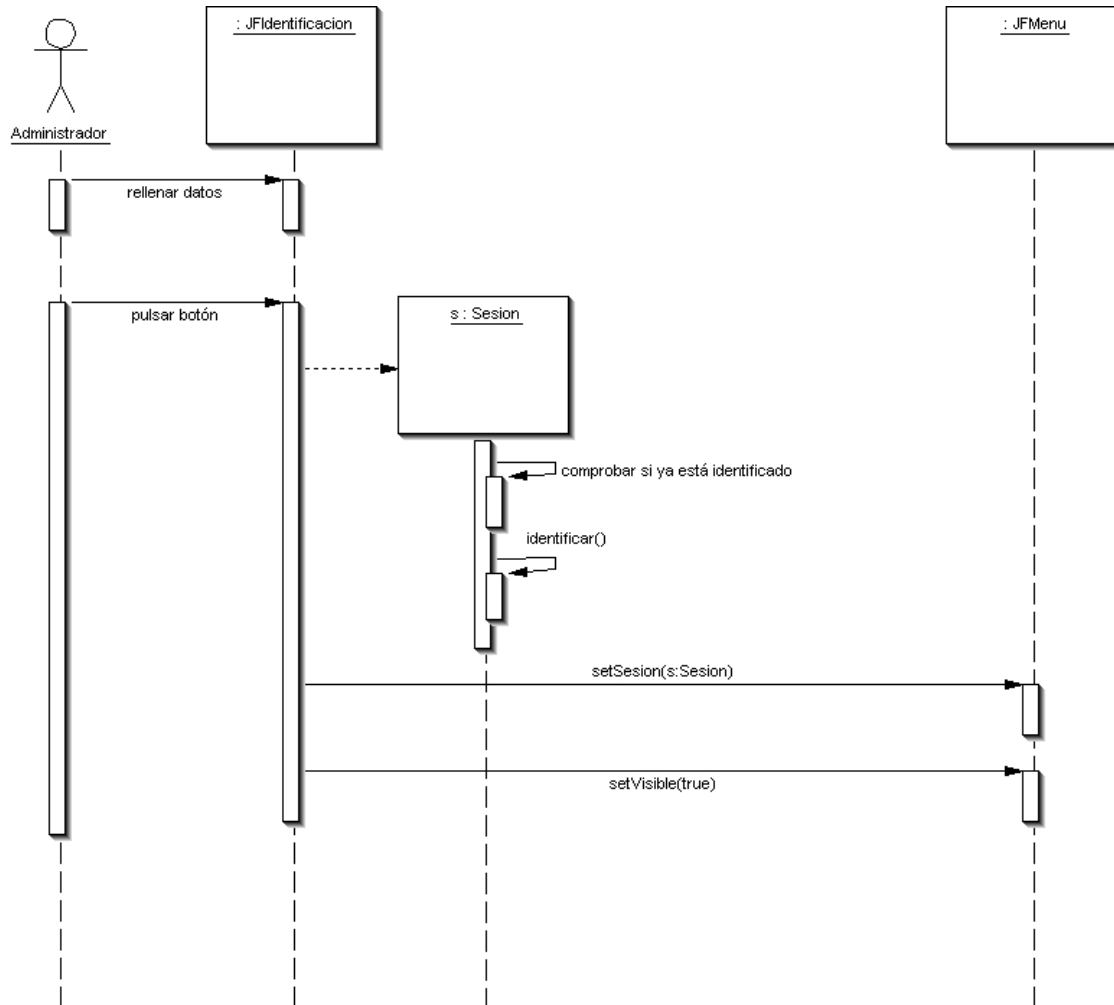


Figura 44. Diagrama de secuencia modificado

En esta aplicación, el *Agente* se ha implementado como un *Proxy* que, además, es un *Singleton*. El *Proxy* es un patrón que se utiliza para representar, en el sistema orientado a objetos, un dispositivo externo (la base de datos, en este caso); el *Singleton* se utiliza cuando se desea disponer de una clase de la que existirá, a lo sumo, una sola instancia. En la Sección 8 se discute la implementación dada al *Agente* como un *Singleton*.

1.2.2 Diagramas de colaboración

Los diagramas de colaboración representan exactamente lo mismo que los diagramas de secuencia, sólo que desde otro punto de vista: si éstos representan las secuencias de mensajes con una clara ordenación temporal,

aquéllos resaltan los mensajes que se pasan entre objetos, lo que puede ayudar a detectar más fácilmente los objetos que tienen un número excesivo de responsabilidades.

De cada diagrama de secuencia se puede obtener un diagrama de colaboración, y viceversa. A ambos se los conoce como diagramas de interacción. De hecho, muchas herramientas CASE obtienen el uno a partir del otro, y el otro a partir del uno, como en la figura siguiente:

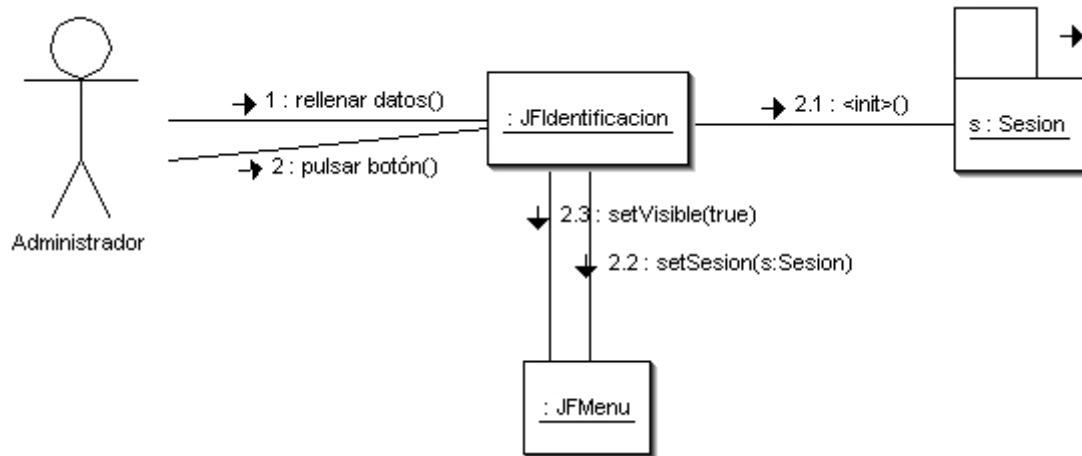


Figura 45. Diagrama de colaboración procedente de la transformación automática del diagrama de secuencia de la Figura 44

En los diagramas de colaboración, la ordenación temporal viene representada por los números de cada mensaje. Los focos de control, que en los diagramas de secuencia representaban el anidamiento de mensajes, se sustituyen aquí por la numeración anidada y ordenada de los mensajes.

1.2.3 Diagramas de de secuencia para flujos de eventos alternativos

En la descripción textual de este caso de uso, se indicaba en los flujos de eventos alternativos que se muestra un mensaje de error si el usuario ya estaba identificado o si las credenciales son incorrectas

La Figura 46 muestra estos dos escenarios. Nótese que *Sesión* devuelve una excepción a *JFIdentificacion*. Las excepciones se representan en UML como señales asíncronas, denotadas con la punta de flecha ligeramente distinta. Al capturar la excepción, *JFIdentificacion* muestra un mensaje de error en una ventana, cuyo mensaje de construcción se ha obviado.

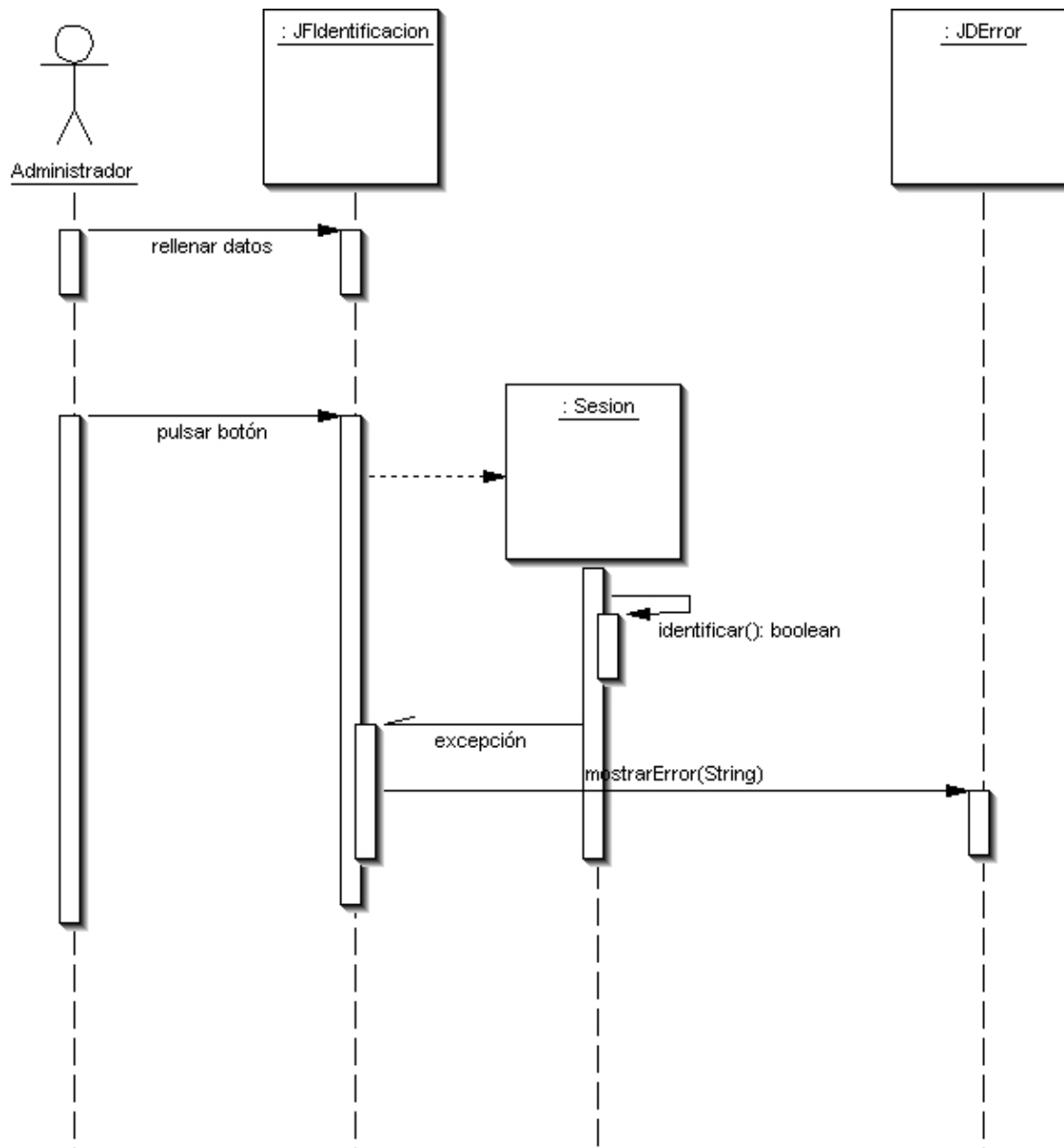


Figura 46. Diagrama de secuencia correspondiente al escenario de error

1.2.4 Escritura de código

Así como los flujos de eventos de las descripciones textuales de casos de uso son la base para la construcción de los diagramas de secuencia, éstos lo son para la escritura del código, en donde ya representamos los pasos de mensajes que hemos dibujado.

El lado izquierdo de la Figura 47 muestra el código de la operación *identificar* incluido en *JFIdentificacion*: como se observa, consta de un bloque *try* con tres *catch*: el *try* incluye la construcción de un objeto de clase *Sesion*, que puede lanzar (véase lado derecho) tres tipos de excepciones: una si el usuario ya estaba identificado (*SesionYaAbiertaException*), otra por combinación incorrecta de nombre y contraseña (la excepción *PasswordInvalidaException*) y otra por error de acceso a la base de datos (*SQLException*).

tion), que a su vez es lanzada por la operación *identificar* (cuyo código ya se ofreció en el Cuadro 12). El código de captura y tratamiento de las excepciones es exactamente el mismo en los tres casos: puesto que las tres excepciones mostradas (dos de ellas creadas ex profeso para este proyecto) son especializaciones de la clase *java.lang.Exception*, se podría haber escrito un único bloque *catch* que capturase una excepción genérica; sin embargo, en ocasiones es conveniente capturar cada excepción de manera distinguida, para realizar un tratamiento particularizado del error.

Si no se lanza excepción, se siguen de forma más o menos fiel los pasos indicados en el diagrama de secuencia: es decir, se crea una instancia de clase *JFMenu*, se le asigna a esta instancia la sesión *s* que acaba de crearse, se oculta la ventana de identificación (mensaje que no estaba representado en el diagrama) y se pone visible la ventana con el menú. Si se captura alguna excepción, se muestra el mensaje de error en un diálogo llamado *JDError*. Como puede observarse, los diagramas de secuencia no son una representación exacta de todos los pasos de mensajes, sino que sólo muestran aquellos que son realmente relevantes para entender el escenario de ejecución.

<pre>// En JFIdentificacion protected void identificar() { try { Session s= new Session(this.jtfLogin.getText(), this.jpfpPassword.getText()); JFMenu j=new JFMenu(); j.setSession(s); this.setVisible(false); j.setVisible(true); } catch (SQLException e) { JDError j=new JDError(); j.mostrarError(e.toString()); j.setModal(true); j.setVisible(true); } catch (PasswordInvalidaException e) { JDError j=new JDError(); j.mostrarError(e.toString()); j.setModal(true); j.setVisible(true); } catch (SesionYaAbiertaException e) { JDError j=new JDError(); j.mostrarError(e.toString()); j.setModal(true); j.setVisible(true); } }</pre>	<pre>package dominio; import java.sql.Connection; import java.sql.PreparedStatement; import java.sql.ResultSet; import java.sql.SQLException; import java.util.Hashtable; import dominio.exceptions. PasswordInvalidaException; import dominio.exceptions. SesionYaAbiertaException; import persistencia.Agente; public class Sesion { private String nombre; private String password; private static Hashtable sesiones= new Hashtable(); public Sesion(String nombre, String password) throws SQLException, PasswordInvalidaException, SesionYaAbiertaException { if (sesiones.get(nombre)!=null) throw new SesionYaAbiertaException(nombre); this.nombre=nombre; this.password=password; if (!identificar()) throw new PasswordInvalidaException(); sesiones.put(nombre, this); } public void cerrarSesion() { if (sesiones.get(this.nombre)!=null) sesiones.remove(this.nombre); } protected boolean identificar() throws SQLException { ... } }</pre>
---	---

Figura 47. Parte del código procedente del diagrama de secuencia

1.2.5 Obtención de diagramas de clase a partir de diagramas de secuencia

Cuando un objeto le dice algo a otro (es decir, cuando le envía un mensaje) es porque, de alguna manera, lo conoce. El conocimiento de un objeto respecto de otro puede estar materializada en la forma de una relación de asociación, dependencia o agregación (recuérdese que la composición es un tipo de agregación “fuerte”). Ya que un diagrama de secuencia muestra paso de mensajes entre objetos, puede servirnos para deducir relaciones entre sus respectivas clases.

Debemos tener un cierto nivel de experiencia para conocer si el mensaje que se pasa de una instancia a otra se corresponde con una relación de asociación, de agregación o de dependencia, aunque de todos modos la naturaleza iterativa del proceso nos permite la introducción progresiva de ciertas modificaciones.

Para este caso de uso, hay una relación de dependencia entre *JFIdentificacion* y *Sesión*, ya que la ventana conoce a la instancia de la clase de dominio únicamente durante la ejecución de su operación *identificar*. *JFIdentificacion* conoce también a *JFMenu* y a *JError* sólo durante un momento, por lo que se trata también de relaciones de dependencia. La *Sesión*, además, conoce al *Agente*, ya que lo utiliza para acceder a la base de datos, y posee también una tabla hash con las sesiones que se van creando. El diagrama de clases procedente del diagrama de secuencia anterior, por tanto, puede ser el siguiente:

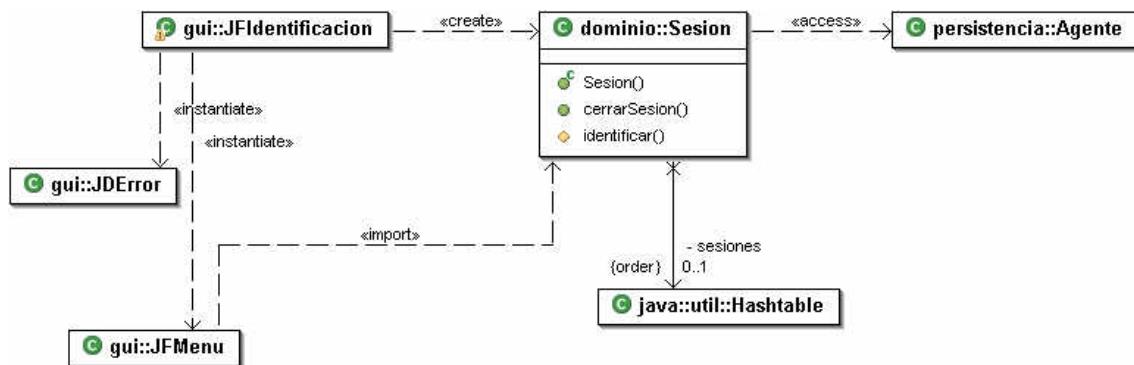


Figura 48. Diagrama de clases procedente de los diagramas de secuencia anteriores

1.2.6 Verificación y validación de diagramas de secuencia

Antes de dar por bueno un diagrama de secuencia, es preciso someterlo a una revisión para comprobar que refleja correctamente el escenario deseado. Podemos utilizar la siguiente lista de comprobación, que nos servirá para repasar las características fundamentales del diagrama (Tabla 13).

Elemento de comprobación	Superado (Sí/No)
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 o, si los hay, su presencia es justificada)	
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 tienen 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	

Tabla 13. Lista de comprobación para diagramas de secuencia

1.3. Obtención de casos de prueba a partir de un flujo de eventos o diagrama de secuencia

Tanto los flujos de eventos que encontramos tanto en las descripciones textuales de casos de uso como en los diagramas de secuencia que describen escenarios del sistema pueden (y deben) utilizarse para ir definiendo casos de prueba que, más adelante, puedan utilizarse para comprobar el funcionamiento de la aplicación.

De manera general, un caso de prueba consistirá en un conjunto de mensajes que se envían a uno o más objetos y una descripción del resultado esperado. Tras la ejecución del caso de prueba, se compara el resultado esperado con el resultado obtenido, considerando que la aplicación ha superado el caso de prueba si el resultado obtenido es igual al resultado esperado.

A partir del último diagrama de secuencia (Figura 44), podemos definir casos de prueba utilizando algún tipo de plantilla, como la siguiente, en la que el último apartado se cumplimentará cuando el caso de prueba sea ejecutado.

Identificador del caso de prueba: 1
Caso de uso/Flujo de eventos: Identificación/Flujo normal
Condiciones de ejecución: El usuario no se encuentra identificado La base de datos se encuentra accesible y operativa
Descripción: El usuario escribe el login y la contraseña de un administrador existente en la ventana y pulsa el botón
Resultado esperado: El usuario queda identificado Se muestra la ventana con el menú principal
Resultado obtenido:

Tabla 14. Caso de prueba 1, “en positivo”

El caso de prueba anterior probará la funcionalidad “en positivo”; sin embargo, es preciso probar también que la aplicación se comporta bien en situaciones de error: en nuestro caso de uso, debemos escribir casos de prueba

para probar que la aplicación se comporta correctamente cuando un usuario intenta identificarse con una combinación errónea de nombre y contraseña (Tabla 15), y cuando un usuario intenta identificarse por segunda vez (Tabla 16).

Identificador del caso de prueba: 2
Caso de uso/Flujo de eventos: Identificación/Flujo alternativo 1
Condiciones de ejecución: El usuario no está dado de alta en el sistema (no autorizado) La base de datos se encuentra accesible y operativa
Descripción: El usuario escribe el login y la contraseña de un usuario existente en la ventana y pulsa el botón
Resultado esperado: Se muestra una ventana con un mensaje de error.
Resultado obtenido:

Tabla 15. Caso de prueba 2, “en negativo”

Identificador del caso de prueba: 3
Caso de uso/Flujo de eventos: Identificación/Flujo alternativo 1
Condiciones de ejecución: El usuario ya se encuentra identificado La base de datos se encuentra accesible y operativa
Descripción: El usuario escribe el login de un usuario existente en la ventana y pulsa el botón
Resultado esperado: Se muestra una ventana con un mensaje de error indicando que el usuario ya está identificado.
Resultado obtenido:

Tabla 16. Caso de prueba 3, “en negativo”

2. Desarrollo del caso de uso *Crear edición*

El análisis de este caso de uso es muy similar al anterior. El siguiente diagrama de secuencia describe el flujo de eventos normal que identificamos en la Tabla 8.

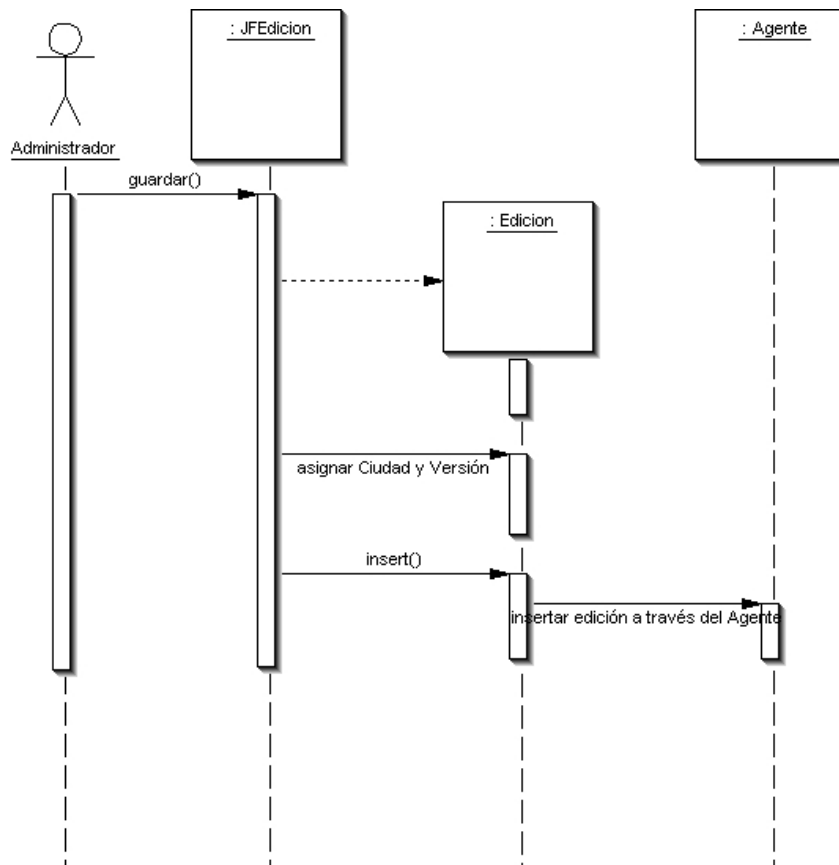


Figura 49. Escenario normal del caso de uso *Crear edición*

Este escenario supone la creación de una nueva clase (*JFEdition*) en el paquete *gui* de clases de la capa de presentación (la Figura 35, en la página 85, muestra el boceto de la ventana que el cliente nos había entregado). En esta ocasión, podemos asumir que *JFEdition* conocerá permanentemente a un objeto de tipo *Edicion* definido en la capa de dominio, por lo que la relación entre ambas clases será representada como una asociación. La instancia de *JFEdition*, además, se hará visible desde la ventana *JFMenu*, por lo que habrá algún tipo de relación desde *JFMenu* hacia *JFEdition*: como ésta es una relación temporal, la representamos con una dependencia.

La nueva versión del diagrama de clases se muestra en la Figura 50.

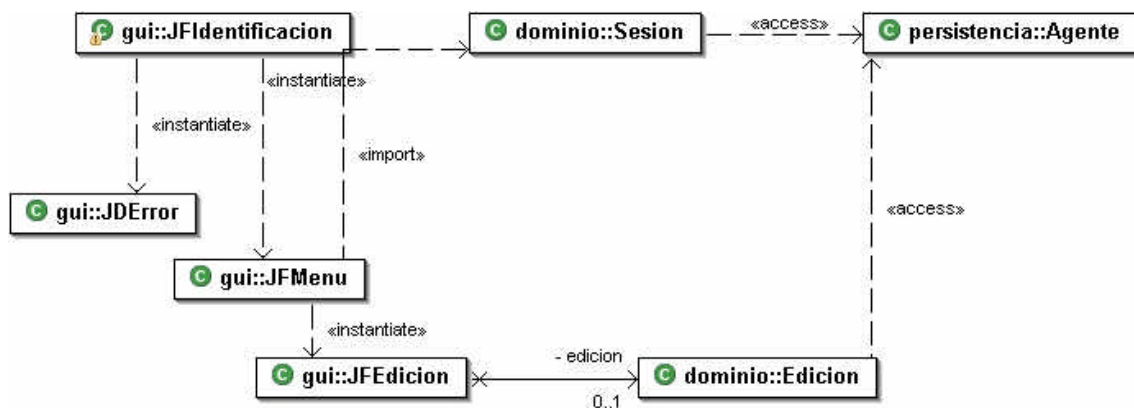


Figura 50. Diagrama de clases con las nuevas adiciones

Este caso de uso supone la inserción de un registro en la tabla *Edicion* de la base de datos. Esto supone la adición de la operación *insert* a la clase *Edicion* (Cuadro 13).

```
// En la clase Edicion
public void insert() throws SQLException {
    PreparedStatement p=null;
    String SQL="Insert into Edicion (Ciudad, Version) values (?, ?)";
    try {
        Connection bd=Agente.getAgente().getDB();
        p=bd.prepareStatement(SQL);
        p.setString(1, this.ciudad);
        p.setString(2, this.version);
        p.executeUpdate();
    }
    catch (SQLException ex) {
        throw ex;
    }
    finally {
        p.close();
    }
}
```

Cuadro 13. Implementación de *insert* en *Edicion*

Como se muestra en el diagrama de secuencia anterior, la primera operación que se ejecuta al guardar la edición es crear una instancia de clase *Edicion* (mediante el mensaje de creación, indicado en la figura con una línea punteada desde *JFEdicion* hasta *Edicion*). El código de este constructor, muy simple, puede ser el siguiente:

```
public Edicion() {
}
```

Cuadro 14. Constructor de la clase *Edicion*

3. Desarrollo del caso de uso *Crear casillas vacías* (primera parte)

Este caso de uso es ejecutado sólo desde el caso de uso *Crear edición*. Asignaremos la responsabilidad de crear las casillas a la propia clase *Edición*, mediante la adición del siguiente método:

```
private void insertarCasillasVacias() throws SQLException {
    casillas=new Vector();
    for (int i=0; i<40; i++) {
        Casilla c=new Casilla(i+1);
        c.setEdicion(this);
        c.insert();
        casillas.add(c);
    }
}
```

Cuadro 15. Método para crear las casillas vacías, en la clase *Edicion*

Evidentemente, el código anterior supone la creación de algún tipo de colección en la clase *Edicion*, que represente que cada *Edicion* está compuesta de muchas casillas. Como, además, la existencia de las casillas no tiene sentido si no están contenidas en una edición, representaremos la relación como una agregación. En el método anterior, proporcionamos a cada casilla

información de la edición en la que está contenida (instrucción *c.setEdicion(this)*), por lo que hay que representar la relación entre *Casilla* y *Edicion*. Como se ve en la Figura 51, el diagrama de clases se va enriqueciendo progresivamente.

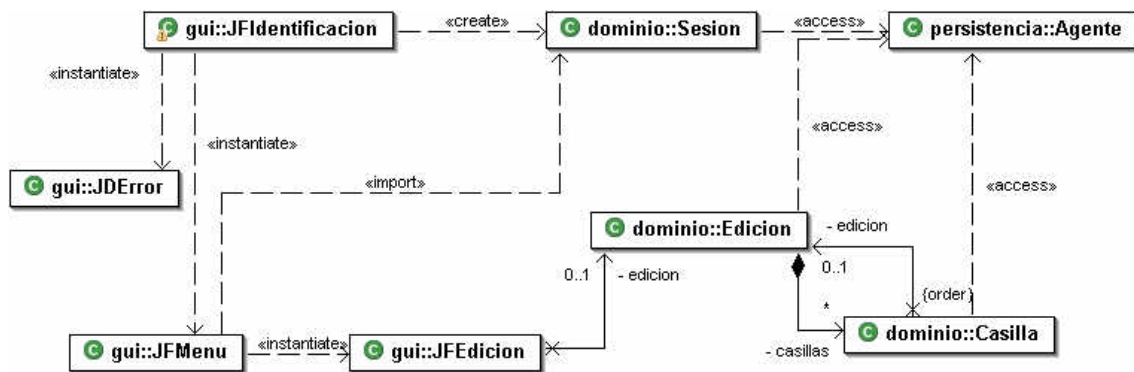


Figura 51. Nueva versión del diagrama de clases.

Por último, quedaría representar en el código la relación entre el caso de uso *Crear edición* y *Crear casillas vacías*. Para ello, debemos introducir en algún lugar una llamada a la operación *insertarCasillasVacías* (mostrada en el Cuadro 15). Como la inserción de la edición supone la inserción de las casillas, podemos modificar el código del método *insert* de *Edicion* y dejarlo como en el

```

public void insert() throws SQLException {
    PreparedStatement p=null;
    String SQL="Insert into Edicion (Ciudad, Version) values (?, ?)";
    try {
        Connection bd=Agente.getAgente().getDB();
        p=bd.prepareStatement(SQL);
        p.setString(1, this.ciudad);
        p.setString(2, this.version);
        p.executeUpdate();
        insertarCasillasVacias();
    }
    catch (SQLException ex) {
        throw ex;
    }
    finally {
        p.close();
    }
}

```

Cuadro 16. Versión modificada de *insert* (en *Edicion*)

4. Creación de la base de datos

En las instrucciones de persistencia que hemos mostrado se ha hecho referencia a nombres de tablas y de columnas de la base de datos que, sin embargo, no se han construido todavía.

Existen muchas formas de manejar las relaciones entre el mundo de clases y el mundo de las tablas, de manera que se mantenga la correspondencia entre instancias y registros. Como el lector ha podido ir observando, es bueno mantener ciertas costumbres, estándares, formas de codificación, etc., durante el desarrollo de un proyecto software; la creación de la base de

datos no es muy diferente: de hecho, un diagrama de clases posee, al menos, el mismo poder expresivo que un diagrama entidad-interrelación, por lo que los mismos mecanismos de transformación de éstos hacia bases de datos relacionales son igualmente válidos para la transformación de aquéllos. A continuación se presentan tres patrones de transformación de diagrama de clases a diagrama relacional; después, esta teoría se aplica a nuestro caso de estudio.

4.1. Patrón *una clase, una tabla*

Mediante este patrón, se construye una tabla por cada clase en el diagrama de clases, las asociaciones, agregaciones y composiciones se transforman a relaciones de clave externa con la multiplicidad indicada por la propia multiplicidad de la asociación, y las relaciones de herencia se transforman a relaciones de clave externa con multiplicidad 1:1.

Supongamos que disponemos del siguiente diagrama de clases, que puede representar el dominio de una biblioteca universitaria, en la que hay diversos tipos de socios y de publicaciones, que pueden prestarse:

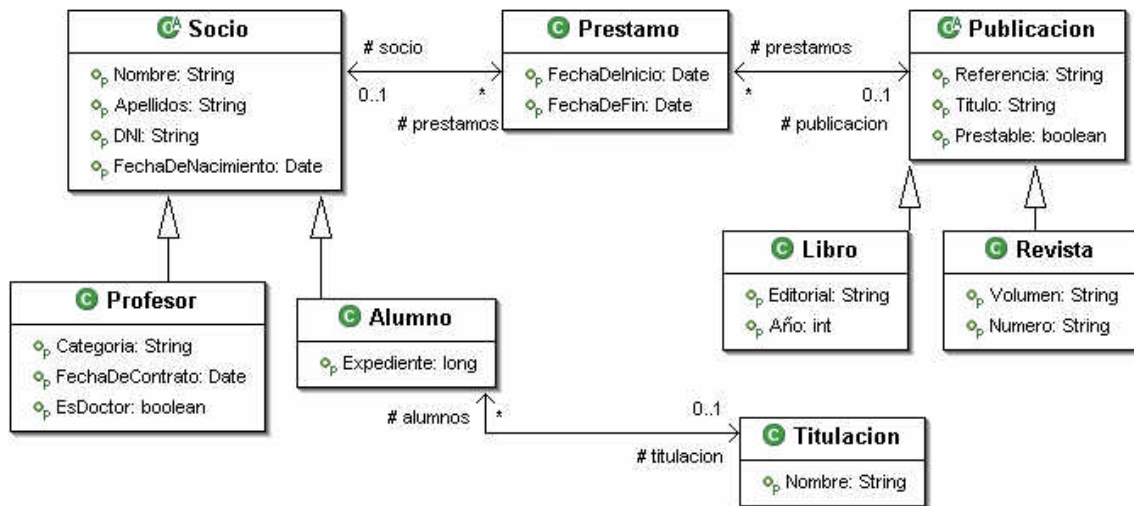


Figura 52. Diagrama de clases de una biblioteca

Mediante el patrón una clase, una tabla, construiremos las tablas correspondientes a *Socio*, *Préstamo*, *Publicación*, *Libro*, *Revista*, *Profesor*, *Alumno* y *Titulación*. En cada tabla crearemos una columna, recomendable homónima, de un tipo que, en el gestor de base de datos que estemos utilizando, sea compatible con el tipo de dato del correspondiente campo en el lenguaje de programación seleccionado. Además, crearemos en las tablas las columnas correspondientes a las claves externas, que apuntarán a las correspondientes columnas que compongan la clave primaria; de este modo, representaremos las distintas relaciones entre clases. El posible diagrama resultante es el siguiente:

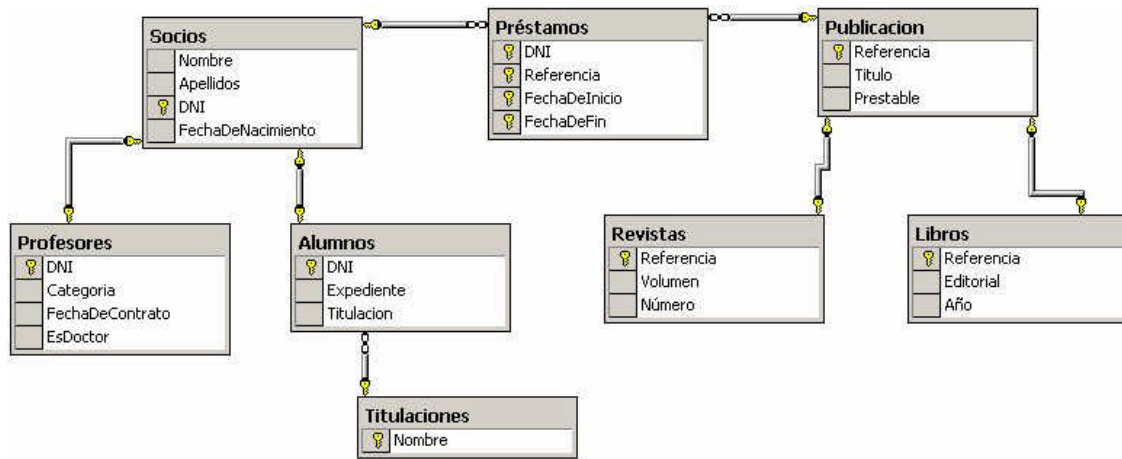




Figura 53. Diagrama resultante de aplicar el patrón una clase, una tabla al diagrama de clases de la Figura 52

Como se observa, para representar las relaciones de herencia, se propaga a las especializaciones la clave principal de la superclase, y se establece una relación de clave externa entre ambas (naturalmente, la clave principal de la subclase apunta a la de la superclase). Las asociaciones se transforman a relaciones de clave externa de la multiplicidad que corresponda (el gestor utilizado para dibujar la figura anterior utiliza el símbolo  para representar cardinalidad 1, y  para cardinalidad mayor que uno). Puede haber situaciones singulares en las que se decida modelar de alguna forma particular algún tipo relación: en nuestro caso, aunque en el diagrama de clases hay una relación bidireccional entre *Alumno* y *Titulación*, se ha decidido por mantener únicamente la clave externa en *Alumno* que almacene la titulación en la que se encuentra matriculado cada alumno. En nuestro caso, la aplicación estricta del patrón nos habría llevado a construir una tabla intermedia, como se muestra en esta figura:

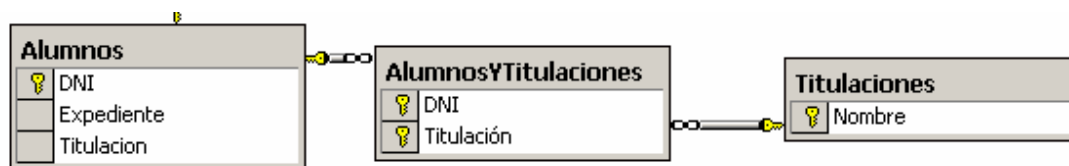


Figura 54. La aplicación estricta del patrón a veces no es recomendable

Del mismo modo, a veces es conveniente crear columnas nuevas, tablas nuevas, etc., como podría ocurrir en el caso de la tabla *Préstamos*: cada vez que insertemos un registro en esta tabla, el gestor de base de datos debe comprobar que la combinación de los valores en las cuatro columnas no viola la unicidad de la clave principal; podríamos eliminar esa clave principal y añadir una nueva columna, más sencilla, como se muestra a continuación, o simplemente no añadir ninguna:

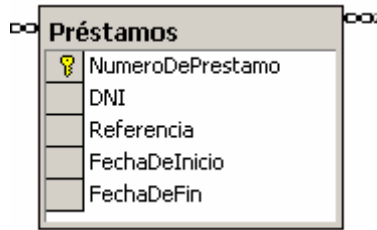


Figura 55. Adición de una columna que no procede de ningún campo

Evidentemente, el diseño que se dé a la base de datos tiene una influencia decisiva en el código SQL que se incrusta en el programa para manipular la base de datos. El Cuadro 17 muestra algunos fragmentos de la implementación de las clases *Socio* y *Alumno*. Como se observa, la primera instrucción del constructor materializador de *Alumno* llama al constructor materializador de la clase de la que hereda. Igualmente, antes de insertar un alumno, la integridad referencial entre las dos tablas nos obliga a insertar primero en *Socio*, lo que se representa con la instrucción `super.insert()`. En este caso pueden surgir problemas adicionales: ¿qué ocurre, por ejemplo, si funciona la operación `super.insert()` pero falla el resto del método? ¿Debe en este caso borrarse el registro que se haya insertado en *Socio*? Una forma de solucionarlo pasa por programar el método para que se capture la posible excepción que se lance y deshacer entonces toda la operación; otra posibilidad es ejecutar ambas operaciones dentro de una transacción, funcionalidad que debe ser soportada por el gestor de base de datos seleccionado y que, por otro lado, puede tener algunos efectos colaterales indeseables, como el bloqueo de la tabla.

```

public abstract class Socio {
    protected String mNombre, mApellidos, mDNI;
    protected Date mFechaDeNacimiento;

    ...
    public Socio(String dni) throws Exception {
        String SQL="Select * from Socios where DNI=?";
        ...
    }

    public void insert() throws Exception {
        String SQL="Insert into Socios (Nombre, Apellidos," +
            "DNI, FechaDeNacimiento) values (?, ?, ?, ?)";
        ...
    }
}

public Alumno extends Socio {
    protected String mExpediente;
    protected long mTitulacion;

    public Alumno(String dni) throws Exception {
        super(dni);
        String SQL="Select Expediente, Titulacion from Alumnos " +
            "where DNI=?";
        ...
    }

    public void insert() throws Exception {
        super.insert();
        String SQL="Insert into Alumnos (DNI, Expediente, " +
            "Titulacion values(?, ?, ?)";
        ...
    }
}

```

Cuadro 17. Fragmento del código de las clases *Socio* y *Alumno* para la Figura 53

4.2. Patrón *un árbol de herencia, una tabla*

Con este patrón, se construye una tabla por cada árbol de herencia que exista en el diagrama de clases. Cada tabla procedente de un árbol de herencia incluye una columna por cada campo en la jerarquía representada. Puesto que en la tabla se van a almacenar instancias de diferentes subtipos, habrá columnas que permanezcan desaprovechadas: así, cuando en la Figura 52 se guarde información de un *Profesor* en la tabla *Socios*, quedarán con valor nulo las columnas *Titulación* y *Expediente*. Además, la fila no contiene información sobre el subtipo del registro, salvo que se cree una columna adicional con este fin.



Figura 56. Diagrama resultante de aplicar el patrón una árbol de herencia, una tabla al diagrama de clases de la Figura 52

A continuación se muestra el fragmento de código correspondiente a las dos mismas clases que se mostraban en el epígrafe anterior, *Socio* y *Alumno*. En este caso, se ha optado por dejar vacías las implementaciones del constructor de *Socio* y de su método *insert*, aunque en la especialización se mantienen las llamadas a *super* por si en un futuro se asignan responsabilidades a estas operaciones en la superclase. Como se ve, la subclase manipula directamente en la tabla *Socios*, aunque sólo utiliza aquellas cadenas que le son de utilidad.

```
public abstract class Socio {
    protected String mNombre, mApellidos, mDNI;
    protected Date mFechaDeNacimiento;

    ...
    public Socio(String dni) throws Exception {
    }

    public void insert() throws Exception {
    }
}

public Alumno extends Socio {
    protected String mExpediente;
    protected long mTitulacion;

    public Alumno(String dni) throws Exception {
        super();
        String SQL="Select Nombre, Apellidos, DNI, " +
            "FechaDeNacimiento, Expediente, Titulacion from " +
            "Socios where DNI=?";
        ...
    }

    public void insert() throws Exception {
        super.insert();
        String SQL="Insert into Socios (Nombre, Apellidos, " +
            "DNI, FechaDeNacimiento, Expediente, " +
            "Titulacion values(?, ?, ?, ?, ?)";
        ...
    }
}
```

Cuadro 18. Fragmento del código de las clases *Socio* y *Alumno* para la Figura 56

4.3. Patrón un camino de herencia, una tabla

Como su nombre indica, con este patrón se construye una tabla por cada rama del árbol de herencia, aunque la casuística puede ser muy variada, en función fundamentalmente de la presencia o ausencia de clases abstractas. En nuestro ejemplo, no existirán instancias de las clases *Socio* y *Publicación* puesto que son abstractas, lo que hace innecesario crear una tabla especial para almacenarlas. La siguiente figura muestra una primera versión del diagrama resultante, en la que la tabla *Préstamos* se utiliza como punto de encuentro de los distintos tipos de *Socios* con los distintos tipos de *Publicaciones*.

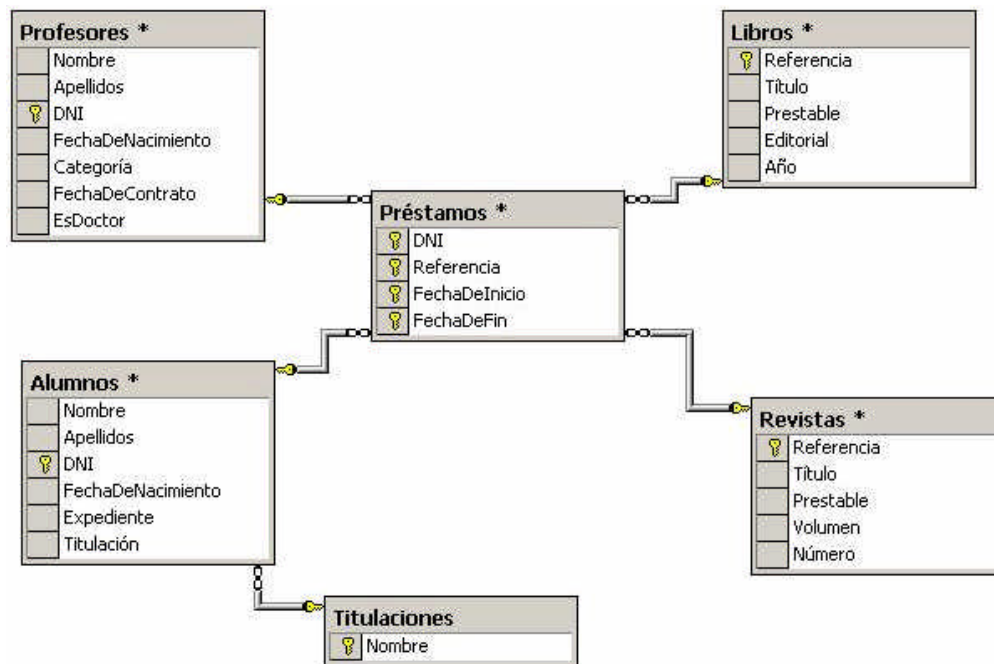


Figura 57. Diagrama resultante (versión 1) de aplicar el patrón un árbol de herencia, una tabla

Un error importante del diseño anterior es que, por ejemplo, la columna *DNI* de *Préstamos* tiene una doble integridad referencial con *Profesores* y con *Alumnos*, lo que obliga a que cada valor de la columna *DNI* que se inserte en esta tabla exista tanto en *Profesores* como en *Alumnos*. Una solución mejor pasa por la creación de dos tablas distintas, una para cada tipo de préstamo. En la siguiente figura se ha creado una tabla específica para los préstamos a profesores, y habría que crear otra con la misma estructura para los de alumnos. El problema que teníamos, sin embargo, se propaga también a esta solución, ya que la columna *Referencia* de *Préstamos* apunta a *Referencia* de *Libros* y de *Revistas*, lo que obliga a que cada referencia que se inserte en *Préstamos* exista tanto en *Libros* como en *Revistas*. Para solventar el problema, sería preciso construir varias tablas nuevas, tipo *PréstamosDeRevistasAProfesores*, *PréstamosDeLibrosAProfesores*, etc.

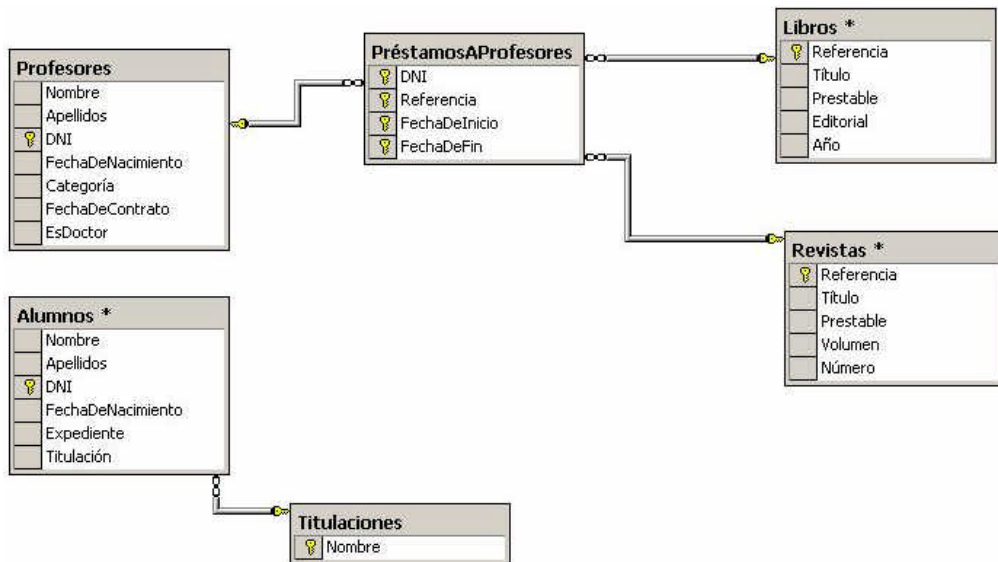


Figura 58. Separación de los distintos préstamos

Para la primera versión del diagrama de base de datos, el posible código de las clases *Socio* y *Alumno* podría tener la forma que se muestra en el Cuadro 19, en donde cada clase manipula su tabla homónima.

```

public abstract class Socio {
    protected String mNombre, mApellidos, mDNI;
    protected Date mFechaDeNacimiento;

    ...

    public Socio(String dni) throws Exception {
    }

    public void insert() throws Exception {
    }
}

public Alumno extends Socio {
    protected String mExpediente;
    protected long mTitulacion;

    public Alumno(String dni) throws Exception {
        super();
        String SQL="Select Nombre, Apellidos, DNI, " +
            "FechaDeNacimiento, Expediente, Titulacion from " +
            "Alumnos where DNI=?";
        ...
    }

    public void insert() throws Exception {
        super.insert();
        String SQL="Insert into Alumnos (Nombre, Apellidos, " +
            "DNI, FechaDeNacimiento, Expediente, " +
            "Titulacion values(?, ?, ?, ?, ?)";
        ...
    }
}
    
```

Cuadro 19. Fragmento del código de las clases *Socio* y *Alumno* para la Figura 57

4.4. Otras influencias del patrón seleccionado en el código de utilización de la base de datos

Además de en la forma de las instrucciones SQL incrustadas en el código fuente, ése se ve influido de otras formas por el diseño físico de la base

de datos. Así, por ejemplo, puede ocurrir que se decida obligar a que todo *Libro* tenga valor en su campo *Editorial*. En la base de datos, basta con restringir el valor de esa columna con la cláusula *not null*. Si se establece esta restricción, en el diagrama procedente del patrón una clase, una tabla (Figura 53), el propio gestor de base de datos lanzará al programa una excepción si se intenta guardar una instancia de *Libro* cuya *Editorial* es nula; en este caso, basta con que el programa capture y procese la excepción como convenga. Sin embargo, en el diagrama de la Figura 56, correspondiente al patrón un árbol de herencia, una tabla, no podrá establecerse esa restricción en la columna *Editorial* de la tabla *Publicaciones*, ya que en ésta se almacenan tanto libros como revistas, y éstas no tienen valor en esa columna. Será preciso, entonces, controlar de otra manera la integridad de la base de datos, bien validando los datos que se envían al gestor antes de que le lleguen, bien construyendo en el gestor instrucciones *check* que pueden ser algo complicadas.

5. Desarrollo del caso de uso *Crear casillas vacías* (segunda parte)

En estos momentos estamos ya en situación de considerar el diseño que vayamos a dar a la base de datos: de hecho, y aunque muy ligeramente, ya hemos asumido parte de su diseño en el código SQL que embebíamos en la implementación de la operación *identificar* (Cuadro 12) y en las dos operaciones *insert* de *Edicion* y *Casilla* (Cuadro 13).

Consideramos que la edición del juego está compuesta, además de por el nombre de la ciudad y del número de versión, de un conjunto de casillas y de un conjunto de tarjetas. En el caso de las casillas, y puesto que hay varios tipos de ellas, podemos optar por:

1. Considerar la creación de una clase abstracta *Casilla* con tantas especializaciones como subtipos hayamos identificado. En la Figura 59 se ha creado una agregación entre *Edición* y las muchas casillas que puede contener (nótese la A situada en el cuadro de *Casilla*, que indica que se trata de una clase abstracta). Con propósitos ilustrativos, se muestran solamente algunas de las posibles especializaciones de *Casilla*.

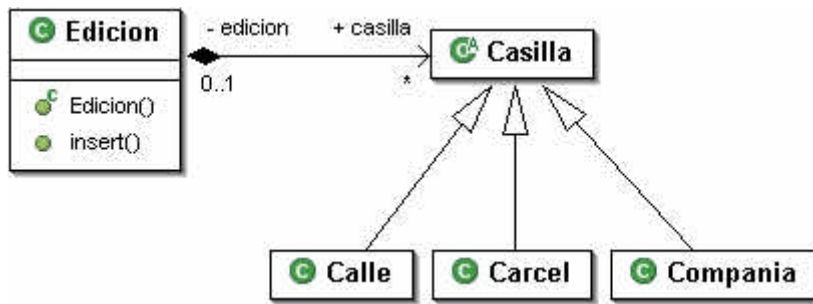


Figura 59. En esta alternativa, cada Edición está compuesta de muchas instancias de *Casilla* (que es abstracta), que posee varias especializaciones

2. Utilizar el patrón *Estado* y delegar el tipo de la casilla a una clase asociada a *Casilla*, que sería concreta.

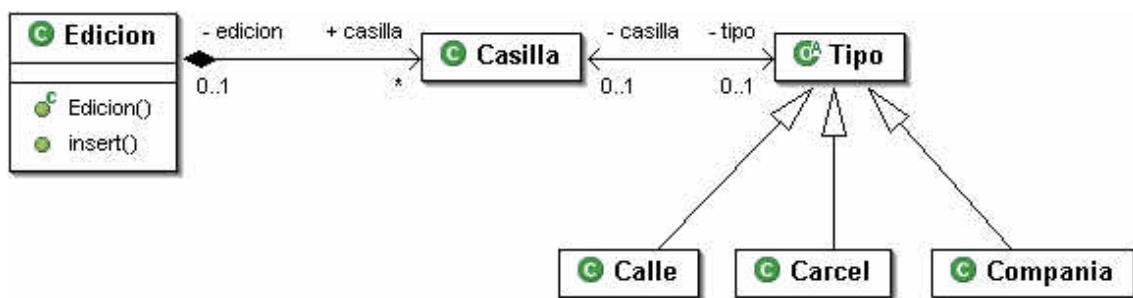


Figura 60. En esta otra, *Casilla* es concreta, y posee un *Tipo* abstracto, especializado al subtipo correspondiente a la casilla

Las dos alternativas tienen ventajas e inconvenientes. Optaremos por utilizar la segunda.

La Figura 61 muestra parte del posible diseño de la base de datos para la aplicación del administrador: se ha utilizado una variante del patrón *una clase, una tabla*, ya que no se ha creado la tabla que, en rigor, correspondería a la clase abstracta *Tipo*. También se ha creado una tabla *Barrio*, en la que se almacenan los diferentes colores de los barrios. Puesto que los colores son comunes a todas las ediciones, la tabla *Barrio* no incluye información de la *Edicion*.

Como se observa, se almacenan dos columnas para cada *Edición*, en las que se almacena información de la ciudad y de la versión correspondientes. Estas dos columnas constituyen la clave primaria de la tabla, que se propaga hacia las tablas relacionadas.

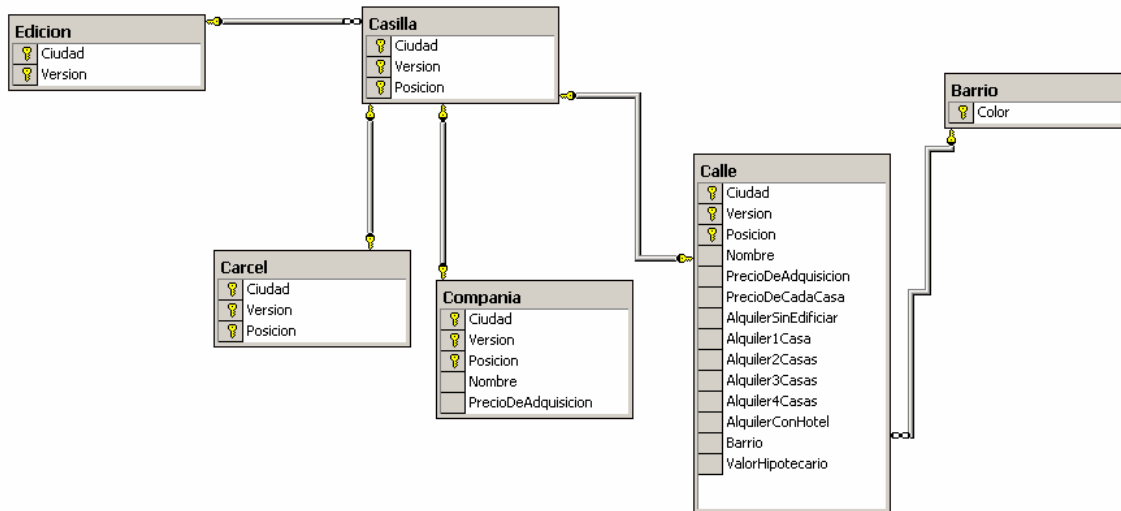


Figura 61. Vista parcial del posible diseño de la base de datos

Edicion está relacionada con *Casilla* mediante una relación de 1 a muchos. La clave primaria de *Casilla* está formada por la clave primaria de *Edicion* más el número de la casilla correspondiente (columna *Posicion*). La relación entre la *Casilla* y sus subtipos se ha representado con relaciones 1 a 1, creando una tabla por cada subtipo. Las tablas de subtipos incluyen la clave primaria de *Casilla*, pero poseen otras columnas para representar la información adicional de cada una: *Calle*, por ejemplo, aporta el nombre de la calle, el precio de adquisición, el precio de cada casa, los diferentes precios de alquiler y el valor hipotecario. Se da la circunstancia de que el valor hipotecario es siempre la mitad del precio de adquisición, por lo que puede eliminarse esta columna de la tabla y añadir a la clase que corresponda una operación *getValorHipotecario()*.

Otra circunstancia relacionada con las casillas de tipo *Calle* es que los precios de las casas son los mismos para todas las calles del barrio. Así, con el diseño de la base de datos anterior, almacenaríamos información redundante en todas las casas de un mismo barrio. Una solución que tendría en cuenta estas consideraciones se muestra en la Figura 62: cada *Edicion* posee varios barrios, que son de un *Color* predeterminado. Las calles pertenecen a un determinado *Barrio*, en donde se almacena el precio de adquisición de cada casa.

En la Figura 62 se ha añadido una nueva columna a *Casilla* que almacena el subtipo de la casilla correspondiente.

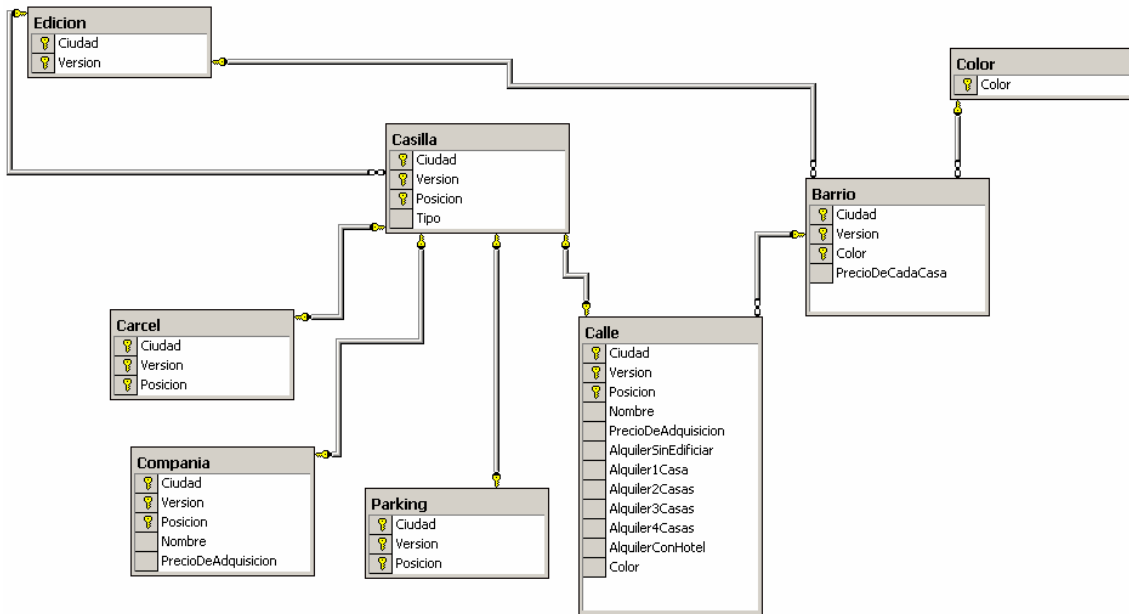


Figura 62. Diseño alternativo de la base de datos

6. Desarrollo del caso de uso *Asignar tipo a casilla*

El primer evento de los flujos dados en la descripción textual del caso de uso (Tabla 10, página 90) indica que el usuario selecciona el tipo de casilla que desea crear (calle, estación, cárcel, etc.), y que luego el controlador del caso de uso comprueba que la edición admite ese tipo de casilla.

Puesto que el tablero de juego del Monopoly está formado por cuarenta casillas, podemos crear una ventana que muestre visualmente la disposición de las diferentes casillas, y dotarla de facilidades para que el administrador pueda configurar el tablero a su gusto. La Figura 63 muestra el posible diseño de esta ventana, que se ajusta bastante el boceto que se dibujó en el flujo de trabajo de *Requisitos* (Figura 38, página 88): posee tantos botones como casillas, numerados desde el 1 al 40. Se desea que, al pulsar el botón correspondiente a una casilla, la etiqueta situada en la esquina superior izquierda indique el número de casilla que se ha pulsado; una vez seleccionada una casilla, si su subtipo aún no le ha sido asignado, éste se podrá elegir utilizando las solapas situadas en la zona central de la ventana. Por ejemplo: si la casilla nº 2 no ha sido aún establecida a *Calle*, *Impuestos*, *Salida*, etc., el usuario podrá seleccionar el subtipo correspondiente eligiendo una de las solapas. Si la casilla sí tenía asignado subtipo, al pinchar en su botón se pondrá en primer plano la solapa correspondiente, que mostrará sus características.

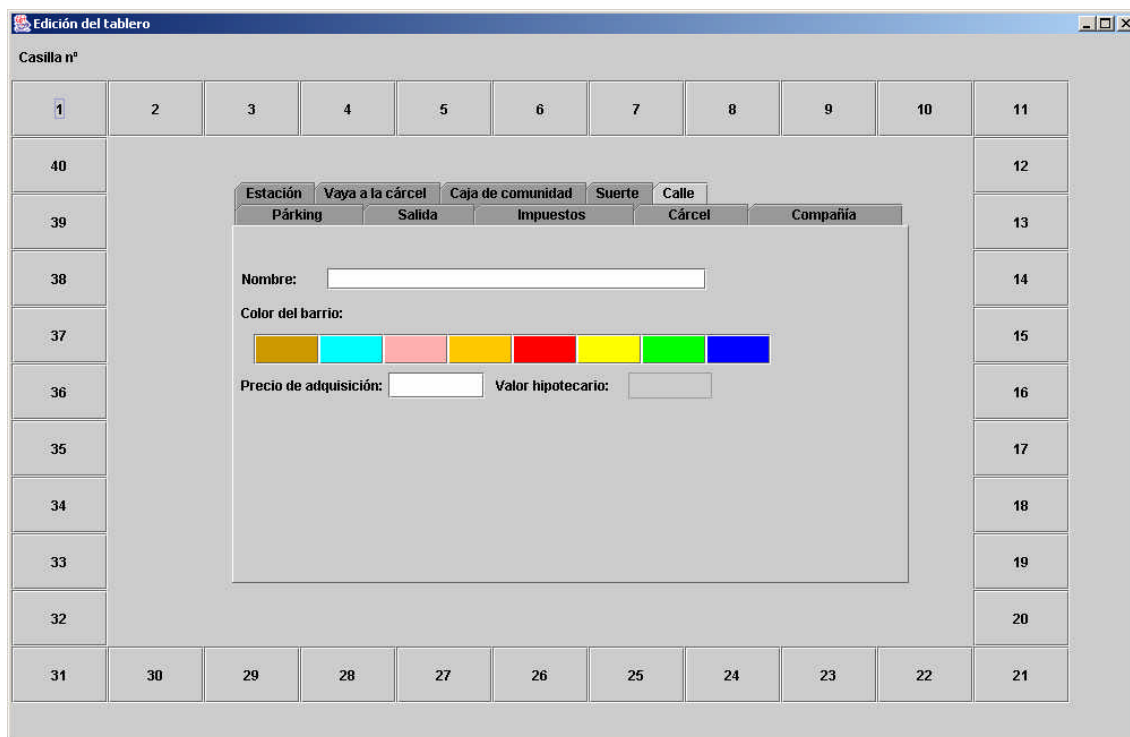


Figura 63. Prototipo de la ventana para configurar las casillas del tablero

Cada uno de los botones, entonces, representa una casilla del tablero: realmente, una instancia de la clase *Casilla* mostrada en el diagrama de clases de la Figura 59. Siguiendo nuestros principios de asignar al dominio las responsabilidades relacionadas con la resolución del problema, podemos entender que los botones que representan las casillas no son sino interfaces entre el usuario y la instancia correspondiente de la clase *dominio.Casilla*.

Para controlar las casillas, podemos crear un tipo especial de botón al que podemos llamar *JBCasilla* (*JB* de *JButton*) que conozca a una instancia de *dominio.Casilla* (Figura 64):

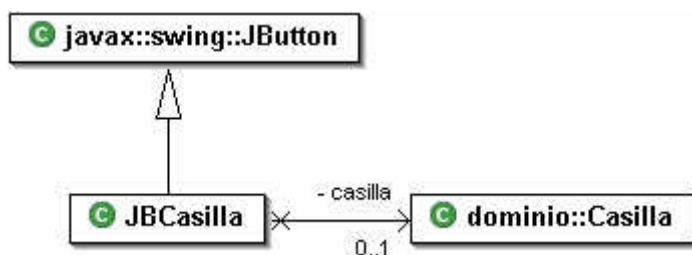


Figura 64. Creación de un subtipo de *JButton* para manipular casillas

Según se ha dicho, queremos que, cuando se pulse el botón correspondiente a una casilla en la ventana de la Figura 63, la etiqueta superior izquierda debe indicar el número de casilla seleccionado y, si la casilla ya tenía un subtipo asignado, poner en primer plano la solapa correspondiente y cargarla con su información. Gráficamente:

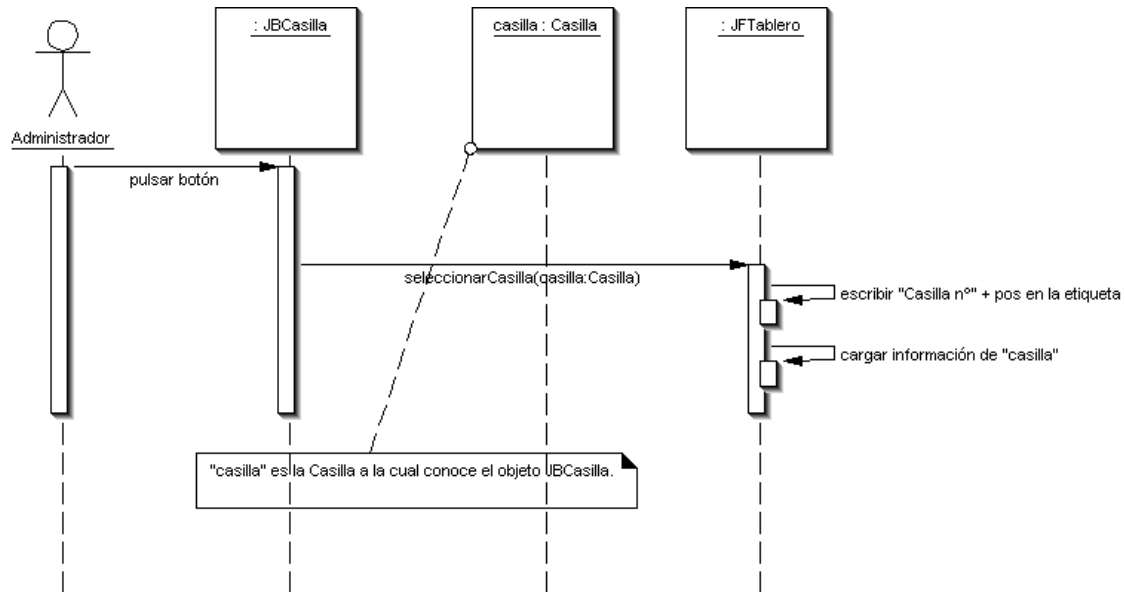


Figura 65. Acciones realizadas al pulsar el botón correspondiente a una casilla

Como se observa en el diagrama, el botón (instancia de *JBCasilla*) debe conocer al tablero sobre el que está situado. Por tanto, enriqueceremos el diagrama de clases creando una asociación entre *JBCasilla* y *JFTablero*:

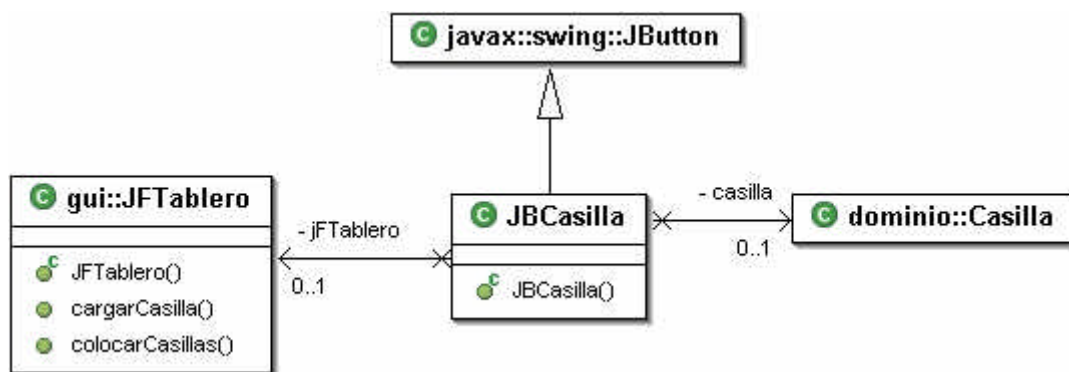


Figura 66. Cada objeto *JBCasilla* conoce el tablero sobre el que está situado

La clase *JFTablero* de la Figura 66 incluye una operación *colocarCasillas* que distribuye por el tablero los botones correspondientes a las cuarenta casillas. El cuerpo de esta operación incluirá cuatro bucles para crear las instancias de *JBCasilla* y situarlas en las coordenadas adecuadas. Al crear cada instancia de *JBCasilla*, se le pasa como parámetro el tablero sobre el que está situadas. Este paso de mensajes se ilustra en el Cuadro 20: dentro del bucle (lado izquierdo) se crean los botones correspondientes a las casillas; dentro del código del constructor del botón se crea la instancia de *dominio.Casilla* correspondiente, a la que se le pasa como parámetro el tablero; a continuación, se le dice al botón cuál es la casilla a la que representa (que se toma de la colección de casillas del objeto de tipo *Edicion*).

<pre> public void colocarCasillas() { for (int i=1; i<=11; i++) { Casilla c=(Casilla) edicion.getCasillas().get(i-1); JBCasilla boton=new JBCasilla(this); boton.setCasilla(c); boton.setSize(85, 50); boton.setLocation((i-1)*85, 40); jContentPane.add(boton, null); } for (int i=12; i<=21; i++) { ... } } </pre>	<pre> public JBCasilla(JFTablero v) { super(); initialize(); this.tablero=v; } public void setCasilla(Casilla casilla) { this.casilla = casilla; setText(""+casilla.getPos()); } </pre>
--	--

Cuadro 20. Parte del código del método *colocarCasillas()* de *JFTablero*

En *JBCasilla* escribiremos un método que se ejecute cuando el usuario pulse el botón. De acuerdo con el diagrama de secuencia de la Figura 65, este método le dirá al tablero que cargue la información correspondiente a la casilla seleccionada:

```

protected void seleccionado() {
    this.tablero.setCasilla(casilla);
}

```

Cuadro 21. Método correspondiente a la pulsación del botón (en *JBCasilla*)

Obviamente, el código mostrado en el Cuadro 21 requiere la existencia de una operación *setCasilla* en *JFTablero*. La implementación, de momento, puede ser la del Cuadro 22, suficiente para que la ventana muestre en la etiqueta el número de la casilla seleccionada.

```

public void setCasilla(Casilla c) {
    this.jlCasillaNumero.setText("Casilla nº " + c.getPos());
}

```

Cuadro 22. Método *setCasilla(int)* en *JFTablero*

Para que el tablero muestre la información correspondiente a la casilla seleccionada es preciso añadir más código al método del Cuadro 22: el tablero deberá recuperar la información del tipo de la casilla pasado como parámetro. Puesto que el tipo de la casilla está delegado al objeto *Tipo* (recuérdese que estamos utilizando el patrón *Estado*, Figura 60), el método *setCasilla* interrogará a la casilla que recibe como parámetro por el valor de su *Tipo*.

```

public void cargarCasilla(Casilla c) {
    this.jlCasillaNumero.setText("Casilla nº " + c.getPos());
    if (c.getTipo()!=null) {
        ...
    }
}

```

Cuadro 23. Método *cargarCasilla(int)* en *JFTablero*, con una pequeña adición

En este momento estamos ya en condiciones de crear ediciones con sus casillas sin tipo, así como de crear casillas de tipo *Parking*.

7. Desarrollo del caso de uso *Crear parking*

El tablero admite casilla de *Parking* gratuito si no la tenía. Cuando se guarde la casilla, se debe insertar un registro en la tabla *Parking* (que debe-

mos crear en la base de datos) almacenando la posición de la casilla en el tablero junto a la ciudad y el número de versión de esta edición del juego.

De acuerdo con el diseño de la base de datos discutido antes, añadiremos una tabla *Parking* sin más columnas que las tres que constituirán la clave primaria (Ciudad, Versión y Posición), que hará referencia a la tabla principal *Casilla*. Ésta, a su vez, hace referencia a la tabla principal *Edición*: por tanto, antes de insertar un registro en la tabla *Parking*, habrá que insertarlo en *Casilla* y, antes, en *Edicion*.

Igualmente tendremos en cuenta el prototipo de interfaz de usuario que mostramos en la Figura 63. La Figura 67 muestra el mismo prototipo, pero con la solapa correspondiente a la casilla del *Parking* en primer plano. En las ediciones estándar del Monopoly, la casilla del parking suele estar en la casilla nº 21; no obstante, nosotros daremos libertad al usuario para que la sitúe en donde quiera. Se desea que, al pulsar el botón correspondiente a una casilla, si se elige la solapa del *Parking*, se muestre en la etiqueta el texto “El parking estará situada en la casilla X”, donde X es el número de la casilla.

Además, queremos que al pulsar el botón *Guardar*, se inserte el registro correspondiente en la base de datos.

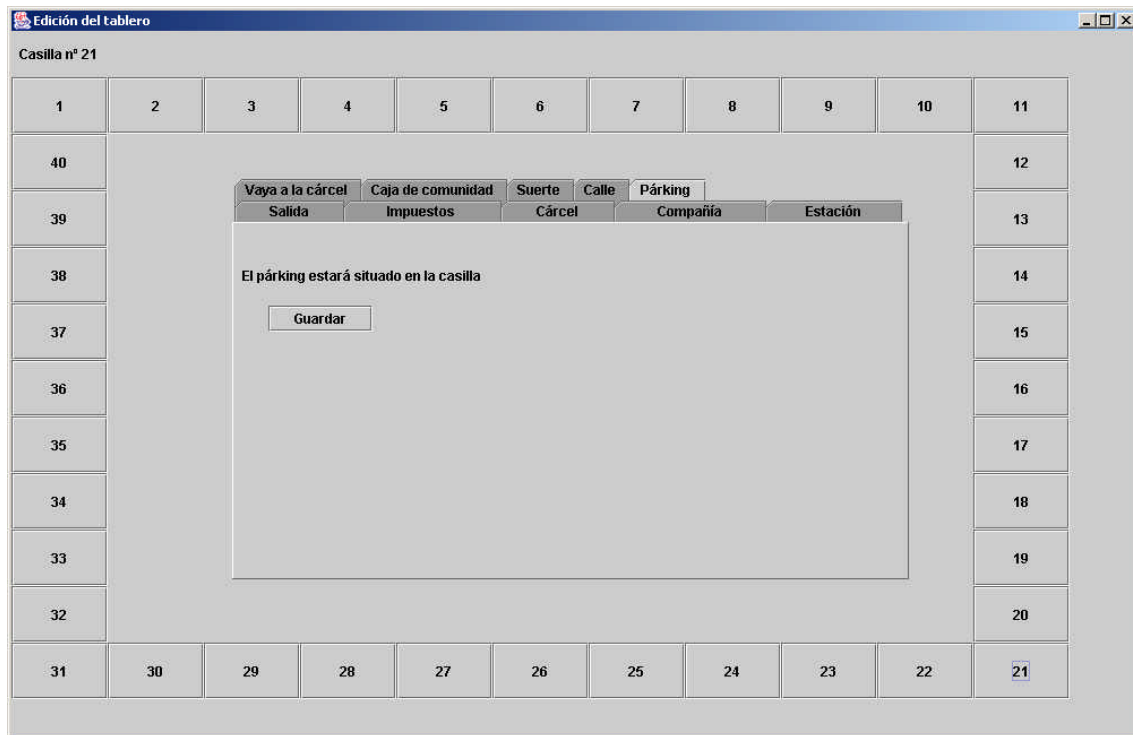


Figura 67. La interfaz de usuario considerando la casilla del *Parking*

Gráficamente, el comportamiento deseado es el siguiente:

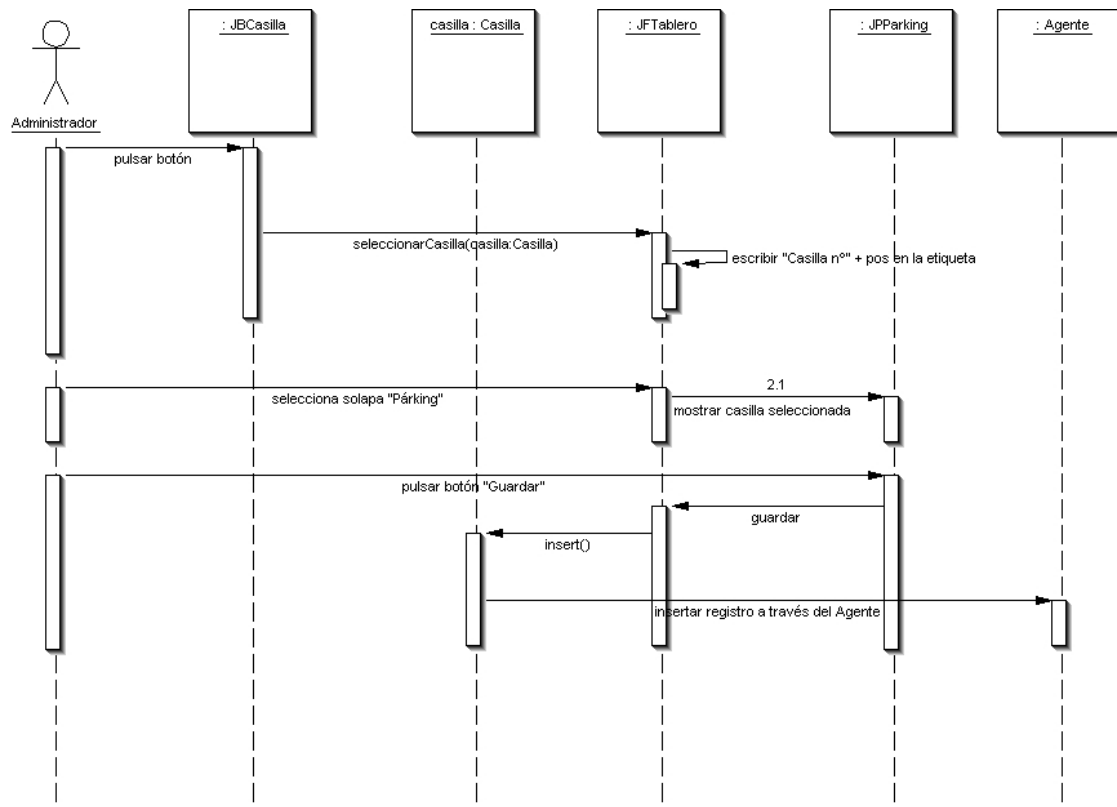


Figura 68. Comportamiento deseado para la casilla del *Parking*

De acuerdo con el planteamiento mostrado, *JFTablero* le debe pasar información de la casilla seleccionada a *JPParking* (la solapa del parking). En el prototipo de ventana que se ha construido, las solapas son especializaciones de *JPanel*. De forma general, todos estos *JPanels* deben ser capaces de responder al mensaje que, en la Figura 68, se ha etiquetado “mostrar casilla seleccionada”. Con el fin de que *JFTablero* pueda enviar ese mensaje a cualquiera de las solapas independientemente de su tipo (es decir, independientemente de que se trate de *JPParking*, *JPCalle*, *JPSalida*, etc.), construiremos una interfaz con la operación *setCasilla(Casilla)*, que será implementada por todos estos *JPanels* (Figura 69).

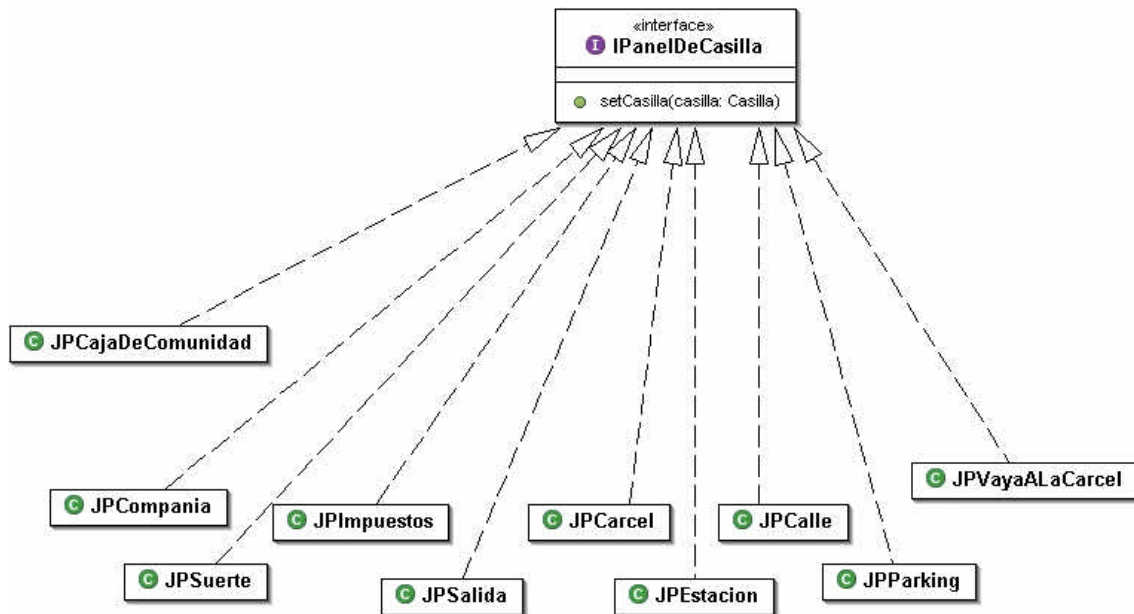


Figura 69. Hacemos que los diferentes *JPanels* implementen una interfaz

Para el paso de mensaje se produzca (es decir, para que el *JFTablero* le diga al *JPPanel* seleccionado que muestre la información de la casilla), añadimos un evento al *JTabbedPane* del *JFTablero* (es decir, al conjunto de solapas) para que, cuando se seleccione una solapa, se ejecute sobre ella el método *setCasilla(Casilla)*. En el entorno de desarrollo Eclipse lo hacemos como se muestra en la siguiente figura: utilizando el editor visual, en el lado izquierdo de la Figura 70 añadimos el evento; en el lado derecho, llamamos a un método nuevo como respuesta a la ejecución de ese evento.

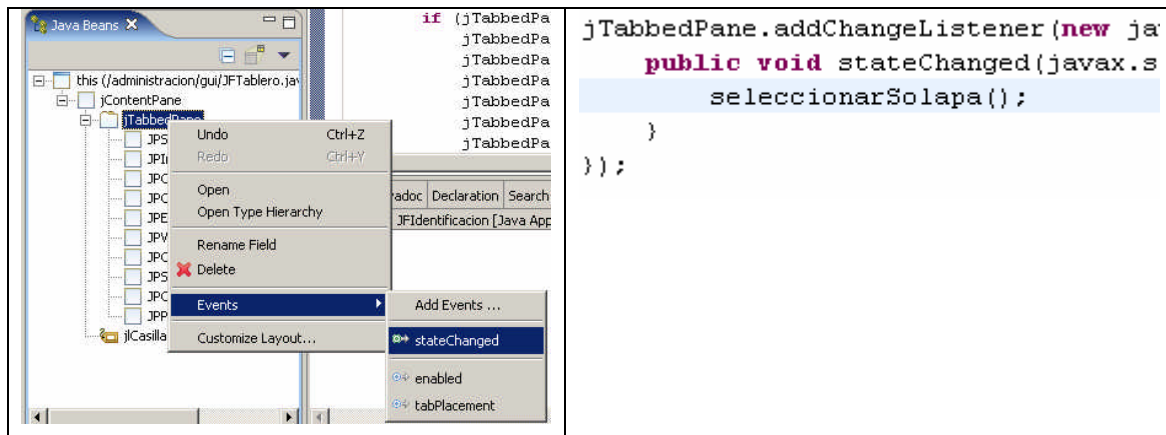


Figura 70. Adición de un evento a un componente visual en Eclipse

El código del método *seleccionarSolapa* debe recuperar la solapa seleccionada en forma del interfaz *IPanelDeCasilla* mostrado en la Figura 69, y que sobre tal objeto ejecute su operación *setCasilla(Casilla)*. Una versión básica de este código es la siguiente:

```
protected void seleccionarSolapa() {
    IPanelDeCasilla solapa=(IPanelDeCasilla) this.jTabbedPane.getSelectedComponent();
    solapa.setCasilla(this.casillaSeleccionada);
}
```

Figura 71. Implementación del método *seleccionarSolapa*, que se ejecuta al seleccionar una solapa en *JFTablero*

Como se observa en el cuadro anterior, a la solapa se le pasa la casilla seleccionada, que es aquella correspondiente al botón *JBCasilla* sobre la que el usuario ha hecho click. Sin embargo, el código con el que *JFTablero* responde al evento click sobre un *JBCasilla* no asigna ninguna referencia a ningún campo llamado *casillaSeleccionada* (véase el método del Cuadro 22, página 123): por tanto, debemos añadir un campo de tipo *dominio.Casilla* a *JFTablero* al que llamaremos *casillaSeleccionada*, y modificaremos el código del método *setCasilla* para que realice la asignación a este campo.

```
public void setCasilla(Casilla c) {
    this.casillaSeleccionada=c;
    this.jlCasillaNumero.setText("Casilla nº " + casillaSeleccionada.getPos());
    if (casillaSeleccionada.getTipo()!=null) {
    }
}
```

Cuadro 24. Modificación del método *setCasilla* en *JFTablero*

Puesto que estamos en el desarrollo del caso de uso *Crear parking*, implementaremos la operación *setCasilla* en la solapa *JPParking*.

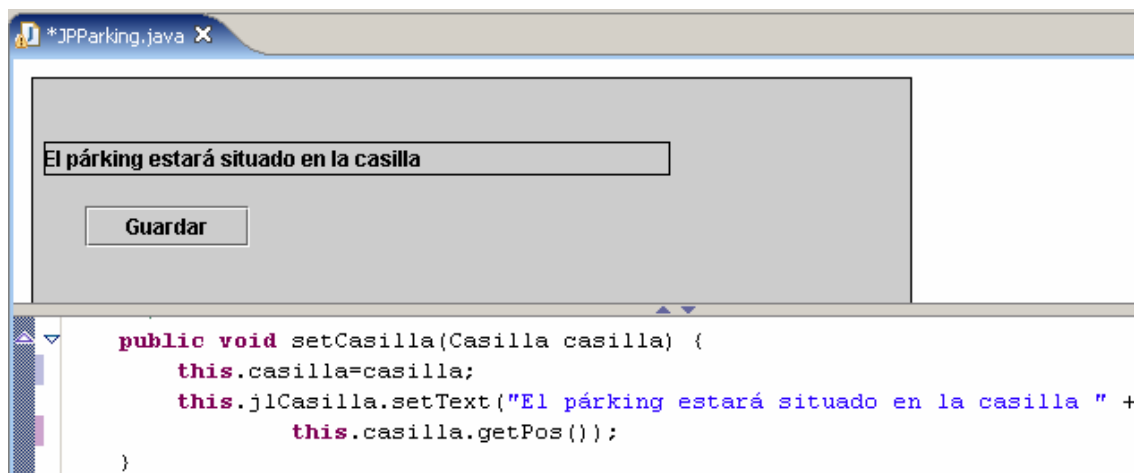


Figura 72. Implementación de *setCasilla* en *JPParking*

Según decíamos y mostrábamos en el diagrama de la Figura 68, cuando se pulse el botón *Guardar* en *JPParking* debe decirse a la casilla correspondiente que se inserte en la base de datos. Entonces, escribimos un método en *JPParking* al que llamaremos *guardarCasilla*, para que se ejecute cuando se pulse el botón. El método asignará a la casilla seleccionada el tipo "P  king". La implementaci  n de esta operaci  n se ofrece en el lado izquierdo del Cuadro 25: como vemos, se llama al m  todo *setTipo(String)* de la clase *Casilla*, cuya implementaci  n (lado derecho) delega la ejecuci  n de la ope-

ración a su atributo *tipo* el tipo *Parking*, que es una especialización de la clase *Tipo* (recuérdese que estamos utilizando el patrón *Estado*, ilustrado en la Figura 60, página 118). A continuación, la operación *setTipo(String)* crea la relación en sentido contrario (es decir, le dice a *this.tipo* que su casilla es la misma que está ejecutando la operación); luego, ejecuta la operación *update* (lado derecho, abajo, del cuadro), que realiza dos operaciones sobre la base de datos: primero borra la casilla (que tal vez tuviera un tipo asignado) y luego la inserta con el nuevo tipo.

<pre>protected void guardarCasilla() { try { this.casilla.setTipo("Parking"); } catch (Exception e) { JError j=new JError(); j.mostrarError(e.toString()); j.setModal(true); j.setVisible(true); } }</pre>	<pre>public void setTipo(String nombreDelTipo) throws SQLException { this.tipo=new Parking(); this.tipo.setCasilla(this); update(); } private void update() throws SQLException { delete(); insert(); }</pre>
--	--

Cuadro 25. Método *guardarCasilla* en *JPParking*, e implementación de dos operaciones en *Casilla*

Tipo poseerá un método *insert* abstracto, que tendrá implementación en sus diferentes especializaciones. En la subclase *Parking*, que es la que nos interesa en este caso de uso, una primera versión de la implementación de *insert* es la siguiente:

<pre>public void insert() throws SQLException { PreparedStatement p=null; boolean result=false; String SQL="Insert into Parking (Ciudad, Version, Posicion) values (?, ?, ?)"; try { Connection bd=Agente.getAgente().getDB(); p=bd.prepareStatement(SQL); p.setString(1, casilla.getCiudad()); p.setString(2, casilla.getVersion()); p.setInt(3, casilla.getPos()); p.executeUpdate(); } catch (SQLException ex) { throw ex; } finally { p.close(); } }</pre>
--

Cuadro 26. Implementación de *insert* en *Parking*

Obsérvese, en el código del Cuadro 26, que la instrucción SQL inserta un registro en la tabla *Parking*, almacenando los valores de la ciudad, versión y posición de la casilla correspondiente. Para recuperar estos valores, la instancia de *Parking* utiliza el campo *casilla* definido en *Tipo*, en donde el campo es sin embargo, privado (véase, en la Figura 60, que el modificador de visibilidad de *casilla* en *Tipo* es un signo menos): es preciso, entonces, aumentar la visibilidad (al menos a protegido) de *casilla* para que puedan acceder al campo las especializaciones de *Tipo*.

El diseño de clases que vamos teniendo de momento es el de la Figura 73: obsérvese el modificador de visibilidad de la *casilla* a la cual conoce *Tipo* (ahora es una almohadilla, que denota que es un campo protegido) y que el

entorno de desarrollo indica errores en la clase *Parking*. Estos errores se deben a que la implementación del método *insert* accede a los métodos *getCiudad* y *getVersion* de *casilla* que, sin embargo, no están definidos.

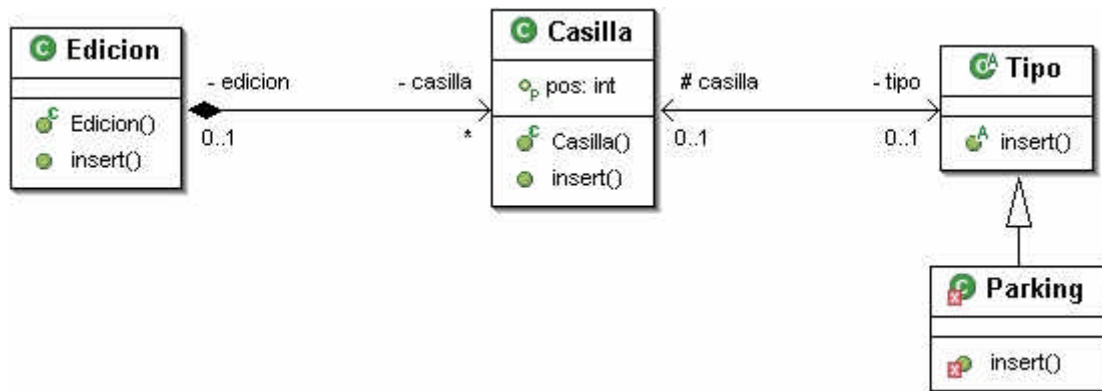


Figura 73. Fragmento del diseño de clases

La implementación que podemos dar a los dos métodos que necesitábamos en la clase *Casilla* es la siguiente:

```

public String getCiudad() {
    return this.edicion.getCiudad();
}

public String getVersion() {
    return this.edicion.getVersion();
}

```

Cuadro 27. Implementación de dos operaciones en *Casilla*

De acuerdo con la discusión realizada al presentar el diseño de la base de datos, antes de insertar un registro en la tabla *Parking* es preciso insertarlo en *Casilla*. Podemos modificar el código del método *insert* de *Casilla* (que ofrecíamos en el lado derecho del Cuadro 25) y añadirle al principio unas instrucciones para insertar previamente en la tabla *Casilla*, de forma que quede como en el Cuadro 28: cuando la casilla no tiene aún tipo (es decir, pertenece a una edición recién creada), se ejecuta la primera versión de la sentencia SQL (que no da valor a la columna *Tipo*); cuando ya lo tiene, además de asignar valor a esta columna, ejecuta *tipo.insert()* (última línea).

```

public void insert() throws SQLException {
    PreparedStatement p=null;
    boolean result=false;
    String SQL="Insert into Casilla (Ciudad, Version, Posicion, Tipo) values " +
        "(?, ?, ?, ?)";
    if (tipo==null)
        SQL="Insert into Casilla (Ciudad, Version, Posicion) values (?, ?, ?)";
    try {
        Connection bd=Agente.getAgente().getDB();
        p=bd.prepareStatement(SQL);
        p.setString(1, getCiudad());
        p.setString(2, getVersion());
        p.setInt(3, pos);
        if (tipo!=null)
            p.setString(4, tipo.getNombreDeTipo());
        p.executeUpdate();
    }
    catch (SQLException ex) { throw ex; }
    finally { p.close(); }
    if (tipo!=null)
        this.tipo.insert();
}

```

Cuadro 28. Implementación de *insert* en *Casilla*

Con estos cambios, el usuario está en disposición de crear ya la casilla del parking y es capaz de almacenarla en la base de datos.

8. El patrón *Singleton*

El patrón *Singleton* se utiliza cuando se desea crear una clase de la cual se creará, como máximo, una instancia. Para hacerlo, se crea una clase con un atributo estático cuyo tipo es la propia clase *Singleton* y al que inicialmente se le asigna el valor *null*; a la clase se la dota de un constructor no público que asigna valores al resto de atributos. Cuando una clase externa desea utilizar el *Singleton*, recupera la instancia a través de un método público y estático que pregunta por el valor del atributo estático: si es *null*, crea una instancia llamando al constructor y la devuelve; si no es *null*, la devuelve directamente.

El siguiente código muestra la clase *Agente* implementada como un *Singleton*. Se ha destacado el código típico de los “singletons”.

```

package persistencia;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Agente {
    protected static Agente mInstancia=null;
    protected Connection mBD;
    protected String mURL=
        "jdbc:microsoft:sqlserver://127.0.0.1:1433;DatabaseName=Monoploy";

    protected Agente() throws SQLException, ClassNotFoundException {
        Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
        mBD=DriverManager.getConnection(mURL,
            "administrador", "admin");
    }

    public static Agente getAgente() throws SQLException, ClassNotFoundException {
        if (mInstancia==null)
            mInstancia=new Agente();
        return mInstancia;
    }

    public Connection getBD() throws SQLException {
        if (mBD.isClosed())
            mBD=DriverManager.getConnection(mURL,
                "administrador", "admin");
        return mBD;
    }
}

```

Cuadro 29. Código de la clase *Agente*, implementada como un *Singleton*

9. Pruebas del código

Como ya hemos indicado en varias ocasiones, y además se ilustraba en, por ejemplo, la Figura 16 (página 52), cada iteración conlleva la realización de actividades de verificación y validación de los diferentes artefactos producidos. Los artefactos no ejecutables se someten a inspecciones y revisiones (deseablemente por parte de terceros) y a listas de comprobación (como la mostrada en la Tabla 13 para diagramas de secuencia). El código, además de ser sometido también a inspecciones, revisiones y listas de comprobación, debe ser ejecutado mediante un proceso de pruebas sistemático. Durante éste, se definirán casos de prueba que serán ejecutados contra el código y, deseablemente, serán archivados para ser reejecutados cuando, durante posteriores iteraciones o en la futura etapa de mantenimiento, se realicen modificaciones en el código. A esta reejecución de antiguos casos de prueba se la denomina pruebas de regresión.

Normalmente se identifican dos enfoques diferentes en la realización de pruebas:

- a) En las pruebas funcionales o de caja negra, el ingeniero de pruebas se centra únicamente en ejecutar servicios de la clase que se está probando con ciertos datos de entrada, comparar los resultados obtenidos con los esperados, y actuar de un modo u otro según se encuentren o no errores.

- b) En las pruebas de caja blanca, al ingeniero de pruebas le interesa seguir la traza de las instrucciones que está ejecutando cada caso de prueba.

Ambos tipos de pruebas son complementarios, en el sentido de que utilizamos las pruebas de caja negra para detectar errores, y las de caja blanca para comprobar cuánto código han ejecutado los casos de caja negra: por muchos casos de prueba que ejecutemos sobre una clase, nadie nos garantiza que se hayan ejecutado, por ejemplo, todas sus instrucciones a menos que, de algún modo, guardemos y observemos el registro de ejecución. Piénsese en un método que contenga varios *if* seguidos y anidados con bucles de diversos tipos: si no le hacemos una traza cuidadosa durante las pruebas, ¿tendremos la garantía de que los casos de prueba han sido capaces de recorrer todas las posibles ramas de todos los posibles caminos de ejecución? ¿No tendrá quizás el método vicios ocultos que nuestros casos de prueba no hayan sido capaces de encontrar?

Cuando disponemos del código fuente, lo ideal es someterlo a los dos tipos de pruebas. Es preciso notar, no obstante, que probar correctamente un sistema puede suponer en torno a la mitad del esfuerzo total empleado en su desarrollo: piénsese en el método que describíamos en el párrafo anterior: progresivamente, tendremos que ir construyendo casos de prueba hasta conseguir que se alcance el umbral predeterminado del criterio de cobertura seleccionado.

9.1. Un proceso de pruebas combinando caja negra con caja blanca

Como acaba de decirse, los casos de prueba de caja negra se utilizan para detectar errores en la clase que se está probando, mientras que los de caja blanca miden la cobertura que los de caja negra han alcanzado en la clase bajo prueba. Así, cuando se desea probar una clase K , se crea una batería T de casos de prueba de caja negra, que son ejecutados sobre K . Si T detecta algún fallo sobre K , entonces K se corrige; en otro caso, se mide la cobertura alcanzada por T sobre K utilizando técnicas de caja blanca. Si T alcanza el nivel de cobertura deseado, entonces K se da por buena y el proceso de pruebas termina. Pero si T no alcanza el nivel de cobertura deseado, deben construirse más casos de prueba (que se agregan a T , formando así T') que se ejecutan sobre K con el fin de encontrar fallos. Si T' encuentra fallos, se corrige K ; en otro caso, se mide la cobertura alcanzada por T' . Estos pasos se repiten hasta que la batería de casos de prueba no encuentra fallos y se alcanza el nivel de cobertura deseado.

9.2. Pruebas funcionales con *junit*

*junit*⁵ es un framework que automatiza la ejecución de las pruebas unitarias de código Java. Si se desea probar una clase *A*, se construye una clase de (que llamamos, por ejemplo, *TestA*) en la que se implementa un conjunto de métodos dirigidos a realizar pruebas de *A*. Normalmente, en cada uno de los métodos de *TestA* se crean dos objetos: manualmente, un objeto de clase *A* que se corresponderá con el resultado que se espera obtener de la prueba; mediante la escritura de código, se construye y manipula un objeto de clase *A* ejecutando las operaciones definidas en *A*, que se corresponde con el resultado obtenido. Si el objeto construido manualmente es igual que el construido automáticamente, entonces la prueba ha sido superada.

Supongamos que deseamos probar con *junit* el método *ingresar* de la clase *Cuenta* de la Figura 74:

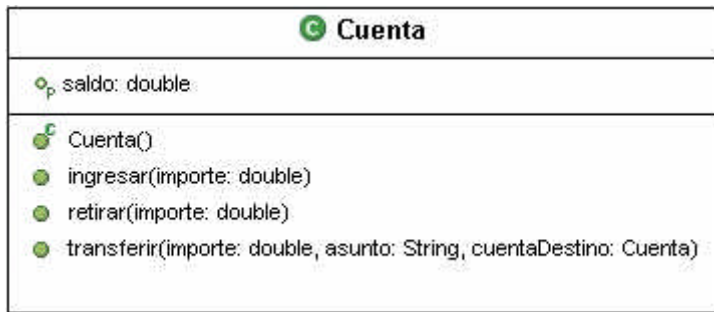


Figura 74. La clase *Cuenta* que deseamos probar

De momento, escribiremos un solo método en la clase de prueba: crearemos manualmente una instancia a la que asignaremos un saldo de 1000 euros, que se corresponderá con el objeto esperado; igualmente, crearemos una cuenta vacía sobre la que ejecutaremos el método bajo prueba con 1000 como argumento. Por último, compararemos los dos objetos haciendo uso de la biblioteca de clases que incorpora *junit*.

⁵ *junit* se puede descargar gratuitamente de <http://www.junit.org>. Existen frameworks de prueba similares para otros lenguajes, como NUnit para los lenguajes de .NET.

```
package dominio.tests;

import junit.framework.TestCase;
import dominio.Cuenta;

public class CuentaTest extends TestCase {

    public void testIngresar() {
        Cuenta esperado=new Cuenta();
        esperado.setSaldo(1000.0);
        Cuenta obtenido=new Cuenta();
        obtenido.ingresar(1000.0);
        assertEquals(esperado, obtenido);
    }

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

Cuadro 30. Código de una clase de prueba de *junit*

Del código anterior podemos destacar varios elementos:

1. Es recomendable ubicar las clases de prueba en un paquete separado.
2. Es preciso importar la clase *junit.framework.TestCase*
3. Es preciso importar la clase que se está probando.
4. La clase de prueba es una especialización de *TestCase*.
5. Para que *junit* ejecute los métodos con los casos de prueba, el nombre de éstos debe comenzar por *test*.
6. Normalmente, en los métodos se utilizan operaciones heredadas de *TestCase*, como *assertEquals(Object, Object)*, que compara los dos objetos pasados como parámetros utilizando el método *equals(Object)* de la clase bajo prueba.
7. La clase de prueba puede ejecutarse utilizando la propia interfaz gráfica de *junit*.

La Figura 75 muestra el resultado de ejecutar la clase de prueba anterior. La barra roja denota que, al menos, ha fallado uno de los casos de prueba contenidos en ella (en nuestro ejemplo, el único que hay).

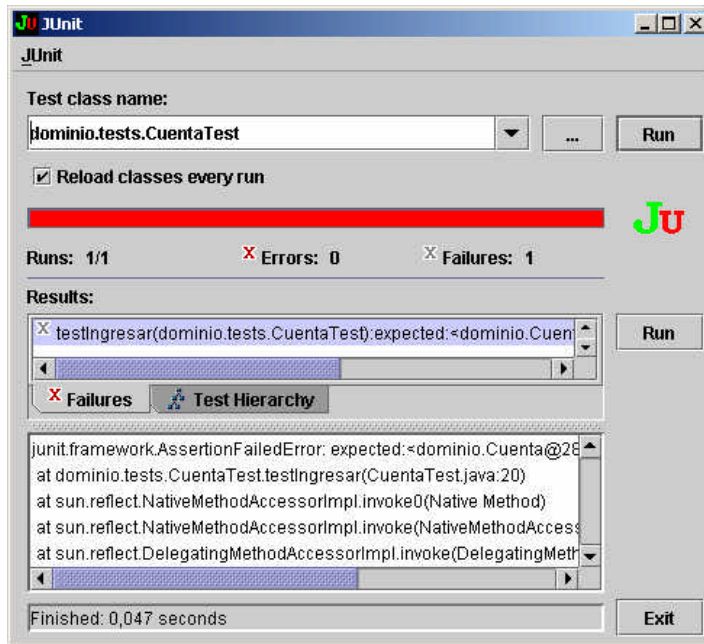


Figura 75. Resultado de ejecutar la clase de prueba del Cuadro 30

Conveniencia de redefinir *equals*

A pesar de que el código de la clase es correcto, el caso de prueba falla porque, como hemos indicado, la operación *assertEquals* compara los dos parámetros utilizando el método *equals* de la clase bajo prueba: al no haber implementado en *Cuenta* esta operación, *junit* ejecuta la versión de *equals* que *Cuenta* hereda de la clase *Object* (todas las clases Java son especializaciones de *Object*). Por tanto, es conveniente redefinir el método *equals* en todas las clases que vayamos a probar con *junit* para que devuelva *true* cuando los dos objetos comparados sean iguales desde el punto de vista del dominio de la aplicación. Si, para este ejemplo, asumimos que dos cuentas son iguales cuando tienen saldos iguales, con el código del Cuadro 31, *junit* muestra una barra verde en lugar de la roja.

```
public boolean equals(Object otro) {
    if (!(otro instanceof Cuenta))
        return false;
    Cuenta aux=(Cuenta) otro;
    return aux.getSaldo()==this.getSaldo();
}
```

Cuadro 31. Redefinición de *equals* en *Cuenta*

9.2.1 Fixtures

Es muy probable que, para las pruebas de la clase *Cuenta*, utilizemos más de una vez los mismos objetos: por ejemplo, para probar los métodos *ingresar*, *retirar* y *transferir*, usaremos quizás una instancia de *Cuenta* con 1000 euros de saldo. Para casos como éste conviene utilizar *fixtures*, que son campos que se declaran en la clase de prueba y que son del tipo de la clase que se está probando. Las instrucciones de creación de estas instancias se escriben en el método *setUp*, un método que la clase de prueba hereda de

TestCase y que puede redefinirse: antes de ejecutar cada método de prueba *junit* ejecuta *setUp*; después, ejecuta *tearDown*, otro método que se encuentra definido en *TestCase*. En *tearDown* se suelen poner instrucciones que terminan el procesamiento de las *fixtures*: si en el *setUp*, por ejemplo, abriéramos una conexión a la base de datos, en *tearDown* la cerraríamos.

Supongamos que deseamos probar las funcionalidades de ingreso y retirada de dinero. En lugar de crear varios objetos en cada método de prueba, declaramos unas *fixtures* que creamos en *setUp*, como se hace en el Cuadro

32

```
package dominio.tests;

import junit.framework.TestCase;
import dominio.Cuenta;

public class CuentaTest2 extends TestCase {
    Cuenta cuenta1000;
    Cuenta cuentaCero;
    Cuenta cuentaMenos1000;

    protected void setUp() throws Exception {
        cuenta1000=new Cuenta(); cuenta1000.setSaldo(1000);
        cuentaCero=new Cuenta();
        cuentaMenos1000=new Cuenta(); cuentaMenos1000.setSaldo(-1000);
    }

    protected void tearDown() throws Exception {
        // Podríamos suprimir este método porque no nos hace falta
    }

    public void testIngresar() {
        Cuenta obtenido=new Cuenta();
        obtenido.ingresar(1000.0);
        assertEquals(cuenta1000, obtenido);
    }

    public void testRetirar() {
        cuenta1000.retirar(500);
        assertTrue(cuenta1000.getSaldo()==500);
    }

    public static void main(String[] args) {
        junit.swingui.TestRunner.run(CuentaTest2.class);
    }
}
```

Cuadro 32. Utilización de *fixtures*

Cuando *junit* ejecuta la clase anterior, el framework llama a *setUp()*, luego a *testIngresar()*, a *tearDown()*, a *setUp()*, a *testRetirar()* y a *tearDown()*.

9.2.2 Pruebas “en negativo” con *fail*

Para que una clase sea correcta, debe comportarse bien también cuando está en una situación de error. Supongamos que la *Cuenta* debe lanzar una excepción cuando su saldo es menor que el importe que se desea retirar. Supongamos también que, al ejecutar la operación *transferir(double importe, String asunto, Cuenta cuentaDestino)*, además de restar al *saldo* de la cuenta origen (la que ejecuta el método) el *importe*, se le cobra una comisión

del 3%, con un mínimo de 1,5 euros. Para que la transacción pueda cursarse, la cuenta origen debe disponer también de saldo suficiente.

Para estas operaciones, lo correcto es que la clase lance una excepción si, de la *fixture cuenta1000*, intentamos retirar 2000 o transferir 1000 (en este caso, por el importe de la comisión).

Para estas situaciones se utiliza la instrucción *fail*, que fuerza la aparición de la barra roja. En el código del Cuadro 33 hacemos dos usos distintos de *fail*: en el método *testRetirar* la llamada a *retirar* no debe lanzar excepción (porque de *cuenta1000* pueden retirarse 500 euros), de modo que, si la lanza, el caso de prueba habrá encontrado un fallo y así lo detectará *junit* al ejecutar la instrucción *fail*. En el método *testRetirar2000* intentamos retirar 2000 euros de *cuenta1000*, que tiene sólo 1000 euros de saldo, por lo que, si la llamada *cuenta1000.retirar(2000)* no lanza excepción, el control del programa saltará a la instrucción *fail*, forzando la aparición de la barra roja. El uso que hacemos de *fail* en *testTransferir* es similar al de *testRetirar*: *cuenta1000* debe soportar sin problemas una transferencia de 500 euros, por lo que si lanza excepción el control del programa salta al bloque *catch* y se fuerza la aparición de la barra roja.

```
public void testRetirar() {
    try {
        cuenta1000.retirar(500);
    } catch (Exception e) {
        fail("No debería haber fallado porque hay saldo suficiente");
    }
    assertTrue(cuenta1000.getSaldo()==500);
}

public void testRetirar2000() {
    try {
        cuenta1000.retirar(2000);
        fail("Debería haber fallado porque no hay saldo suficiente");
    } catch (Exception e) {
    }
}

public final void testTransferir() {
    try {
        cuenta1000.transferir(500, "Alquiler", cuentaCero);
    } catch (Exception e) {
        fail("No debería haber fallado");
    }
    super.assertTrue(cuenta1000.getSaldo()==500-1.5);
    super.assertTrue(cuentaCero.getSaldo()==500);
}
```

Cuadro 33. Casos de prueba que utilizan la instrucción *fail*

9.2.3 Pruebas con *JUnit* del caso de uso *Identificación*

En la Tabla 14 (página 105) se mostraba uno de los casos de prueba para la funcionalidad *Identificación*. De acuerdo con él, el usuario debe quedar identificado cuando se identifica con un login y una contraseña válidos. En la Tabla 15 se describía un caso de prueba en negativo, indicando que, al

pasar una combinación errónea de nombre y contraseña, el usuario no debía quedar identificado.

El siguiente cuadro muestra la clase *IdentificacionTests* con dos casos de prueba, uno en positivo y otro en negativo (compárese con los datos de la Figura 76). El primero no espera que se excepción, por lo que si se lanza alguna, se fuerza a que JUnit muestre la barra roja con sendas instrucciones *fail*. El segundo caso de prueba pasa una contraseña inválida, por lo que espera que lance una *PasswordInvalidaException* (recuérdese el código del constructor de *Sesion*, en el Cuadro 12, página 99): si la llamada al constructor lanza tal excepción, la ejecución del método salta al segundo bloque *catch*, que no incluye *fail*, por lo que el caso de prueba sería superado.

```
package tests;

import java.sql.SQLException;

import dominio.Sesion;
import dominio.exceptions.PasswordInvalidaException;
import junit.framework.TestCase;

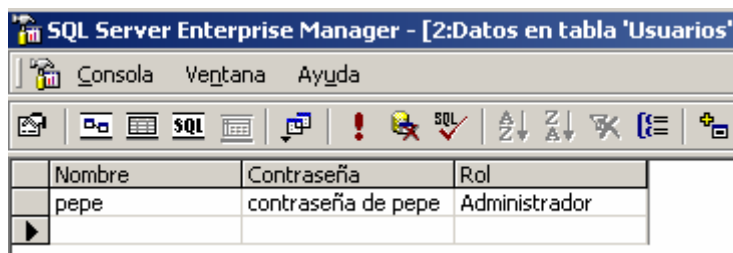
public class IdentificacionTests extends TestCase {

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

    public void testSesionCorrecta() {
        try {
            Sesion s=new Sesion("pepe", "contraseña de pepe");
        } catch (SQLException e) {
            fail("Se ha lanzado una SQLException inesperada");
        } catch (PasswordInvalidaException e) {
            fail("Se ha lanzado una PasswordInvalidaException inesperada");
        }
    }

    public void testSesionIncorrecta() {
        try {
            Sesion s=new Sesion("pepe", "falsa contraseña de pepe");
        } catch (SQLException e) {
            fail("Se ha lanzado una SQLException inesperada");
        } catch (PasswordInvalidaException e) {
        }
    }
}
```

Cuadro 34. Código par probar la clase *Identificacion*



Nombre	Contraseña	Rol
pepe	contraseña de pepe	Administrador

Figura 76. Registro con información de un usuario en la tabla *Usuarios*

9.3. Pruebas de caja blanca

Las pruebas de caja blanca realizan un seguimiento del código fuente según se van ejecutando los casos de prueba, de manera que se pueda determinar de manera concreta qué instrucciones, condiciones, etc. se han recorrido. Así, si los casos de prueba no han recorrido la rama *else* de una instrucción condicional, será necesario construir nuevos casos de prueba hasta que se pase por ella. A estos criterios por los que nos guiamos para determinar qué instrucciones han recorrido los casos de prueba se los denomina *criterios de cobertura* y, entre otros, suelen utilizarse los siguientes:

- Sentencias: número de sentencias ejecutables que se han ejecutado.
- Decisiones: número de decisiones ejecutadas, considerando que se ha ejecutado una decisión cuando se han recorrido todas sus posibles ramas (la que la hace *true* y la que la hace *false*, pero también todas las posibles ramas de un *switch*).
- Condiciones: número de condiciones ejecutadas, considerando que se ha ejecutado una condición cuando se han ejecutado todas sus correspondientes ramas con todas las posibles variantes de la instrucción condicional.
- Caminos: número de caminos linealmente independientes que se han ejecutado en el grafo de flujo de la unidad que se está probando. El número de caminos linealmente independientes coincide con la complejidad ciclomática de McCabe.
- Métodos: número de métodos que han sido llamados.
- Llamadas: número de llamadas a funciones y procedimientos que se han ejecutado. No debe confundirse con la cobertura de funciones: en la cobertura de funciones contamos cuántas funciones de las que hay en nuestro programa han sido llamadas, mientras que la cobertura de llamadas cuenta cuántas de las llamadas a funciones que hay en el programa se han ejecutado.
- Bucles: número de bucles que han sido ejecutados cero veces (excepto para bucles *do..while*), una vez y más de una vez.

9.3.1 Pruebas de caja blanca 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 *). Así, el mutante representa una versión defectuosa del programa original: es decir, el mutante es el programa original, pero con un fallo. El objetivo de las pruebas utilizando mutación consiste en construir ca-

sos de prueba que descubran el fallo existente en cada mutante. Así pues, los casos de prueba serán buenos cuando, al ser ejecutados sobre el programa original y sobre los mutantes, la salida de éstos difiera de la salida de aquél, ya que esto significará que las instrucciones mutadas (que contienen el fallo) han sido alcanzadas.

Los mutantes se generan aplicando “operadores de mutación”, que realizan modificaciones en el código de la clase que se está probando. En la Tabla 17 se muestran algunos operadores habituales. El mismo operador de mutación puede ser aplicado varias veces en un mismo programa, por lo que el número de mutantes que se genera para una clase dada puede ser muy grande.

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 17. Algunos operadores de mutación

9.3.2 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.

9.3.3 Pruebas de caja blanca de *Identificación*

MuJava es una herramienta que permite la realización de pruebas mediante mutación de clases en lenguaje Java. Se aplica en dos pasos: generación de mutantes (la Figura 77 muestra los generados para la clase *Sesion*) y ejecución de casos de prueba sobre la clase original y sobre los mutantes.

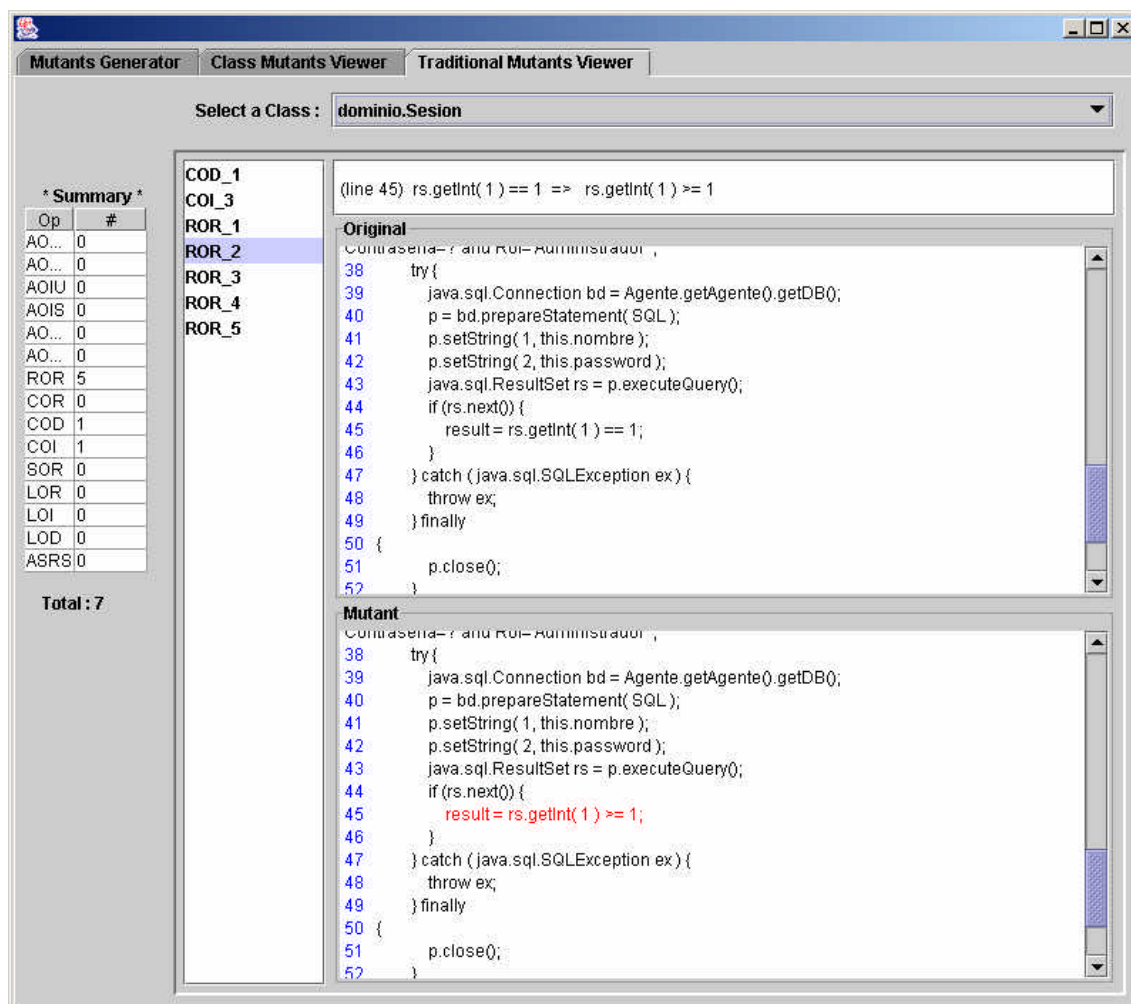


Figura 77. Mutantes generados para la clase *Sesion*

MuJava aplica dos tipos de operadores de mutación (tradicionales y de clase): los primeros son del estilo de los indicados en la Tabla 17, mientras que los segundos siembran los fallos aprovechando las características de los lenguajes de programación orientados a objetos (por ejemplo: eliminación en la subclase de un método que se hereda de una superclase).

Como casos de prueba, utilizaremos los dos utilizados para descubrir fallos (Cuadro 34), sólo que transcritos a la notación utilizada por MuJava: los casos de prueba de MuJava son métodos que, al igual que los de JUnit, ejecutan funcionalidades de la clase que se está probando, pero:

1. Devuelven una cadena, que puede entenderse como la representación del objeto resultante de la ejecución del caso de prueba.
2. No tienen código de comprobación de aserciones.

El siguiente cuadro muestra la clase con los dos casos de prueba, ahora en formato MuJava.

```
package tests;

import java.sql.SQLException;

import dominio.Sesion;
import dominio.exceptions.PasswordInvalidaException;

public class MujavaTests {
    public String testSesionCorrecta() {
        try {
            Sesion s=new Sesion("pepe", "contraseña de pepe");
            return "pepe" + "contraseña de pepe";
        } catch (SQLException e) {
            return "Se ha lanzado una SQLException inesperada";
        }
        catch (PasswordInvalidaException e) {
            return "Se ha lanzado una PasswordInvalidaException inesperada";
        }
    }

    public String testSesionIncorrecta() {
        try {
            Sesion s=new Sesion("pepe", "falsa contraseña de pepe");
            return "pepe" + "falsa contraseña de pepe";
        } catch (SQLException e) {
            return "Se ha lanzado una SQLException inesperada";
        } catch (PasswordInvalidaException e) {
            return "Se ha lanzado una PasswordInvalidaException esperada";
        }
    }
}
```

Cuadro 35. Código con los dos métodos de prueba

La Figura 78 muestra los resultados de la ejecución de los casos de prueba anteriores: de los siete mutantes tradicionales generados queda uno vivo, y dos de entre los mutantes de clase. Si asumimos que el nivel de cobertura debe ser el 100%, entonces debemos, de acuerdo con el proceso descrito en la sección 9.1, construir nuevos casos de prueba para matar todos los mutantes.

Puede ocurrir, sin embargo, que los mutantes que quedan vivos sean funcionalmente equivalentes al programa original: es decir, que no exista ningún caso de prueba que mate al mutante. Como se ve en la figura siguiente, el mutante ROR_2 es el único que queda vivo. Comparando el código del mutante con el del programa original (para lo cual ayuda la Figura 77), y teniendo en cuenta la implementación de la tabla *Usuarios* (dada en la Figura 43, página 99), en la que la columna *Nombre* es clave primaria, podemos concluir que ROR_2 es funcionalmente equivalente a la clase *Sesion* original. Podemos hacer la misma afirmación de los dos mutantes de clase que han quedado vivos, por lo que podríamos dar por concluidas las pruebas de la clase *Sesion*.

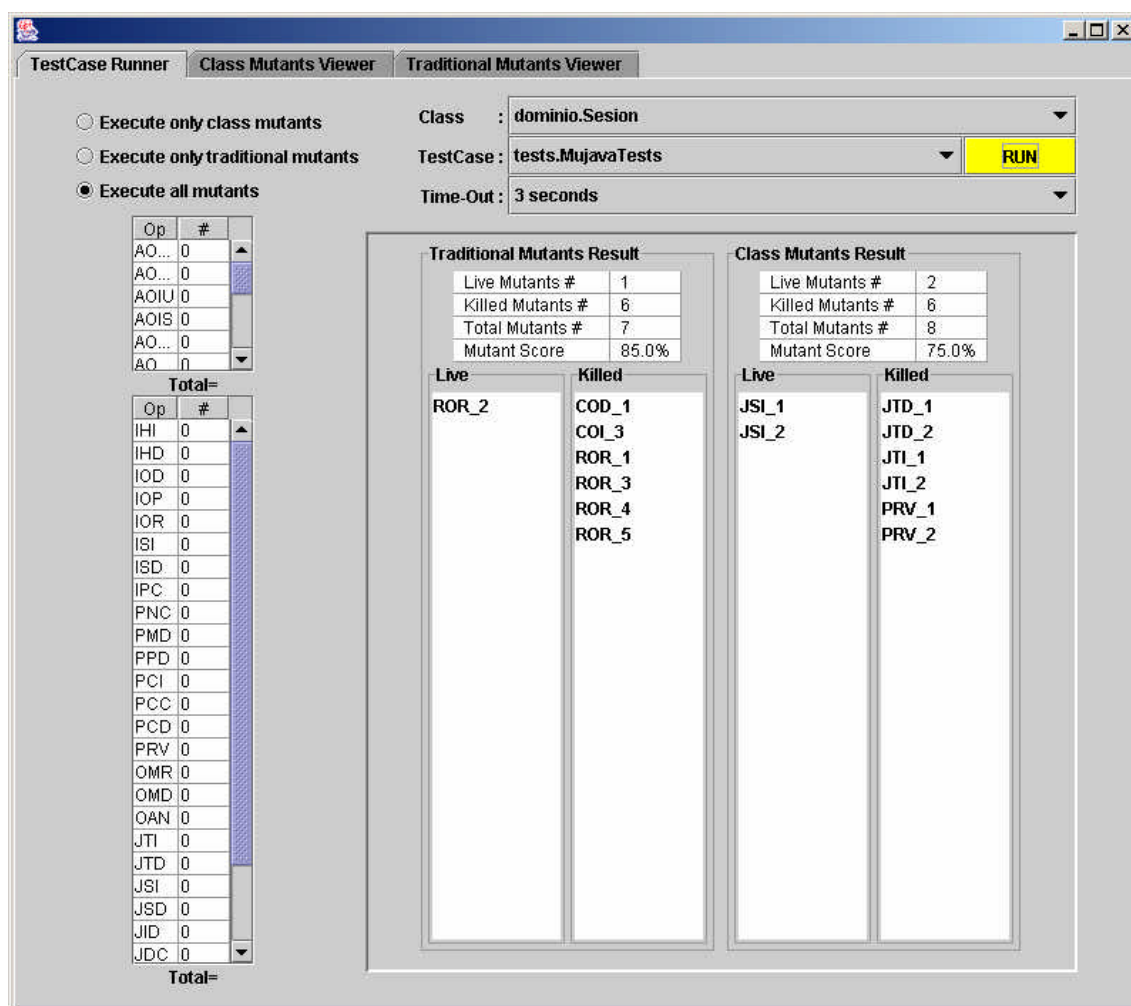
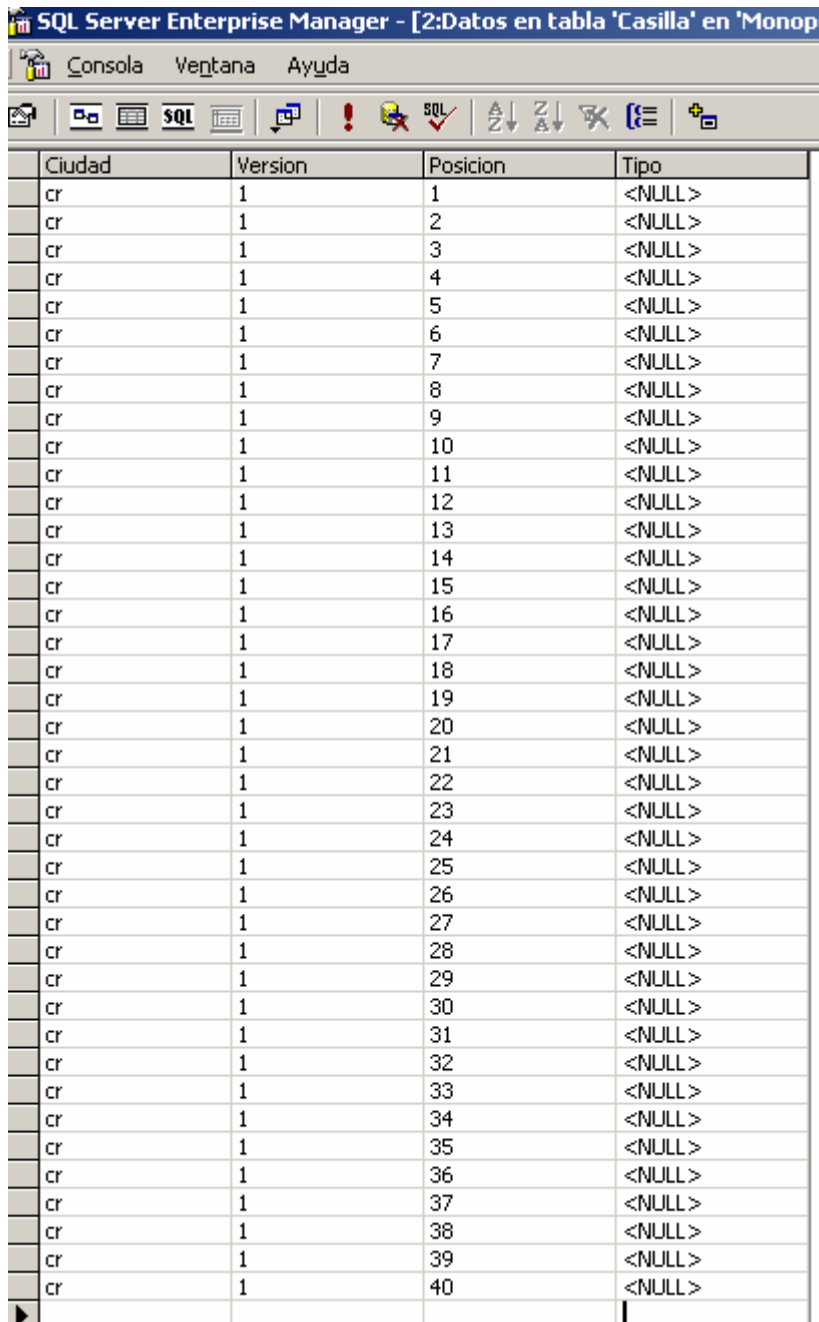


Figura 78. Resultados de la ejecución de los casos de prueba

9.4. Pruebas del caso de uso *Crear páking*

Utilizaremos los flujos normal y alternativo para preparar las pruebas de este caso de uso.

Para el flujo normal, partiremos de que la base de datos ya dispone de la información de la edición recién introducida en la tabla *Edición* y en la tabla *Casilla*. Para este primer flujo, asumimos que ninguna de las casillas tiene tipo asignado: es decir, el valor almacenado en la columna *Tipo* de todas las filas de la tabla *Tipo* es *null*. La Figura 79 muestra los datos de la tabla *Casilla* para la edición de la ciudad CR con versión 1.



Ciudad	Version	Posicion	Tipo
cr	1	1	<NULL>
cr	1	2	<NULL>
cr	1	3	<NULL>
cr	1	4	<NULL>
cr	1	5	<NULL>
cr	1	6	<NULL>
cr	1	7	<NULL>
cr	1	8	<NULL>
cr	1	9	<NULL>
cr	1	10	<NULL>
cr	1	11	<NULL>
cr	1	12	<NULL>
cr	1	13	<NULL>
cr	1	14	<NULL>
cr	1	15	<NULL>
cr	1	16	<NULL>
cr	1	17	<NULL>
cr	1	18	<NULL>
cr	1	19	<NULL>
cr	1	20	<NULL>
cr	1	21	<NULL>
cr	1	22	<NULL>
cr	1	23	<NULL>
cr	1	24	<NULL>
cr	1	25	<NULL>
cr	1	26	<NULL>
cr	1	27	<NULL>
cr	1	28	<NULL>
cr	1	29	<NULL>
cr	1	30	<NULL>
cr	1	31	<NULL>
cr	1	32	<NULL>
cr	1	33	<NULL>
cr	1	34	<NULL>
cr	1	35	<NULL>
cr	1	36	<NULL>
cr	1	37	<NULL>
cr	1	38	<NULL>
cr	1	39	<NULL>
cr	1	40	<NULL>

Figura 79. Para esta prueba, requerimos estos datos en la tabla *Casilla*

Utilizaremos JUnit para crear los casos de prueba correspondientes al flujo de eventos normal. El código de la clase con el caso de prueba se muestra en el Cuadro 36. El método *testSetTipo* instancia en primer lugar la edición con la que vamos a hacer las pruebas llamando a su constructor materializador; a continuación, recupera la casilla vigésima y le asigna el tipo *Parking*.

```

package tests;

import java.sql.SQLException;
import dominio.Casilla;
import dominio.Edicion;
import dominio.Parking;
import junit.framework.TestCase;

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

    public void testSetTipo() {
        try {
            Edicion e=new Edicion("cr", "1");
            Casilla c=e.getCasilla(20);
            c.setTipo("P rking");
            assertTrue(c.getTipo() instanceof Parking);
            assertTrue(e.numeroDeCasillasDeTipo("P rking")==1);
        } catch (SQLException e) {
            fail("Fallo");
        }
    }
}

```

Cuadro 36. C digo JUnit para la prueba del flujo normal

El constructor materializador de *Edicion* (Cuadro 37) crea un objeto de esta clase a partir de la informaci n almacenada en la base de datos. Como una edici n contiene casillas, su implementaci n incluye la materializaci n de todas  stas mediante una llamada al m todo *loadCasillas*, que se muestra en el Cuadro 38.

```

public Edicion(String ciudad, String version) throws SQLException {
    PreparedStatement p=null;
    String SQL="Select Ciudad, Version from Edicion where Ciudad=? and Version=?";
    try {
        Connection bd=Agente.getAgente().getDB();
        p=bd.prepareStatement(SQL);
        p.setString(1, ciudad);
        p.setString(2, version);
        ResultSet rs=p.executeQuery();
        if (rs.next()) {
            this.ciudad=rs.getString(1);
            this.version=rs.getString(2);
            this.loadCasillas();
        } else throw new SQLException("Edici n no encontrada");
    }
    catch (SQLException ex) {
        throw ex;
    }
    finally {
        p.close();
    }
}

```

Cuadro 37. Constructor materializador de la clase *Edicion*

El m todo *loadCasillas* recupera de la tabla *Casilla* todos los registros correspondientes a la instancia de *Edicion* que ejecuta la operaci n. Como se observa en el c digo de la instrucci n SQL, se leen los valores de las columnas *Posicion* y *Tipo*; en funci n del valor de esta columna, el tipo del objeto *Casilla* debe instanciarse al subtipo correspondiente. De esta asignaci n se encarga la operaci n *buildCasilla*, que consiste realmente en un m todo de materializaci n de instancias. Se podr a haber optado por implementar un

constructor materializador también en *Casilla*, pero por razones didácticas se ha preferido crear esta operación estática (Cuadro 39).

```
private void loadCasillas() throws SQLException {
    casillas=new Vector();
    PreparedStatement p=null;
    String SQL="Select Posicion, Tipo from Casilla where Ciudad=? and Version=?";
    try {
        Connection bd=Agente.getAgente().getDB();
        p=bd.prepareStatement(SQL);
        p.setString(1, this.ciudad);
        p.setString(2, this.version);
        ResultSet rs=p.executeQuery();
        while (rs.next()) {
            int posicion=rs.getInt(1);
            String tipo=rs.getString(2);
            Casilla casilla=Casilla.buildCasilla(posicion, tipo);
            casilla.setEdicion(this);
            casillas.add(casilla);
        }
    }
    catch (SQLException ex) {
        throw ex;
    }
    finally {
        p.close();
    }
}
```

Cuadro 38. Método encargado de materializar las casillas de una edición (en la clase *Edicion*)

```
public static Casilla buildCasilla(int posicion, String nombreDelTipo) {
    Casilla result=new Casilla(posicion);
    Tipo tipo=null;
    if (nombreDelTipo!=null) {
        if (nombreDelTipo.equals("P rking")) {
            tipo=new Parking();
        }
        result.setTipo(tipo);
    }
    return result;
}
```

Cuadro 39. Un materializador de casillas, que decide el tipo en funci n de un par metro (en la clase *Casilla*)

Regresando a la discusi n sobre el c digo del caso de prueba, tras la asignaci n del tipo a la casilla se realizan dos comprobaciones: en primer lugar, que el tipo asignado a la casilla vig sima es de clase *Parking*; en segundo lugar, que la edici n posee s lo una casilla de ese tipo. Tal y como se ha planteado, la segunda comprobaci n exige la creaci n de la operaci n *numeroDeCasillasDeTipo(String tipoBuscado)*, que devuelve el n mero de casillas de un cierto tipo que hay en el tablero. El c digo de esta operaci n, que se a ade a la clase *Edicion*, se muestra en el Cuadro 41; no obstante, puesto que es una operaci n cuyo prop sito es facilitar las pruebas en JUnit, se podr a haber creado en la clase *TestCasilla* con la siguiente cabecera:

```
public int numeroDeCasillasDeTipo(Edicion e, String tipoBuscado) throws SQLException
```

Cuadro 40. Cabecera alternativa de una operaci n de apoyo a las pruebas (en *TestCasilla*)

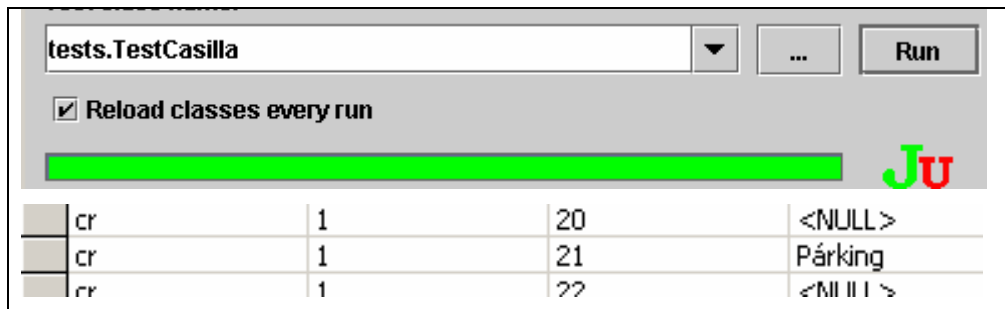
```

public int numeroDeCasillasDeTipo(String tipoBuscado) throws SQLException {
    PreparedStatement p=null;
    String SQL="Select count(*) from Casilla where Ciudad=? and Version=? and Tipo=?";
    int result=0;
    try {
        Connection bd=Agente.getAgente().getDB();
        p=bd.prepareStatement(SQL);
        p.setString(1, ciudad);
        p.setString(2, version);
        p.setString(3, tipoBuscado);
        ResultSet rs=p.executeQuery();
        if (rs.next()) {
            result=rs.getInt(1);
        } else throw new SQLException("Edición o tipo no encontrado");
    }
    catch (SQLException ex) { throw ex; }
    finally { p.close(); }
    return result;
}

```

Cuadro 41. Implementación de una operación con el fin de facilitar las pruebas (en Edición)

La clase supera la ejecución del caso de prueba. La siguiente figura muestra el resultado en JUnit y en la base de datos.



Casilla	Edición	Versión	Tipo	Resultado
cr	1	20	<NULL>	
cr	1	21	Parking	
cr	1	??	<NULL>	

Figura 80. Superación del caso de prueba

De acuerdo con la especificación del flujo alternativo, si optamos por cambiar la ubicación de la casilla del parking, debe eliminarse la ubicación anterior y establecerse la nueva. Es decir: si primero colocamos el parking en la casilla 20ª y luego en la 21ª, el tipo de la 20ª debe establecerse a null.

El código de este nuevo caso de prueba, que añadimos a la clase *Test-Casilla*, se muestra en el siguiente cuadro. Obsérvese que ahora, añadimos a los métodos *assert* un mensaje con una descripción del error encontrado.

```

public void testSetTipo2() {
    try {
        Edicion e=new Edicion("cr", "1");
        Casilla c20=e.getCasilla(20);
        c20.setTipo("Parking");
        Casilla c21=e.getCasilla(21);
        c21.setTipo("Parking");
        assertTrue("A la casilla 20 no se le quita el tipo", c20.getTipo()==null);
        assertTrue("La casilla 21 no es de tipo Parking",
            c21.getTipo() instanceof Parking);
        assertTrue("Hay más de un Parking", e.numeroDeCasillasDeTipo("Parking")==1);
    } catch (SQLException e) {
        fail("Fallo");
    }
}

```

Cuadro 42. Código para la prueba del flujo de eventos alternativo

El caso de prueba encuentra un error, y JUnit muestra el mensaje correspondiente (Figura 81).

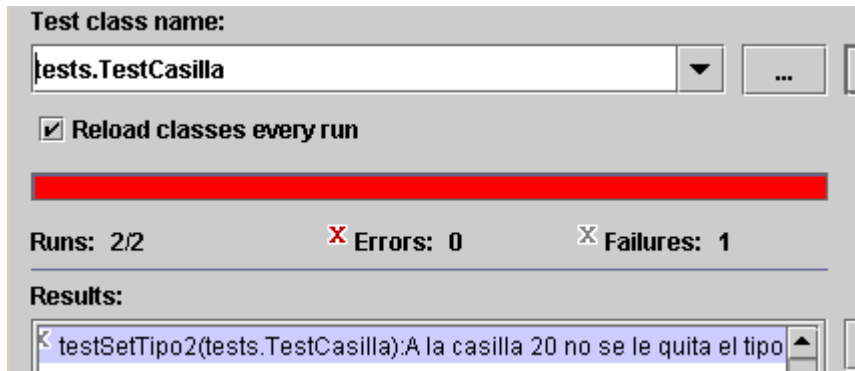


Figura 81. El segundo caso de prueba encuentra un error

Tras una depuración del código, observamos que el error se encuentra en la implementación del método *update* de *Casilla*, que mostrábamos en el Cuadro 25 (página 128): este método borra la casilla y luego la inserta con su nuevo tipo; pero, en el caso de la casilla de tipo *Parking*, puesto que sólo puede haber una en todo el tablero, tiene también que borrar la antigua casilla del *parking*. Es decir, que el escenario de ejecución debe ser el siguiente:

- 1) Partimos de una edición en la que una casilla (por ejemplo, la 20) es de tipo *Parking*.
- 2) Elegimos otra casilla (por ejemplo, la 21).
- 3) Marcamos la casilla 21 como de tipo *Parking*.
- 4) Se elimina la casilla 21 de la tabla *Casilla*.
- 5) Se elimina el tipo de la casilla 20. Esto supone establecer a *null* la columna *Tipo* de la tabla *Casilla*, y borrar el registro de la tabla *Parking*.
- 6) Se inserta la casilla 21 como de tipo *Parking*. Esto supone asignar el valor "Parking" a la columna *Tipo* de la tabla *Casilla* e insertar un registro en la tabla *Parking*.

La corrección de este error supone la modificación del código de varias clases. En particular, el método *setTipo* de *Casilla* pasa a ser el del siguiente cuadro, en el que primero se recupera la casilla del *parking* de esta edición (método *getCasillaDeParking*, mostrado en el Cuadro 44); si la hay, se le quita el tipo y se asigna el tipo *Parking* a la nueva casilla; por último, se delega al *tipo* la actualización de la base de datos.

```
public void setTipo(String nombreDelTipo) throws SQLException {
    Casilla parking=this.edicion.getCasillaDeParking();
    if (parking!=null)
        parking.tipo=null;
    this.tipo=new Parking();
    this.tipo.setCasilla(this);
    this.tipo.update();
}
```

Cuadro 43. Nueva implementación de *setTipo* en *Casilla*


```

public Casilla getCasillaDeParking() {
    for (int i=0; i<this.casillas.size(); i++) {
        Casilla c=(Casilla) this.casillas.get(i);
        if (c.getTipo() instanceof Parking)
            return c;
    }
    return null;
}

```

Cuadro 44. Implementación de *getCasillaDeParking* (en Edición)

La delegación mencionada ejecuta los pasos mencionados arriba. En *Tipo*, la operación *update* es abstracta, y habrá que implementarla en sus especializaciones: en el caso de la clase *Parking*, el código es el del Cuadro 45.

```

public void update() throws SQLException {
    PreparedStatement p=null;
    try {
        // En la tabla Casilla, quitamos el tipo Párking de la casilla que lo tuviera
        String SQL="Update Casilla set Tipo=null where Ciudad=? and Version=? " +
            "and Tipo=?";
        Connection bd=Agente.getAgente().getDB();
        p=bd.prepareStatement(SQL);
        p.setString(1, casilla.getCiudad());
        p.setString(2, casilla.getVersion());
        p.setString(3, "Párking");
        p.executeUpdate();
        // En la tabla Casilla, indicamos que el tipo de la nueva casilla es Párking
        SQL="Update Casilla set Tipo=? where Ciudad=? and Version=? and Posicion=?";
        p=bd.prepareStatement(SQL);
        p.setString(1, "Párking");
        p.setString(2, casilla.getCiudad());
        p.setString(3, casilla.getVersion());
        p.setInt(4, casilla.getPos());
        p.executeUpdate();
        // Borramos la antigua casilla de Párking de la tabla Párking
        SQL="Delete from Parking where Ciudad=? and Version=?";
        p=bd.prepareStatement(SQL);
        p.setString(1, casilla.getCiudad());
        p.setString(2, casilla.getVersion());
        p.executeUpdate();
        // Guardamos la nueva casilla de Párking en la tabla Párking
        SQL="Insert into Parking (Ciudad, Version, Posicion) values (?, ?, ?)";
        p=bd.prepareStatement(SQL);
        p.setString(1, casilla.getCiudad());
        p.setString(2, casilla.getVersion());
        p.setInt(3, casilla.getPos());
        p.executeUpdate();
    }
    catch (SQLException ex) { throw ex; }
    finally { p.close(); }
}

```

Cuadro 45. Implementación de la operación *update* en la clase *Parking*, subclase de *Tipo*, en donde *update* es abstracta

Ahora, los dos casos de prueba no encuentran fallos, por lo que estaríamos en disposición de pasar a realizar las pruebas de caja blanca correspondientes.

A pesar de lo dicho, la operación *update* mostrada en el último cuadro es, en cierto modo, insegura, ya que podría producirse un error en una de las instrucciones SQL que interrumpiera la ejecución completa del método, dejando la base de datos en un estado inconsistente. Algunas posibles soluciones pasarían por la ejecución de las cuatro instrucciones SQL en una sola transacción, o bien por la delegación de la responsabilidad a la base de datos,

mediante la creación de procedimientos almacenados o disparadores. Los procedimientos almacenados son operaciones guardadas en el gestor de base de datos a las que se pueden pasar parámetros, que son ejecutadas por el propio gestor y a las que se las puede llamar desde programas externos.

En nuestro caso, podríamos crear el procedimiento almacenado de la Figura 82, que viene a realizar las mismas acciones que el método *update* mostrado en el cuadro anterior:

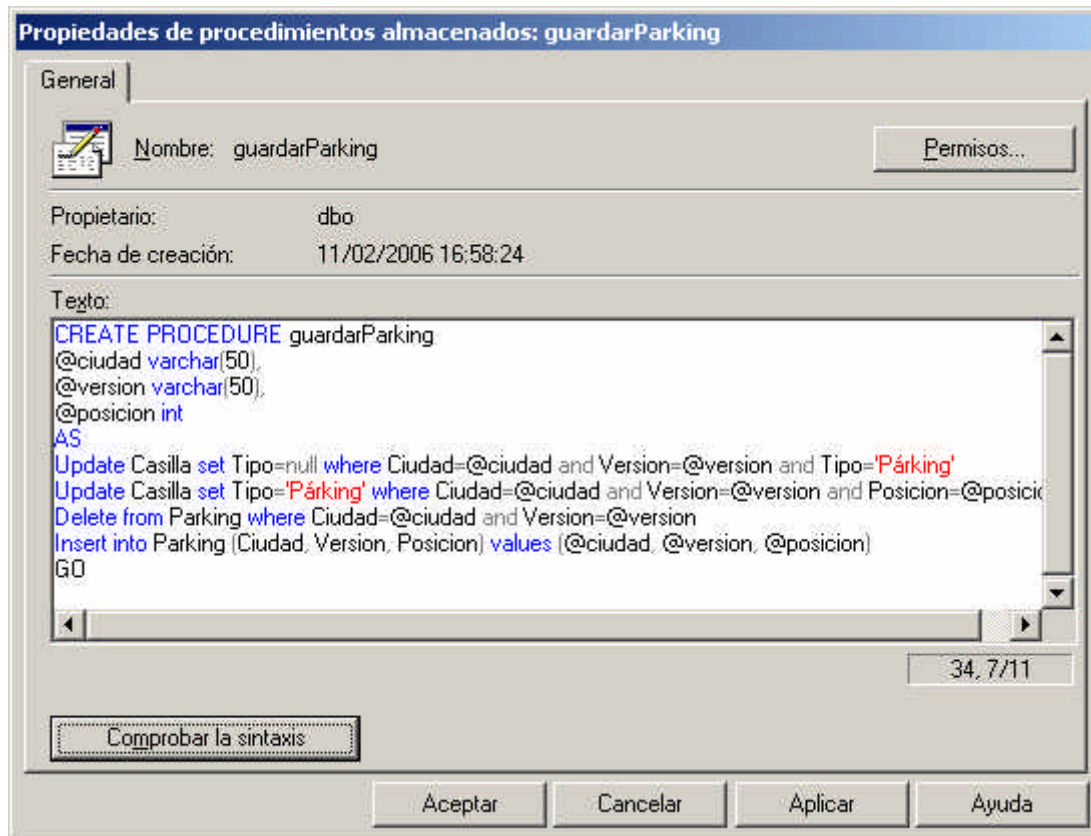


Figura 82. Un procedimiento almacenado

Todas las operaciones incluidas en el procedimiento almacenado se ejecutan dentro de una transacción, por lo que, en caso de que falle una de ellas, se deshacen todos los cambios y el estado de la base de datos no se altera. La implementación del método *update* debería modificarse: ahora, en lugar de embeber el código SQL en el programa y utilizar objetos de tipo *PreparedStatement*, recuperaremos una referencia al procedimiento almacenado mediante una instancia de *CallableStatement*, a la que pasaremos los parámetros correspondientes. Esta implementación alternativa se muestra en la Figura 83.

```
public void update() throws SQLException {
    CallableStatement cs=null;
    try {
        Connection bd=Agente.getAgente().getDB();
        cs=bd.prepareCall("{call guardarParking (?, ?, ?)}");
        cs.setString(1, this.casilla.getCiudad());
        cs.setString(2, this.casilla.getVersion());
        cs.setInt(3, this.casilla.getPos());
        cs.executeUpdate();
    }
    catch (SQLException ex) { throw ex; }
    finally { cs.close(); }
}
```

Figura 83. Llamada al procedimiento almacenado de la Figura 82

Al hacer esta modificación, se deben reejecutar los casos de prueba para comprobar que no se han introducido nuevos errores.

El proceso de pruebas, evidentemente, se repetiría para el resto de clases implementadas en esta iteración.

10. Lecturas recomendadas

Mantener la correspondencia entre el mundo de tablas relacionales y sus registros con el mundo de clases y sus instancias ha producido y sigue produciendo ríos de tinta. En la página <http://www.objectarchitects.de/> pueden consultarse algunas propuestas. En particular, el artículo titulado “Persistence Options for Object-Oriented Programs”, de Wolfgang Keller, presenta de forma resumida algunas de ellas. De todos modos, no se quede el lector únicamente en esa lectura, y dése un paseo por esta web.

En <http://www.inf-cr.uclm.es/www/mpolo/> se presenta una estrategia de generación en tiempo de ejecución del código de las operaciones CRUD utilizando programación reflexiva (también conocida como programación introspectiva, o simplemente reflexión o *reflection*).

Capítulo 6. APLICACIÓN DE ADMINISTRACIÓN:

FASE DE ELABORACIÓN (II)

En este capítulo se desarrollan los casos de uso *Crear salida*, *Crear impuestos* y *Crear calle*, ubicados en esta fase según el plan de iteraciones. El capítulo también presenta OCL (*Object Constraint Language*), un lenguaje de restricciones para UML que permite describir con más formalidad y menos ambigüedades los sistemas software.

1. Desarrollo del caso de uso *Crear salida*

La descripción textual de este caso de uso es muy similar a la de *Crear parking*, ya que el juego del Monopoly sólo permite la existencia de una casilla de este tipo en cada tablero. Así, podemos reutilizar prácticamente la totalidad del análisis y diseño realizados para ese caso de uso; gracias a la herencia, bastante de su implementación, incluyendo los casos de prueba.

Nombre: Crear salida
Abstracto: no
Precondiciones: 1. Hay una edición del juego que se está creando
Postcondiciones:
Rango: 5
Flujo normal: <i>No había casilla de Salida en la edición</i> 1. En una ventana como la mostrada en la sección de Descripción, el usuario selecciona el número de la casilla en la que desea situar la casilla de Salida. 2. La ventana muestra en la etiqueta situada en arriba a la izquierda el número de la casilla seleccionada. 3. El usuario elige la solapa Salida 4. El usuario pulsa el botón guardar. 5. La ventana, que conoce a la Edición que se está creando, le dice a ésta que el tipo de la casilla seleccionada es Salida 7. La Edición comprueba que aún no dispone de casilla tipo Salida 8. La Edición asigna a la <i>Casilla</i> el tipo Salida 9. La casilla actualiza su tipo en la base de datos a través del Agente
Flujo alternativo 1: <i>Ya había una casilla de Salida en la edición</i> 1. En una ventana de edición del tablero, el usuario selecciona el número de la casilla en la que desea situar la Salida. 2. La ventana muestra en la etiqueta situada en arriba a la izquierda el número de la casilla seleccionada. 3. El usuario elige la solapa Salida 4. El usuario pulsa el botón guardar. 5. La ventana, que conoce a la Edición que se está creando, le dice a ésta que el tipo de la casilla seleccionada es Salida. 7. La Edición comprueba que ya dispone de casilla tipo Salida. 8. La Edición elimina de la base de datos (a través del Agente) su casilla de Salida. 8. La Edición asigna a la <i>Casilla</i> seleccionada en el paso 1 el tipo Salida 9. La casilla actualiza su tipo en la base de datos a través del Agente
Descripción:

Tabla 18. Descripción textual del caso de uso *Crear salida*

El diseño del sistema necesita la adición de la clase *Salida* como especialización de *Tipo* (Figura 84), así como la creación de la solapa correspondiente para la ventana, en la que será necesario escribir el código que proceda.

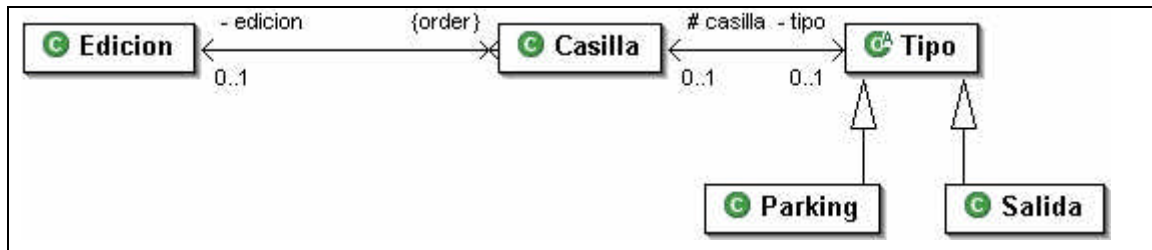


Figura 84. Se crea la clase *Salida* como especialización de *Tipo*

El lado izquierdo del Cuadro 46 muestra el código del método que se ejecuta al pulsar el botón Guardar en la solapa correspondiente a la casilla de salida: únicamente establece el tipo de la casilla al texto “Salida” mediante una llamada a *setTipo* de *Casilla*; en el lado derecho se muestra la implementación de este método en *Casilla* (que ya ofrecíamos en el Cuadro 43, página 148). Este método ahora deja de ser válido, ya que establece de manera fija el tipo a *Parking*.

<pre> protected void guardarCasilla() { try { this.casilla.setTipo("Salida"); } catch (Exception e) { ... } } </pre>	<pre> public void setTipo(String nombreDelTipo) throws SQLException { Casilla parking= this.edicion.getCasillaDeParking(); if (parking!=null) parking.tipo=null; this.tipo=new Parking(); this.tipo.setCasilla(this); this.tipo.update(); } </pre>
--	--

Cuadro 46. Código que se ejecuta al pulsar el botón Guardar en *JPSalida* (la solapa correspondiente a la casilla de salida)

El método *setTipo* debe modificarse. Una forma de hacerlo es añadiendo una serie de instrucciones condicionales *if* de manera que se ejecuten unas u otras operaciones en función del valor del parámetro *nombreDelTipo* (Cuadro 47). Como se observa en el segundo *if*, es preciso añadir a la clase *Edicion* la operación *getCasillaDeSalida*, cuyo código será muy parecido al de *getCasillaDeParking*, que se ofrecía en el Cuadro 44 (página 149). También es necesario modificar el código de *buildCasilla* (Cuadro 39) para que materialice instancias de tipo *Salida*.

```

public void setTipo(String nombreDelTipo) throws SQLException {
    if (nombreDelTipo.equals("P rking")) {
        Casilla parking=this.edicion.getCasillaDeParking();
        if (parking!=null)
            parking.tipo=null;
        this.tipo=new Parking();
    } else if (nombreDelTipo.equals("Salida")) {
        Casilla salida=this.edicion.getCasillaDeSalida();
        if (salida!=null)
            salida.tipo=null;
        this.tipo=new Salida();
    }
    this.tipo.setCasilla(this);
    this.tipo.update();
}

```

Cuadro 47. Nueva implementaci n de *setTipo* (en *Casilla*)

Por otro lado, habr  que implementar el m todo *update* en la clase *Parking*, y crear el correspondiente procedimiento almacenado *guardarSalida* en la base de datos.

Para finalizar, crearemos casos de prueba nuevos que prueben esta funcionalidad, y reejecutaremos todos los que llevamos contruidos para comprobar que las modificaciones no han dado lugar a la introducci n de nuevos errores.

2. Desarrollo del caso de uso *Crear impuestos*

Este caso de uso tiene una diferencia importante con respecto a los dem s, y es que una edici n debe tener dos casillas de impuestos, cada una con el nombre del impuesto y un importe. En la descripci n textual del caso de uso, podemos a adir como precondici n el hecho de que no la edici n no tenga dos casillas de impuestos.

Nombre: Crear impuestos
Abstracto: no
Precondiciones: 1. Hay una edici�n del juego que se est� creando 2. La edici�n tiene una o ninguna casillas de impuestos
Postcondiciones:
Rango: 5
Flujo normal: 1. En la ventana de definici�n del tablero, el usuario selecciona el n�mero de la casilla en la que desea situar la casilla de Impuestos, le asigna un nombre y un importe. 2. La ventana muestra en la etiqueta situada en arriba a la izquierda el n�mero de la casilla seleccionada. 3. El usuario elige la solapa Impuestos 4. El usuario pulsa el bot�n guardar. 5. La ventana, que conoce a la Edici�n que se est� creando, le dice a �sta que el tipo de la casilla seleccionada es Impuestos 6. La Edici�n asigna a la <i>Casilla</i> el tipo Impuestos, asign�ndole el nombre del impuesto y el tipo 9. La casilla actualiza su tipo en la base de datos a trav�s del Agente
Descripci�n:

Tabla 19. Descripci n textual del caso de uso *Crear impuestos*

Comenzaremos este caso de uso ilustrando la forma en que un entorno de desarrollo moderno, como Eclipse, puede facilitar la escritura del c digo.

Partiendo del diagrama de clases del lado izquierdo de la siguiente figura, dibujamos la relaci n de herencia desde *Impuestos* hacia *Tipo*. Al tra-

zar la línea (lado derecho, arriba), el entorno de desarrollo añade el texto *extends Tipo* a la cabecera de la clase *Impuestos*, subrayando “Impuestos” porque la clase, al haberse convertido en subclase de *Tipo* y no ser abstracta, debe implementar las operaciones abstractas heredadas de *Tipo*. El entorno sugiere varias soluciones, entre ellas dos interesantes: añadir los métodos heredados de *Tipo* no implementados o convertir la clase *Impuestos* en abstracta. Optamos por la primera opción, de forma que se añaden las cabeceras de las operaciones al cuerpo de la clase *Impuestos*.

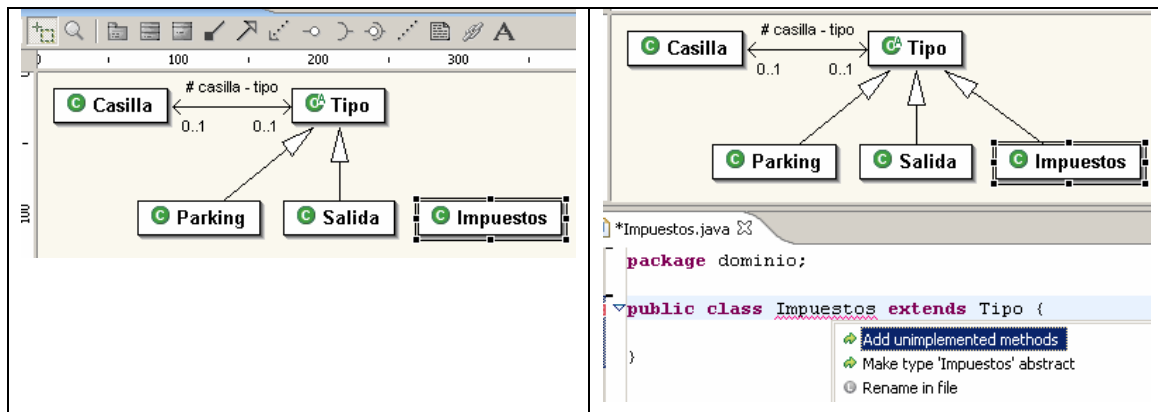


Figura 85. Un buen entorno de desarrollo facilita la escritura del código

Ahora, añadimos a la clase los campos *nombre* e *importe*, creamos la tabla *Impuestos* en la base de datos y la hacemos dependiente de *Casilla* (Figura 86). Obsérvese ahora que la nueva tabla incorpora dos columnas adicionales para representar los dos valores que aportan las casillas de impuestos.

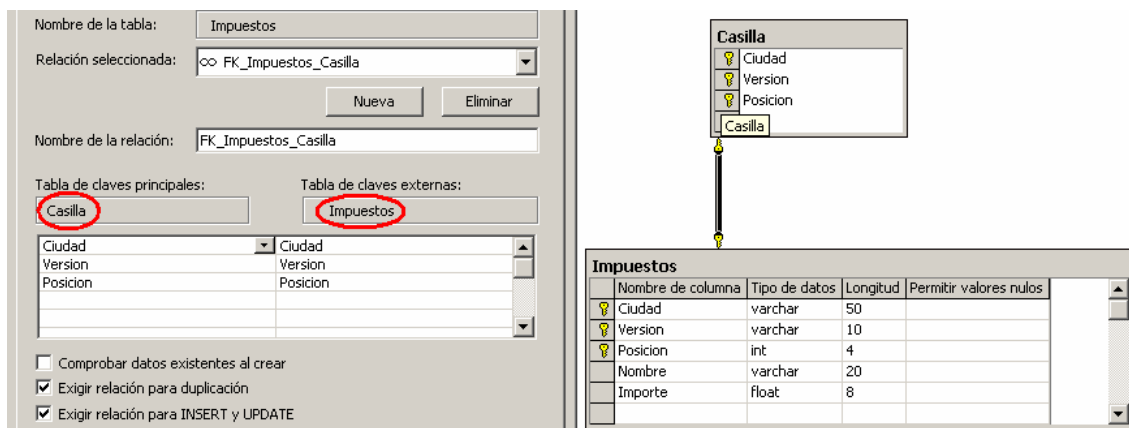


Figura 86. Adición de una nueva tabla a la base de datos

También debemos crear la solapa de creación de la casilla. La siguiente figura muestra el aspecto que decidimos darle y el código del método *guardarCasilla*, que hemos construido copiando y pegando de las solapas anteriores:

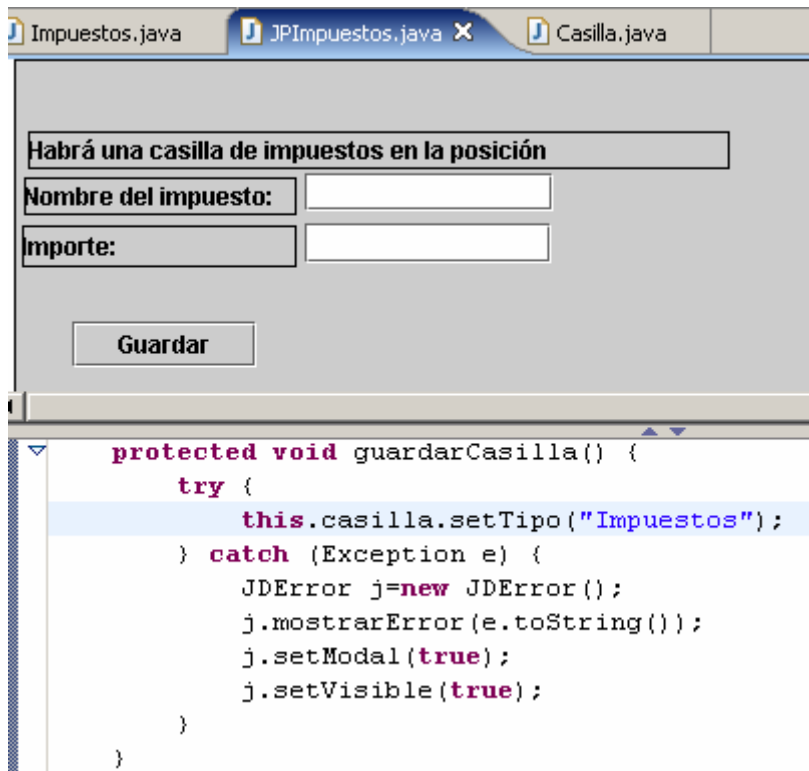


Figura 87. Diseño de la solapa *JPImpuestos* y parte de su código

El método *guardarCasilla* deja ahora de ser válido, ya que asigna a la casilla el tipo, pero no le pasa valores a los campos *nombre* e *importe*. Una mala solución sería crear las operaciones *setNombre(String)* y *setImporte(double)* en la clase *Casilla*: sería mala porque no todas las casillas requieren un nombre y un importe. El siguiente cuadro muestra una de las posibles implementaciones de esta operación: obsérvese que, primero, se establece el tipo, luego se recupera con el *cast* en la forma del subtipo *Impuestos*; entonces, se le asignan los valores a sus dos campos *nombre* e *importe*, y se actualizan sus valores en la base de datos.

```
protected void guardarCasilla() {
    try {
        this.casilla.setTipo("Impuestos");
        Impuestos tipo=(Impuestos) this.casilla.getTipo();
        tipo.setNombre(jtfNombreDelImpuesto.getText());
        double importe=Double.parseDouble(this.jtfImporte.getText());
        tipo.setImporte(importe);
        tipo.updateNombreEImporte();
    } catch (Exception e) {
        ...
    }
}
```

Cuadro 48. Implementación de *guardarCasilla* en *JPImpuestos*, la solapa correspondiente a las casillas de impuestos

updateNombreEImporte llama a un procedimiento almacenado que actualiza los valores de las columnas en la tabla *Impuestos* (Cuadro 49).

<pre> public void updateNombreEImporte() throws SQLException { CallableStatement cs=null; try { Connection bd= Agente.getAgente().getDB(); cs=bd.prepareCall("{call " + "actualizarNombreEImporte " + "(?, ?, ?, ?, ?)}"); cs.setString(1, casilla.getCiudad()); cs.setString(2, casilla.getVersion()); cs.setInt(3, this.casilla.getPos()); cs.setString(4, this.nombre); cs.setDouble(5, this.importe); cs.executeUpdate(); } catch (SQLException ex) { throw ex; } finally { cs.close(); } } </pre>	<pre> CREATE PROCEDURE actualizarNombreEImporte @ciudad varchar(50), @version varchar(50), @posicion int, @nombre varchar(20), @importe float AS Update Impuestos set Nombre=@nombre, Importe=@importe where Ciudad=@ciudad and Version=@version and Posicion=@posicion GO </pre>
--	---

Cuadro 49. Método de la clase *Impuestos* que llama al procedimiento almacenado del lado derecho

3. Introducción a OCL

El lenguaje OCL (*Object Constraint Language* o *Lenguaje de Restricciones sobre Objetos*) se utiliza como complemento a UML (de hecho, OCL es parte de la especificación 2.0 de UML). OCL aumenta la precisión y disminuye la ambigüedad de las descripciones de sistemas realizadas con UML.

Con OCL se pueden escribir los siguientes tres tipos de restricciones sobre modelos UML:

- Precondiciones: para un 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.

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 las siguientes secciones se presentan los tipos predefinidos en OCL para, más adelante, anotar con algunas restricciones un ejemplo.

3.1. Tipos predefinidos de OCL

En OCL existe el conjunto de tipos básicos que se muestra en la Tabla 20, y que son independientes del modelo de objetos que se esté desarrollando.

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	* (Real):Real floor():Real min(Real):Real
Boolean	and or xor not implies if-then-else-endif = <>		
String	concat(String):String substring(Integer, Integer):String toLower():String toReal():Real Además: = <>	size():Integer toInteger():Integer toUpper():String	

Tabla 20. Tipos básicos escalares y sus operaciones

Todos los tipos tanto del modelo UML como de OCL tienen un atributo *type* de un tipo especial de OCL llamado *OclType*, que permite acceder a las características del tipo. Haciendo un símil, el atributo *type:OclType* se corresponde con el método *getClass():Class* de la clase *Object* de Java, y *OclType* equivale a la propia clase *Class*.

OclType tiene los atributos y operaciones que se muestran en la Tabla 21.

Atributo u operación	Tipo	Resultado/descripción
Name	String	Nombre del tipo (en la clase <i>Persona</i> , por ejemplo, este atributo vale "Persona").
Attributes	Set(String) ⁶	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.). El resultado es un <i>Set</i> , un tipo de OCL que se describe a continuación.
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.
Operations	Set(String)	Conjunto formado por todas las operaciones de este tipo.
Supertypes	Set(OclType)	Conjunto formado por los supertipos directos de este tipo.
allSupertypes	Set(OclType)	Conjunto formado por todos los supertipos de este tipo.
allInstances	Set(type)	Conjunto de todas las instancias de <i>type</i> y de sus supertipos

Tabla 21. 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 (continuando con la equivalencia, sería algo parecido a la clase *Object* de Java). Por tanto, todos los miembros definidos en *OclAny* (Tabla 22) son accesibles en todos los tipos que vayamos definiendo en nuestro modelo.

⁶ Las expresiones de la Tabla 21 *Set(String)*, *Set(OclType)* y *Set(type)* denotan conjuntos cuyos elementos son, respectivamente, de los tipos *String*, *OclType* y *type*.

Atributo u operación	Tipo	Resultado/descripción
objeto = (x : OclAny)	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 : OclAny)	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	OclType	Tipo de este objeto
objeto.oclIsKindOf(type:OclType)	Boolean	Devuelve <i>true</i> si <i>type</i> es supertipo del tipo de <i>objeto</i> o el mismo tipo
objeto.oclIsTypeOf(type:OclType)	Boolean	Devuelve <i>true</i> si el tipo del objeto es <i>type</i>
objeto.oclInState(state:OclState)	Boolean	Devuelve <i>true</i> si el objeto está en el estado <i>state</i> . <i>state</i> es de tipo <i>OclState</i> , un tipo de OCL sin propiedades ni operaciones y cuyo único propósito es ser utilizado en esta operación.
objeto.oclIsNew()	Boolean	Devuelve <i>true</i> si el objeto se crea durante la ejecución de la operación. Sólo puede utilizarse en pre-condiciones

Tabla 22. Atributos y operaciones de *OclAny*

El tipo *OclState* (mencionado en la operación *oclInState* de la tabla anterior) se utiliza para representar estados de aquéllos en los que puede encontrarse una instancia del tipo que se está describiendo. *OclState* no tiene campos ni operaciones. En la sección 1.2 del Capítulo 8 (página 268) se explica cómo obtener especificaciones OCL a partir de máquinas de estados, utilizándose este tipo y la operación correspondiente.

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:

Atributo u operación	Tipo	Resultado/descripción
expr.evaluationType	OclType	Tipo del objeto obtenido al evaluar <i>expr</i>

Tabla 23. El único miembro de *OclExpression*

3.2. Colecciones

Existe también la posibilidad de manipular colecciones de elementos (por ejemplo, para hacer referencia al lado de multiplicidad “muchos” de una asociación), para lo que se definen el tipo abstracto *Collection* y sus subtipos *Set*, *Bag* y *Sequence*.

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

Operación	Tipo del resultado	Resultado/descripción
size	Integer	número de elementos
includes(x : OclAny)	Boolean	<i>true</i> si la colección incluye el parámetro
count(x : OclAny)	Integer	número de veces que el parámetro se encuentra incluido en la colección
includesAll(c : Collection)	Boolean	<i>true</i> si todos los elementos del parámetro están en la colección
isEmpty	Boolean	<i>true</i> si la colección está vacía
notEmpty	Boolean	<i>true</i> si la colección no está vacía
sum	Integer o Real	suma de los elementos de la colección, si éstos son de tipo <i>Integer</i> o <i>Real</i>
exists(expr:OclExpression)	Boolean	<i>true</i> si la colección incluye al menos un elemento que satisfaga la expresión pasada como parámetro
forall(expr : OclExpression)	Boolean	<i>true</i> si todos los elementos de la colección satisfacen la expresión pasada como parámetro
iterate(expr : OclExpression)	expr.evaluationType	realiza operaciones (descritas en la expresión pasada como parámetro) con todos los elementos de la colección

Tabla 24. Operaciones de *Collection*

El subtipo *Set* (conjunto) representa un conjunto de elementos en el que sólo puede haber un ejemplar de cada elemento.

Operación	Tipo devuelto	Resultado/descripción
union(s : Set(T))	Set(T)	conjunto formado por la unión de este conjunto y del pasado como parámetro
union(b : Bag(T))	Bag(T)	bolsa formada por la unión de este conjunto y de la bolsa pasada como parámetro
= (s : Set(T))	Boolean	<i>true</i> si los dos conjuntos contienen los mismo elementos
intersection(s : Set(T))	Set(T)	conjunto formado por la intersección de este conjunto y del pasado como parámetro
intersection(b : Bag(T))	Bag(T)	bolsa formada por la intersección de este conjunto y de la bolsa pasada como parámetro
- (s : Set(T))	Set(T)	elementos que están en el conjunto pero no en el pasado como parámetro
including(x : T)	Set(T)	añade <i>x</i> al conjunto (si no estaba)
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 <i>del conjunto</i> para los que <i>expr</i> es cierta
reject(expr : OclExpression)	Set(expr.type)	subconjunto de los elementos para los que <i>expr</i> es falsa
collect(expr : OclExpression)	Bag(expr.oclType)	construye una nueva <i>bag</i> a partir de la expresión pasada como parámetro
count(x : T)	Integer	número de apariciones de <i>x</i> . Como es un <i>Set</i> , el resultado es 0 ó 1
asSequence	Sequence(T)	secuencia formada por los elementos del conjunto
asBag	Bag(T)	bolsa formada por los elementos del conjunto

Tabla 25. Operaciones de *Set(T)*, donde *T* es el tipo de los elementos del conjunto

Una *Bag* (bolsa) representa un conjunto de elementos en el que cada elemento puede aparecer más de una vez.

Operación	Tipo devuelto	Resultado/descripción
union(s : Set(T))	Bag(T)	bolsa con la unión de esta bolsa con el conjunto pasado. Como puede observarse, unir un <i>Set</i> y un <i>Bag</i> produce siempre un <i>Bag</i>
union(b : Bag(T))	Bag(T)	bolsa formada por los elementos de esta bolsa y la pasada como parámetro
= (s : Bag(T))	Boolean	<i>true</i> si las dos bolsas son iguales (todos los elementos son iguales y, además, cada uno aparece el mismo número de veces en cada bolsa)
intersection(s : Set(T))	Set(T)	bolsa formada por la intersección de esta bolsa con el conjunto pasado como parámetro
intersection(b : Bag(T))	Bag(T)	bolsa formada por la intersección de esta bolsa con la pasada como parámetro
including(x : T)	Bag(T)	añade <i>x</i> a la bolsa (aunque ya esté)
excluding(x : T)	Bag(T)	elimina todas las apariciones de <i>x</i> de la bolsa
select(expr : OclExpression)	Bag(T)	bolsa con los elementos para los que <i>expr</i> es cierta
reject(expr : OclExpression)	Set(T)	bolsa con los elementos para los que <i>expr</i> es falsa
collect(expr : OclExpression)	Bag(expr.oclType)	construye una nueva <i>bag</i> a partir de la expresión pasada como parámetro
count(x : T)	Integer	Número de apariciones de <i>x</i>
asSequence	Sequence(T)	secuencia formada por los elementos de la bolsa
asSet	Set(T)	conjunto formado por los elementos de la bolsa

Tabla 26. Operaciones de *Bag(T)*, en donde *T* es el tipo de los elementos de la bolsa

Una *Sequence* (secuencia) es una *Collection* que se comporta como una *Bag*, pero en la que los elementos están indexados, poseyendo un número de orden que podemos utilizar para acceder a un elemento dado.

Operación	Tipo devuelto	Resultado/descripción
count(x : T)	Integer	Número de apariciones de <i>x</i>
= (s : Sequence(T))	Boolean	<i>true</i> si las dos secuencias son iguales (como en <i>Bag</i> , pero además los elementos están en el mismo orden)
union(s : Sequence(T))	Sequence(T)	secuencia formada por los elementos de ambas secuencias
append(x : T)	Sequence(T)	añade <i>x</i> al final de la secuencia
prepend(x : T)	Sequence(T)	quita todas las apariciones de <i>x</i>
subsequence(inf:Integer, sup:Integer)	Sequence(T)	secuencia con los elementos situados entre las posiciones <i>inf</i> y <i>sup</i>
at(pos : Integer)	T	elemento en la posición <i>pos</i>
first	T	primer elemento de la secuencia
last	T	último elemento de la secuencia
including(x : T)	Sequence(T)	coloca <i>x</i> al final (como <i>append</i>)
excluding(x : T)	Sequence(T)	elimina todas las apariciones de <i>x</i>
select(expr : OclExpression)	Sequence (T)	secuencia formada por los elementos que cumplen <i>expr</i>
reject(expr : OclExpression)	Sequence (T)	secuencia formada por los elementos que no cumplen <i>expr</i>
collect(expr : OclExpression)	Sequence (expr.oclType)	construye una nueva secuencia a partir de la expresión pasada como parámetro
asBag	Bag(T)	bolsa producida al transformar la secuencia
asSet	Set(T)	conjunto producido al transformar la secuencia

Tabla 27. Operaciones de *Sequence(T)*, siendo *T* el tipo de los elementos de la secuencia

3.3. La operación *iterate*

iterate(expr : OclExpression) es la operación más compleja (pero también la más potente) para trabajar con colecciones. La *OclExpression* que toma como parámetro tiene tres partes: (1) el tipo de los elementos de la colección; (2) el tipo del resultado y la asignación inicial; y (3) la operación que aplicamos a los elementos de la colección. Su sintaxis es la siguiente:

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

El *Tipo1* es el tipo de los elementos de la colección; *resultado* es la variable en la que se almacenará el resultado devuelto por la operación, que será de *Tipo2* y a la que se le habrá asignado el *valor_inicial*; la *operación* representa el cómputo que se realiza combinando el *resultado* con cada *elemento* de la colección. La operación *iterate* debe entenderse como una función que devuelve un valor del tipo del resultado (*Tipo2*, en este caso). Un fragmento de código Java equivalente es:

```
Tipo2 resultado = valor_inicial;
for (int i=0; i<colección.size(); i++) {
    Tipo1 elemento= (Tipo1) colección.elementAt(i);
    resultado=operación(elemento, resultado);
}
return result;
```

3.4. Navegación entre asociaciones

En un diagrama de clases podemos navegar desde una clase cualquiera hasta cualquier otra clase con la que esté asociada. Si hay nombres de rol en las asociaciones, se usa éste para hacer referencia al elemento de destino; si no la hay, se utiliza el nombre del tipo para hacer referencia al elemento.

Al navegar desde un tipo hacia otro, si la cardinalidad en el tipo de destino es 0, 1 ó 0..1, el tipo devuelto es el tipo del objeto de destino. Si la cardinalidad es mayor que uno y se recorre una sola asociación, entonces se obtiene un *Set*; si se recorre más de una asociación y la cardinalidad en el tipo de destino es mayor que uno, se obtiene una *Bag*. No obstante, si la relación está ordenada, se obtiene una *Sequence*.

Así, en la Figura 88:

- *A.B* es de tipo *B*.
- *A.B.C* es de tipo *Sequence(C)*.
- *B.C* es de tipo *Sequence(C)*.
- *D.E* es de tipo *Set(E)*.
- *E.F* es de tipo *Set(F)*.
- *D.E.F* es de tipo *Bag(F)*.

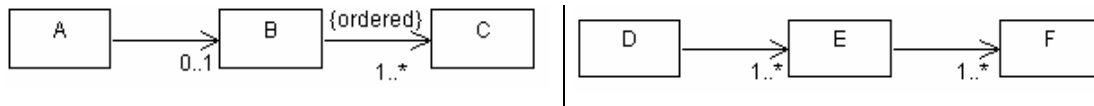


Figura 88. Dos ejemplos para ilustrar la navegación entre asociaciones

3.5. Ejemplos y notación

Supongamos que disponemos del siguiente diagrama de clases, que representa un sistema bancario. El banco dispone de una serie de cuentas que pueden tener hasta tres titulares; cada cliente puede operar sobre varias cuentas y puede tener una cuenta distinguida en la que se le ingresa la nómina. Se almacenan todos los movimientos que se realizan sobre las cuentas y sobre sus tarjetas asociadas, que pueden ser de débito y de crédito.

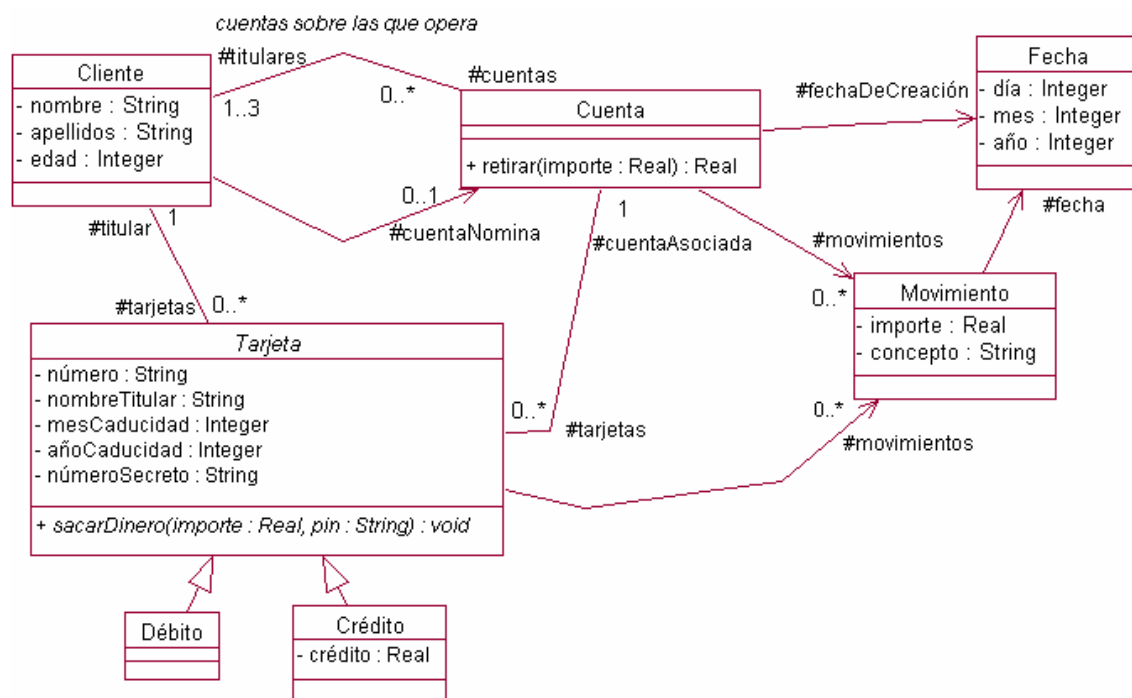


Figura 89. Un diagrama de clases para anotar con OCL

A continuación, vamos enumerando, explicando y escribiendo algunas restricciones sobre este diagrama.

1) La *edad* de cualquier *Cliente* es siempre mayor o igual a cero.

Esta restricción se refiere al valor de un atributo determinado de la clase *Cliente*, por lo que se trata de una invariante. Todas las restricciones OCL anotan un elemento del modelo UML al que se conoce como “contexto”, que puede ser un tipo en el caso de una invariante, o una operación si se trata de precondiciones o postcondiciones. En OCL, esta restricción se escribiría de la siguiente forma, en la que el término *inv* denota que se trata de una invariante:


```
Context Cliente
inv:
    edad>=0
```

Restricción 1

Igualmente podíamos haber añadido el término *self* antes de *edad*. *self* hace referencia a la instancia actual (equivale al *this* de Java, C y otros lenguajes, o al *Me* de Visual Basic):

```
Context Cliente
inv:
    self.edad>=0
```

Restricción 2. La Restricción 1, con el término *self*

2) En toda *Cuenta* debe haber al menos un *titular* de 18 años o más.

Se trata de una invariante sobre la colección de *titulares* de cada *Cuenta*, por lo que éste es su contexto. Su descripción podría ser:

```
Context Cuenta
inv:
    self.titulares→select(t:Cliente | t.edad>=18)→size()>=1
```

Restricción 3. Un titular de 18 años o más, versión 1

En la Restricción 3 navegamos desde la instancia de *Cuenta* en la que nos encontramos hasta su colección de *titulares* y, de entre éstos, seleccionamos todos aquellos cuya edad sea mayor o igual a 18. A continuación los contamos, obligando a que haya al menos 1. Obsérvese el operador flecha (\rightarrow), que se utiliza para representar las operaciones que se ejecutan sobre colecciones.

Otra posible forma es del siguiente cuadro. Como se observa, el término *self* puede omitirse.

```
Context Cuenta
inv:
    titulares→exists(c : Cliente | c.edad>=18)
```

Restricción 4. Un titular de 18 años o más, versión 2

Una forma incorrecta de anotar la misma restricción es, por ejemplo, la siguiente, en la que pasamos a *count* una expresión en lugar de un objeto.

```
Context Cuenta
inv:
    self.titulares→count(t:Cliente | t.edad>=18)
```

Restricción 5. Restricción incorrecta por el tipo del parámetro en *count*

3) La *cuentaNómina* de un *Cliente* es una de sus *cuentas*.

```
Context Cliente
inv:
    cuentas→includes(cuentaNómina)
```

Restricción 6

4) El saldo de toda *Cuenta* debe ser siempre mayor o igual a cero (el saldo se calcula como la suma del importe de los movimientos asociados).

En la siguiente restricción, iteramos tomando cada elemento *Movimiento m* de la colección de movimientos del objeto *self* (una *Cuenta*). Antes

de entrar al bucle declaramos la variable acumuladora *saldo* y la inicializamos a cero; en cada vuelta vamos sumando el importe de cada *Movimiento*. La operación *iterate* de este ejemplo devuelve un resultado de tipo *Real*, al que se impone que sea mayor o igual a cero.

```
Context Cuenta
inv:
    self.movimientos→iterate(m:Movimiento;
                             saldo : Real = 0 | saldo=saldo+m.importe)>=0
```

Restricción 7. El saldo de toda cuenta es mayor o igual a cero, versión 1

Otra posibilidad es la Restricción 8, en la que declaramos la variable *importes* (mediante la palabra reservada *let*) de tipo colección de números reales. *importes* se construye seleccionando el campo *importe* de la colección *movimientos* de esta *Cuenta*. Declarada y construida la variable de esa forma, imponemos (véase la palabra reservada *in*) la restricción de que la suma de los importes (función *sum*, que puede aplicarse ya que el tipo de dato almacenado en la colección es numérico) sea mayor o igual a cero.

```
Context Cuenta
inv:
    let importes : Collection(Real) = movimientos→importe in
    importes→sum()>=0
```

Restricción 8. El saldo de toda cuenta es mayor o igual a cero, versión 2

Otra posibilidad más es la siguiente, en la que declaramos dos variables:

```
Context Cuenta
inv:
    let importes : Collection(Real) = movimientos→importe,
    let saldo=importes→sum() in
    saldo>=0
```

Restricción 9. El saldo de toda cuenta es mayor o igual a cero, versión 3

5) El *titular* de toda *Tarjeta* de *Crédito* debe tener la nómina domiciliada en la misma *Cuenta* que está asociada a la *Tarjeta*.

```
Context Crédito
inv:
    self.titular.cuentaNomina=self.cuentaAsociada
```

Restricción 10

6) El nombre del titular que figura en toda *Tarjeta* (campo *nombreTitular*) es la concatenación del nombre y apellidos del *titular*.

```
Context Tarjeta
inv:
    self.nombreTitular = titular.nombre.concat(" ").concat(titular.apellidos)
```

Restricción 11

7) Para retirar dinero con una *Tarjeta* de *Débito*, el número secreto pasado a la operación (parámetro *pin*) debe coincidir con el *númeroSecreto* de la tarjeta; además, la *cuentaAsociada* debe tener saldo suficiente para

hacer frente al *importe* que se desea sacar, que también tiene que ser positivo.

En este caso, la restricción se trata de una precondition (obsérvese el término *pre* en el siguiente recuadro) sobre la operación *sacarDinero* de la clase *Débito*, que ésta hereda de *Tarjeta*.

```
Context Débito :: sacarDinero(importe: Real, pin:String) : void
  pre:
    importe>0 and
    pin=self.numeroSecreto and
    self.cuentaAsociada.getSaldo()>=importe
```

Restricción 12

Como se observa en la Restricción 12, cuando el contexto de una restricción es una operación, la sintaxis general es la siguiente:

```
context Tipo :: operación(parámetro1 : Tipo1, ...) : TipoDevuelto
  pre : ...
  post : ...
```

Tabla 28. Sintaxis de la restricciones sobre operaciones (pre y postcondiciones)

En la Restricción 12 utilizamos la operación *getSaldo*, que no aparece en el diagrama de clases. Si apareciese, podría especificarse como una postcondición el hecho de que debe devolver la suma del importe de los movimientos de la *Cuenta*, como se hace en la Restricción 13: obsérvese que utilizamos la palabra reservada *result* para hacer referencia al resultado devuelto por la operación.

```
Context Cuenta :: getSaldo() : Real
  post:
    result=self.movimientos->iterate(m:Movimiento;
                                     saldo : Real = 0 | saldo=saldo+m.importe)
```

Restricción 13

8) Supongamos que deseamos ampliar la descripción de la operación *sacarDinero* de *Débito* para indicar que el saldo de la *cuentaAsociada* se decrementa por el importe retirado y que, además, se ejecuta la operación *retirar* sobre la *cuentaAsociada*. En las postcondiciones, se utiliza el símbolo *@pre* para hacer referencia a los valores anteriores a la ejecución de la operación, y el símbolo *^* (acento circunflejo) para denotar el envío de mensajes. Se escribe el nombre del objeto receptor, el acento y el nombre del mensaje con sus parámetros.

```
Context Débito :: sacarDinero(importe: Real, pin:String) : void
  pre:
    importe>0 and
    pin=self.numeroSecreto and
    self.cuentaAsociada.getSaldo()>=importe
  post:
    cuentaAsociada.getSaldo()=(cuentaAsociada.getSaldo())@pre-importe
    and cuentaAsociada^retirar(importe)
```

Restricción 14. La Restricción 12, con un par de adiciones

Si el mensaje devuelve un resultado, podemos hacer referencia a éste mediante la operación *result()*. Supongamos que el resultado devuelto por

Cuenta::retirar():Real es el saldo que queda en la *Cuenta*. Podríamos añadir a la restricción el hecho de que el saldo disponible de la tarjeta será el saldo disponible en la cuenta asociada:

```
Context Débito :: sacarDinero(importe: Real, pin:String) : void
  pre:
    importe>0 and
    pin=self.numeroSecreto and
    self.cuentaAsociada.getSaldo()>=importe
  post:
    let mensaje : OclMessage = cuentaAsociada.retirar(importe) in
    mensaje.result()==self.getSaldo()
```

Restricción 15

En la restricción anterior utilizamos un objeto de tipo *OclMessage*, que OCL utiliza para manipular mensajes. Lo que hacemos en la postcondición es asignar a la variable *mensaje* una referencia al mensaje que se envía; luego, se impone que el resultado devuelto por este mensaje sea igual al saldo disponible en la tarjeta. Obsérvese que no hemos definido la operación *getSaldo* para ningún tipo de *Tarjeta* (la Restricción 13 la especifica, pero para la clase *Cuenta*). Si queremos indicar que el saldo disponible de una tarjeta de *Débito* es el saldo disponible en la cuenta asociada, escribiríamos:

```
Context Débito :: getSaldo() : Real
  post:
    result=cuentaAsociada.getSaldo()
```

Restricción 16

9) Al sacar dinero con tarjeta de *Crédito* se cobra una comisión del 3% con un mínimo de 2,50 euros. Para sacar dinero, la suma de estas dos cantidades debe ser menor o igual al crédito disponible de la tarjeta, que es el *crédito* menos el importe gastado en el mes en curso (suma del importe de los movimientos realizados con la tarjeta en el mes en curso). Además, el importe que se desea sacar debe ser positivo.

En lugar de escribir toda la restricción de una sola vez, intentaremos ir desglosándola en operaciones más simples. Una transcripción casi textual de la restricción es la siguiente:

```
Context Crédito :: sacarDinero(importe:Real, pin:String) : void
  pre :
    importe+getComisión(importe)<=getCréditoDisponible() and
    importe>0
```

Restricción 17

Ahora debemos ir definiendo las dos operaciones a las que hacemos mención en la Restricción 17. *getComisión* puede especificarse al menos de estas tres formas:

```
Context Crédito :: getComisión(importe: Real) : Real
  post:
    result=(importe*0,03).min(2,50)
```

Restricción 18. Especificación de *getComisión*, versión 1

En la versión 1 de la operación, utilizamos la operación *min* (disponible para los tipos *Integer* y *Real*, según la Tabla 20) para determinar el valor.

```
Context Crédito :: getComisión(importe: Real) : Real
  post:
    let porcentaje : Real = 0,03*importe in
    result=porcentaje.min(2,50)
```

Restricción 19. Especificación de *getComisión*, versión 2

En la versión 2, declaramos una variable *porcentaje* y sobre ella aplicamos la operación *min(Real) : Real*.

```
Context Crédito :: getComisión(importe: Real) : Real
  post:
    if (importe*0,03<2,50)
    then result=2,50
    else result=0,03*importe
```

Restricción 20. Especificación de *getComisión*, versión 3

En esta última versión de la operación utilizamos la operación *if-then-else*, válida para operar sobre el tipo *Boolean* (Tabla 20). Es importante notar que la instrucción *if-then-else*, en OCL, debe llevar siempre la rama cierta y la rama falsa.

La operación *getCréditoDisponible* puede anotarse de esta manera, en la que seleccionamos aquellos movimientos producidos en el mismo mes que la fecha del sistema (asumimos que disponemos de un tipo *System* que nos ofrece estas dos operaciones) con la instrucción *select*, sumamos sus importes y lo asignamos al resultado.

```
Context Crédito :: getCréditoDisponible() : Real
  post : result = self.movimientos→
    select(m:Movimiento|m.fecha.año=System.getYear()
    and m.fecha.mes=System.getMonth)→importe→sum
```

Restricción 21. Especificación de *getCréditoDisponible()*, versión 1

Todas las operaciones sobre las colecciones pueden especificarse con la operación *iterate*. La selección que hacemos en la restricción anterior puede escribirse de otra manera, con lo que la restricción quedaría así:

```
Context Crédito :: getCréditoDisponible() : Real
  post : result=self.movimientos→iterate(
    m : Movimiento;
    total : Real = 0 |
    if (m.fecha.Año=System.getYear() and
    m.fecha.Mes=System.getMonth())
    then total=total+m.importe
    else total=total+0
  )
```

Restricción 22. Especificación de *getCréditoDisponible()*, versión 2

10) La fecha de los movimientos de una *Cuenta* es mayor o igual a la fecha de creación de la *Cuenta*.

```
Context Cuenta inv:
  self.movimientos→forall(m:Movimiento | m.fecha.mayorOIgualA(self.fechaDeCreación))
```

Restricción 23. Restricción sobre fechas, versión 1

En el recuadro anterior utilizamos la operación *Fecha::mayorOIgual(Fecha)*, cuya especificación en OCL se propone al lector. Asumiendo que disponemos ya de esta operación correctamente anotada, otra forma de escribir la misma restricción, pero ahora desde el contexto de *Movimiento*, es la siguiente:

```
Context Movimiento inv:
    self.fecha.mayorOIgual(cuenta.fecha)
```

Restricción 24. Restricción sobre fechas, versión 2

La Restricción 24, sin embargo, no es coherente con el diagrama de clases, ya que la asociación entre *Cuenta* y *Movimiento* no es navegable desde ésta hacia aquélla. Reescribamos la versión correcta, pero ahora utilizando el cuantificador existencial en lugar del universal:

```
Context Cuenta inv:
    not(self.movimientos->exists(m: Movimiento | m.fecha.menor(fechaDeCreacion)))
```

Restricción 25. Restricción sobre fechas, versión 3

Si dispusiéramos de la operación *Fecha::mayorOIgual(Fecha)* debidamente especificada, la operación *Fecha::menor(Fecha)* que usamos en la restricción anterior podría describirse de este modo:

```
Context Fecha :: menor(x : Fecha) : Boolean
    post:
        if (x.mayorOIgual(self)) then
            result=self
        else
            result=x
```

Restricción 26

11) En toda *Cuenta* debe haber entre 1 y 3 titulares.

No debemos olvidar que OCL complementa las descripciones de UML. Por tanto, si una restricción ya se encuentra descrita en el modelo (como en este caso, en el que hay una multiplicidad 1..3 en el lado de la clase *Cliente* de la asociación), es innecesario escribir la restricción.

Por otro lado, podemos utilizar la operación *collect* para construir colecciones a partir de otras colecciones. La siguiente línea, por ejemplo, construiría una bolsa con las fechas de apertura de las cuentas de cada cliente:

```
cuentas->collect(c:Cuenta | c.fechaDeCreacion)
```

Cuadro 50. Utilización de la operación *collect*

No obstante, OCL permite abreviar la operación *collect* haciendo referencia directamente al campo que queramos “recolectar”. La siguiente instrucción sería equivalente a la anterior:

```
cuentas.fechaDeCreacion
```

Cuadro 51. Abreviación de la expresión del Cuadro 50

De hecho, ya utilizamos este tipo de abreviación en, por ejemplo, la Restricción 8.

3.6. Escritura de aserciones en el programa

Idealmente, todas las restricciones OCL con que se anota un modelo UML deben ser luego implementadas en el código, con el fin de mantener la integridad entre los diferentes artefactos del sistema. A partir de la versión 1.4, Java incluye la instrucción *assert* para comprobar aserciones. Su formato es el siguiente:

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

La *expresiónBooleana* representa la condición que se está verificando, mientras que la *expresiónConValor* (que es opcional) representa el mensaje de error que se lanzará en caso de que la condición sea falsa. 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 paquete.ClaseConMain.class      para compilar
java -ea paquete.ClaseConMain                    para ejecutar
```

Las operaciones que incluyen asertos arrojan el error *java.lang.AssertionError*, que son objetos “lanzables” (es decir, especializaciones de *java.lang.Throwable*), pero no exactamente excepciones (Figura 90). Por consiguiente, ni la cláusula *throws Exception* ni el bloque *catch(Exception e)* son suficientes para capturar y procesar este tipo de errores.

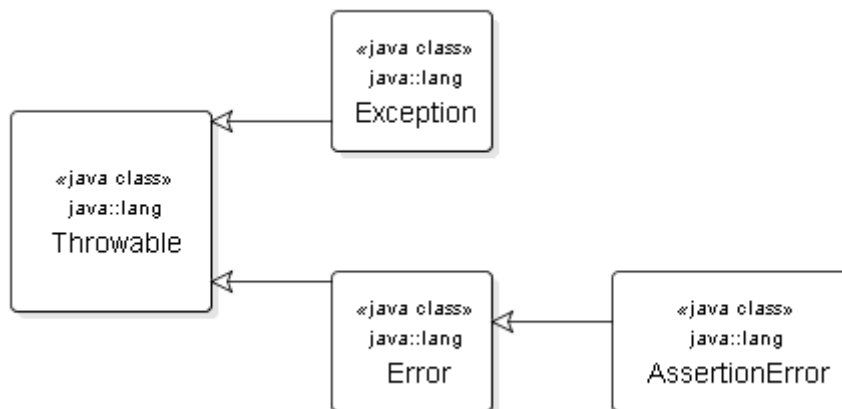


Figura 90. Jerarquía en Java de los objetos *Throwables*

Supongamos que la clase *Solicitud* dispusiera de la operación *solicitar(Proyecto, Partida, double)*. Si quisiéramos anotarla con una aserción que comprobara la disponibilidad de presupuesto, podríamos completar el código de la siguiente forma:

```
public void solicitar(Proyecto p, String partida, double importe)
    throws Exception, AssertionError {

    Partida partidaSolicitada=new Partida(p, partida);
    assert partidaSolicitada.getDisponible()>=getImporte() : "Disponible insuficiente";
    ...
}
```

Cuadro 52. Adición de una aserción a un método

**Tratamiento
distinguido de
excepciones y
errores**

El programador de un método que llama al código anterior sabrá que, cuando capture una *AssertionError*, quizás deba darle un tratamiento distinto que si se tratara de una *Exception* genérica. Por este motivo, en el lenguaje Java no debe utilizarse la instrucción *assert* para hacer comprobaciones sobre los valores de los parámetros, sino que deben comprobarse con instrucciones *if* y, en caso necesario, lanzar objetos de clase *IllegalArgumentException*; de este modo, y si es preciso, el programador procesará el error de manera diferente. Entonces, si en el mismo método anterior fuera preciso comprobar que el importe solicitado es positivo, añadiríamos una instrucción *if* de este estilo:

```
public void solicitar(Proyecto p, String partida, double importe)
    throws IllegalArgumentException, SQLException, AssertionError {
    if (importe<0)
        throw new IllegalArgumentException("El importe solicitado " +
                                           "debe ser mayor que cero");

    Partida partidaSolicitada=new Partida(p, partida); // Lanza SQLException
    assert partidaSolicitada.getDisponible()>=getImporte() : "Disponible insuficiente";
    ...
}
```

Cuadro 53. Lanzamiento de distintos tipos de errores

3.7. Utilización de las aserciones para derivar casos de prueba

Dos técnicas bien conocidas para la creación de casos de prueba son la creación de particiones o clases de equivalencia y el análisis de valores límite.

Para el primer caso, se divide el conjunto de valores de entrada de cada función en un número finito de clases de equivalencia o particiones disjuntas. De cada partición se toma un valor, que se asume será suficientemente representativo de los restantes valores de la clase de equivalencia. A continuación, se construye un caso de prueba por cada valor. Si la función que deseamos probar tiene varios parámetros, se construyen tantos casos de prueba como combinaciones tengamos.

En la segunda técnica, se utilizan como valores para ser pasados al caso de prueba los valores límite de cada clase de equivalencia.

Los valores utilizados en las restricciones del sistema (estén o no descritas en OCL) pueden y suelen emplearse para construir los casos de prueba. De este modo, si tuviéramos que construir casos de prueba para probar la operación mostrada en el Cuadro 53, podríamos identificar las siguientes clases de equivalencia y valores límite:

	Clases de equivalencia	Valores límite
Proyecto p	{Un proyecto caducado} {Un proyecto activo}	Un proyecto que cadu- que hoy
String partida	{"Fungible", "Inventariable", "Viajes y dietas", "Personal"} {“Partida que no existe y con el nombre demasiado largo”}	No aplicable
double importe	{-∞, 0} {0, máximo disponible en la partida} { máximo disponible en la partida+1, ∞}	0 máximo disponible en la partida 1+ máximo disponible en la partida

Tabla 29. Valores obtenidos para probar el supuesto método *solicitar* del Cuadro 53

3.8. Interpretación de las aserciones como contratos

En muchas ocasiones, las aserciones se interpretan como contratos de utilización de las operaciones: anotar una operación con pre y postcondiciones es como decir al programador que la utiliza: “te garantizo que si la instancia se encuentra en este estado y los parámetros cumplen tales condiciones, entonces la operación va a comportarse de este modo y la instancia va a quedar en este estado”.

4. Desarrollo del caso de uso *Crear calle*

Además de con la descripción textual, los casos de uso pueden describirse mediante máquinas de estados, que representan los pasos que el sistema ejecuta para servir la funcionalidad que se está describiendo.

Una máquina de estados es un grafo que, como todos los grafos, posee nodos y arcos. Cuando se utilizan máquinas de estados para representar el comportamiento de casos de uso, los nodos representan pasos en la ejecución del caso de uso, y los arcos representan las transiciones entre esos pasos. La descripción de las máquinas de estados dada por UML es muy poderosa, y se verá más a fondo en el Capítulo 13; sin embargo, para describir casos de uso, suele ser suficiente con utilizar un subconjunto de su notación.

La Figura 91 describe el caso de uso *Crear calle* mediante una máquina de estados. Cada estado representa un paso en la ejecución del caso de uso; las transiciones, que no se encuentran etiquetadas, se asume que se disparan cuando terminan las operaciones del estado origen. El rombo es un nodo *fork* (literalmente, “tenedor”: nodos que tienen una entrada y varias salidas), y se utilizan para representar caminos alternativos de ejecución (en función de la condición o guarda que etiqueta las transiciones que salen de ellos). Además, hay dos nodos distinguidos que representan los estados inicial y final. En la máquina de la figura, se selecciona una casilla, se indica que

va a ser de tipo Calle y se le asigna el barrio al que pertenecerá y el resto de datos (precio de adquisición, precio de cada casa, etc.); entonces se comprueba si el barrio elegido admite más calles (el máximo de calles por barrio es tres, aunque puede haber barrios de dos calles) y si la edición también admite más calles (el número de calles en un tablero es de veintidós). Si el barrio o el tablero no admiten más calles, entonces se informa del error.

No todas las herramientas CASE disponen de las mismas capacidades de dibujo. El plugin para Eclipse que veníamos utilizando no permite la adición al diagrama de nodos *fork*, por lo que hemos optado por construir esta figura con el entorno de desarrollo Oracle JDeveloper.

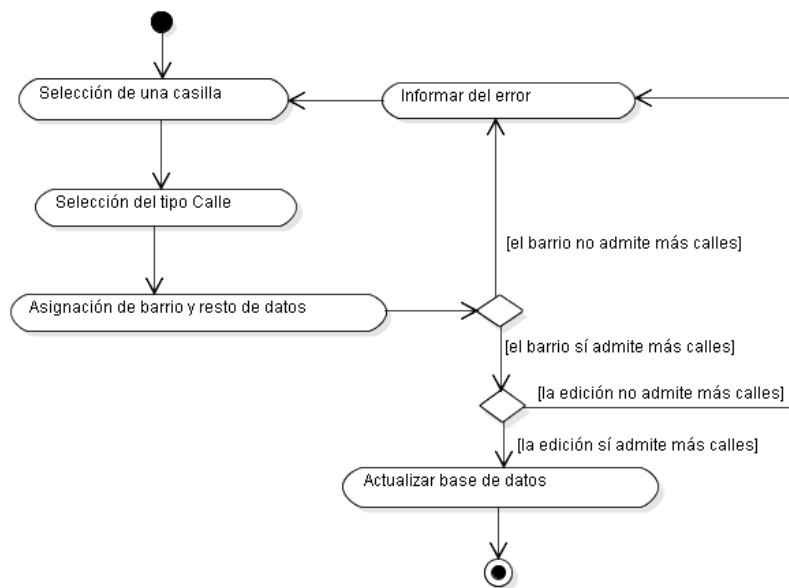


Figura 91. Descripción de *Crear calle* con una máquina de estados

La descripción textual de la que procede la máquina anterior podría ser la siguiente:

Nombre: Crear calle
Abstracto: no
Precondiciones: 1. Hay una edición del juego que se está creando 2. Los barrios del juego han sido creados
Postcondiciones:
Rango:
Flujo normal: 1. En la ventana de definición del tablero, el usuario selecciona el número de la casilla en la que desea situar una calle 2. La ventana muestra en la etiqueta situada en arriba a la izquierda el número de la casilla seleccionada. 3. El usuario elige la solapa Calle y cumplimenta los datos correspondientes, incluyendo el barrio 4. El usuario pulsa el botón guardar. 5. La ventana, que conoce a la Edición que se está creando, le dice a ésta que el tipo de la casilla seleccionada es Calle 6. Edición comprueba que el barrio admite más calles (porque tiene una o dos), y que el tablero admite más calles (porque tiene menos de 22) 7. La Edición asigna a la <i>Casilla</i> el tipo Calle, asignándole el nombre, precio, barrio, etc. 8. La casilla actualiza su tipo en la base de datos a través del Agente
Flujo alternativo 1: 1. En la ventana de definición del tablero, el usuario selecciona el número de la casilla en la que desea situar una calle 2. La ventana muestra en la etiqueta situada en arriba a la izquierda el número de la casilla seleccionada. 3. El usuario elige la solapa Calle y cumplimenta los datos correspondientes, incluyendo el barrio 4. El usuario pulsa el botón guardar. 5. La ventana, que conoce a la Edición que se está creando, le dice a ésta que el tipo de la casilla seleccionada es Calle 6. La Edición comprueba que el barrio no admite más calles 7. Se informa del error al usuario
Flujo alternativo 2: 1. En la ventana de definición del tablero, el usuario selecciona el número de la casilla en la que desea situar una calle 2. La ventana muestra en la etiqueta situada en arriba a la izquierda el número de la casilla seleccionada. 3. El usuario elige la solapa Calle y cumplimenta los datos correspondientes, incluyendo el barrio 4. El usuario pulsa el botón guardar. 5. La ventana, que conoce a la Edición que se está creando, le dice a ésta que el tipo de la casilla seleccionada es Calle 6. La Edición comprueba que el barrio admite más calles (porque tiene una o dos) 7. La Edición comprueba que el tablero ya tiene 22 calles, por lo que no admite más casillas de este tipo 8. Se informa del error al usuario
Descripción:

Tabla 30. Descripción textual del caso de uso *Crear calle*

Se observa por la precondición que este caso de uso requiere la existencia previa de barrios en la edición, por lo que habrá que añadir un nuevo caso de uso *Crear barrio*.

4.1. Anotación del diagrama de clases con OCL

El siguiente diagrama muestra parte de la estructura del sistema, una vez se añadida la clase *Calle* como especialización de *Tipo*. A *Calle* se le han añadido, entre otros, los campos *nombre* y *precio* y la operación *getValorHipotecario*, que devuelve el valor hipotecario de la calle, que es la mitad del precio de adquisición. Otros campos que también se han añadido aunque no se muestran son los precios de los alquileres con cero, 1, 2, 3 y 4 casas, y con hotel.

Se ha creado además una clase *Barrio*; cada barrio posee dos o tres calles, y cada calle conoce el barrio al que pertenece. El barrio está identificado

por un color; como el precio de adquisición de casas para todas las calles de un barrio es el mismo, se ha añadido el campo *precioDeCadaCasa* a la clase *Barrio*.

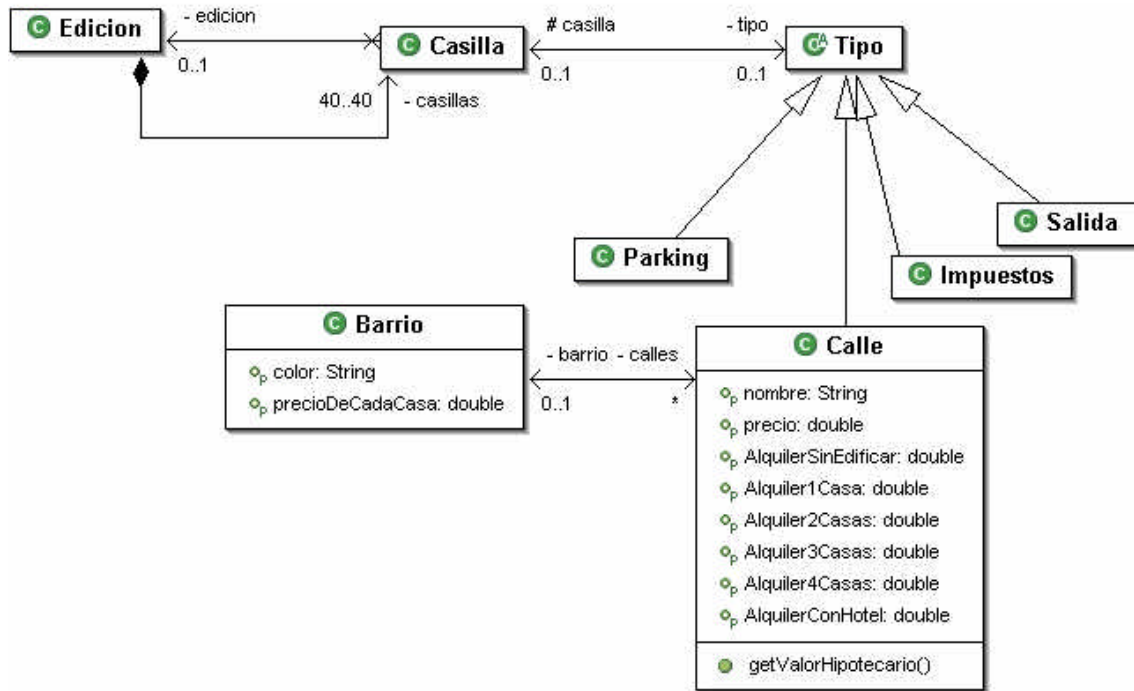


Figura 92. Fragmento del diagrama de clases, con *Calle* como subclase de *Tipo*

También se ha anotado la multiplicidad de la agregación entre *Edicion* y *Casilla* con el valor exacto 40. Podemos enriquecer la descripción del sistema anotando con OCL las siguientes restricciones:

- 1) El precio de adquisición de una calle es igual o mayor al precio de adquisición de las calles situadas en las posiciones anteriores.
- 2) Un barrio debe tener dos o tres calles.
- 3) Cuando se termina de crear una edición, ésta debe disponer de 22 calles.

4.1.1 Primera restricción

Una forma de describir esta restricción es en el contexto *Calle*. Indicaremos que el precio de esta calle (*self*) es mayor o igual que el de las calles situadas en posiciones anteriores. Para ello, creamos una colección *calles* en la que colocamos todas las instancias de *Calle* correspondientes a casillas en posiciones anteriores a la de ésta (*self*). Después obligamos a que todas las instancias de la colección *calles* tengan un precio menor o igual al de *self*.

```

context Calle inv:
  let calles : Collection(Calle) =
    self.casilla.edicion->casillas->select(tipo : Tipo | c.tipo.ocIsTypeOf(Calle)
      and c.posicion<self.posicion)
  in
    calles->forall(calle:Calle; calle.precio<=self.precio)

```

Restricción 27

4.1.2 Segunda restricción

Puesto que el diagrama de clases ya incluye mención expresa multiplicidad en la relación de *Barrio* con *Calle*, no sería preciso realizar la anotación. Si se quiere anotar, no obstante, una posible forma es la siguiente.

```
context Barrio inv: calles->size()>=2 and calles.size()<=3
```

Restricción 28

4.1.3 Tercera restricción

Esta restricción parece implicar la existencia de una operación de finalización de la Edición, algo parecido a una validación. Podemos añadir la operación *finalizar* a la clase *Edicion* y anotar la restricción como una post-condición.

```
context Edicion :: finalizar() : void
  let calles=self.casillas->select(c:Casilla | c.tipo.OclType=Calle)
  in
    post: calles->size()==22
```

Restricción 29

4.2. Creación de las tablas *Calle* y *Barrio*

La base de datos debe permitir el almacenamiento de las calles y de los barrios de cada edición, manteniendo las relaciones entre ambos tipos de elementos. El resultado de esta adición se muestra en la siguiente figura: cada *Edicion* posee una serie de barrios, que tienen a su vez calles.

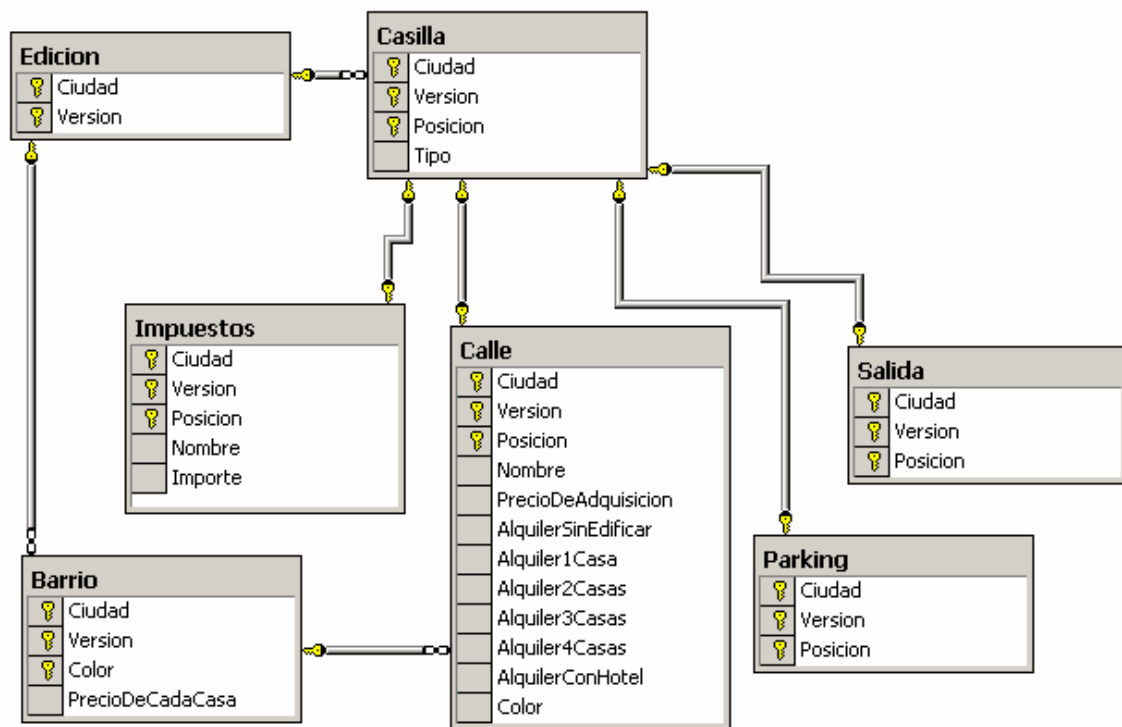


Figura 93. Adición de las tablas a la base de datos

5. Lecturas recomendadas

Algo de inspecciones y revisiones de código (libro verde, quizás).

Algo de OCL: traducción de OCL a Java, p.ej.

Algo de Design by contract

Capítulo 7. APLICACIÓN DE ADMINISTRACIÓN:

FASE DE ELABORACIÓN (III)

En este capítulo se presentan algunas de las actividades con las que finalizar la fase de Elaboración. Al finalizar esta fase, debe tenerse claro tanto el diseño arquitectónico del sistema como los principales aspectos de su diseño detallado.

1. Resto de actividades de la fase de Elaboración

Recordando las palabras de la sección 3.1.2 del Capítulo 2 (página 55), en la fase de elaboración “se analiza de forma detallada la práctica totalidad de los requisitos, lo cual llevará a terminar de establecer la base arquitectónica del producto software y a tener más base de conocimiento para concluir el plan del proyecto, que dirija especialmente a las dos próximas fases”.

De acuerdo además con la Figura 18 (página 53), en este fase habremos hecho bastante esfuerzo en los flujos de trabajo de Requisitos, Análisis y Diseño, pero también en los de Implementación y Pruebas, lo que nos habrá permitido encontrarnos con problemas técnicos reales que nos sirven para ir refinando tanto el diseño detallado como el arquitectónico.

Arquitectura software

La arquitectura de un sistema software, según los creadores del Proceso Unificado de Desarrollo⁷, es el “conjunto de las decisiones significativas sobre la organización del sistema mediante: (1) selección de los elementos estructurales e interfaces de los que el sistema está compuesto; (2) especificación de su comportamiento indicado por la colaboración entre esos elementos; (3) composición de estos elementos estructurales y de comportamiento en subsistemas cada vez más grandes; (4) estilo arquitectónico que guíe a esta forma de organización: los elementos mencionados y sus interfaces, colaboraciones y composiciones. Además, a la arquitectura también le conciernen las restricciones tecnológicas, de uso, de funcionalidad, de rendimiento, de reutilización, y de facilidad de comprensión, así como trade-offs y la estética”.

A partir de esta definición, la descripción de la arquitectura de un sistema estará formada por distintas vistas: su estructura se puede describir

⁷ Véase glosario general de “The Unified Software Development Software”, de I. Jacobson, G. Booch y J. Rumbaugh. Addison-Wesley, 1999.

mediante una vista funcional, una vista de análisis o una vista de diseño; para el comportamiento, usaremos diagramas de interacción o máquinas de estados que representen el comportamiento de casos de uso. Incluirá también las decisiones de diseño más importantes, el software externo utilizado o los mecanismos de almacenamiento y recuperación de la base de datos.

2. Análisis de algunos casos de uso

La Tabla 31 muestra agrupadamente algunos casos de uso cuyo análisis no se ha tratado en el texto, agrupados por el tipo de caso de uso (creación de casillas o creación de tarjetas). De aquí en adelante, asumiremos que, de acuerdo con el plan de iteraciones (Tabla 6, página 81), en esta fase ya se ha abordado el análisis de los marcados con asterisco. Nótese la introducción del caso de uso *Crear barrio* antes de *Crear calle*.

Creación de casillas	Crear cárcel (*) Crear compañía (*) Crear estación (*) Crear barrio Crear calle (*) Crear vaya a la cárcel (*) Crear caja de comunidad (*) Crear suerte (*)
Creación de tarjetas	Crear tarjeta Ir a una casilla determinada Avanzar o retroceder Cobrar cantidad fija Pagar cantidad fija Pagar cantidad variable Cobrar de otros jugadores

Tabla 31. Algunos casos de uso cuyo análisis no se ha abordado

3. Consideraciones generales sobre el diseño detallado

Al finalizar la fase de Elaboración, los miembros del equipo de desarrollo deben tener claros los principios del diseño detallado del sistema. Bajo esta idea se encierran conceptos como las políticas de acceso y gestión de la base de datos, de asignación de responsabilidades a clases, de estilo de codificación, etc.

3.1. Diseño de la base de datos

En el Capítulo 5 se discutieron diferentes políticas de creación de la base de datos a partir del diseño de la capa de dominio del sistema. Se decidió, por ejemplo, crear tablas específicas para cada tipo de casilla. Así, se han creado tablas para almacenar las casillas de tipo *Parking*, *Salida*, *Impuestos* y *Calle*, todas relacionadas con la tabla *Casilla*, en la que se almacena la información común a todas las casillas.

Algunas tablas (*Calle* e *Impuestos*) almacenan información adicional respecto de la tabla *Casilla*, mientras que otras (*Parking* y *Salida*) no. Ade-

más, la información sobre el tipo de las casillas ya se encuentra almacenada en la columna *Tipo* de la tabla *Casilla*. Por tanto, podríamos eliminar las tablas de la base de datos las tablas que no aportan nada. En las sucesivas fases e iteraciones, será preciso tener en cuenta que, de acuerdo con esta idea, no se crearán tablas para *Cárcel*, *Vaya a la cárcel*, *Caja de comunidad* y *Suerte*.

Así pues, el nuevo diseño de la base de datos, en el que se mantienen las tablas estrictamente necesarias, será el siguiente:

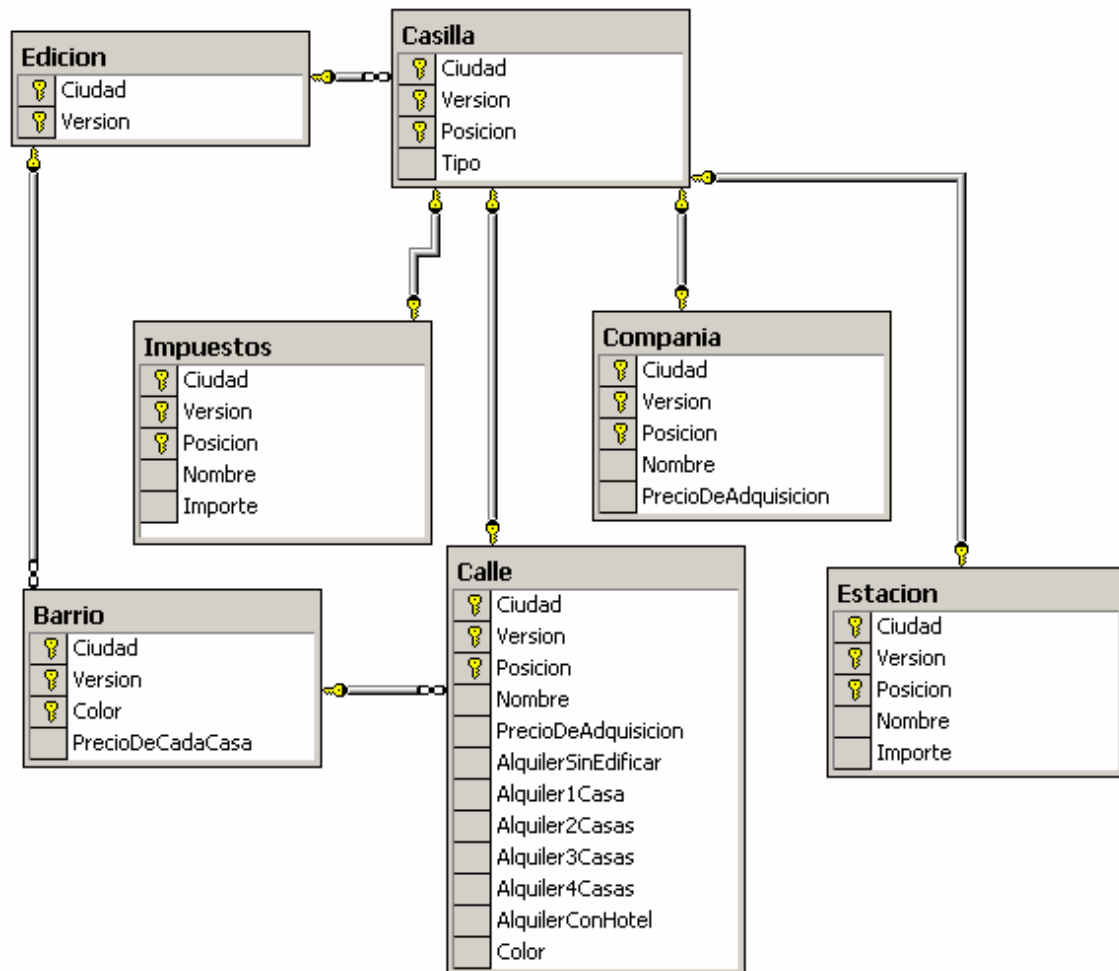


Figura 94. Diseño de la base de datos

Claramente, es necesario actualizar tanto el código de la aplicación como de los procedimientos almacenados de la base de datos. El Cuadro 54 muestra el nuevo código de dos de ellos: comparando el nuevo código de *guardarParking* con el de la versión anterior (Figura 82, página 150), se observa que ha perdido las dos últimas instrucciones *Delete* e *Insert*, que actuaban sobre la tabla *Parking*.

<pre>CREATE PROCEDURE guardarParking @ciudad varchar(50), @version varchar(50), @posicion int AS Update Casilla set Tipo=null where Ciudad=@ciudad and Version=@version and Tipo='Párking' Update Casilla set Tipo='Párking' where Ciudad=@ciudad and Version=@version and Posicion=@posicion GO</pre>	<pre>CREATE PROCEDURE guardarSalida @ciudad varchar(50), @version varchar(50), @posicion int AS Update Casilla set Tipo=null where Ciudad=@ciudad and Version=@version and Tipo='Salida' Update Casilla set Tipo='Salida' where Ciudad=@ciudad and Version=@version and Posicion=@posicion GO</pre>
--	---

Cuadro 54. Nuevo código de dos de los procedimientos almacenados

Además, habrá que reejecutar todos los casos de prueba.

3.2. Métodos de acceso a la base de datos

Durante el desarrollo del código hemos utilizado objetos de tipos tanto *PreparedStatement* como *CallableStatement* para manipular los datos de la base de datos. Aunque ambas formas de acceso pueden coexistir, es conveniente mantener una cierta uniformidad, por lo que éste puede ser un buen momento para decantarse por una u otra forma.

3.3. Utilización de herencia

En la Figura 69 (página 126) ilustrábamos la creación de la interfaz *IPanelDeCasilla*, que sería implementada por todas las solapas correspondientes a tipos de casillas. Tras la codificación de los casos de uso que llevamos hasta este momento, hemos observado que todas las solapas poseen un botón etiquetado “Guardar” que, al ser pulsado, ejecuta la operación *guardarCasilla*. Además, todas las solapas tienen un campo de tipo *Casilla*, que es la casilla sobre la que actúan.

Podríamos sustituir la interfaz *IPanelDeCasilla* de la Figura 69 por una nueva solapa de la que heredaran las demás. En esta nueva solapa (*JPCasilla*, por ejemplo, que sería un *JPanel*) crearíamos el campo *casilla* de tipo *Casilla* y las operaciones comunes a todas las solapas. El lado izquierdo del siguiente cuadro muestra el código de algunas operaciones incluidas en la solapa *JPParking* (una de las solapas más sencillas, similar a *JPSalida*); el derecho muestra el código de las mismas operaciones en *JPCalle*⁸ (una de las más complicadas).

⁸ Con propósitos ilustrativos adelantamos parte del código del caso de uso *Crear calle*

<pre>protected void guardarCasilla() { try { this.casilla.setTipo("Parking"); } catch (Exception e) { ... } } public void setCasilla(Casilla casilla) { this.casilla = casilla; this.jlCasilla.setText("El párking " + "estará situado en la casilla " + this.casilla.getPos()); } public Tipo getTipo() { return new Parking(); }</pre>	<pre>protected void guardarCasilla() { try { this.casilla.setTipo("Calle"); Calle tipo= (Calle) this.casilla.getTipo(); tipo.setNombre(jtfNombre.getText()); double precio=Double.parseDouble(this.jtfPrecioDeAdquisicion. getText()); double alquiler0=Double.parseDouble(this.jtfAlquiler0.getText()); ... tipo.setPrecio(precio); tipo.setAlquilerSinEdificar(alquiler0); tipo.setAlquiler1Casa(alquiler1); tipo.setAlquiler2Casas(alquiler2); tipo.setAlquiler3Casas(alquiler3); tipo.setAlquiler4Casas(alquiler4); tipo.setAlquilerConHotel(alquiler5); tipo.updateImportes(); } catch (Exception e) { ... } } public void setCasilla(Casilla casilla) { this.casilla = casilla; if (casilla.getTipo()!=null) { Calle tipo=(Calle) casilla.getTipo(); this.jtfNombre.setText(tipo.getNombre()); this.jtfPrecioDeAdquisicion.setText(""+tipo.getPrecio()); this.jtfValorHipotecario.setText(...); } } public Tipo getTipo() { return new Calle(); }</pre>
--	---

Cuadro 55. Código de dos de las solapas

Analicemos el código de las tres operaciones con el fin de intentar subir la mayor cantidad posible de implementación a la clase *JPCasilla* cuya creación nos estamos planteando:

1) *guardarCasilla* establece el tipo de la casilla pasando como parámetro el nombre del tipo y, en caso necesario, asigna al tipo los parámetros específicos que lo determinan. En *JPCasilla* se podría crear una operación concreta *guardarCasilla* que llamara a una operación abstracta *establecerTipo*, que dé valor al tipo de la casilla y a los restantes parámetros. Además, *JPCasilla* puede de paso incorporar el botón etiquetado *Guardar*, así como el código encargado de manejar el evento producido cuando éste se pulsa. El lado izquierdo de la Figura 95 muestra un pantallazo del entorno de desarrollo con parte del diseño y del código de la clase abstracta *JPCasilla*, la solapa genérica para manipular casillas, en donde la operación *establecerTipo* es abstracta; en el lado derecho se muestra la implementación de la operación en la solapa *JPCalle*.

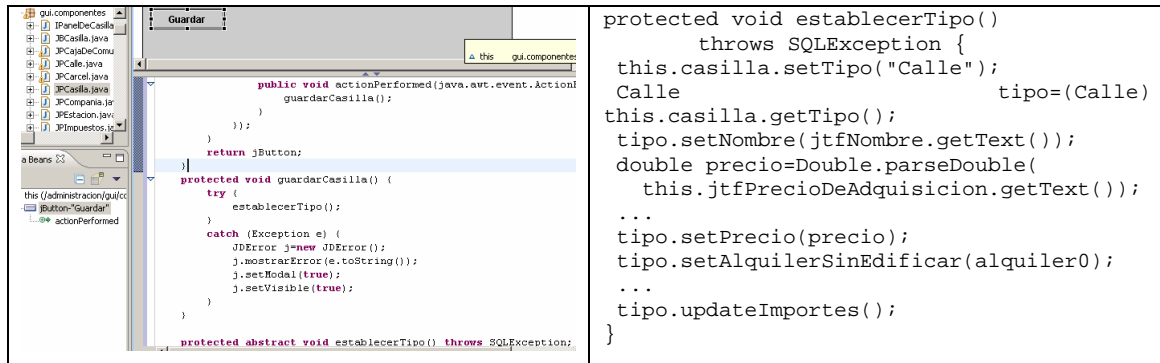


Figura 95. Reestructuración de las solapas para aprovechar la herencia

2) *setCasilla* asigna a la solapa la casilla que se pasa como parámetro y, si es necesario, muestra en la propia solapa información de la casilla. La operación puede subirse a *JPCasilla* y hacer que ejecute dos acciones: por un lado, asignar a la solapa la casilla pasada como parámetro y, por otro, mostrar la información mediante una llamada a una nueva operación *mostrarCasilla*, que será abstracta en la superclase (*JPCasilla*) y concreta en sus especializaciones. Esta solución se muestra en el Cuadro 56. La operación *mostrarCasilla* es redefinida de distintas formas en las diversas especializaciones de *JPCasilla*.

<pre> // En JPCasilla public void setCasilla(Casilla casilla) { this.casilla = casilla; if (casilla.getTipo() != null) mostrarCasilla(); } protected abstract void mostrarCasilla(); </pre>	<pre> // En JPCalle protected void mostrarCasilla() { Calle tipo = (Calle) casilla.getTipo(); this.jtfNombre.setText(tipo.getNombre()); this.jtfPrecioDeAdquisicion.setText("+" + tipo.getPrecio()); this.jtfValorHipotecario.setText("+" + tipo.getValorHipotecario()); this.jtfAlquiler0.setText("+" + tipo.getAlquilerSinEdificar()); this.jtfAlquiler1.setText("+" + tipo.getAlquiler1Casa()); this.jtfAlquiler2.setText("+" + tipo.getAlquiler2Casas()); this.jtfAlquiler3.setText("+" + tipo.getAlquiler3Casas()); this.jtfAlquiler4.setText("+" + tipo.getAlquiler4Casas()); this.jtfAlquiler5.setText("+" + tipo.getAlquilerConHotel()); } </pre>
<pre> // En JPParking protected void mostrarCasilla() { this.jlCasilla.setText("El parking estará situado en la casilla " + this.casilla.getPos()); } </pre>	

Cuadro 56. Implementación de *setCasilla* y de *mostrarCasilla*

3) *getTipo* devuelve simplemente una instancia del tipo correspondiente a la solapa. La operación puede declararse directamente abstracta en *JPCasilla* y redefinirla en las especializaciones.

<pre> // En JPCasilla public abstract Tipo getTipo(); </pre>	<pre> // En JPCalle public Tipo getTipo() { return new Calle(); } </pre>
--	--

Cuadro 57. Implementación de *getTipo* en *JPCasilla* y en *JPCalle*

Una vez se han hecho estos cambios, de casa solapa concreta habrá que eliminar el campo *casilla* (ya que *JPCasilla* posee un campo protegido y, por tanto, accesible para sus especializaciones, de tipo *Casilla*), el botón *Guardar* y el método *guardarCasilla*.

Además, en *JFTablero* sustituiremos el tipo del campo *solapaSeleccionada* (que era de tipo *IPanelDeCasilla*) por *JPCasilla*.

La Figura 96 muestra el diseño que se consigue con estos cambios: *JPCasilla* es una clase abstracta que conoce a una instancia de *Casilla*; las solapas concretas (*JPParking*, *JPCalle*, etc.) heredan de *JPCasilla*. La interfaz *IPanelDeCasilla* (que antes era implementada por las solapas concretas) deja ahora de tener utilidad y puede ser eliminada del sistema.

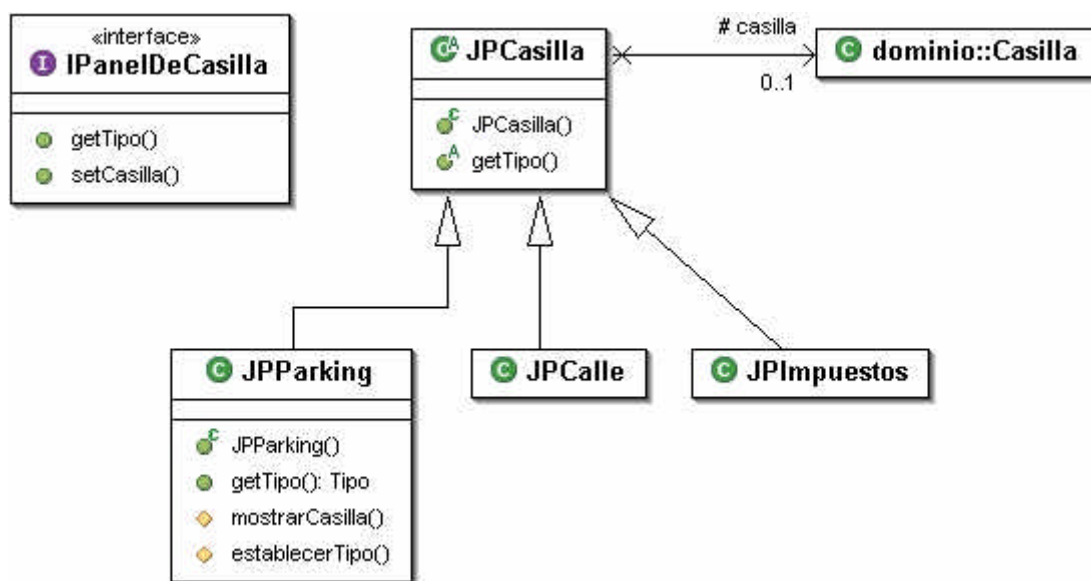


Figura 96. Fragmento del nuevo diseño de la capa de presentación

3.4. Decisiones sobre el desarrollo de otros casos de uso

Respecto de los casos de uso de creación de tarjetas (ver Tabla 31, página 180), podemos aprovechar las lecciones aprendidas de la creación de casillas: puesto que para esta aplicación únicamente debemos almacenar información de las tarjetas, optaremos por agrupar la creación de todas ellas en un solo caso de uso no abstracto *Crear tarjeta*. Además, y con propósitos ilustrativos, optaremos ahora por representar las tarjetas mediante una clase *Tarjeta* con varias especializaciones, en lugar de utilizar el patrón Estado para delegar el tipo (como sí hicimos con las casillas). Para su almacenamiento en la base de datos, optaremos por aplicar el patrón *Un árbol de herencia, una tabla*, de manera que se almacenarán todas en una única tabla *Tarjeta* de la base de datos.

Estas decisiones suponen la supresión de casos de uso y la modificación del plan de iteraciones.

3.5. Modificación del plan de iteraciones

La siguiente tabla muestra la nueva versión del plan de iteraciones, una vez agrupados todos los casos de uso dedicados a la creación de tarjetas en *Crear tarjeta*.

Fase	Iteración	Caso de uso	Flujos de trabajo				
			R	A	D	I	P
Comienzo	1	Identificación	X	X			
		Crear edición	X	X			
		Crear casillas vacías	X	X			
		Asignar tipo a casilla	X	X			
		Crear parking	X	X			
Elaboración	2	Identificación		X	X	X	X
		Crear edición		X	X	X	X
		Crear casillas vacías		X	X	X	X
		Asignar tipo a casilla		X	X	X	X
		Crear parking		X	X	X	X
		Crear salida		X	X	X	X
		Crear impuestos		X	X	X	X
		Crear cárcel		X			
		Crear compañía		X			
		Crear estación		X			
		Crear barrio		X			
		Crear calle		X			
		Crear vaya a la cárcel		X			
Construcción	3	Crear cárcel		X	X	X	X
		Crear compañía		X	X	X	X
		Crear estación		X	X	X	X
		Crear calle		X	X	X	X
		Crear vaya a la cárcel		X	X	X	X
	4	Crear caja de comunidad		X	X	X	X
		Crear suerte		X	X	X	X
	5	Crear tarjeta		X	X	X	X
	6	Gestionar usuarios	X	X	X	X	X
	7	Arrancar sistema	X	X	X	X	X
		Parar sistema	X	X	X	X	X
Transición							

Tabla 32. Nueva versión del plan de iteraciones

4. Descripción arquitectónica del sistema

4.1. Vista funcional de la arquitectura

La vista funcional de la arquitectura del sistema contendrá los casos de uso y los actores más relevantes del sistema, incluyendo también una descripción de los escenarios más importantes. En la descripción de los casos de uso se pueden incluir las máquinas de estados que representen su comportamiento.

Al finalizar la fase, debemos tener realizado el análisis de la práctica totalidad de los requisitos del sistema. De acuerdo con el plan de iteraciones (Tabla 6, página 81), hasta ahora se han analizado desde *Identificación* hasta *Crear impuestos*. Los restantes requisitos que consisten en la creación de determinados tipos de casillas son muy similares a los vistos en páginas anteriores; pueden diferir bastante los de creación de tarjetas con sus subtipos.

4.2. Vista de subsistemas con diagramas de paquetes

Si asimilamos que cada paquete Java es un subsistema, la aplicación hasta ahora construida constará ya de seis: el de presentación (al que llamaremos *gui*), el de dominio, el que contiene las excepciones creadas para manejar la apertura de la sesión del usuario, el de persistencia y el que almacena los casos de prueba de *junit*.

En principio, las únicas relaciones permitidas por UML en los diagramas de paquetes son las dependencias. Un paquete *A* depende de otro *B* cuando algún elemento de *A* depende de algún elemento de *B*. La Figura 97 muestra las relaciones de dependencia entre los paquetes de nuestro sistema.

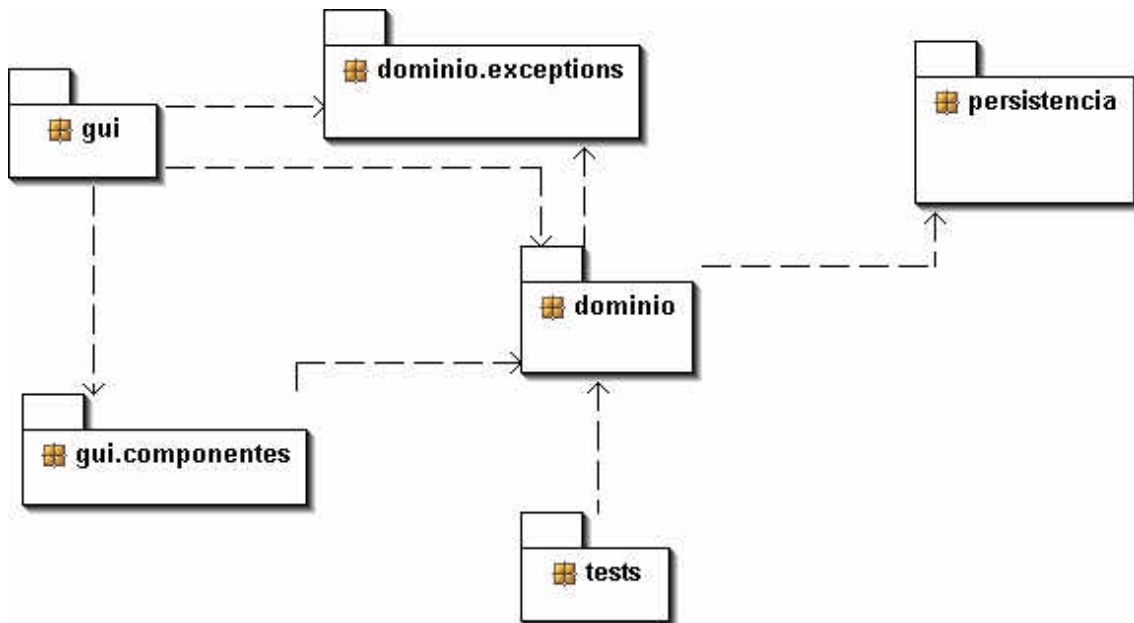


Figura 97. Diagrama de paquetes de la Aplicación para investigadores

5. Lecturas recomendadas

Capítulo 4 (An Architecture-Centric Process) de The Unified Software Development Process, de Jacobson, Booch y Rumbaugh. Addison-Wesley. Se define y describe con profundidad el concepto de arquitectura software

Capítulo 8 (Analysis) de The Unified Software Development Process, de Jacobson, Booch y Rumbaugh. Addison-Wesley. El capítulo explica las principales actividades del análisis, artefactos que se construyen, personal involucrado, estructuración en subsistemas, etc.

Capítulo 8. APLICACIÓN DE ADMINISTRACIÓN:

FASE DE CONSTRUCCIÓN

En la fase de Construcción se realiza el análisis detallado de los requisitos restantes, se desarrolla el software y se prueba. Las actividades que se desarrollan en esta fase son similares a las de la fase de Elaboración; no obstante, se debe llegar a este punto con las ideas muy claras acerca del diseño arquitectónico del sistema, diseño detallado, etc.

1. Estado actual de los requisitos

La siguiente tabla muestra el estado en que se encuentra el desarrollo de cada caso de uso en este momento.

Caso de uso	Análisis	Diseño	Codif.	Pruebas
Identificación				
Crear edición				
Crear casillas vacías				
Asignar tipo a casilla				
Crear parking				
Crear salida				
Crear impuestos				
Crear cárcel				
Crear compañía				
Crear estación				
Crear barrio				
Crear calle				
Crear vaya a la cárcel				
Crear caja de comunidad				
Crear suerte				
Crear tarjeta				
Gestionar usuarios				
Arrancar sistema				
Parar sistema				

Tabla 33. Estado actual de los requisitos

2. Desarrollo del caso de uso *Crear barrio*

Este caso de uso surge por la necesidad de tener los barrios creados antes de crear las calles.

Para crear los barrios seguiremos la misma idea que para crear las casillas vacías: al crear la Edición e insertarla en la base de datos, crearemos también los barrios con los colores por defecto, y los iremos insertando en la base de datos (Cuadro 58).

<pre> public void insert() throws SQLException { PreparedStatement p=null; String SQL="Insert into ..."; try { Connection bd=...; p=bd.prepareStatement(SQL); p.setString(1, this.ciudad); ... p.executeUpdate(); insertarCasillasVacias(); insertarBarriosVacios(); } catch (SQLException ex) { throw ex; } finally { p.close(); } } </pre>	<pre> private void insertarBarriosVacios() throws SQLException { barrios=new Vector(); Barrio b=new Barrio("Marrón"); barrios.add(b); b=new Barrio("Azul claro"); barrios.add(b); b=new Barrio("Rosa"); barrios.add(b); b=new Barrio("Naranja"); barrios.add(b); b=new Barrio("Rojo"); barrios.add(b); b=new Barrio("Amarillo"); barrios.add(b); b=new Barrio("Verde"); barrios.add(b); b=new Barrio("Azul"); barrios.add(b); for (int i=0; i<barrios.size(); i++) { b=(Barrio) barrios.get(i); b.setEdicion(this); b.insert(); } } </pre>
--	---

Cuadro 58. Fragmento del código de *Edicion::insert()*, al que se ha añadido una llamada a *insertarBarriosVacios* (a la derecha)

Además del color, el barrio tiene el precio de adquisición de cada casa. Para configurar estos precios, añadiremos un botón a la ventana de la Figura 67 (página 124) que habilite el diálogo de la para esta operación.

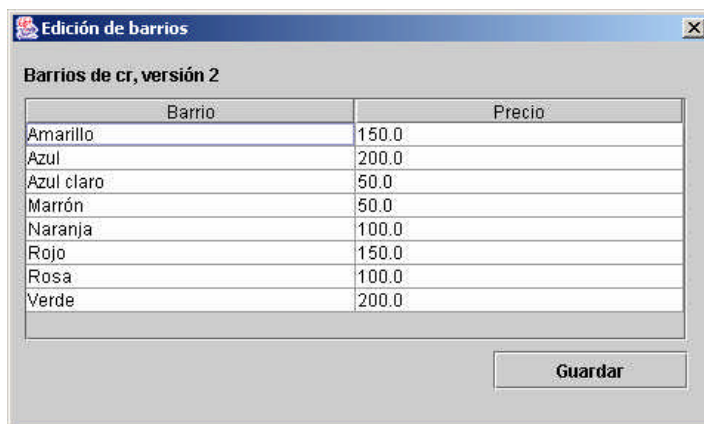


Figura 98. Diálogo para la asignación de precios de adquisición de casas a los barrios

Al pulsar el botón *Guardar*, el diálogo recupera los datos que aparecen en la tabla y va actualizando el precio de las casas de cada barrio en la base de datos.

3. Desarrollo del caso de uso *Crear tarjeta*

Como observa el lector, hemos obviado el desarrollo de los casos de uso de creación de casillas que nos quedaban, ya que son muy similares a los anteriores. El caso de uso *Crear tarjeta* no es que sea muy diferente, pero ya mencionamos en la página 185 que optaríamos por representar las tarjetas sin utilizar el patrón *Estado*, y aplicando el patrón *Un árbol de herencia, una tabla* para su almacenamiento en la base de datos.

La estructura del árbol de herencia de las tarjetas se muestra en la Figura 99: todas las tarjetas tienen un *texto* y son de un *tipo* (*Suerte* o *Caja de comunidad*); las tarjetas de tipo *IrA* envían al jugador a una *posicionDestino* determinada, pudiendo hacer que el jugador cobre o no en caso de que pase por la casilla de salida; las de *Avanzar* incluyen las tarjetas que mandan al usuario avanzar o retroceder un número fijo de casillas; *CobrarFija* y *PagarFija* incluyen el *importe* que el jugador cobra o paga a la banca; las tarjetas de tipo *PagarVariable* obligan al jugador a pagar una cantidad dependiente del número de casas y hoteles que tenga en sus calles; *CobrarDeOtros* indica la cantidad que el jugador debe cobrar de los otros jugadores.

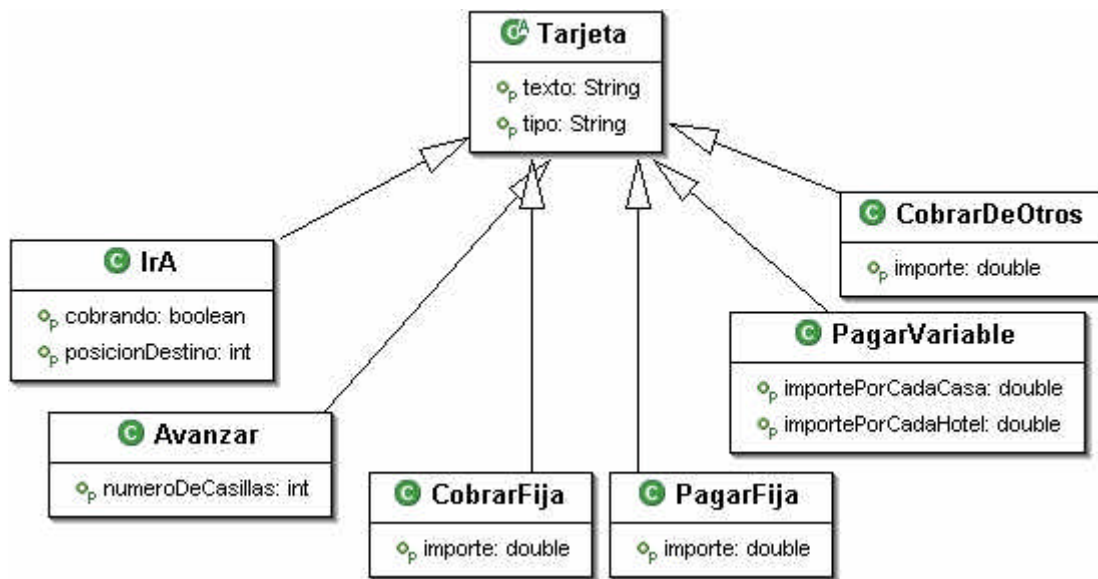


Figura 99. Estructura del subsistema de tarjetas

Respecto de la base de datos, todas las tarjetas se almacenarán en una única tabla *Tarjeta* (Figura 100), que contendrá columnas suficientes para representar los campos de cualquier registro. Además, se añade una columna adicional (*subtipo*) que representa el tipo de la tarjeta (*IrA*, *Avanzar*, etc.), y que es útil para materializar la tarjeta como una instancia del subtipo adecuado. La clave principal de *Tarjeta* estará compuesta por la de la *Edicion* más un número de tarjeta (columna *IdTarjeta*). Se podría haber añadido la columna *Texto*, pero haría una clave principal demasiado compleja.

Edición		Tarjeta			
?	Ciudad	Nombre de columna	Tipo de datos	Longitud	Permitir valores nulos
?	Version	Ciudad	varchar	50	
		Version	varchar	10	
		IdTarjeta	int	4	
		Texto	varchar	50	✓
		Tipo	varchar	50	✓
		Subtipo	varchar	20	✓
		Cobranco	bit	1	✓
		PosicionDestino	int	4	✓
		NumeroDeCasillas	int	4	✓
		Importe	float	8	✓
		ImportePorCadaCasa	float	8	✓
		ImportePorCadaHotel	float	8	✓

Figura 100. La tabla *Tarjeta*

La tabla puede ser anotada con restricciones *check* del lenguaje SQL para limitar los valores de las columnas. El Cuadro 59 muestra las cuatro restricciones añadidas:

- La primera indica que el tipo de tarjeta es *Suerte* o *Caja de comunidad*.
- La segunda indica que el importe por cada casa es o bien nulo, o bien un valor positivo.
- La tercera indica lo mismo, referido al importe por cada hotel.
- La cuarta indica que la posición de destino es o nula o un número entre 1 y 40.
- La quinta indica que el número de casillas que puede avanzarse debe estar entre -5 y 5 y ser además distinto de cero.

```

CONSTRAINT [CK_Tarjeta] CHECK ([Tipo] = 'Suerte' or [Tipo] = 'Caja de comunidad'),
CONSTRAINT [CK_Tarjeta_1]
    CHECK ([ImportePorCadaCasa] = null or [ImportePorCadaCasa] > 0),
CONSTRAINT [CK_Tarjeta_2]
    CHECK ([ImportePorCadaHotel] = null or [ImportePorCadaHotel] > 0),
CONSTRAINT [CK_Tarjeta_3]
    CHECK ([PosicionDestino] = null or [PosicionDestino] >= 1 and [PosicionDestino] <= 40),
CONSTRAINT [CK_Tarjeta_4]
    CHECK ([NumeroDeCasillas] = null or [NumeroDeCasillas] >= (-5) and
        [NumeroDeCasillas] <= 5 and [NumeroDeCasillas] <> 0)

```

Cuadro 59. Restricciones *check* añadidas la tabla *Tarjeta*

Añadir restricciones a la tabla libera al programador de la necesidad de controlar valores en el programa: si se intenta, por ejemplo, introducir un valor de *ImportePorCadaCasa* menor que cero, la base de datos lanzará una *SQLException*, que podrá ser capturada y procesada por el programa.

La Figura 101 muestra un pantallazo obtenido mientras se diseñaba la ventana para creación de tarjetas: la etiqueta situada arriba mostrará información sobre la Edición para la cual se están creando las tarjetas; el recuadro situado debajo es una tabla (de tipo *JTable*) que, en ejecución, se cargará con las tarjetas de esta edición, si es que las tiene. Si el usuario selecciona una tarjeta en la tabla y pulsa *Eliminar*, la tarjeta se elimina de la base de datos; si pulsa *Ver*, se pondrá en primer plano la solapa correspondiente al tipo de

tarjeta y se cargará con su información; si se pulsa *Insertar*, se creará un objeto de clase *Tarjeta* (instanciado al subtipo correspondiente a la solapa seleccionada) y se insertará en la base de datos.

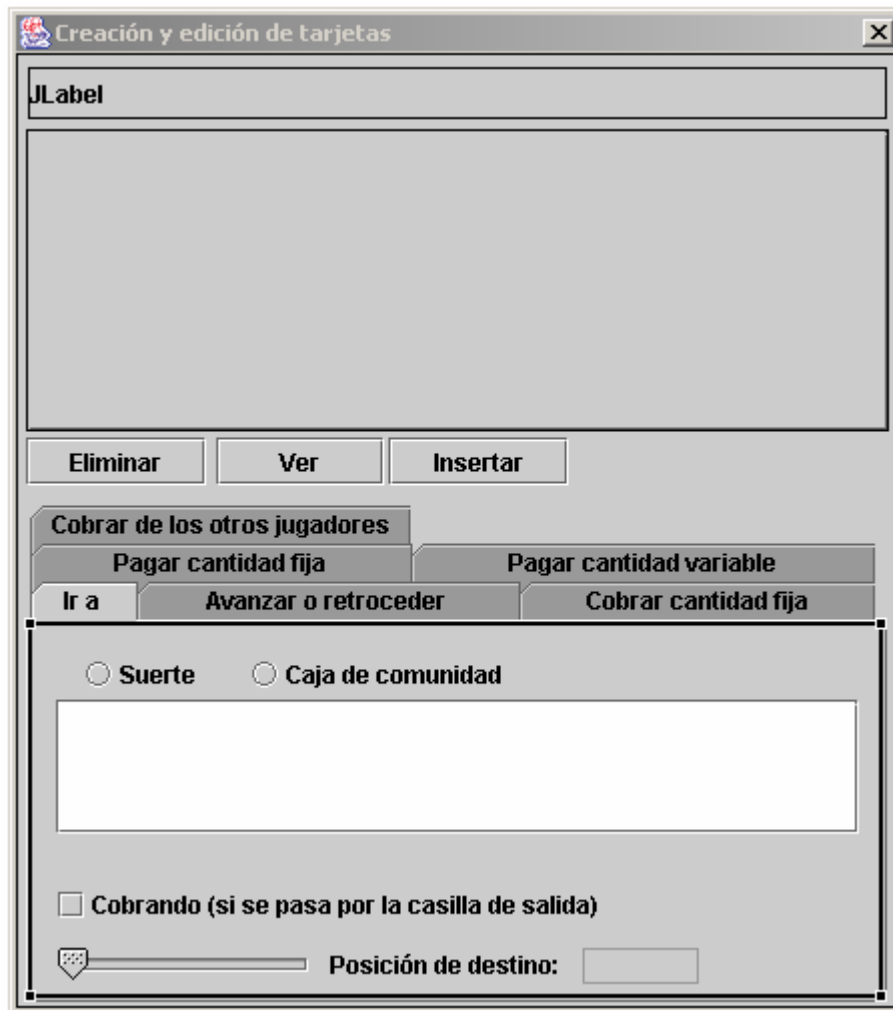


Figura 101. Diseño de la ventana de creación de tarjetas (clase *JDTarjetas*)

La ventana anterior se mostrará pulsando un botón etiquetado *Tarjetas* situado en la ventana de edición del tablero. Como se ha dicho, al mostrarse el diálogo *JDTarjetas*, debe cargarse la tabla con la lista de tarjetas de la edición. De esta operación se encarga el constructor *JDTarjetas(Edicion)* mostrado en el Cuadro 60, que llama al método *loadTarjetas* mostrado en el mismo cuadro. El elemento *modelTarjetas* utilizado en el código de las dos operaciones es un objeto de tipo *DefaultTableModel*, una clase que se utiliza para manipular el contenido de las *JTable*.

```
public JDTarjetas(Edicion edicion) throws SQLException {
    super();
    initialize();
    this.edicion=edicion;
    this.jlEdicion.setText("Tarjetas de " + edicion.getCiudad() + ", versión " +
        edicion.getVersion());
    String[] cabecera={"Id", "Tipo", "Subtipo"};
    modelTarjetas.setColumnIdentifiers(cabecera);
    loadTarjetas();
}

private void loadTarjetas() throws SQLException {
    Vector datos=Tarjeta.loadTarjetas(this.edicion);
    for (int i=0; i<datos.size(); i++) {
        Vector fila=(Vector) datos.get(i);
        modelTarjetas.addRow(fila);
    }
}
```

Cuadro 60. Constructor de *JDTarjetas*

Para mantener la independencia de la capa de presentación con respecto de la de persistencia, *loadTarjetas* no accede directamente a la base de datos, sino que carga los datos a partir de la operación estática *loadTarjetas(Edicion)* definida en *Tarjeta* y mostrada en el Cuadro 61: como se ve, el método recupera de la tabla *Tarjeta* las columnas *IdTarjeta*, *Tipo* y *Subtipo*, que son las que luego se muestran en la tabla del diálogo.

```
public static Vector loadTarjetas(Edicion edicion) throws SQLException {
    PreparedStatement p=null;
    Vector result=new Vector();
    String SQL="Select IdTarjeta, Tipo, Subtipo from Tarjeta where Ciudad=? " +
        "and Version=? order by Tipo, IdTarjeta";
    try {
        Connection bd=Agente.getAgente().getDB();
        p=bd.prepareStatement(SQL);
        p.setString(1, edicion.getCiudad());
        p.setString(2, edicion.getVersion());
        ResultSet rs=p.executeQuery();
        while (rs.next()) {
            int id=rs.getInt(1);
            String tipo=rs.getString(2);
            String subtipo=rs.getString(2);
            Vector fila=new Vector();
            fila.add(new Integer(id));
            fila.add(tipo);
            fila.add(subtipo);
            result.add(fila);
        }
    }
    catch (SQLException ex) { throw ex; }
    finally { p.close(); }
    return result;
}
```

Cuadro 61. Código de *loadTarjetas*, definido en *Tarjeta*

Las solapas correspondientes a los tipos de tarjeta tienen todas un fragmento común y un fragmento distinto. La siguiente figura muestra el diseño de las solapas *JPIrA* (para tarjetas de tipo *IrA*) y *JPAvanzar* (para tarjetas de tipo *Avanzar*): los dos fragmentos recuadrados son un elemento de la misma clase, un panel llamado *JPComun*, que incluye los botones de selección para el tipo de tarjeta y la caja para su texto.

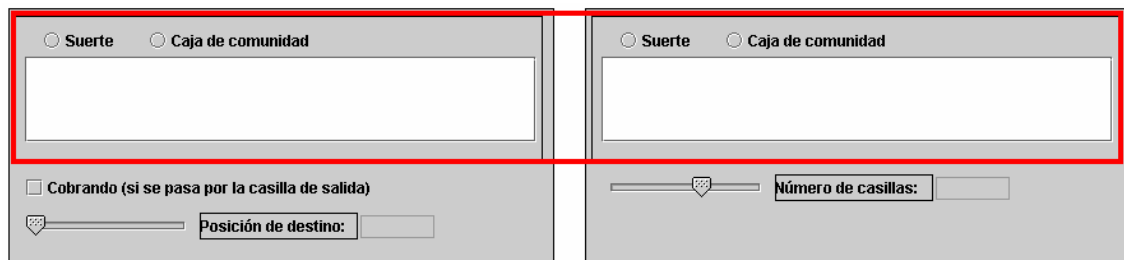


Figura 102. Las solapas tienen un elemento común (un *JPComun*)

A nivel de clases, las consideraciones anteriores se representan de la siguiente forma:

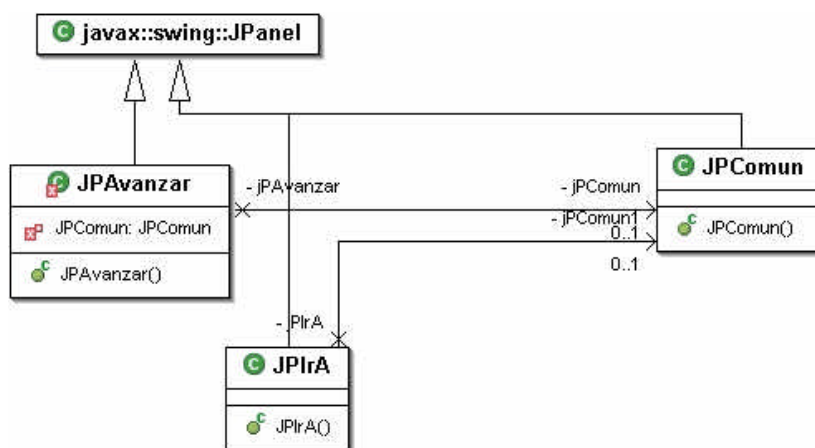


Figura 103. Las solapas conocen a un *JPComun*

Describamos a continuación los escenarios normales de la inserción, visualización y eliminación de tarjetas.

3.1. Escenario correspondiente a la inserción de una tarjeta

Supondremos, para este escenario, que el usuario decide crear una tarjeta de tipo *IrA*. La experiencia que ya tenemos de anteriores casos de uso nos permite describir mucho más detalladamente (en términos del diálogo *JDTarjetas*, la solapa *JPirA*, el subtipo *IrA*, etc.) la secuencia de operaciones:

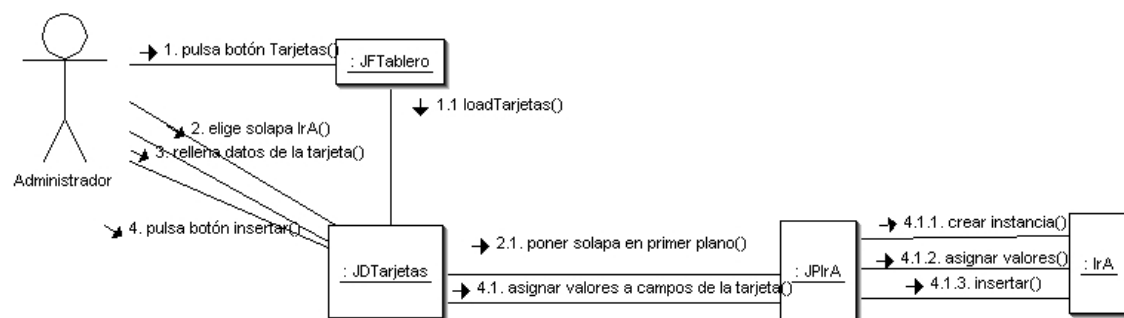


Figura 104. Diagrama de colaboración correspondiente al inserción de una Tarjeta de tipo *IrA*

A la vista del diagrama, al pulsar el botón *Insertar* se debe crear una instancia de tipo *IrA*, cargarla con los valores escritos por el usuario y ejecutar su operación *insert*. Para que la solapa *JPIrA* cree la instancia de tipo *IrA*, puede hacerse que *JPIrA* conozca a una instancia de tipo *IrA*. Análogamente, *JPAvanzar* conocerá a otra de tipo *Avanzar*, etc. Es decir:

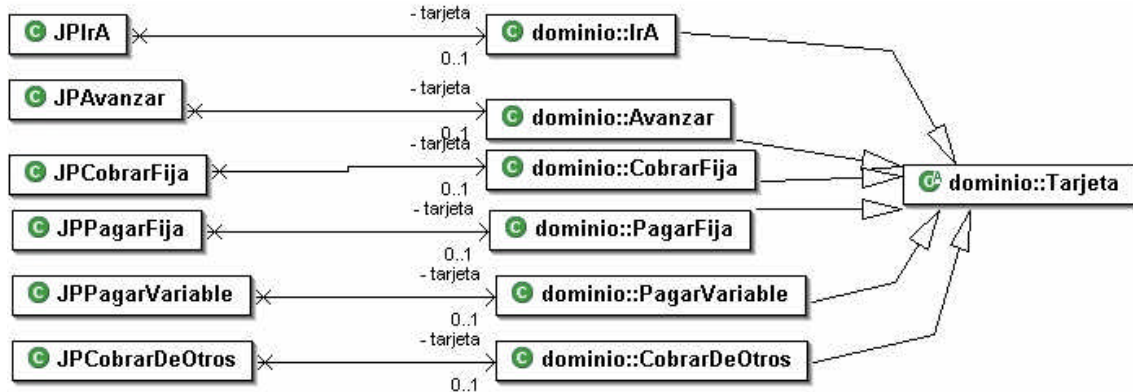


Figura 105. Cada tipo de solapa conoce a un determinado tipo de tarjeta

Para desencadenar los mensajes correspondientes al evento 4, el diálogo debe decirle a la solapa situada en primer plano que asigne a la *tarjeta* a la cual conoce los valores que el usuario ha indicado en la propia solapa (tipo de tarjeta, texto, etc.). El código podría ser el mostrado en el siguiente cuadro: del *JTabbedPane*, que es el componente contenedor de solapas, tomamos la solapa seleccionada en la forma de un *JPanel* (véase en la Figura 103 que las solapas son especializaciones de *JPanel*) y, sobre ella, ejecuta la operación *actualizarTarjeta*.

```
protected void insertar() {
    JPanel solapa=(JPanel) this.jTabbedPane.getSelectedComponent();
    solapa.actualizarTarjeta();
}
```

Cuadro 62. Intento de pasar el mensaje de inserción de la tarjeta a la solapa (en *JDtarjetas*)

Sin embargo, *JPanel*, que es una clase de biblioteca del paquete *javax.swing*, no dispone de la operación *actualizarTarjeta*, por lo que debemos decidírnos por alguna alternativa.

La Figura 106 muestra el diseño de la que podría ser la primera alternativa: de forma parecida a la solución que se dio al gestionar las solapas de los diferentes tipos de casilla, se crea una clase abstracta *JPTarjeta* con una operación abstracta *actualizarTarjeta*; se construyen tantas solapas como tipos de tarjetas, cada una implementando la operación abstracta heredada. A diferencia de la solución dada para las casillas (en las que se utilizaba el patrón *Estado*), cada solapa conoce a un subtipo de *Tarjeta*, sobre el que actúa. Además, se aprovecha la creación de la solapa abstracta *JPTarjeta* para

colocar en ella la solapa de información común (campo *comun*, de tipo *JPComun*), que es heredada por las especializaciones.

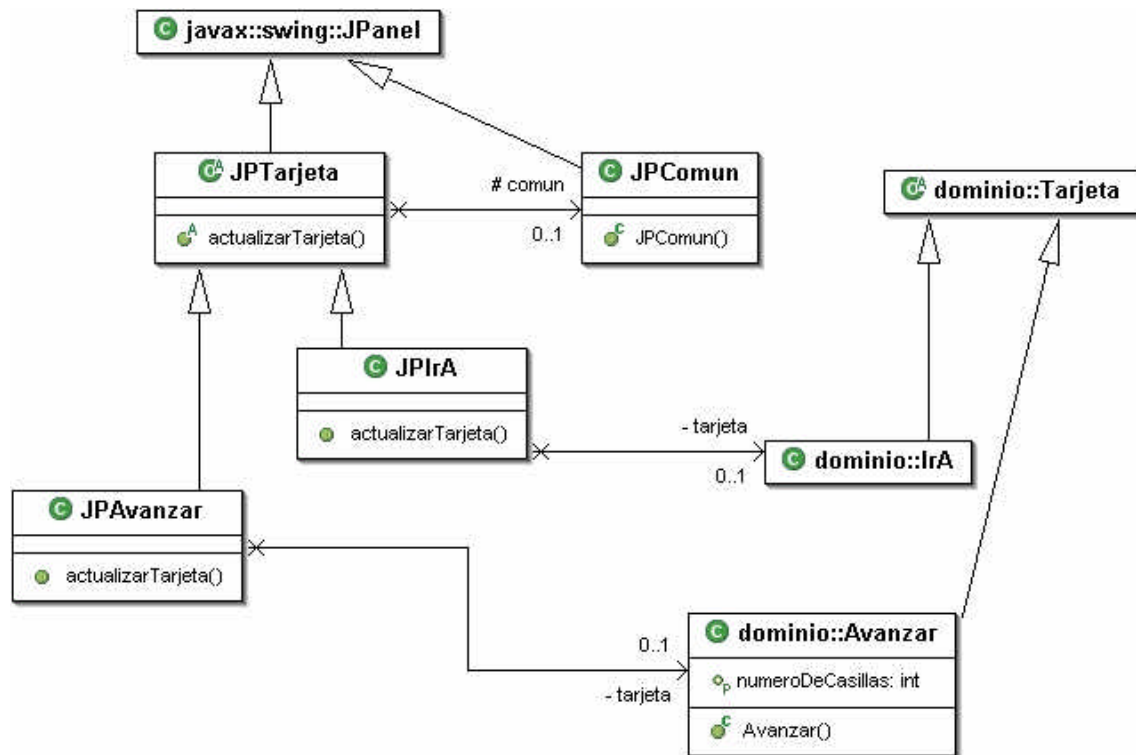


Figura 106. Esquema de la alternativa 1

Tal vez se recuerde que, para solucionar el tema de las casillas, se creó inicialmente una interfaz llamada *IPanelDeCasilla* que implementaban las solapas. La interfaz declaraba una operación abstracta que proporcionaba un mecanismo uniforme de comunicación con las solapas (Figura 69, página 126). Una solución similar puede ser válida para este caso, como se muestra en la Figura 107.

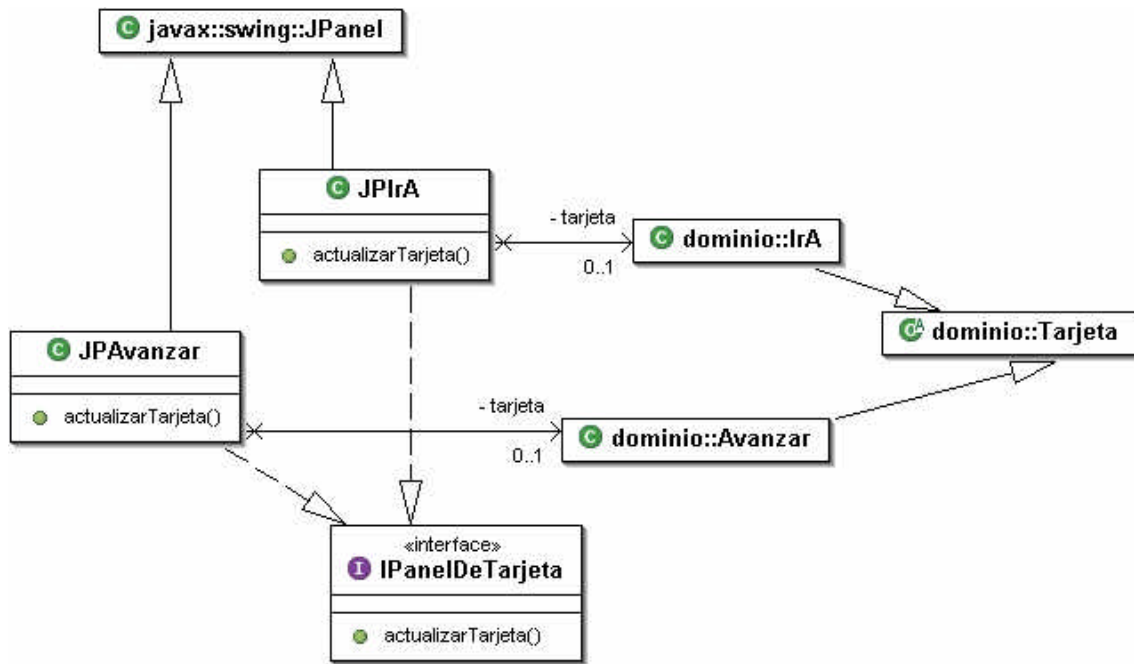


Figura 107. Esquema de la alternativa 2

Con el fin de diversificar el tipo de soluciones adoptadas, utilizaremos la segunda alternativa para la implementación. Así, el código que hemos mostrado en el Cuadro 62, y que decíamos que no era válido, lo reescribimos de esta forma:

```

protected void insertar() {
    IPanelDeTarjeta solapa=(IPanelDeTarjeta) this.jTabbedPane.getSelectedComponent();
    solapa.actualizarTarjeta();
}

```

Cuadro 63. Versión modificada del Cuadro 62

La redefinición de *actualizarTarjeta* en la solapa que nos ocupa podría ser la siguiente, en la que se asigna valor a los campos de *tarjeta* y luego se le dice que se inserte.

```

public void actualizarTarjeta() {
    this.tarjeta=new IrA();
    this.tarjeta.setTipo(this.JPComun.getTipo());
    this.tarjeta.setTexto(this.JPComun.getTexto());
    this.tarjeta.setCobrandeo(this.jchCobrandeo.isSelected());
    this.tarjeta.setPosicionDestino(this.jsPosicionDeDestino.getValue());
    this.tarjeta.insertar();
}

```

Cuadro 64. *actualizarTarjeta* (en *IrA*)

La tarjeta necesita conocer la ciudad y versión correspondientes a la edición para la que se está creando (véase Figura 100, página 192), por lo que debemos pasar este objeto de alguna manera: por ejemplo, como argumento a la operación *insertar*. Pero para ello, primero hay que pasarlo como parámetro a la operación *actualizarTarjeta*, por lo que es necesario redefinirlo en la interfaz y en todas sus especializaciones. Por último, habrá que declarar la

excepción *SQLException* en la cabecera de algunas operaciones, y darle el tratamiento con un *try-catch* en otras.

Para finalizar con esta sección, se ofrece en el código del método *insertar* de la clase *IrA*: de acuerdo con el patrón *Un árbol de herencia, una tabla*, inserta un registro en la tabla *Tarjeta*, pero sólo en aquellas columnas utilizadas para representar valores de los objetos de clase *IrA*.

```
public void insertar(Edicion edicion) throws SQLException {
    PreparedStatement p=null;
    try {
        Connection bd=Agente.getAgente().getDB();
        String SQL="Insert into Tarjeta " +
            "(Ciudad, Version, Texto, Tipo, Subtipo, Cobrando, PosicionDestino) " +
            " values (?, ?, ?, ?, ?, ?, ?)";
        p=bd.prepareStatement(SQL);
        p.setString(1, edicion.getCiudad());
        p.setString(2, edicion.getVersion());
        p.setString(3, this.texto);
        p.setString(4, this.tipo);
        p.setString(5, Tarjeta.IR_A);
        p.setBoolean(6, this.cobrando);
        p.setInt(7, this.posicionDestino);
        p.executeUpdate();
    }
    catch (SQLException ex) { throw ex; }
    finally { p.close(); }
}
```

Cuadro 65. Código de *insertar* (en *IrA*)

3.2. Escenario correspondiente a la visualización de las tarjetas

La Figura 108 muestra el diálogo *JDTarjetas* mostrando la información de las tarjetas creadas para la edición de la ciudad *cr*, versión *1* (de momento, sólo se ha creado una tarjeta de tipo *Suerte* con el escenario anterior).

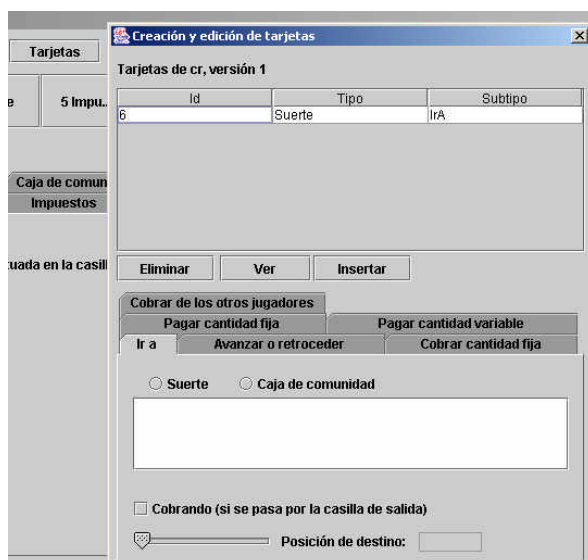


Figura 108. El diálogo *JDTarjetas*, mostrando información de una tarjeta

Ahora se desea que, al hacer clic sobre la fila correspondiente a la tarjeta cuyo *Id* es 6, se ponga en primer plano la solapa *Ir a* y se cargue la información de la tarjeta. Gráficamente, el proceso es el siguiente:

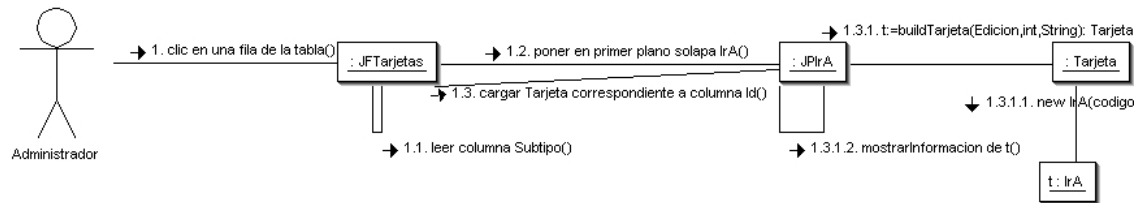


Figura 109. Descripción del escenario de visualización de la información de una tarjeta de tipo *IrA*

De este escenario, puede ser interesante destacar que se ha añadido la operación *cargarTarjeta(int idTarjeta)* a la interfaz *IPanelDeTarjeta*, lo que ha supuesto su implementación en todas las solapas. En el caso de *JPIrA*, la operación queda como en el Cuadro 66, en donde:

- En primer lugar, se instancia el campo *tarjeta* a una instancia del subtipo *IrA*.
- En segundo lugar, se carga el *JPComun* al cual conoce (Figura 102 y Figura 103) con la información del tipo y texto de la tarjeta.
- Por último, se muestra la información propia de este tipo de tarjeta.

```

public void cargarTarjeta(int id) throws SQLException {
    this.tarjeta=new IrA(id);
    this.JPComun.cargarTarjeta(this.tarjeta);
    this.jchCobrando.setSelected(this.tarjeta.isCobrando());
    this.jsPosicionDeDestino.setValue(this.tarjeta.getPosicionDestino());
    this.jtfPosicionDeDestino.setText(""+this.tarjeta.getPosicionDestino());
}
  
```

Cuadro 66. Implementación de *cargarTarjeta* en *JPIrA*

El constructor materializador de *IrA*, al que se llama en la primera instrucción del Cuadro 66 llama en primer lugar al materializador de su superclase (*Tarjeta*), y luego lee aquellas columnas de *Tarjeta* que le interesan.

```

public IrA(int id) throws SQLException {
    super(id);
    PreparedStatement p=null;
    try {
        Connection bd=Agente.getAgente().getDB();
        String SQL="Select Cobrando, PosicionDestino from Tarjeta where IdTarjeta=?";
        p=bd.prepareStatement(SQL);
        p.setInt(1, id);
        ResultSet rs=p.executeQuery();
        if (rs.next()) {
            this.cobrando=rs.getBoolean(1);
            this.posicionDestino=rs.getInt(2);
        } else throw new SQLException("Casilla no encontrada");
    }
    catch (SQLException ex) { throw ex; }
    finally { p.close(); }
}
  
```

Cuadro 67. Constructor materializador de *IrA*

3.3. Escenario correspondiente a la eliminación de las tarjetas

Para este escenario, supondremos que, antes de ser eliminada, es necesario mostrar información de la tarjeta que se desea eliminar. Así pues, podemos describirlo ampliando el diagrama de colaboración de la Figura 109 con el mensaje nº 2 de la siguiente figura y aquellos que le siguen:

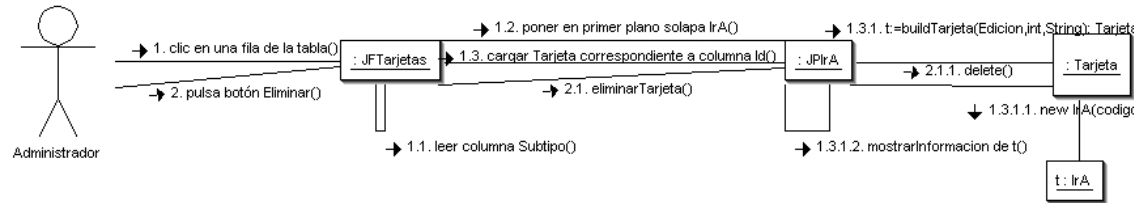


Figura 110. Descripción del escenario de eliminación de una tarjeta de tipo IrA

Este diagrama supone la adición de operaciones a varios elementos: el lado izquierdo del siguiente cuadro muestra la posible implementación de la operación *eliminar* en el diálogo, que se ejecuta al pulsar el botón etiquetado *Eliminar* de la Figura 108; como se observa a la derecha, se requiere la adición de dos operaciones a la interfaz *IPanelDeTarjeta* (mostradas a la derecha, arriba); estas operaciones deben implementarse en cada solapa (derecha, centro); a diferencia de la operación *insert*, que era redefinida en cada especialización de tarjeta, la operación *delete* puede tener una implementación en la clase abstracta *Tarjeta* que sea válida para todas las especializaciones (derecha, abajo).

<pre>protected void eliminar() { int fila=this.jTable.getSelectedRow(); if (fila==-1) return; String sId=this.jTable.getValueAt(fila, 0).toString(); int id=Integer.parseInt(sId); IPanelDeTarjeta solapa= (IPanelDeTarjeta) this.jTabbedPane.getSelectedComponent(); if (solapa.getTarjeta()==null) return; try { solapa.eliminarTarjeta(id); this.loadTarjetas(); } catch (SQLException ex) { JOptionPane.showMessageDialog(this, ex.getMessage(),"Error", JOptionPane.OK_OPTION); } }</pre>	<pre>Tarjeta getTarjeta(); void eliminarTarjeta(int id) throws SQLException; public void eliminarTarjeta(int id) throws SQLException { this.tarjeta.delete(id); } public void delete(int id) throws SQLException { PreparedStatement p=null; try { Connection bd= Agente.getAgente().getDB(); String SQL= "Delete from Tarjeta where IdTarjeta=?"; p=bd.prepareStatement(SQL); p.setInt(1, id); p.executeUpdate(); } catch (SQLException ex) { throw ex; } finally { p.close(); }</pre>
---	---

Cuadro 68. Código añadido a diferentes elementos de la aplicación

4. Posibles refactorizaciones

La Figura 111 muestra las clases contenidas en el paquete de dominio del sistema. Además de la clase *Sesion*, En él se mezclan las clases correspondientes a casillas y tarjetas. Un agrupamiento más razonable crearía dos

subpaquetes dentro de dominio, en donde se colocarían las clases correspondientes a cada tipo. A este tipo de cambios, que modifican la calidad del sistema (al menos, desde el punto de vista del ingeniero de software) pero sin alterar la funcionalidad se los llama refactorizaciones.

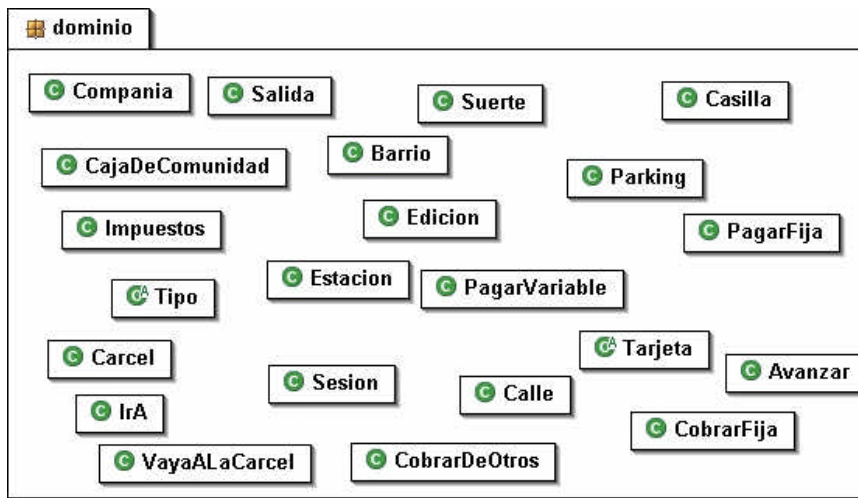


Figura 111. Clases contenidas en el paquete de dominio

Algunos entornos de desarrollo facilitan la refactorización. Para llevar a cabo la refactorización mencionada, Eclipse incluye la posibilidad de seleccionar las clases e interfaces que se desean mover y elegir el paquete de destino; el entorno, entonces, actualiza todas las instrucciones en las que se hace referencia a los elementos cambiados de sitio para que la aplicación continúe operativa (Figura 112).

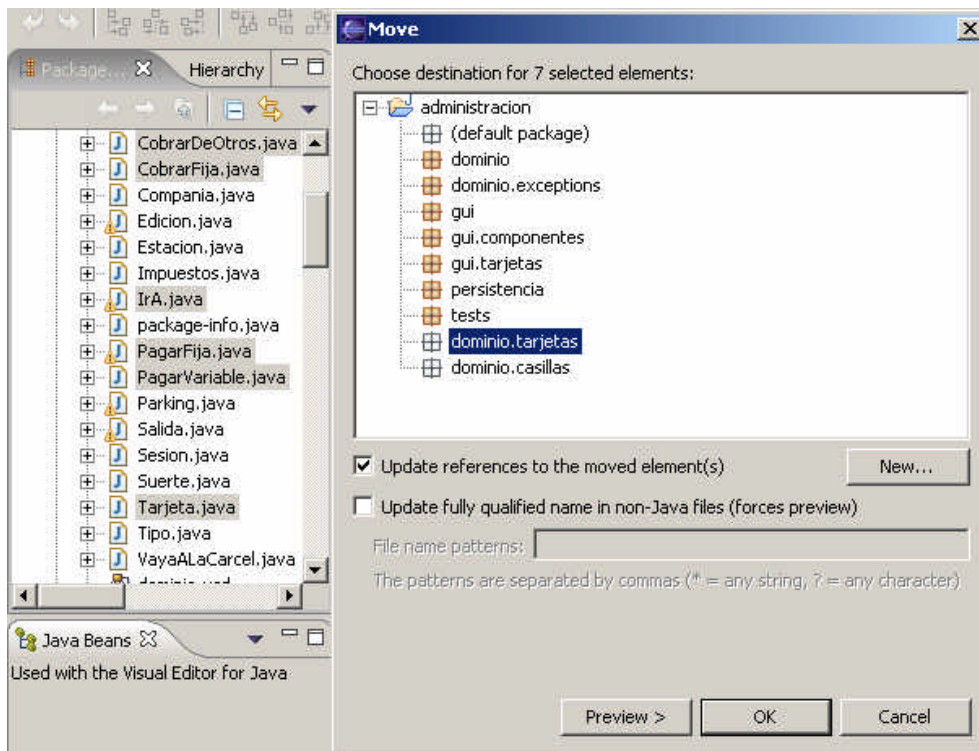


Figura 112. Asistente para la reubicación de clases de Eclipse

Hechas las dos refactorizaciones para mover las tarjetas al subsistema *dominio.tarjetas* y las casillas a *dominio.casillas*, la organización del código queda mucho más clara:

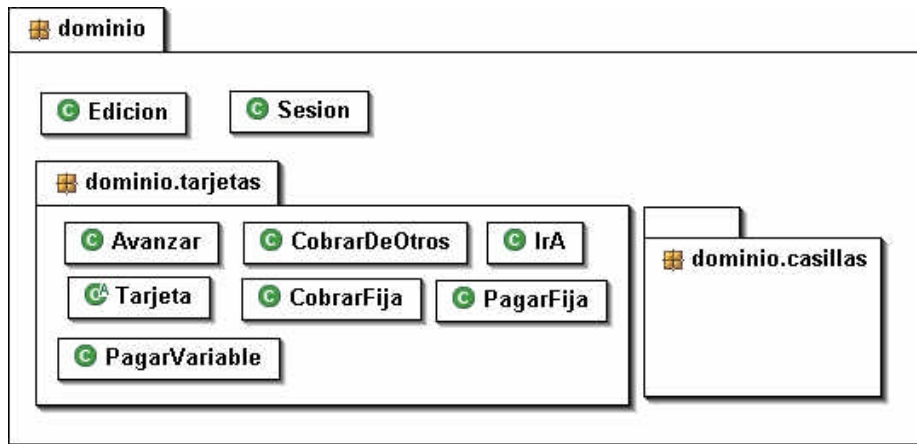


Figura 113. Estructura del paquete de dominio después de refactorizar

5. Pruebas de clases abstractas con JUnit

Como siempre, al término de cada iteración deben realizarse pruebas para comprobar que el funcionamiento del sistema se ajusta a las expectativas. En esta ocasión, probaremos los diferentes tipos de tarjetas creando una clase abstracta que contendrá casos de prueba abstractos para probar tarjetas.

El siguiente cuadro muestra el posible código de la clase abstracta *TestTarjeta*, que contiene casos de prueba abstractos:

```

package tests.tarjetas;

import junit.framework.TestCase;
import ...;

public abstract class TarjetaTest extends TestCase {
    public abstract void testInsertarTarjetaConTipo();
    public abstract void testInsertarTarjetaSinTipo();
    public abstract void eliminarTarjeta();
}
  
```

Cuadro 69. Una clase de prueba abstracta con casos de prueba abstractos

Si queremos reutilizar la clase anterior para probar, por ejemplo, la clase *IrA*, debemos crear una especialización *TestIrA* de *TestTarjeta*, en la que redefinimos los tres métodos abstractos anteriores.

El Cuadro 70 muestra el código de dos de los tres métodos implementados en la especialización: obsérvese que en su implementación, los objetos bajo prueba son instancias de las clases concretas. Por cierto, el caso de prueba *testInsertarTarjetaSinTipo* encuentra un error al ser ejecutado por primera vez, ya que se la tarjeta se inserta sin tipo, por lo que se alcanza el primer *fail*. Corregida la base de datos para que la columna *Tipo* de la tabla

Tarjeta no admita nulos (Figura 100, página 192), el caso de prueba se supera con éxito.

<pre> public void testInsertarTarjetaConTipo() { try { IrA tarjeta=new IrA(); tarjeta.setCobrando(true); tarjeta.setPosicionDestino(11); tarjeta.setTexto("Vaya a la cárcel"); tarjeta.setTipo("Suerte"); Edicion edicion=new Edicion("cr", "1"); int numPre=numeroDeTarjetas(edicion); tarjeta.insertar(edicion); int numPost=numeroDeTarjetas(edicion); assertTrue(numPost==numPre+1); } catch (Exception ex) { fail("No debería haber fallado"); } } </pre>	<pre> public void testInsertarTarjetaSinTipo() { try { IrA tarjeta=new IrA(); tarjeta.setCobrando(true); tarjeta.setPosicionDestino(11); tarjeta.setTexto("Vaya a la cárcel"); /* tarjeta.setTipo("Suerte"); */ Edicion edicion= new Edicion("cr", "1"); tarjeta.insertar(edicion); fail("Debería haber lanzado " + "una SQLException"); } catch (SQLException ex) { } catch (Exception ex) { fail("Debería haber lanzado " + "una SQLException"); } } </pre>
--	---

Cuadro 70. Implementación en *TestIrA* de dos de los casos de prueba abstractos

Como se observa, el primer caso de prueba utiliza el método *numeroDeTarjetas(Edicion)*, que se espera devuelva el número de tarjetas de la edición que se pasa como parámetro. Este método puede ser concreto y encontrarse implementado en la superclase abstracta *TestTarjeta*.

6. Lecturas recomendadas

Capítulo 15 (Construction leads to initial operational capability) de The Unified Software Development Process, de Jacobson, Booch y Rumbaugh. Addison-Wesley. Se explican las principales actividades de la fase de construcción.

Capítulo 9. APLICACIÓN DE ADMINISTRACIÓN:

FASE DE TRANSICIÓN

El objetivo principal de la fase de transición es la entrega al cliente de la versión del sistema producida en este ciclo. Para ello, se realizarán pruebas de aceptación, se crearán los procedimientos de instalación del sistema, manuales de usuario, sistema de ayuda, etc. Adicionalmente, se revisan y actualizan los modelos elaborados durante el desarrollo.

Las pruebas de aceptación tienen como objetivo comprobar que el sistema construido se adapta a las necesidades de los usuarios, por lo cual son realizadas por ellos mismos. Si el número final de usuarios es muy grande pueden realizarse las llamadas pruebas *beta*, mediante las cuales el sistema final se pone a disposición de los usuarios para que devuelvan errores, fallos y sugerencias al equipo de desarrollo.

En principio, el sistema debe llegar a esta fase muy estable y con pocas posibilidades de cambio. Jacobson, Booch y Rumbaugh, en su libro dedicado al Proceso Unificado de Desarrollo, indican que las modificaciones al sistema en la fase de transición no deberían afectar a más de un 5% de éste.

Para realizar las pruebas de aceptación, el sistema debe estar disponible para los usuarios finales. Para ello, es preciso haber realizado con anterioridad la implantación del sistema, la carga de datos y, posiblemente, sesiones de formación.

Las pruebas de aceptación se superan cuando los usuarios han cubierto las funcionalidades principales (por ejemplo, las representadas en los casos de uso).

Capítulo 10. RESUMEN

En este capítulo se presenta un resumen de las actividades realizadas en cada una de las fases, haciéndolas corresponder con la estructura del Proceso Unificado de Desarrollo.

1. XXX

La aplicación de administración se ha desarrollado en un solo ciclo.

Tercera parte

Capítulo 11. APLICACIÓN SERVIDORA (I)

En este capítulo se comienza el desarrollo de la aplicación servidora, que debe permitir a los usuarios jugar al Monopoly conectándose al servidor.

Desarrollaremos las funcionalidades utilizando desarrollo dirigido por las pruebas (*test-driven development*).

1. Recolección de requisitos

La aplicación que debemos construir ahora es una aplicación servidora, en donde los clientes serán las aplicaciones utilizadas por cada jugador, que desarrollaremos en ciclos próximos. El servidor deberá encargarse de gestionar el desarrollo completo de cada partida.

A primera vista, los requisitos deseados para esta aplicación serán los siguientes:

1. Un jugador previamente registrado podrá identificarse.
2. Un jugador podrá proponer una partida y estar a la espera de que ésta alcance el número deseado de jugadores para empezar la partida.
3. Un jugador podrá consultar la lista de partidas con vacantes, para poder unirse a ellas.
4. El usuario que propone una nueva partida podrá comenzarla desde el momento en que se haya apuntado a ella, al menos, un jugador más.
5. Un jugador podrá tirar los dados para mover su ficha. En función de lo que le haya salido en los dados y de la casilla de destino, podrá realizar una serie de acciones.

De este modo, una primera vista funcional podría ser la mostrada en la Figura 114.

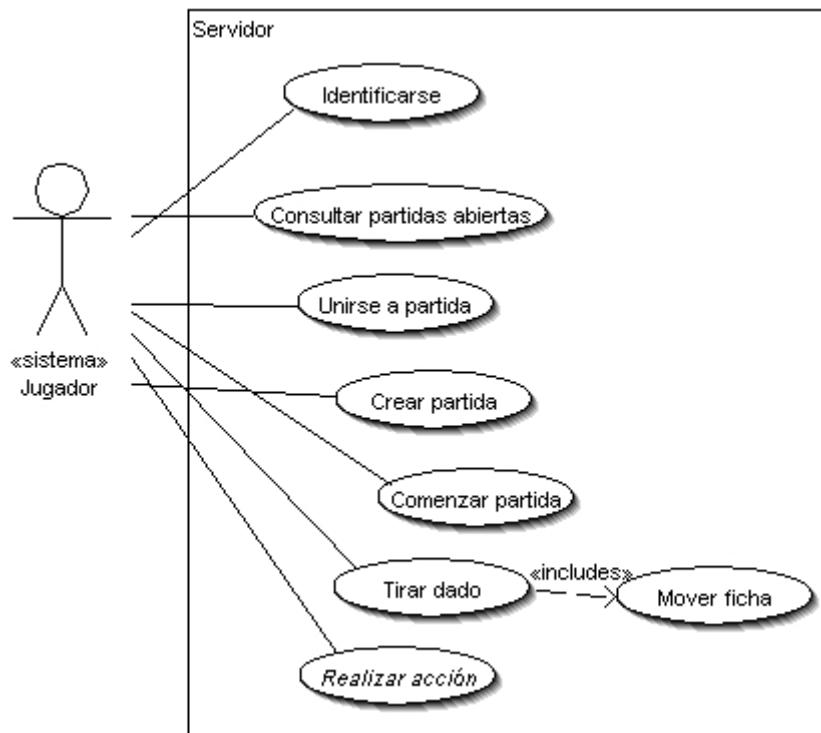


Figura 114. Funcionalidades principales de la aplicación servidora

En la figura anterior se observa que el jugador es considerado un sistema, no un humano, ya que quien se comunicará con el Servidor será un conjunto de aplicaciones cliente que, éstas sí, serán manejadas por personas. De alguna manera, entonces, deberemos exponer las funcionalidades ejecutables por los clientes mediante algún *middleware* que permita la comunicación de aplicaciones remotas.

2. El patrón *Fachada*

La fachada es un patrón de Gamma et al.⁹ que nos aconseja utilizar una clase como punto de exposición de un subsistema, a la que se denomina *fachada*. Cuando alguien desea acceder a alguno de los servicios prestados por el subsistema, lo hace a través de la fachada, que encamina la petición al elemento adecuado del subsistema. La Figura 115 muestra un subsistema genérico con tres clases de dominio (*A*, *B* y *C*) que ofrecen al resto del sistema un subconjunto de sus servicios mediante una fachada. Cuando *ClaseExterna1* o *ClaseExterna2* desea utilizar alguna de las operaciones de *A*, *B* o *C*, lo hace a través de la fachada.

⁹ E. Gamma, R. Helm, R. Johnson y J. Vlissides. *Patrones de Diseño*. Editorial Addison-Wesley, 2003.

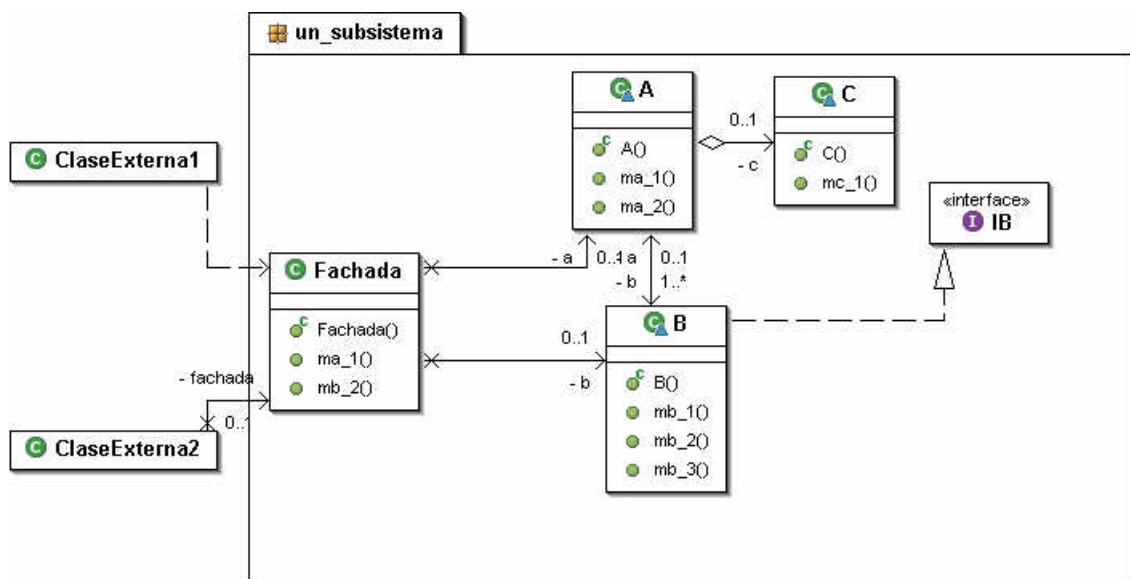


Figura 115. Un subsistema genérico con una fachada

El uso de esta solución es adecuada para el desarrollo del servidor: en este caso, las clases externas estarán constituidas por el sistema externo *Jugador*, al que la fachada expondrá los servicios accesibles de forma remota, cuyas llamadas encaminará posteriormente a la clase correspondiente. La Figura 116 muestra una primera aproximación a la utilización del patrón *Fachada* en el servidor: como se observa, la *Fachada* queda constituida por la clase *Servidor*, que debe exponer las operaciones indicadas en la figura de forma remota.

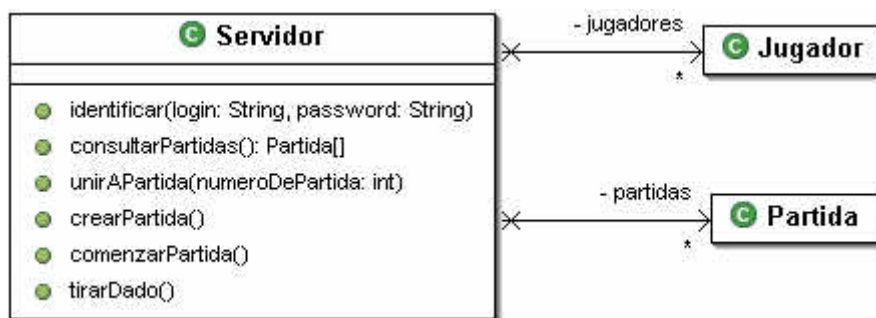


Figura 116. El patrón Fachada, adaptado a nuestro problema

3. Elección del middleware

Existen muchas maneras de comunicar aplicaciones remotas. Una posible forma de comunicación asíncrona consistiría en que una aplicación dejase en algún lugar pactado un fichero con información útil para la otra aplicación, que ésta recogería para, posteriormente, procesarla. En nuestro caso precisamos que ambas aplicaciones se comuniquen de manera inmediata, por lo que debemos considerar otras posibilidades.

3.1.1.1 **Sockets**

Los *sockets* representan los extremos de una línea de comunicación entre dos máquinas, y proporcionan mecanismos para enviar y recibir ristas de bytes. De hecho, cuando enviamos una petición *http* a un servidor web, lo que ocurre realmente es que éste recibe una ristra de bytes con un formato predeterminado.

En nuestro caso, si optáramos por la utilización de *sockets* para comunicar el cliente y el servidor, podríamos determinar una estructura de las ristas de bytes que se intercambiarían las dos aplicaciones. Por ejemplo, se podría establecer que, para identificarse, la aplicación del *Jugador* debe enviar una ristra de bytes con la cadena *identificarse#nombre#contraseña*. El *socket* del servidor, al recibir esa cadena, analizará sus componentes y determinará cuál es la acción de que debe realizar.

Una de las desventajas de utilizar *sockets* puros (es decir, no bibliotecas de clases o *frameworks* que, apoyados en *sockets*, faciliten las tareas de comunicación) es, precisamente, que debe dedicarse un esfuerzo adicional al análisis de cadenas, incluyendo la captura de posibles errores, etc.

3.1.1.2 **RMI**

Una de las alternativas que ofrece Java para la comunicación remota (que, naturalmente, se basa en la utilización de *sockets*) es la “invocación remota de métodos” o RMI (*remote message invocation*).

RMI se basa en que un objeto *A*, para comunicarse con otro *B*, necesita conocer únicamente un subconjunto del conjunto de operaciones públicas de *B*. *B*, entonces, recoge este subconjunto de operaciones en una interfaz, que ofrece de forma remota. De este modo, la interfaz remota será una especialización de la interfaz *java.rmi.Remote*, que no define ningún método y que es, por tanto, una interfaz de nombrado; la clase que implementa la interfaz remota será una especialización de alguna de las clases diseñadas ex profeso para este tipo de comunicaciones, como *java.rmi.server.UnicastRemoteObject*. Aplicando estos conceptos a nuestro ejemplo, *Servidor* deberá ser una especialización de *UnicastRemoteObject* e implementará una interfaz (a la que, por ejemplo, llamaremos *ISolicitud*) en la que se encontrarán las cabeceras de las tres operaciones antes citadas y que, a su vez, especializará a la interfaz *Remote*. Gráficamente:

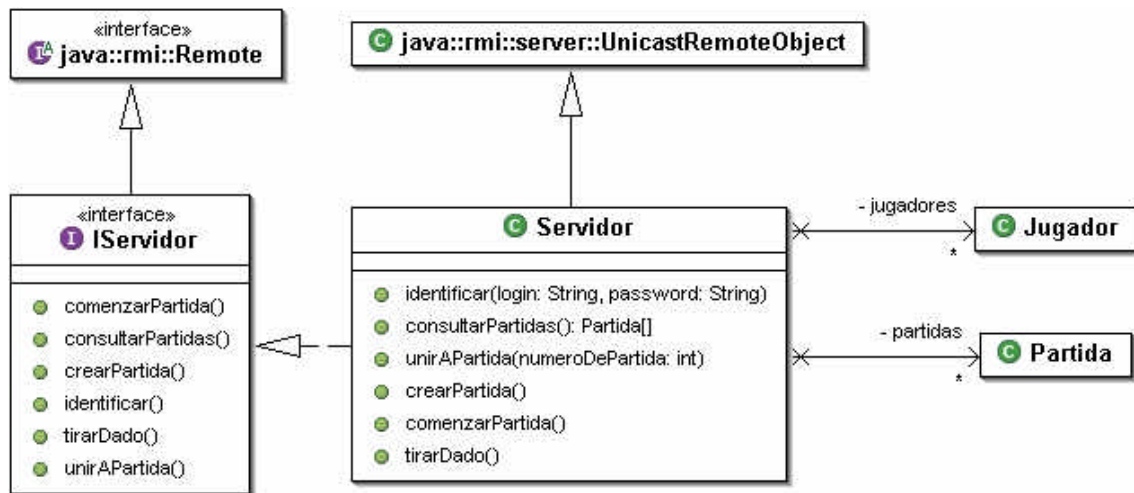


Figura 117. Exposición de los servicios de *Servidor* con *RMI*

Cuando la aplicación del jugador invoque cualquiera de las operaciones ofrecidas por *IServidor*, será realmente una instancia de *Sservidor* quien ejecute las operaciones. Antes de ejecutarlas, la otra aplicación deberá localizar la clase remota, que previamente deberá haberse publicado en algún puerto de alguna máquina utilizando un servidor *RMI*.

Un objeto remoto se publica utilizando el método *bind(String, Remote)* de la clase *java.rmi.Naming*: el primer parámetro es la URL en la que se publica el objeto, al que además se asigna un nombre; el segundo representa la instancia del objeto remoto que se publica. Por ejemplo, si escribimos el siguiente código en la clase *Servidor*, se publica en la dirección local, puerto 2006, el objeto de tipo *Servidor* que se pasa en el segundo parámetro (*this*) con el nombre *servidordemonopoly*:

```
Naming.bind("rmi://127.0.0.1:2006/servidordemonopoly", this);
```

Previamente, habrá sido preciso lanzar el servidor *RMI*:

```
LocateRegistry.createRegistry(2006);
```

La clase que desea acceder de forma remota al objeto publicado debe localizarlo, lo que consigue utilizando el método *Naming.lookup(String)*, que devuelve una referencia a un objeto *Remote* y al que debe hacerse un *cast* para transformarlo al objeto deseado, que debe estar en la forma de una interfaz remota. Es decir:

```
IServidor servidor=(IServidor) Naming.lookup
    ("rmi:// 127.0.0.1:2006/servidordemonopoly");
```

Una vez obtenida la referencia, se pueden realizar las llamadas que se deseen a las operaciones ofrecidas en la interfaz remota, como si se tratara de un objeto local:

```
servidor.identificar("fulano", "contraseñaDeFulano");
```

Todas las operaciones ofrecidas en la interfaz remota deben arrojar, de forma obligatoria, una *RemoteException*.

A pesar de su sencillez de uso, una de las desventajas de *RMI* es que se trata de una tecnología propietaria que funciona únicamente en entornos Java.

3.1.1.3 Servicios web (*web services*)

Los servicios web son una tecnología relativamente reciente en cuyo desarrollo ha participado un amplio grupo de empresas e instituciones que han conseguido un mecanismo de comunicación de aplicaciones robusto y, lo que es más importante, completamente estándar.

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*, 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 es 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 que la estructura de los mensajes que intercambian se encuentra estandarizada.

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*. Si queremos ofrecer las tres operaciones *void validar(long numeroDeSolicitud long)*, *void invalidar(long numeroDeSolicitud)* y *long[] getSolicitudesPendientesDeValidar()* mediante un servicio web y el entorno de desarrollo que utilizamos es relativamente moderno, poseerá un asistente con el que generar un documento *WSDL* más o menos similar al incluido en el Cuadro 71, construido por el entorno Oracle JDeveloper.

El consorcio de empresas e instituciones que ha participado en el desarrollo de los servicios web ha estandarizado, por ejemplo, los tipos de datos básicos que pueden intercambiarse entre clientes y servidores. Para intercambiar datos de tipos complejos (como *Solicitud*, *Investigador*, etc.) es necesario especificar éstos en el *WSDL* en función de tipos simples estándares. El tipo *long*, por ejemplo, es estándar, pero no así el array de *long*, que es el tipo devuelto por el método *getSolicitudesPendientesDeValidar*. Así, en el Cuadro 71 se especifica el *complexType long[]*.

```

<?xml version = '1.0' encoding = 'UTF-8'?>
<!--Generated by the Oracle JDeveloper 10g Web Services WSDL Generator-->
<!--Date Created: Thu Aug 17 18:02:22 CEST 2006-->
<definitions
  name="SolicitudWS"
  targetNamespace="http://investigadores/dominio/Solicitud.wsdl"
  ...
  xmlns:ns1="http://investigadores.dominio/ISolicitudWS.xsd">
  <types>
    <schema
      targetNamespace="http://investigadores.dominio/ISolicitudWS.xsd"
      ...">
      <complexType name="ArrayOflong" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
        <complexContent>
          <restriction base="SOAP-ENC:Array">
            <attribute ref="SOAP-ENC:arrayType" wsdl:arrayType="xsd:long[]"/>
          </restriction>
        </complexContent>
      </complexType>
    </schema>
  </types>
  <message name="validar0Request">
    <part name="numeroDeSolicitud" type="xsd:long"/>
  </message>
  <message name="validar0Response"/>
  <message name="invalidar1Request">
    <part name="numeroDeSolicitud" type="xsd:long"/>
  </message>
  <message name="invalidar1Response"/>
  <message name="getSolicitudesPendientesDeValidar2Request"/>
  <message name="getSolicitudesPendientesDeValidar2Response">
    <part name="return" type="ns1:ArrayOflong"/>
  </message>
  <portType name="SolicitudPortType">
    <operation name="validar">
      <input name="validar0Request" message="tns:validar0Request"/>
      <output name="validar0Response" message="tns:validar0Response"/>
    </operation>
    <operation name="invalidar">
      <input name="invalidar1Request" message="tns:invalidar1Request"/>
      <output name="invalidar1Response" message="tns:invalidar1Response"/>
    </operation>
    <operation name="getSolicitudesPendientesDeValidar">
      <input name="getSolicitudesPendientesDeValidar2Request"
        message="tns:getSolicitudesPendientesDeValidar2Request"/>
      <output name="getSolicitudesPendientesDeValidar2Response"
        message="tns:getSolicitudesPendientesDeValidar2Response"/>
    </operation>
  </portType>
  <binding name="SolicitudBinding" type="tns:SolicitudPortType">
    ...
  </binding>
  <service name="SolicitudWS">
    <documentation>
      @author Macario Polo Usaola, 12-ago-2005
    </documentation>
    <port name="SolicitudPort" binding="tns:SolicitudBinding">
      <soap:address location="http://127.0.0.1:8888/libro/SolicitudWS"/>
    </port>
  </service>
</definitions>

```

Cuadro 71. Fragmento del documento WSDL por el que se publican ciertas operaciones de *Solicitud*

Igual que el usuario de un objeto remoto *RMI* debe localizarlo con el método *lookup* de *Naming*, el usuario de un servicio web debe también buscarlo y ubicarlo para hacer uso de los servicios que ofrece. Los entornos *JDeveloper* y *Microsoft Visual Studio .NET* incluyen asistentes para agregar referencias a servicios web en los proyectos. Estas referencias son realmente clases que brindan a las restantes clases del proyecto el acceso a las operaciones

ofrecidas por el servicio web. Al asistente de JDeveloper 10, por ejemplo, es preciso pasarle la URL del documento WSDL que describe el servicio web; si logra conectarse, muestra la lista de operaciones accesibles, que el usuario elige, y entonces crea un *proxy*¹⁰ que representa el punto de acceso que la aplicación tiene para acceder al servicio web.

En la Figura 118 se representa la estructura de la clase generada por JDeveloper para conectar al servicio web representado por el documento WSDL anterior. Incluye los tres métodos de negocio antes indicados, además de otros varios que se encargan de establecer y gestionar la conexión al servidor remoto. El acceso al servicio web, por tanto, se llevaría a cabo mediante la clase mostrada en la figura.

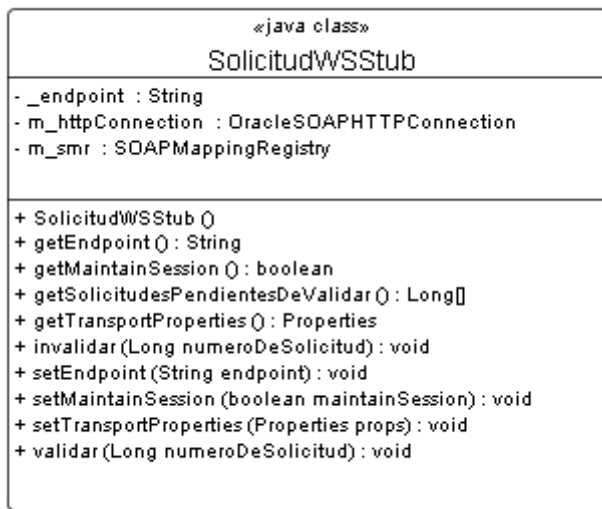


Figura 118. Representación en UML del proxy de acceso al servicio web de Solicit

3.1.1.4 Otros *middlewares*

XXX: Corba...

3.1.1.5 Decisión sobre el *middleware*

Asumiremos que, tras una reunión, el equipo de desarrollo del proyecto ha decidido utilizar RMI como medio de comunicación de las dos aplicaciones.

4. Desarrollo del caso de uso *Identificarse*

La figura siguiente muestra el escenario normal correspondiente a la identificación de un usuario mediante un diagrama de secuencia. Se explicita

¹⁰ Se habla del patrón *Proxy* en la página 100.

el hecho de que el mensaje desde el *Jugador* hacia el servidor consiste en una llamada remota, que posteriormente se traduce en una llamada a un servicio local. No se muestra el paso de mensajes hacia el Agente de base de datos; no obstante, se menciona que éste utiliza la misma base de datos que en la aplicación de administración.

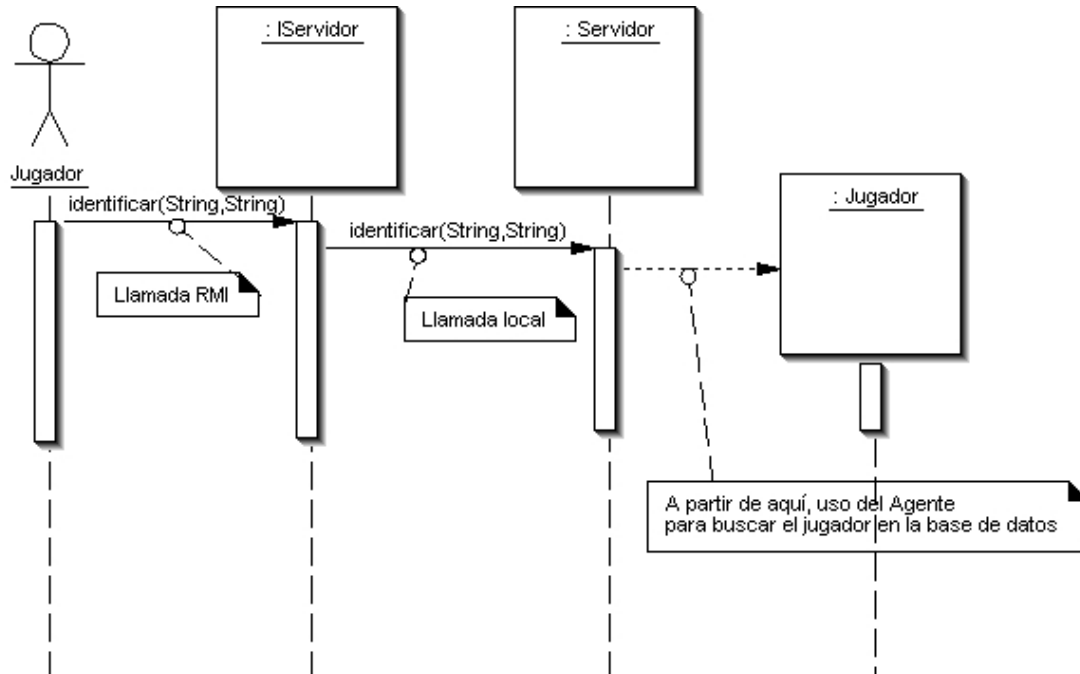


Figura 119. Identificación por RMI

Podemos construir una ventana para que el administrador del sistema vea las diferentes operaciones que van ejecutándose. Cuando el *Servidor* reciba una petición de ejecución de un servicio remoto, mostrará un mensaje en dicha ventana. La ventana, además, servirá para que el administrador arranque el servidor. La Figura 120 muestra parte del diagrama de clases del servidor, en el que la ventana (llamada, igual que en la aplicación para administradores, *JFMain*) implementa una interfaz a la cual conoce la clase *Servidor*. Los mensajes recibidos por *Servidor* serán notificados a *JFMain*, en su forma de *IVentana*.

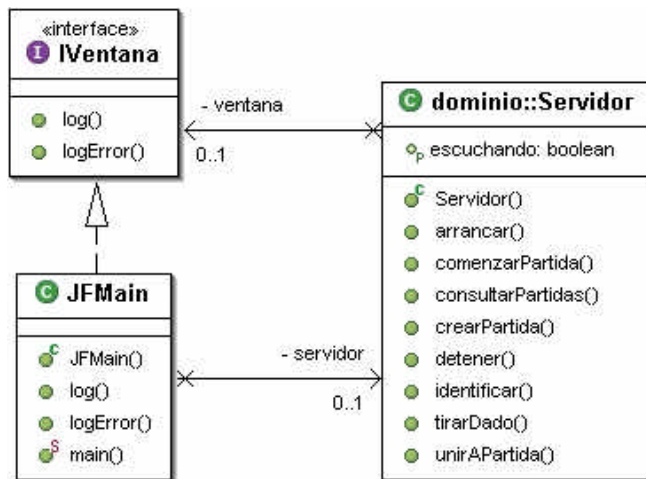


Figura 120. Conexión entre el dominio y la presentación

La figura siguiente muestra el escenario de arranque del servidor, que empieza por una llamada al método *main* de *JFMain*: se observa que, en el cuerpo de esta función, se construye una instancia de *Servidor* a cuyo constructor, la propia ventana se pasa como parámetro en forma de la *IVentana* de la Figura 120; una vez identificado el usuario, la instancia de *Servidor* realiza la notificación correspondiente a la ventana. A la derecha aparece la implementación de los mensajes más destacables.

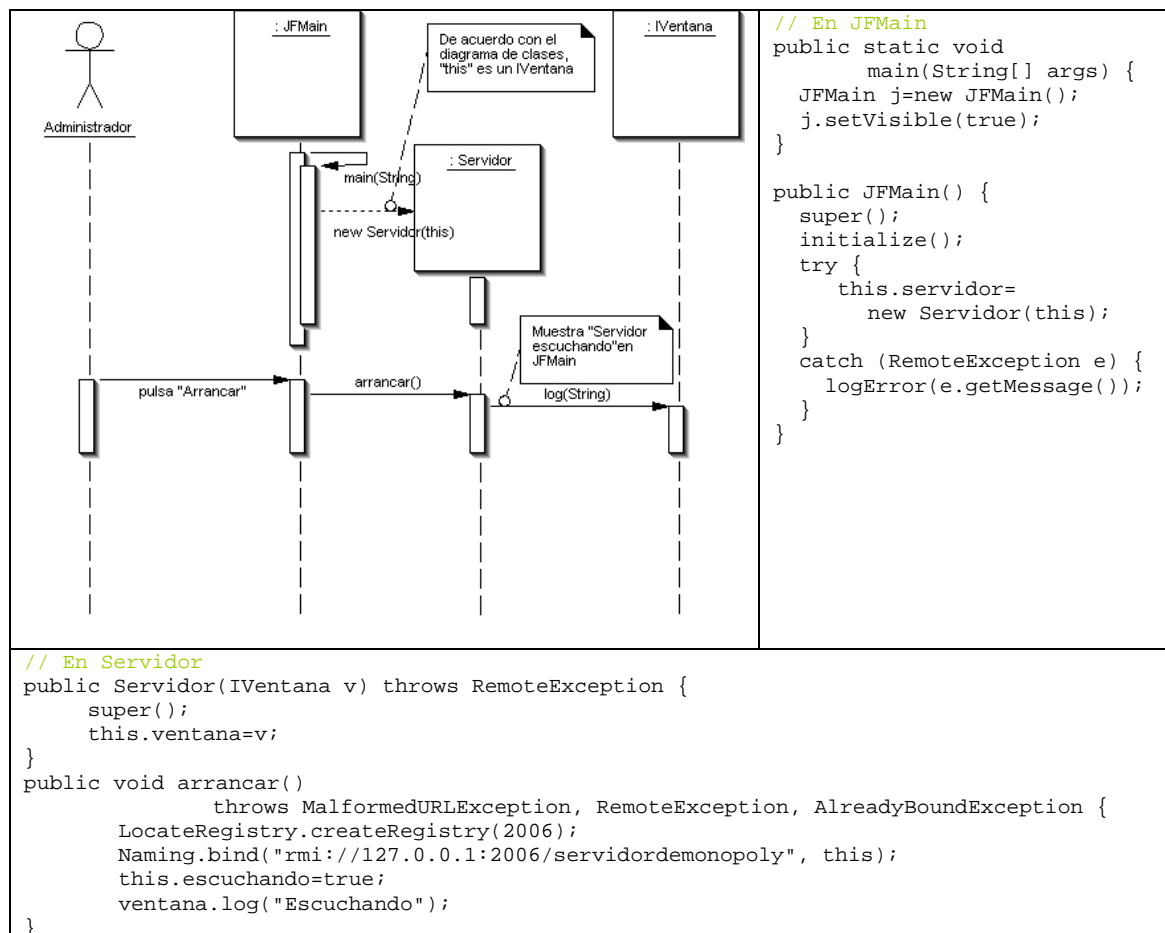


Figura 121. Escenario de arranque del *Servidor* y parte de la implementación

El escenario de identificación que mostrábamos en la Figura 119, con la ampliación que acabamos de mencionar para que haya notificación a la ventana del administrador, puede redibujarse como en la Figura 122. En ésta también se indica que, una vez identificado, la instancia de *Jugador* (que no debe confundirse con el actor homónimo) se añade a la lista de *jugadores* (Figura 116, página 213) a la que conoce el *Servidor*.

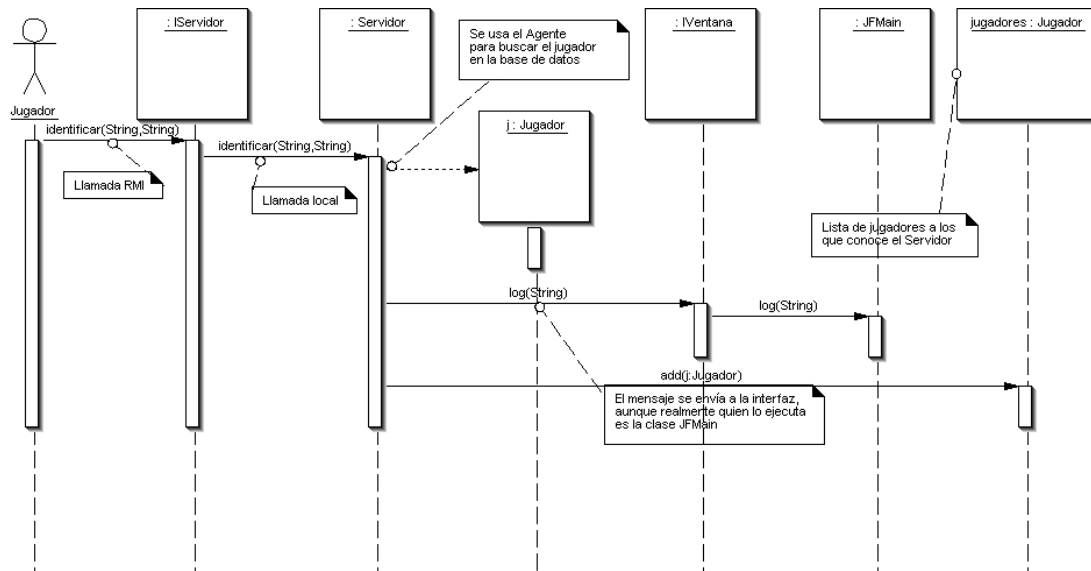


Figura 122. Escenario de identificación con notificación a la ventana

La implementación de la operación *identificar* en *Servidor* puede ser la mostrada en el siguiente cuadro, en donde la comprobación de las credenciales del jugador se comprueba en el constructor de *Jugador*; de este modo, el *Servidor* juega el rol de mero receptor y repartidor de mensajes con pocas responsabilidades más, que es la idea del patrón *Fachada*.

```
public void identificar(String login, String password) throws
    JugadorYaIdentificadoException, SQLException, UsuarioNoRegistrado {
    Jugador j=new Jugador(login, password);
    if (jugadores.contains(j))
        throw new JugadorYaIdentificadoException(login);
    jugadores.add(j);
    this.ventana.log(login + " conectado");
}
```

Cuadro 72. Código de *identificar* en *Servidor*

4.1. Pruebas de *identificar*

La prueba de la operación *identificar* se podría realizar en local o en modo remoto. Puesto que el servicio va a ser ejecutado mediante *RMI*, construiremos una clase *TestServidor* que acceda a la operación de forma remota.

4.1.1 Compilación de la clase *Servidor* con *rmic*

Para que la conexión entre las dos máquinas funcione, el desarrollador del cliente debe disponer de diversos elementos, que deberán ser entregados

por el desarrollador de la aplicación servidora. Éste debe compilar la clase remota (*Servidor*, en este ejemplo) con el compilador *rmic*, y entregar al cliente el resultado de esta compilación junto a la interfaz remota, también compilada. Los archivos resultantes pueden ser empaquetados en un archivo *jar* para su distribución.

rmic se ejecuta desde la línea de comando (Figura 123), toma como parámetro la clase que expone los servicios remotos y produce dos archivos, el *skeleton* y el *stub*, que en nuestro caso se llaman *Servidor_Skel.class* y *Servidor_Stub.class*. Es importante notar que todas las operaciones incluidas en la interfaz remota implementada por la clase que se desea compilar deben arrojar la excepción *java.rmi.RemoteException*.

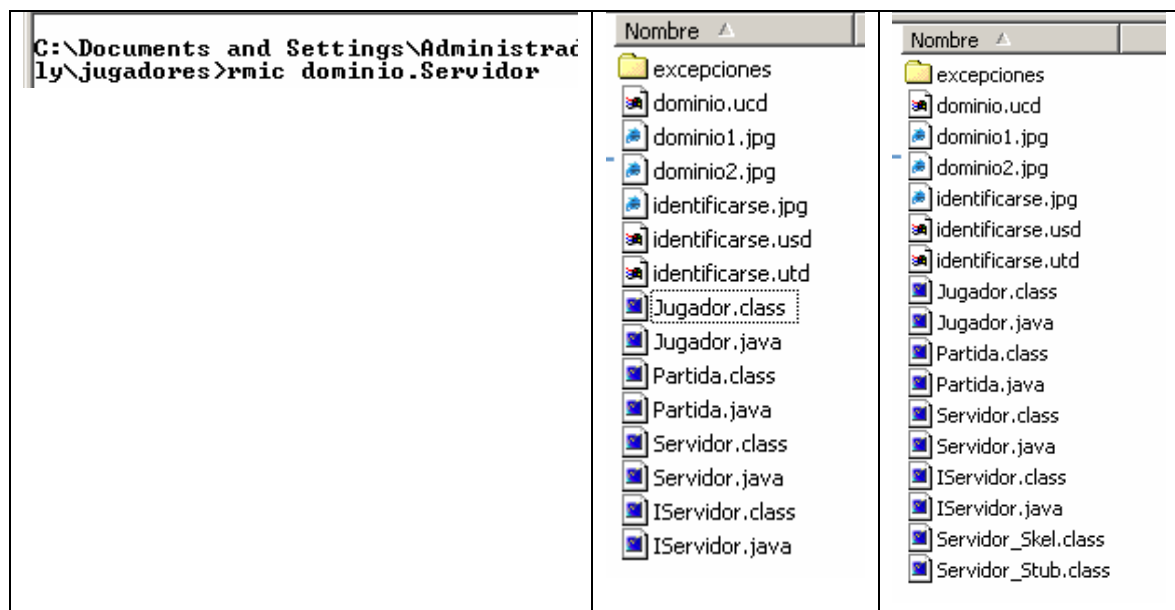


Figura 123. Comando de compilación con *rmic* (izquierda), y contenidos de la carpeta *dominio* antes (centro) y después (derecha) de la compilación

4.1.2 Creación y ejecución de la clase con los casos de prueba

Construiremos como normalmente la clase con el código de los casos de prueba. No obstante, y puesto que accederemos al servicio de forma remota, el código de los casos de prueba no incluirá, como normalmente, una llamada al constructor de *Servidor* para crear la instancia de la clase bajo prueba: en su lugar, habrá una instrucción que nos devuelva una referencia remota al *Servidor*, que estará escuchando en algún lugar del ciberespacio.

El Cuadro 73 muestra tres casos de prueba incluidos en la clase *IServidorTest*: el primero corresponde a la identificación correcta de un usuario, por lo que colocamos un *fail* para que la barra roja aparezca ante cualquier excepción que se lance; el segundo se ejecuta después del primero e intenta la identificación de *fulano*, el mismo usuario, sobre una referencia al mismo *servidor*: por tanto, se espera que lance una excepción porque el jugador ya

está identificado; el tercero comprueba que, si se intenta registrar un usuario que no existe, se lanza la excepción *UsuarioNoRegistrado*.

```
public void testIdentificar() {
    try {
        IServidor servidor=(IServidor)
            Naming.lookup("rmi://127.0.0.1:2006/servidordemonopoly");
        servidor.identificar("fulano", "contraseña de fulano");
    } catch (Exception e) { fail("No se esperaba excepción: " + e.toString()); }
}

public void testIdentificar2() {
    try {
        IServidor servidor=(IServidor)
            Naming.lookup("rmi://127.0.0.1:2006/servidordemonopoly");
        servidor.identificar("fulano", "contraseña de fulano");
        fail("Se esperaba una JugadorYaIdentificadoException");
    } catch (JugadorYaIdentificadoException e) {
    } catch (Exception e) { fail("Se esperaba una JugadorYaIdentificadoException"); }
}

public void testIdentificar3() {
    try {
        IServidor servidor=(IServidor)
            Naming.lookup("rmi://127.0.0.1:2006/servidordemonopoly");
        servidor.identificar("mengano", "contraseña de fulano");
        fail("Se esperaba una UsuarioNoRegistrado");
    } catch (UsuarioNoRegistrado e) {
    } catch (Exception e) { fail("Se esperaba una UsuarioNoRegistrado"); }
}
}
```

Cuadro 73. Código de tres casos de prueba con acceso remoto

Obviamente, para que los casos de prueba sean superados, el servidor tiene que haber sido arrancado previamente.

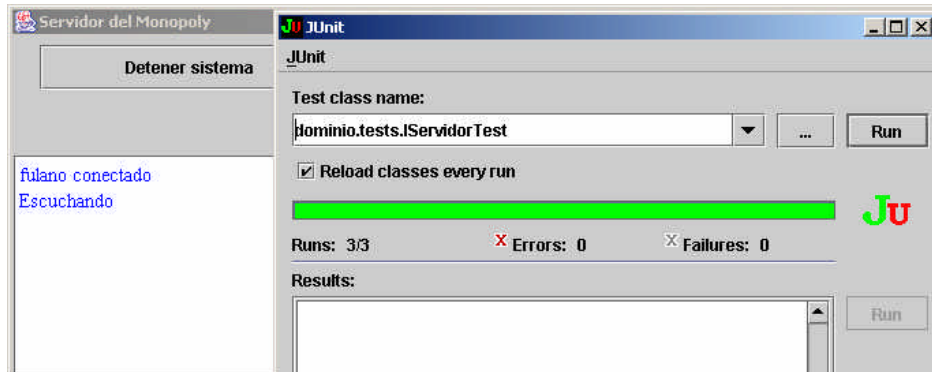


Figura 124. Al fondo, la instancia de *JFMain*; en primer plano, JUnit

5. Desarrollo del caso de uso *Crear partida*

Podemos imponer la restricción de que un jugador no juegue más de una partida simultáneamente. Así, un jugador no podrá ejecutar esta operación si ya se encuentra en la lista de jugadores de alguna partida. Esto supone la creación de una relación de 1 a muchos entre *Partida* y *Jugador*; además, y puesto que el creador de una partida es el jugador que más tarde podrá decidir comenzarla, crearemos otra relación de 1 a 1 para que la partida mantenga una referencia a su creador (Figura 125).

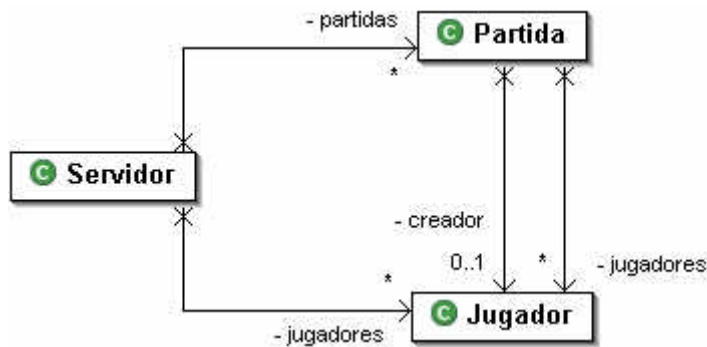


Figura 125. Adición de relaciones entre *Partida* y *Jugador*

Para que el *Servidor* sea capaz de comprobar si el jugador ya está participando en alguna *Partida* el *Servidor* debe, de alguna manera, conocer el login del usuario que intenta ejecutar la operación *crearPartida*. Una forma de hacerlo es añadir a la operación *crearPartida* el parámetro *login*, y utilizar éste para buscarlo en la lista de jugadores. No obstante, si tenemos en cuenta que el *Servidor* debe ser capaz de comunicarse con los jugadores, podemos aprovechar desde este momento para implementar un mecanismo alternativo de comunicación, para que el *Servidor* les pueda enviar mensajes a los clientes, también por *RMI*.

Así, construiremos un cliente en un proyecto separado que nos permita enviar mensajes al *Servidor* por *RMI*, pero también recibirlos. El *Cliente*, entonces, será también un servidor *RMI* que recibirá mensajes del *Servidor* a través de la interfaz remota que ofrezca. La siguiente figura muestra el diseño de este cliente: la clase de dominio *Cliente* es un objeto remoto (especialización de *UnicastRemoteObject* e implementación de la interfaz remota *ICliente*) que, a la vez, posee una referencia a la interfaz remota *IServidor*.

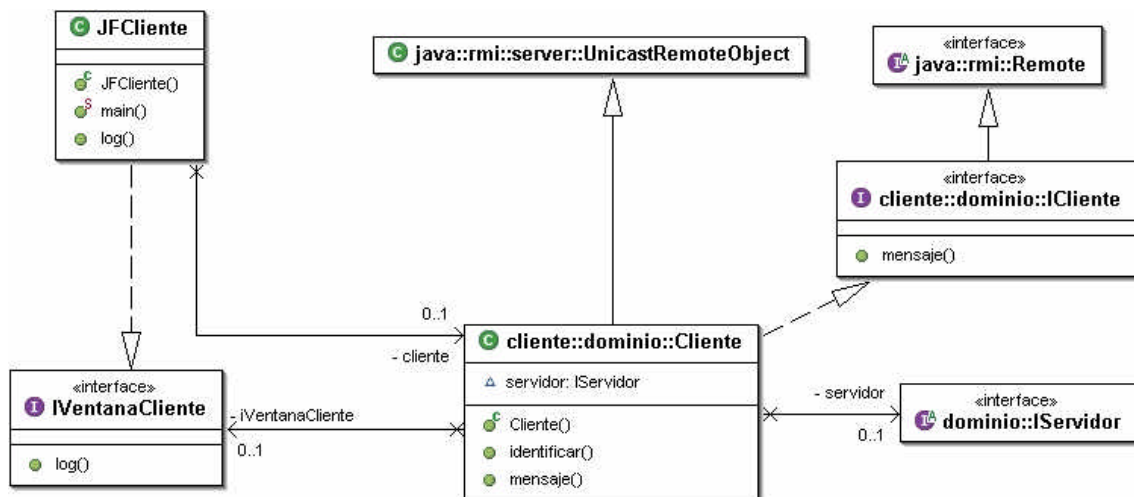


Figura 126. Diseño del cliente de prueba, basado también en RMI

En la figura anterior, el usuario escribe sus credenciales en *JFCliente* y pulsa un botón *Conectar*, lo que produce una llamada a la operación *identifi-*

car(login:String, pwd:String) de *Cliente*. Esta operación se encuentra en esta clase implementada como en el Cuadro 74: la operación “busca al servidor por el ciberespacio” y, una vez localizado, ejecuta sobre él la operación *identificar*, a la que ahora se le ha añadido un tercer parámetro, *this*, que es el cliente remoto.

```
public void identificar(String login, String pwd) throws Exception {
    servidor=(IServidor) Naming.lookup("rmi://127.0.0.1:2006/servidordemonopoly");
    servidor.identificar(login, pwd, this);
}
```

Cuadro 74. Código de *identificar* en *Cliente*

La solución para que el *Servidor* pueda conocer a los diferentes clientes puede pasar por añadir un campo de tipo *ICliente* a la clase *Jugador*. Servidor utilizará *Jugador* como proxy para comunicar mensajes al cliente asociado. Estos cambios suponen una modificación al diagrama de la Figura 125 y se representan en la Figura 127. Obsérvese que a la operación *identificar* se le ha añadido el parámetro de tipo *ICliente*.

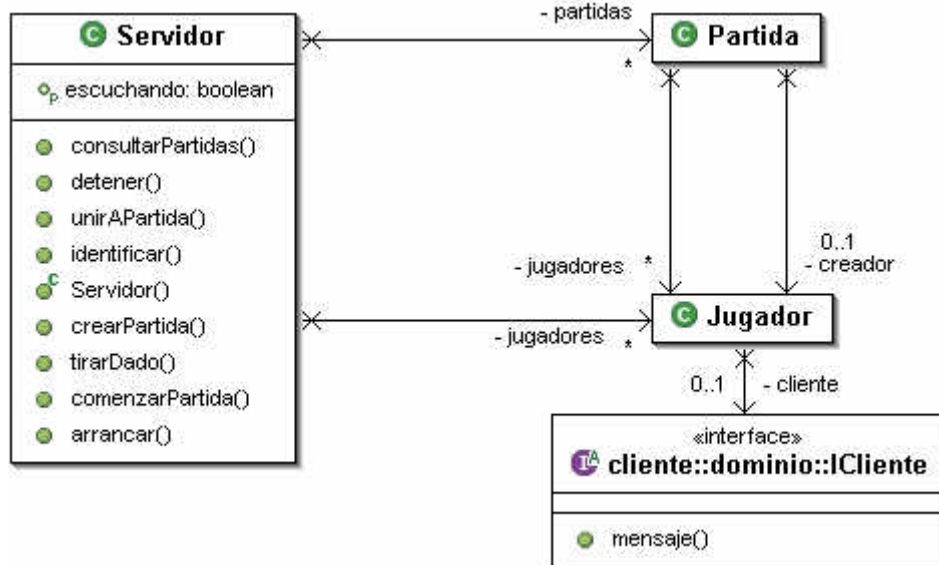


Figura 127. En el servidor, la clase *Servidor* conoce ahora a muchos clientes remotos

El nuevo código de la operación *identificar* en *Servidor* es el siguiente:

```
public void identificar(String login, String password, ICliente cliente) throws
    JugadorYaIdentificadoException, SQLException, UsuarioNoRegistrado, RemoteException {
    Jugador j=new Jugador(login, password);
    if (jugadores.get(login)!=null)
        throw new JugadorYaIdentificadoException(login);
    j.setCliente(cliente);
    jugadores.put(login, j);
    this.ventana.log(login + " conectado");
    cliente.mensaje("Bienvenido al servidor del Monopoly");
}
```

Cuadro 75. Código de *identificar* en *Servidor*

Cuando un cliente identificado desea crear una partida, el *Servidor* comprobará que no esté ya en ninguna de las que hay en ejecución; si no lo

está, creará una instancia de clase *Partida* a la que añadirá el *Jugador* correspondiente. El código de la operación *crearPartida* podemos ir escribiéndolo sobre la marcha, como en la siguiente figura, en la que se resaltan muchos errores debidos a elementos que aún no se encuentran implementados.

```
public void crearPartida(ICliente cliente) {
    Jugador j=(Jugador) this.jugadores.get(cliente.getLogin());
    if (j==null)
        throw new UsuarioNoRegistrado(cliente.getLogin());
    if (estaJugando(j))
        throw new YaEstaJugando(cliente.getLogin());
    Partida p=Partida.nuevaPartida();
    p.setCreador(j);
}
```

Figura 128. Código que vamos escribiendo en *crearPartida* (en *Servidor*)

No obstante, también podemos escribir una clase *JUnit* para probar el servidor. La siguiente figura muestra los tres proyectos que tenemos ahora en el workspace de Eclipse: *administración* es el proyecto que se ha desarrollado en capítulos anteriores; *jugadores* es el proyecto que venimos desarrollando en este; *clienteJugador* es el cliente de pruebas que hemos construido y cuyo diseño se ofrecía en la Figura 126. El código mostrado es el de la clase *TestCliente*, que prueba, desde *clienteJugador*, la funcionalidad del *Servidor* a través de *Cliente*. Obsérvese que la *TestCliente* implementa la interfaz *IVentanaCliente*.

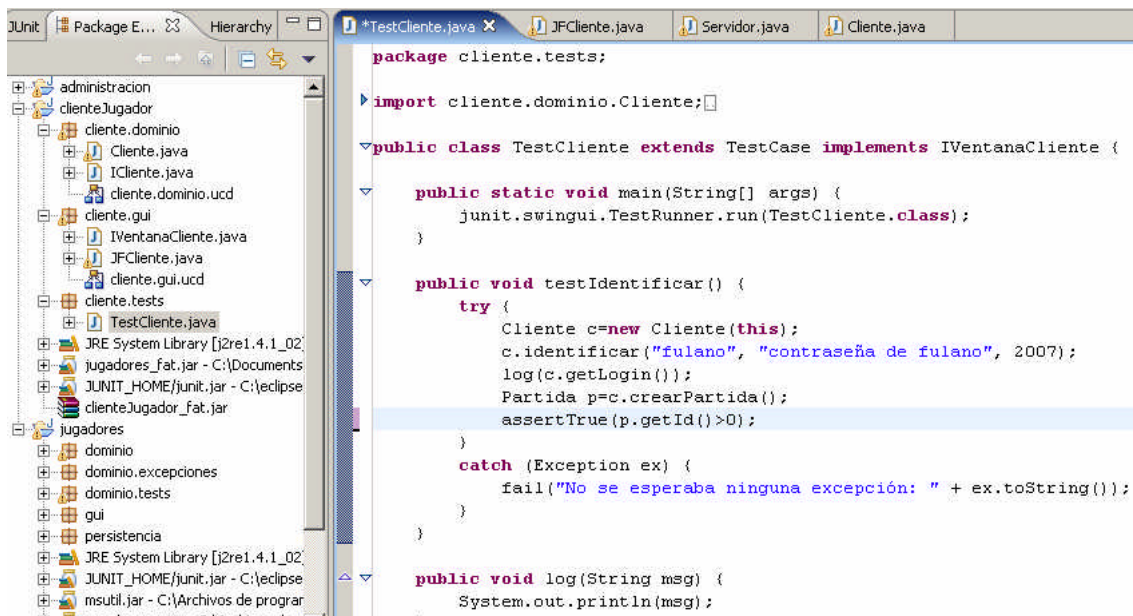


Figura 129. El workspace consta de tres proyectos

5.1. Pruebas del código con varios clientes

Puesta que se trata de un sistema distribuido cliente-servidor, nos interesa probar el funcionamiento del sistema con varios clientes conectados

simultáneamente. Si estamos desarrollando y probando en una misma máquina, el servidor y los posibles clientes compartirán la misma dirección IP, podrán escuchar en el mismo puerto, peor deberán registrarse con nombres distintos.

Así, el servidor puede utilizar, por ejemplo, el puerto 2006, y los clientes los puertos 2007, 2008, etc. Para la implementación del cliente de prueba, entonces, podemos añadir un parámetro que represente el puerto en el que el cliente se pone a la escucha. La siguiente muestra el aspecto de la ventana del cliente de prueba, en donde se introduce el login del jugador, su contraseña y el puerto en el que se ubicará el cliente.



Figura 130. Aspecto de la ventana del cliente de prueba

En el siguiente cuadro se muestra, por un lado, el código que se ejecuta al pulsar el botón *Conectar* en la ventana de la figura anterior, que lanza una llamada al método *identificar* de la clase *Cliente*: éste registra al cliente como servidor *rmi* en el puerto pasado como parámetro y con el nombre correspondiente al *login* del jugador (instrucción *Naming.bind*); a continuación, asigna al campo *servidor* una referencia remota al servidor del Monopoly, que escucha de manera fija en el puerto 2006 con el nombre *servidordemonopoly*, y usa esta referencia remota para identificarse, pasándose a sí mismo como parámetro.

<pre>protected void conectar() { try { int puerto; if (jrbPuerto2007.isSelected()) puerto=2007; else puerto=2008; cliente.identificar(this.jtfLogin.getText(), this.jtfPwd.getText(), puerto); } catch (Exception e) { log(e.toString()); } }</pre>	<pre>public void identificar(String login, String pwd, int puerto) throws Exception { this.puerto=puerto; LocateRegistry.createRegistry(puerto); Naming.bind("rmi://127.0.0.1:" + puerto + "/" + login, this); this.login=login; servidor=(IServidor) Naming.lookup("rmi://127.0.0.1:2006/servidordemonopoly"); servidor.identificar(login, pwd, this); }</pre>
--	--

Cuadro 76. Código correspondiente al botón *Conectar* (izquierda) y al método *identificar* en *Cliente* (derecha)

La siguiente figura muestra un escenario de ejecución, en el que dos clientes (con logins *fulano* y *fulano2*) se conectan al servidor satisfactoriamente, cada uno en distinto puerto:

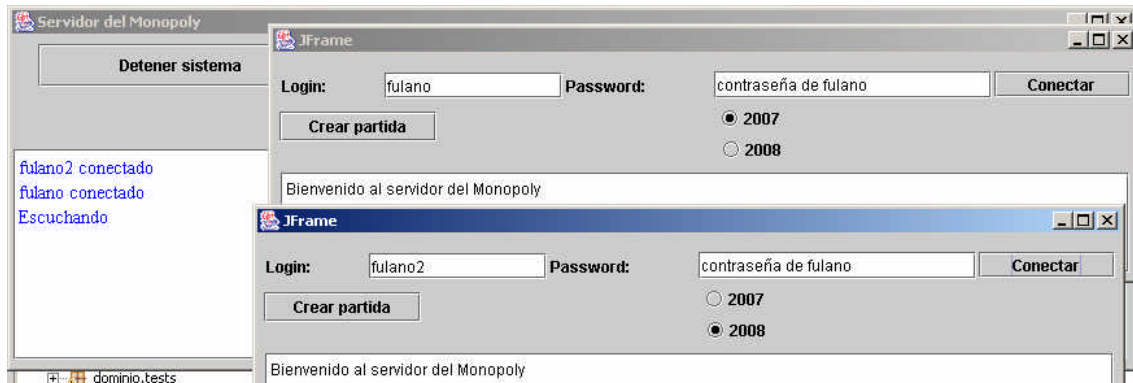


Figura 131. Escenario de conexión de dos clientes (al fondo, la ventana del servidor)

En la ventana de los clientes hemos colocado un botón para crear partidas; de acuerdo con el código del método de prueba de la Figura 129, esperamos que el servidor nos devuelva un objeto de clase *Partida*. Para que se puedan enviar objetos mediante RMI, éstos deben ser *serializables*. Un objeto *serializable* es un objeto que se puede traducir a una ristra bytes para, por ejemplo, ser almacenado en disco y posteriormente recuperado en el mismo estado o, como es este caso, para ser transmitido por la red.

Así, *Partida* debe ser serializable, lo que se consigue haciendo que implemente la interfaz *java.io.Serializable*, que es una interfaz de nombrado, sin ninguna operación. Como *Partida* posee referencias a objetos de clase *Jugador*, esta clase debe implementar también la misma interfaz.

<pre>package dominio; import java.io.Serializable; import java.util.Hashtable; public class Partida implements Serializable { ... }</pre>	<pre>package dominio; import java.io.Serializable; import ...; public class Jugador implements Serializable { ... }</pre>
---	---

Cuadro 77. Los objetos que se envían por RMI deben ser serializables

6. Desarrollo del caso de uso *Consultar partidas abiertas*

El desarrollo de este caso de uso será muy similar al anterior: podemos ir escribiendo en el cliente el código necesario para recuperar la lista de partidas abiertas en un array, e ir escribiendo la implementación de las operaciones a medida que las vayamos necesitando.

El siguiente cuadro muestra el código de la operación *consultarPartidasAbiertas*, que se ejecuta cuando se pulsa un botón para tal efecto ubicado

en la ventana *JFCliente*. Se observa que la ventana pasa el mensaje al objeto de clase *Cliente*, que actúa de proxy para comunicar con el *Servidor*.

```
protected void consultarPartidasAbiertas() {
    try {
        Partida[] pp=this.cliente.consultarPartidasAbiertas();
        if (pp==null) return;
        for (int i=0; i<pp.length; i++) {
            Partida p=pp[i];
            String texto="Partida nº " + p.getId() + ", creada por " +
                p.getCreador().getNombre();
            log(texto);
        }
    } catch (RemoteException e) {
        this.log(e.toString());
    }
}
```

Cuadro 78. Implementación de *consultarPartidasAbiertas* en la ventana del cliente

El método *consultarPartidasAbiertas* en *Cliente* pasa la llamada al *Servidor*, al cual conoce:

```
public Partida[] consultarPartidasAbiertas() throws RemoteException {
    return this.servidor.consultarPartidas();
}
```

Cuadro 79. Implementación de la operación en *Cliente*

Las operaciones de los dos cuadros anteriores pertenecen a clases del proyecto *clienteJugador* (recuérdese de la Figura 129 que tenemos tres proyectos); en el proyecto *Jugadores*, en donde está incluido el *Servidor*, implementamos la operación, que se ofrece de forma remota, de la siguiente manera:

```
public Partida[] consultarPartidas() {
    Vector partidasAbiertas=new Vector();
    Enumeration e=partidas.elements();
    while (e.hasMoreElements()) {
        Partida p=(Partida) e.nextElement();
        if (!p.estaCerrada()) {
            partidasAbiertas.add(p);
        }
    }
    Partida[] result=new Partida[partidasAbiertas.size()];
    for (int i=0; i<partidasAbiertas.size(); i++)
        result[i]=(Partida) partidasAbiertas.get(i);
    return result;
}
```

Cuadro 80. Implementación de *consultarPartidas* (que devuelve la lista de partidas abiertas) en la clase *Servidor*

7. Desarrollo del caso de uso *Unirse a partida*

Este caso de uso lo podemos desarrollar exactamente de la misma manera que en el caso anterior. Cuando trabajamos con *RMI* y alteramos alguna de las clases remotas es necesario recompilarlas con *rmic* para ofrecer a las aplicaciones usuarias sus versiones actualizadas.

En este caso, hemos añadido un parámetro de tipo *ICliente* a la operación *unirAPartida* de la clase *Servidor*, lo que supone actualizar la propia

clase, la interfaz *IServidor* (que utiliza el cliente para comunicarse), y los archivos *skel* y *stub* que se generan al compilar con *rmic*.

8. El patrón *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 una operación abstracta tipo *procesar()*, 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 que hemos utilizado como ejemplo en alguna ocasión, deseamos dotar a los empleados de un sencillo lenguaje de comandos para realizar algunas operaciones. Supongamos también que los comandos del lenguaje serán los siguientes:

- ingresar cuenta importe 'descripción';
- retirar cuenta importe 'descripción';
- transferir cuentaOrigen importe cuentaDestino 'descripción';
- movimientos cuenta;
- saldo cuenta;

Mediante el patrón *Intérprete* construiremos la estructura de clases de la Figura 132: la clase *Parser* analiza el texto que recibe como parámetro en su constructor y, en función del valor del primer token, determina la subclase a la que debe instanciar el campo *expresion*; entonces, y si no se detectan errores sintácticos, ejecuta la operación *procesar*, que tendrá la implementación adecuada.

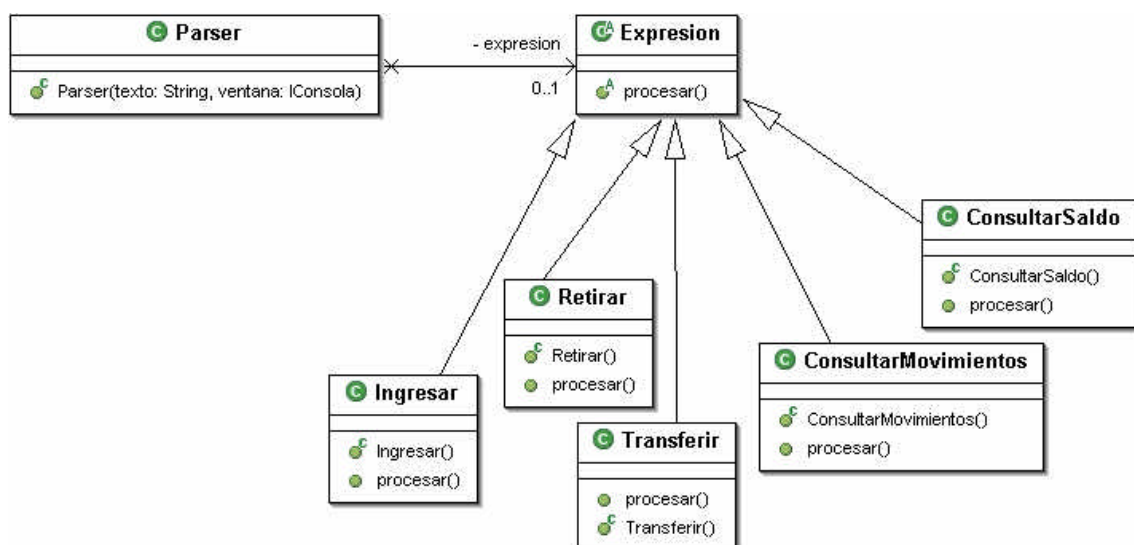


Figura 132. Ejemplo del patrón Intérprete

El siguiente cuadro incluye parte de la implementación del constructor de *Parser*: mediante la clase *java.util.StringTokenizer* partimos el *texto* en tokens delimitados por espacios en blanco.

```
public Parser(String texto, IConsola ventana) throws Exception {
    this.consola=ventana;
    StringTokenizer st=new StringTokenizer(texto, " ");
    if (!st.hasMoreTokens())
        throw new SintaxisIncorrectaException("Comando no reconocido");
    String comando=st.nextToken().toLowerCase();
    if (comando.equals("ingresar")) {
        String cuentaDestino=st.nextToken();
        String sImporte=st.nextToken();
        String descripcion=getDescripcion("ingresar", st);
        double importe=Double.parseDouble(sImporte);
        expresion=new Ingresar(cuentaDestino, importe, descripcion);
    } else if (comando.equals("retirar")) {
        ...
    } else throw new
        SintaxisIncorrectaException("Comando no reconocido: " + texto);
    expresion.procesar(this.consola);
}
```

Cuadro 81. Constructor de *Parser*

Si el primer token (variable *comando*) es “ingresar”, instanciamos *expresion* a la subclase *Expresion*, pasándole los parámetros necesarios para construir el objeto, que proceden de los restantes tokens del comando (Cuadro 82).

```
public Ingresar(String cuentaDestino, double importe, String descripcion)
    throws CuentaInexistenteException, SQLException {
    this.cuentaDestino=new Cuenta(cuentaDestino);
    this.importe=importe;
    this.descripcion=descripcion;
}
```

Cuadro 82. Constructor del subtipo *Ingresar*, especialización de *Expresion*

Finalmente, se ejecuta la operación *procesar*, cuyo código puede tener la siguiente forma:

```
public void procesar(IConsola consola) throws
    IllegalArgumentException, SQLException, CuentaInexistenteException {
    this.cuentaDestino.ingresar(importe, descripcion);
}
```

Cuadro 83. Implementación de *procesar* en la clase *Ingresar***9. Aplicación del patrón *Intérprete* al cliente de pruebas**

Podemos definir un pequeño lenguaje de comandos para que el cliente pueda conectarse, crear partidas, consultar partidas abiertas y unirse a partidas. La gramática del lenguaje puede ser la siguiente:

```
expresión : conectar | crear | consultar | unir
conectar : 'conectar' nombre '<' contraseña '>' puerto ';'
crear    : 'crear' ';'
consultar : 'consultar' ';'
unir     : 'unir' idPartida ';'

```

Aplicando el patrón, construimos una clase abstracta *Expresion* para el símbolo principal de la gramática, con tantas especializaciones como reglas:

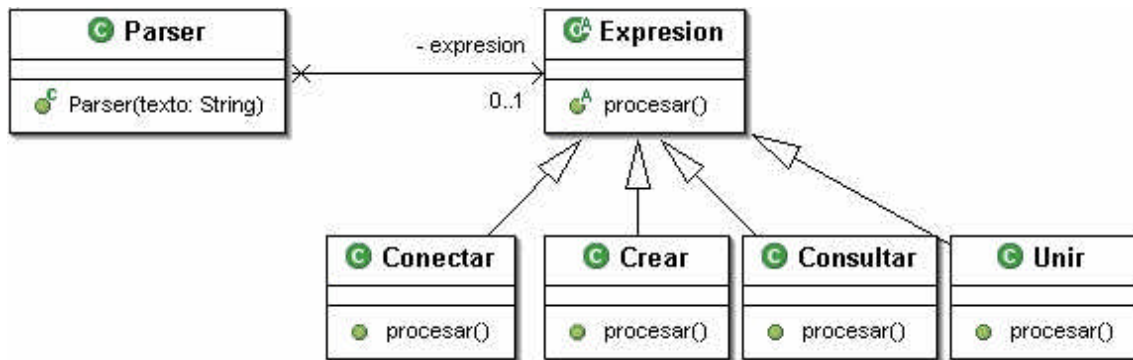


Figura 133. Estructura inicial de clases para el subsistema *cliente.parser*

Centrándonos, por ejemplo, en la implementación de la operación *procesar* en *Conectar*, ésta debe decirle al objeto de clase *Cliente* al que conoce la ventana que conecte. Lo mismo ocurre con el resto de versiones de la operación en los diferentes subtipos. *Expresion*, por tanto, puede disponer de un campo protegido de tipo *Cliente*, que será aquel sobre el que actúen sus especializaciones:

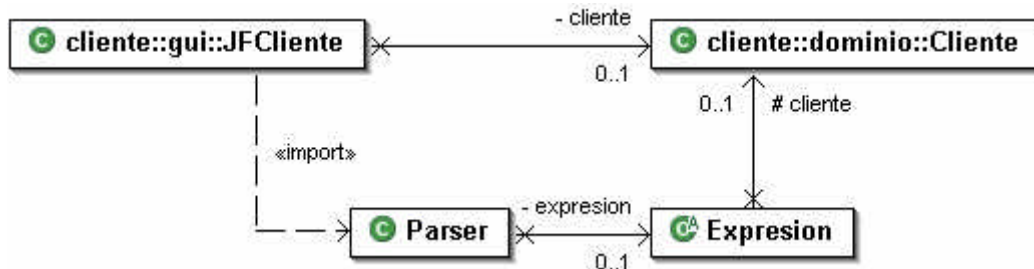


Figura 134. *Expresion* conoce a *Cliente*

El paso de mensajes que tiene lugar para procesar el comando de conexión se ilustra en la Figura 135.

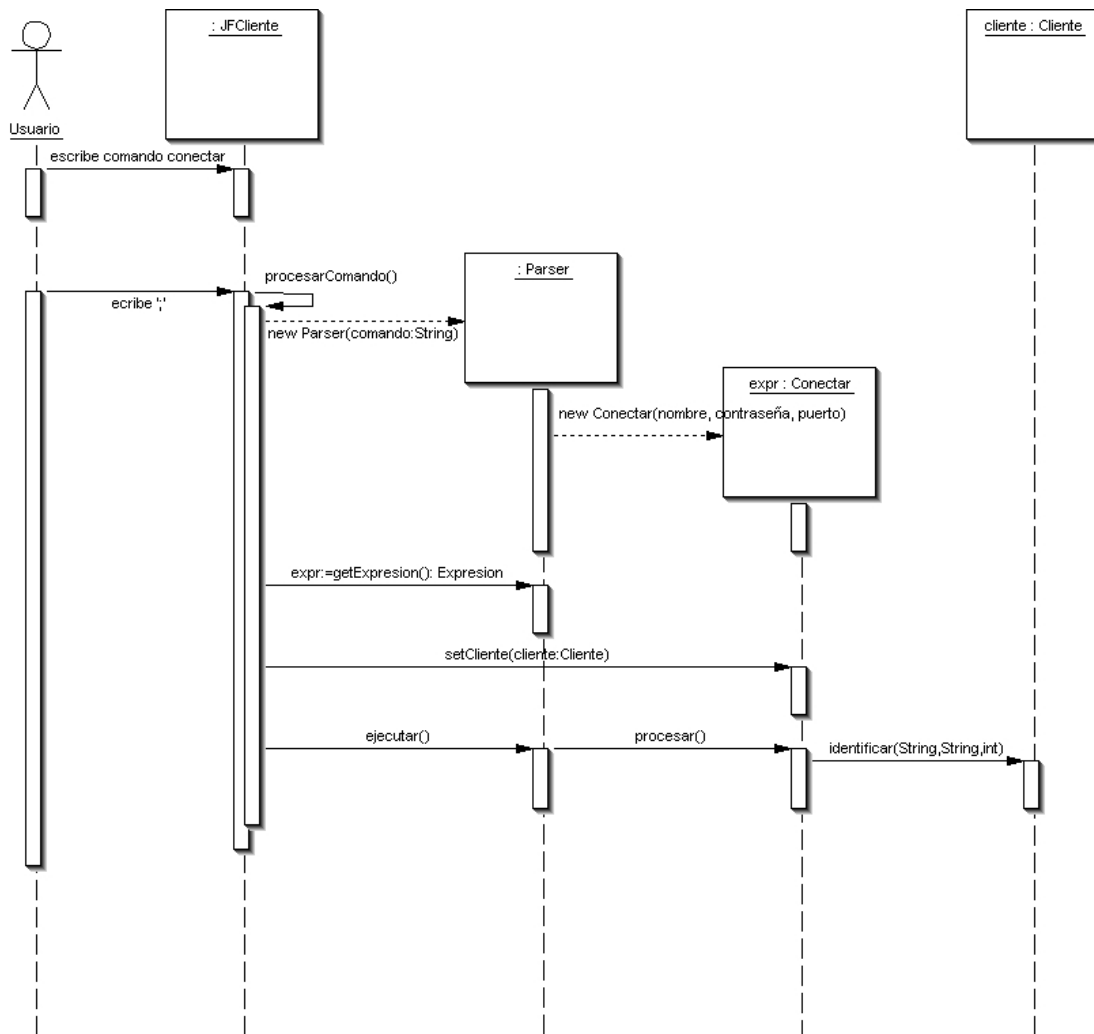


Figura 135. Paso de mensajes para procesar el comando de conexión al *Servidor*

10. Desarrollo de los casos de uso *Comenzar partida* y *Tirar dados*

Para que este caso de uso pueda ejecutarse es preciso que haya, al menos, dos jugadores en la partida. El jugador que la creó elegirá la opción de comenzar la partida. Entonces, cada jugador tirará los dados y comenzará aquél que saque más puntuación. Habrá desempate entre los jugadores que saquen la misma puntuación.

Añadiremos un botón a la ventana del cliente de prueba para que se pueda comenzar una partida., que desencadenará en el cliente las acciones necesarias para indicar al servidor que se desea comenzar la partida. El código de esta operación en el *Servidor* se muestra en el Cuadro 84: obsérvese que la cabecera del método incluye excepciones que también deben encontrarse declaradas en la descripción de la operación que se ofrece al cliente en la interfaz *IServidor*; estas excepciones, además, tendrán que ser capturadas adecuadamente por el *Cliente*, por lo que tienen que incluirse en el fichero

jar que se le entrega. Además, como ya se ha dicho anteriormente, hay que recompilar con *rmic* las clases que ofrecen los servicios remotos.

```
public void comenzarPartida(ICliente cliente) throws
    RemoteException, NoEsCreador, NumeroInsuficienteDeJugadores, PartidaCerrada {
    String login=cliente.getLogin();
    Partida p=getPartidaCreadaPor(login);
    if (p==null)
        throw new NoEsCreador(login);
    if (p.getNumeroDeJugadores()<2)
        throw new NumeroInsuficienteDeJugadores();
    if (p.estaCerrada())
        throw new PartidaCerrada();
    p.setCerrada(true);
    ventana.log("La partida " + p.getId() + " comienza");
    p.comenzar();
}
```

Cuadro 84. Código de *unirAPartida* (en *Servidor*)

La operación *comenzar*, llamada en último lugar, de momento simplemente notifica a los jugadores que la partida comienza.

```
public void comenzar() throws RemoteException {
    notifica("La partida " + this.id + " comienza");
}

private void notifica(String mensaje) throws RemoteException {
    Enumeration jj=getJugadores();
    while (jj.hasMoreElements()) {
        Jugador j=(Jugador) jj.nextElement();
        ICliente c=j.getCliente();
        c.mensaje(mensaje);
    }
}
```

Cuadro 85. Código provisional de *comenzar* y de *notifica* (en *Servidor*)

La operación *notifica* mostrada en el cuadro anterior funciona perfectamente; ahora bien, como hemos dicho que *Jugador* actúa de proxy entre el *Servidor* y los clientes, se adapta más a esta filosofía el hecho de que *Servidor* no acceda al objeto de tipo *ICliente* de cada *Jugador*, sino que envíe el mensaje a través del *Jugador*:

```
private void notifica(String mensaje) throws RemoteException {
    Enumeration jj=getJugadores();
    while (jj.hasMoreElements()) {
        Jugador j=(Jugador) jj.nextElement();
        j.enviaMensaje(mensaje);
    }
}
```

Cuadro 86. Nuevo código de *notifica* en *Servidor*, que envía el mensaje al cliente a través del *Jugador*

El método del cuadro anterior supone la creación de una nueva operación (*enviaMensaje*) en *Jugador*:

```
public void enviaMensaje(String texto) throws RemoteException {
    this.cliente.mensaje(texto);
}
```

Cuadro 87. Nueva operación añadida a *Jugador*, utilizada en el Cuadro 86

A partir del momento en que la partida comienza se hace necesario dotar a ésta de un mecanismo de gestión del turno, para controlar el orden en el que los jugadores realizan las operaciones. El receptor de los mensajes

que los clientes envían para interactuar con la partida que están jugando continuará siendo el *Servidor*, que actúa como una fachada para los clientes, pero pasará el control a la *Partida* para que ésta bregue con ellos como considere. Así, podemos añadir una línea de código al método *comenzar* para que asigne a un nuevo campo *jugadorConElTurno* el valor del primer jugador que tira los dados, que puede ser el creador de la partida:

```
public void comenzar() throws RemoteException {
    notifica("La partida " + this.id + " comienza");
    this.setTurno(this.creador);
}
```

Cuadro 88. Se añade una instrucción de asignación de turno al jugador

Debemos dar una implementación a *setTurno(Jugador)* en *Servidor*, como la mostrada en el lado izquierdo del Cuadro 89. Lo que esta operación puede realizar es transmitir a todos los jugadores de la partida el nombre del jugador que posee el turno; el cliente, al recibir este mensaje, podrá activar o desactivar botones para realizar operaciones en función de si es él el que tiene el turno o no.

```
private void setTurno(Jugador jugador) {
    this.jugadorConElTurno=jugador;
    Enumeration jj=getJugadores();
    while (jj.hasMoreElements()) {
        Jugador j=(Jugador) jj.nextElement();
        j.setTurno(jugadorConElTurno.getNombre());
    }
}
```

```
public void setTurno(String login) {
    this.cliente.setTurno(login);
}
```

Cuadro 89. Implementación de *setTurno* en *Servidor* (izquierda) y en *Jugador*

Evidentemente, la adición de *setTurno* a *Jugador* mostrada en el lado derecho del Cuadro 89 supone también la adición de una operación homónima a la interfaz *ICliente* (Cuadro 90), puesto que el campo *this.cliente* es de tipo *ICliente*, y su implementación en *Cliente* (Cuadro 91).

```
package cliente.dominio;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ICliente extends Remote {
    public void mensaje(String msg) throws RemoteException;
    public String getLogin() throws RemoteException;
    public void setTurno(String login) throws RemoteException;
}
```

Cuadro 90. Adición de *setTurno(String)* a *ICliente*

```
public void setTurno(String login) throws RemoteException {
    if (this.login.equals(login)) {
        ventana.activaBotones();
    } else {
        ventana.desactivaBotones();
    }
}
```

Cuadro 91. Implementación de *setTurno* (en *Cliente*)

Del mismo modo, habrá que implementar las operaciones *activaBotones* y *desactivaBotones* en la ventana del cliente, de tipo *JFCliente*, y en la interfaz utilizada para separar el dominio de la presentación.

La siguiente figura muestra el escenario en el que se arranca el servidor (mostrado en la ventana del fondo) y se lanzan y conectan dos clientes (*fulano* y *fulano2*); *fulano* (en la ventana de primer plano) crea una partida pulsando el botón *Crear partida*; previa consulta de las partidas abiertas, *fulano2* se une a la partida número 1 (la creada por *fulano*); *fulano* elige comenzar la partida pulsando el botón *Conectar*; al recibir este mensaje, el *Servidor* notifica a los dos clientes cuál de ellos tiene el turno: las ventanas de los dos clientes informan de que el turno lo tiene *fulano*, a la vez que se deshabilita el botón *Tirar dado* en la ventana de *fulano2*, quedándose habilitado en la *fulano*.

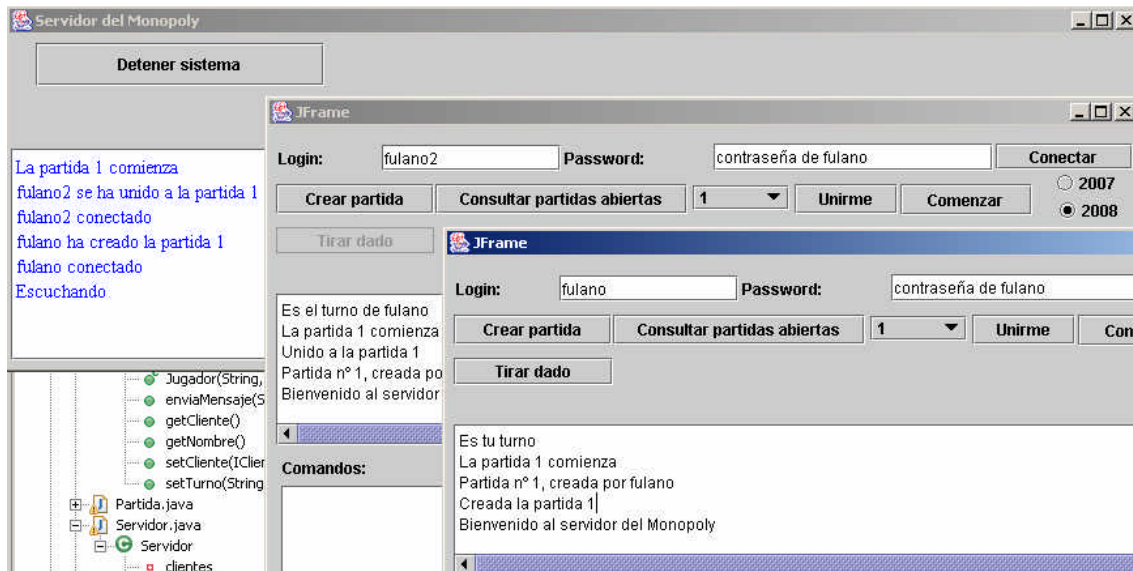


Figura 136. Escenario de conexión de dos jugadores y de comienzo de una partida

Cuando *fulano* tira los dados, se ejecuta en *JFCliente* el método *tirarDados* que genera dos números aleatorios, y se pasan al *Servidor* llamando a su operación *tirarDados(cliente : ICliente, dado1 : int, dado2 : int)*, cuya posible implementación se ofrece en el Cuadro 92: tras comprobar que la tirada es legal y que corresponde al jugador que tiene el turno, se le pasan los valores obtenidos en los dados a la *Partida*.

```
public void tirarDados(ICliente cliente, int dado1, int dado2)
    throws RemoteException, NoEstaJugando {
    String login=cliente.getLogin();
    Partida p=getPartidaEnQueEstaJugando(login);
    if (p==null)
        throw new NoEstaJugando(login);
    if (!p.getJugadorConElTurno().getNombre().equals(login))
        return;
    p.tirarDados(dado1, dado2);
}
```

Cuadro 92. Implementación de *tirarDados* (en *Servidor*)

A partir de este momento nos encontramos con un dilema importante, ya que debemos decidir cómo procesa la partida la tirada de dados. Podemos aplicar el patrón *Estado* a la *Partida* para que procese las tiradas de forma diferente según se estén decidiendo los turnos (como ocurre ahora) o en pleno juego: es decir, los valores de los dados se tratarán de una u otra forma en función del *Momento*: así, crearemos una clase abstracta *Momento* que representará el estado de la *Partida*, con una operación abstracta *tirarDados* que será interpretada por sus especializaciones que, a primera vista, pueden ser *DecidiendoTurno* y *EnJuego* (Figura 137).

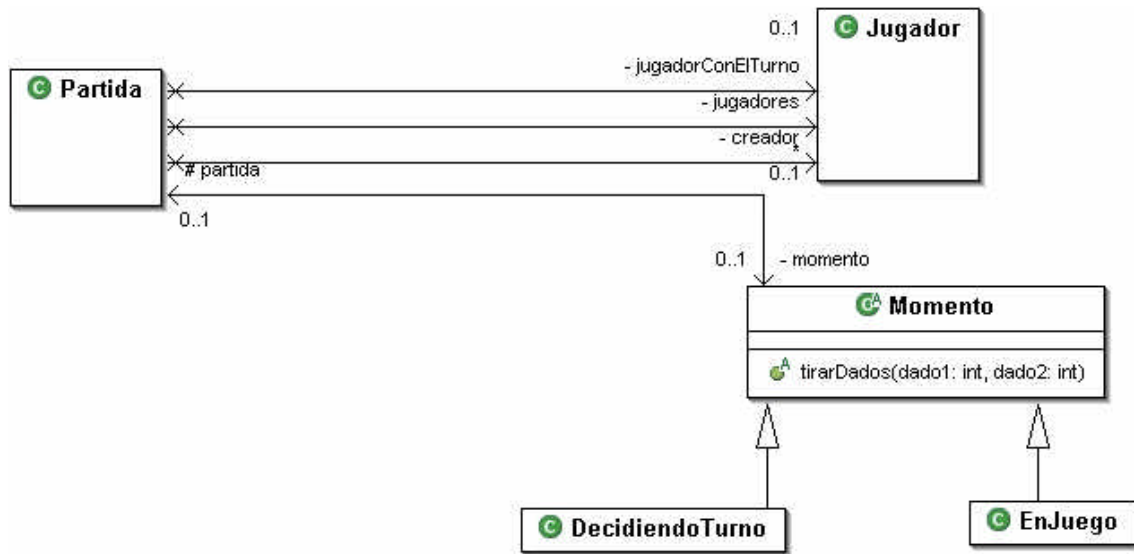


Figura 137. Aplicación del patrón *Estado* a la *Partida*, que delega su estado a *Momento*

En la figura siguiente se muestra un escenario de ejecución en el que se ha comenzado una partida con dos jugadores que están decidiendo quién comienza: el primer jugador tira los dados y el mensaje se pasa al campo *momento*, instanciado al subtipo *DecidiendoTurno*, que almacena en una lista el valor de la tirada de este jugador; el servidor, a continuación, pasa el turno al segundo jugador, que tira los dados, llegando el mensaje también al campo *momento*. Si no quedan jugadores por tirar, se determinan los posibles empates y se construye una lista con los jugadores empatados; posteriormente, se deberá pasar el turno a los jugadores de esta lista para que lancen los dados nuevamente.

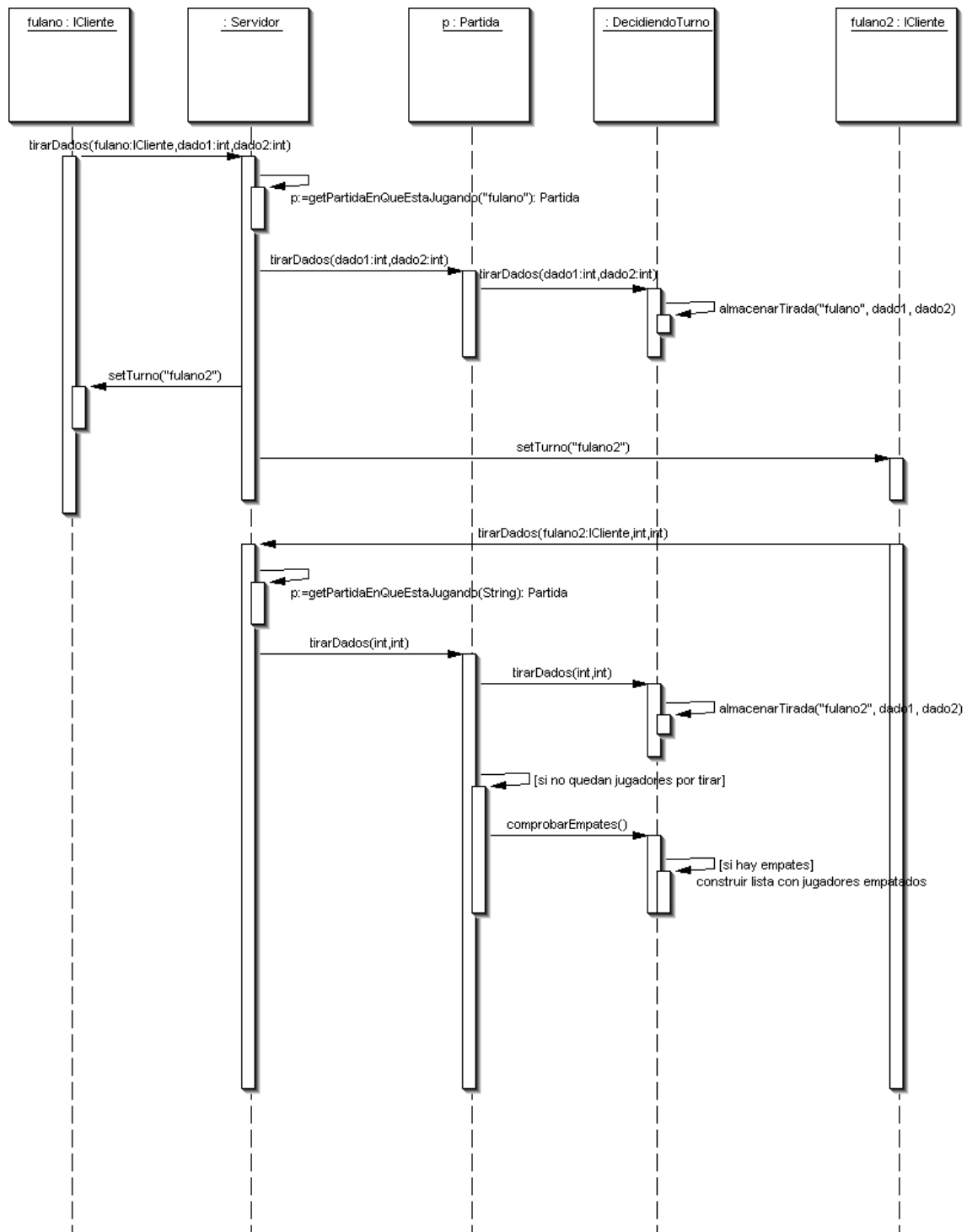


Figura 138. Dos jugadores decidiendo el turno

10.1. Pruebas del caso de uso

Probaremos un escenario similar al anterior creando una clase de prueba *JUnit* en la que dos jugadores se conectan y comienzan a jugar la partida (Cuadro 93).

```

package cliente.tests;

import cliente.dominio.Cliente;
import cliente.gui.IVentanaCliente;
import dominio.DecidiendoTurno;
import dominio.Partida;
import junit.framework.TestCase;
public class TestCliente extends TestCase implements IVentanaCliente {

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

    public void test1() {
        try {
            Cliente fulano=new Cliente(this);
            fulano.identificar("fulano", "contraseña de fulano", 2007);
            Cliente fulano2=new Cliente(this);
            fulano2.identificar("fulano2", "contraseña de fulano", 2007);
            Partida p=fulano.crearPartida();
            fulano2.unirAPartida(p.getId());
            fulano.comenzarPartida();
            fulano.tirarDados(3, 3);
            fulano2.tirarDados(5, 2);
        }
        catch (Exception ex) {
            fail(ex.toString());
        }
    }

    public void log(String msg) {
        System.out.println(msg);
    }

    public void activaBotones() {
        // TODO Auto-generated method stub
    }

    public void desactivaBotones() {
        // TODO Auto-generated method stub
    }
}

```

Cuadro 93. Una clase *JUnit* para probar un escenario

Los mensajes que se muestran por la consola son los de la Figura 139. Algunos aparecen dos veces porque los mismos textos son notificados a las ventanas de ambos jugadores, en este caso la propia clase *TestCliente*.

```

Bienvenido al servidor del Monopoly
Bienvenido al servidor del Monopoly
La partida 1 comienza
La partida 1 comienza
Es el turno de fulano
Es tu turno
fulano ha sacado 3 y 3
fulano ha sacado 3 y 3
Es tu turno
Es el turno de fulano2
fulano2 ha sacado 5 y 2
fulano2 ha sacado 5 y 2
La partida cambia de momento
La partida cambia de momento
Es tu turno
Es el turno de fulano2

```

Figura 139. Mensajes enviados desde el servidor a los clientes al ejecutar el caso de prueba anterior

El mensaje “La partida cambia de momento” lo envía el servidor después del lanzamiento de dados del segundo jugador. Al mostrar ese mensaje, la partida cambia su campo *momento* al subtipo *EnJuego*.

Nos interesa comprobar que, en caso de empate, la partida no cambia de momento. Modificamos el caso de prueba anterior para hacer esta comprobación, sustituyendo la línea `fulano2.tirarDados(5, 5)` por `fulano2.tirarDados(3, 3)`, de manera que los dos jugadores empaten. En esta ocasión, el mensaje no aparece por la consola.

Podemos seguir simulando el hecho de que ambos vuelven a tirar los dados y vuelven a empatar; tras este segundo empate, tiran de nuevo y saca más puntuación *fulano*, con lo que éste toma el turno (Figura 140).

<pre>... fulano.comenzarPartida(); fulano.tirarDados(3, 3); fulano2.tirarDados(2, 4); fulano.tirarDados(5, 5); fulano2.tirarDados(6, 4); fulano.tirarDados(6, 1); fulano2.tirarDados(3, 3); ...</pre>	<pre>... fulano ha sacado 3 y 3 fulano ha sacado 3 y 3 ... fulano ha sacado 6 y 1 fulano ha sacado 6 y 1 Es tu turno Es el turno de fulano2 fulano2 ha sacado 3 y 3 fulano2 ha sacado 3 y 3 La partida cambia de momento La partida cambia de momento Es el turno de fulano</pre>
---	---

Figura 140. Modificación del caso de prueba y resultados obtenidos en la consola

Los dos casos de prueba descritos sirven como pruebas preliminares, pero su control es difícil de mantener ya que no realizan comprobación de aserciones. Podemos añadir la aserción resaltada en el siguiente cuadro:

<pre>... Partida p=fulano.crearPartida(); ... fulano.comenzarPartida(); assertTrue(p.getJugadorConElTurno().getNombre().equals("fulano")); fulano.tirarDados(3, 3); ...</pre>

Cuadro 94. Adición de una aserción al caso de prueba

Al ejecutar el caso aparece sorprendentemente la barra roja. Esto ocurre porque el objeto *p*, de clase *Partida*, no es la misma referencia que el objeto que se utiliza en el servidor. Sin embargo, si recuperamos del servidor la partida de estos dos jugadores, el caso de prueba se supera:

<pre>... Partida p=fulano.crearPartida(); ... fulano.comenzarPartida(); assertTrue(fulano.getPartida().getJugadorConElTurno().getNombre().equals("fulano")); fulano.tirarDados(3, 3); ...</pre>

Cuadro 95. Modificación de la aserción anterior para utilizar la partida en el servidor

10.2. Los valores “interesantes” en la fase de pruebas

La clase *DecidiendoTurno* representa el momento inicial de la partida, una vez que el creador ha decidido darle comienzo. Implementa la operación *tirarDado(int dado, int dado2)*, que representa la puntuación obtenida por el cliente al tirar los dados. El objetivo de esta operación en esta clase es decidir cuál de los jugadores anotados a la partida comienza el juego. Su implementación se da en el cuadro siguiente y, como se observa, es una operación ciertamente compleja con múltiples ramas de decisión (varios *if* y varios bucles). Los casos de prueba deben escribirse con la suficiente “picardía” como para conseguir que se recorran todas las posibles ramas de la operación.

```
public class DecidiendoTurno extends Momento {
    protected Vector tiradas;
    protected Tirada tiradaMaxima;
    protected int numeroDeTiradas;

    public DecidiendoTurno() {
        super();
        tiradas=new Vector();
        tiradaMaxima=null;
        numeroDeTiradas=0;
    }

    public void tirarDados(int dado1, int dado2) throws RemoteException {
        numeroDeTiradas+=1;
        Tirada t=new Tirada(this.partida.getJugadorConElTurno(), dado1, dado2);
        if (tiradaMaxima!=null) {
            if (dado1+dado2>tiradaMaxima.getPuntos()) {
                tiradas.removeAllElements();
                tiradas.add(t);
                tiradaMaxima=t;
            } else if (dado1+dado2==tiradaMaxima.getPuntos()) {
                tiradas.add(t);
            }
        } else {
            tiradaMaxima=t;
            tiradas.add(t);
        }
        if (numeroDeTiradas==jugadores.size()) {
            numeroDeTiradas=0;
            jugadores.removeAllElements();
            for (int i=0; i<tiradas.size(); i++) {
                Tirada tirada=(Tirada) tiradas.get(i);
                Jugador j=tirada.getJugador();
                jugadores.add(j);
            }
            tiradas.removeAllElements();
            tiradaMaxima=null;
            this.setJugadorConElTurno(0);
            if (jugadores.size()==1) {
                EnJuego momento=new EnJuego();
                momento.setPartida(this.partida);
                momento.add(this.getJugadorConElTurno());
                Enumeration e=partida.getJugadores();
                while (e.hasMoreElements()) {
                    Jugador j=(Jugador) e.nextElement();
                    if (j.equals(this.getJugadorConElTurno()))
                        momento.add(j);
                }
                this.partida.setMomento(momento);
            }
        } else turno++;
    }
}
```

Cuadro 96. Código de la clase *DecidiendoTurno*

Los valores “interesantes”¹¹ son aquellos que consiguen ir ejecutando todas las posibles ramas en el camino de ejecución. En el caso concreto de la operación *tirarDado*, cuyo diagrama de flujo es el de la Figura 141, el recorrido completo de la función no es trivial. Debe analizarse la estructura de la operación para determinar los valores interesantes. Si se utilizara mutación para validar la calidad de los casos de prueba, los valores interesantes conseguirían matar porcentajes altos de los mutantes generados.

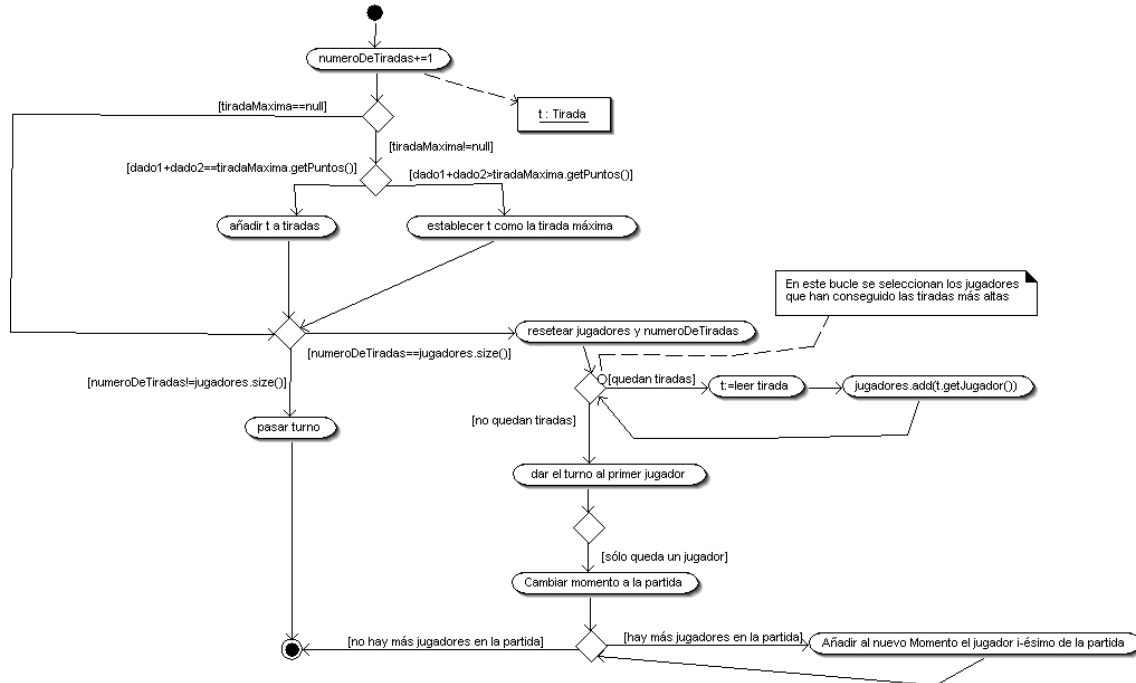


Figura 141. Diagrama de flujo de *tirarDados* (del Cuadro 96)

En lugar de utilizar mutación, instrumentaremos el código de *tirarDado* para comprobar si los casos de prueba lo recorren por completo. En el Cuadro 97 se han añadido instrucciones para que se muestre por la consola el número de las ramas que se van ejecutando. Esperamos que, tras ejecutar los casos de prueba, se hayan recorrido todos ellos.

¹¹ Así se suele llamar a este tipo de valores: Grindal M., Offutt J. y Andler S.F. (2005). "Combination testing strategies: a survey." *Software Testing, Verification and Reliability* (15): 167-199.

```

public void tirarDados(int dado1, int dado2) throws RemoteException {
    ...
    if (tiradaMaxima!=null) {
        if (dado1+dado2>tiradaMaxima.getPuntos()) {
            System.out.print("1-");
            ...
        } else if (dado1+dado2==tiradaMaxima.getPuntos()) {
            System.out.print("2-");
            ...
        }
    } else {
        System.out.print("3-");
        ...
    }
    if (numeroDeTiradas==jugadores.size()) {
        System.out.print("4-");
        ...
        for (int i=0; i<tiradas.size(); i++) {
            System.out.print("5-");
            ...
        }
        ...
        if (jugadores.size()==1) {
            System.out.print("6-");
            ...
            while (e.hasMoreElements()) {
                System.out.print("7-");
                ...
                if (j.equals(this.getJugadorConElTurno())) {
                    System.out.print("8-");
                    ...
                }
            }
            ...
        }
    } else {
        System.out.print("9-");
        ...
    }
}

```

Cuadro 97. Instrumentación de la operación para comprobar el recorrido

En el caso de prueba de *TestCliente* crearemos tres jugadores (*fulano*, *fulano2* y *fulano3*) que realizarán varios lanzamientos de dados, según el siguiente escenario:

```

public void test1() {
    try {
        ...
        fulano.tirarDados(3, 3);
        fulano2.tirarDados(2, 4);
        fulano3.tirarDados(4, 2); // Empatan y vuelven a tirar
        fulano.tirarDados(5, 5);
        fulano2.tirarDados(6, 4);
        fulano3.tirarDados(1, 1); // fulano3 eliminado
        fulano.tirarDados(6, 1);
        fulano2.tirarDados(3, 3);
        ...
    }
}

```

Cuadro 98. Escenario de ejecución

Tas la ejecución del caso, la salida que se obtiene por la consola del servidor es la siguiente:

3-9-2-9-2-4-5-5-5-3-9-2-9-4-5-5-3-9-4-5-6-7-7-7-8-

No se está recorriendo la rama etiquetada con un 1, por lo que hay que generar un caso que fuerce el paso por ella. La rama se ejecuta cuando la tirada del jugador suma más puntos que la máxima tirada hasta el momento,

por lo que modificamos el caso de prueba hasta lograr que se pase por ahí: si sustituimos la última tirada por *fulano2.tirarDados(6, 6)*, el resultado que se obtiene es:

3-9-2-9-2-4-5-5-5-3-9-2-9-4-5-5-3-9-**1**-4-5-6-7-7-8-7-

..que ya sí ha forzado el paso por el nodo 1.

Podemos concluir el caso de prueba añadiéndole las siguientes líneas, para comprobar que la partida ha cambiado de momento y que, como ha sido *fulano* el que más puntuación ha obtenido, es el que tiene el turno:

```
...
fulano.tirarDados(6, 1);
fulano2.tirarDados(3, 1);
Momento m=fulano.getPartida().getMomento();
assertTrue(m instanceof EnJuego);
assertTrue(m.getJugadorConElTurno().getNombre().equals("fulano"));
```

Cuadro 99. Comprobaciones adicionales en el caso de prueba

11. Lecturas recomendadas

Capítulo 12. APLICACIÓN SERVIDORA (II)

En este ...

1. Presentación

En el resto del capítulo iremos construyendo, aplicando desarrollo dirigido por las pruebas, las funcionalidades necesarias para que el juego pueda tener lugar. Lo que haremos será jugar una partida guiando los movimientos de *fulano*, *fulano2* y *fulano3* por el tablero de la siguiente figura. Desde la clase de test, haremos caer a cada jugador en la casilla que nos vaya interesando.

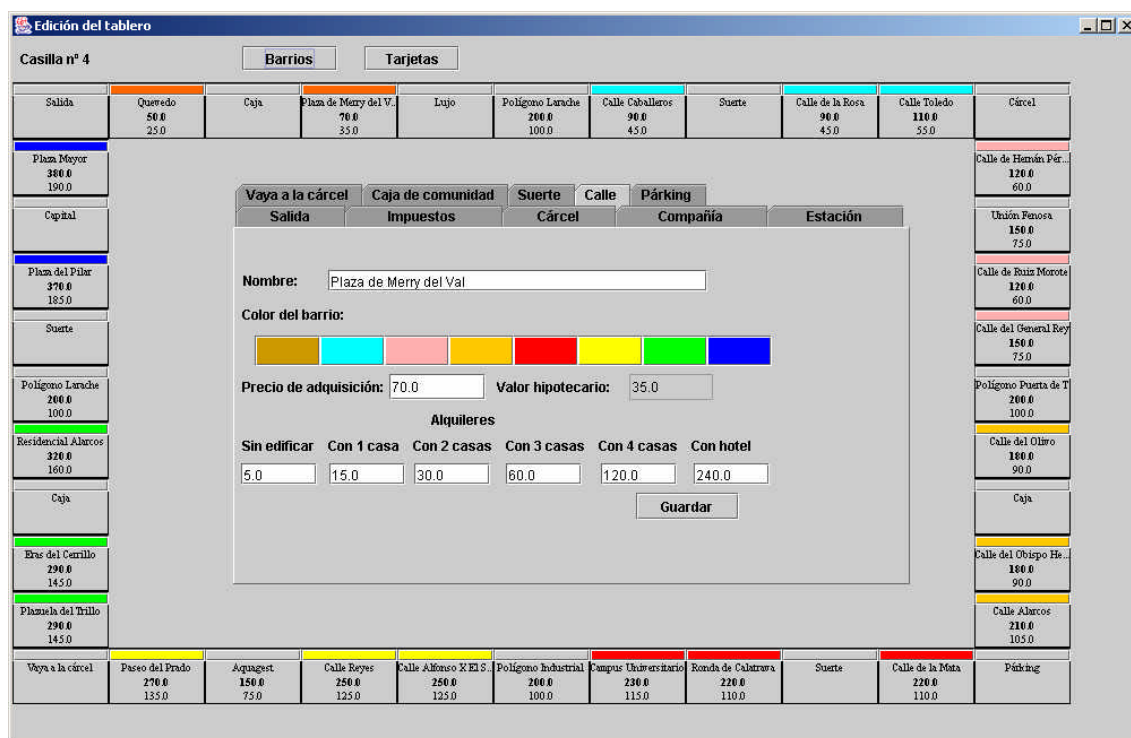


Figura 142. Tablero que utilizaremos para continuar el desarrollo

2. Preparación del tablero

Cuando el *momento* de la partida pasa de *DecidiendoTurno* a *EnJuego*, dotaremos al servidor de la capacidad de instanciar al azar un tablero de juego, de los que previamente se han debido construir en la aplicación del administrador. Para ello, es necesario crear en el servidor el conjunto de clases necesario para soportar la gestión del tablero: es decir, el propio *Tablero*, las diferentes casillas y los dos tipos diferentes de *Tarjeta*.

La siguiente figura muestra la estructura de clases en el servidor: la *Partida* tiene un *Momento* que, en caso de que sea *EnJuego*, conoce al *Tablero* sobre el que se está jugando que, a su vez, tendrá su colección de casillas y tarjetas. La estructura es muy parecida a la que teníamos en la primera aplicación, aunque ahora, por ejemplo, no delegamos el tipo de la *Casilla* con el patrón *Estado*.

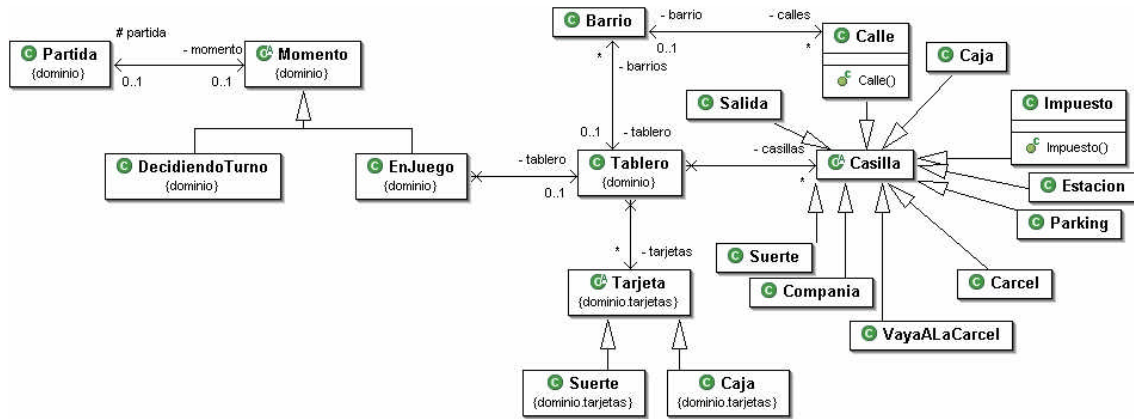


Figura 143. Estructura de clases

Como tal vez recuerde el lector, el cambio de *momento* se realizaba en las últimas líneas del método *tirarDados* de *DecidiendoTurno* (Cuadro 96, página 241), en donde se creaba una instancia de *EnJuego*, que era asignada a la *Partida*. Ahora hemos añadido al constructor de *EnJuego* la línea resaltada en el lado izquierdo del Cuadro 100 para que materialice un tablero al azar.

<pre> public EnJuego() throws SQLException { jugadores=new Vector(); tablero=Tablero.cargaTablero(); } </pre>	<pre> public static Tablero cargaTablero() throws SQLException { Connection bd=null; PreparedStatement p=null; try { String SQL="Select Ciudad, Version from Edicion"; bd=Agente.getAgente().getDB(); p=bd.prepareStatement(SQL); ResultSet r=p.executeQuery(); Vector ciudades=new Vector(); Vector versiones=new Vector(); while (r.next()) { String ciudad=r.getString(1); String version=r.getString(2); ciudades.add(ciudad); versiones.add(version); } p.close(); Random aleatorio=new Random(); int posicion=aleatorio.nextInt(ciudades.size()); String ciudad=ciudades.get(posicion).toString(); String version= versiones.get(posicion).toString(); return new Tablero(ciudad, version); } catch (SQLException ex) { throw ex; } } </pre>
---	--

Cuadro 100. Constructor de *EnJuego* (izquierda), que carga un tablero al azar (derecha)

3. Reparto de dinero

Una vez que el tablero ha sido materializado, repartiremos 1500 euros a cada jugador. Añadiremos las líneas resaltadas al caso de prueba, que suponen la creación de la operación *getDinero* en *Cliente*.

```
...
fulano.tirarDados(6, 1);
fulano2.tirarDados(3, 1);
Momento m=fulano.getPartida().getMomento();
assertTrue(m instanceof EnJuego);
assertTrue(m.getJugadorConElTurno().getNombre().equals("fulano"));
assertTrue(fulano.getDinero()==1500);
assertTrue(fulano2.getDinero()==1500);
assertTrue(fulano3.getDinero()==1500);
```

Cuadro 101. Nuevas restricciones en el caso de prueba

La operación *getDinero* en la clase *Cliente* debe devolver el dinero efectivo que tenga el jugador. El valor de esta cantidad puede ser mantenido en el servidor o en el cliente. Optando por esta segunda opción, debemos crear un campo *dinero* en *Cliente*, que será iniciado a 1500 euros por el servidor en el momento en que la partida adquiere entra en el momento *EnJuego*. Así, creamos también la operación *setDinero(importe:double)* en *Cliente* y, para que la operación sea accesible de forma remota, también en *ICliente*.

4. Lanzamiento de los dados en el estado *EnJuego*

Añadiremos las instrucciones *fulano.tirarDados(2, 1)* y *assertTrue(fulano.getPosicion()==3)* al caso de prueba anterior. De acuerdo con la Figura 137 (página 237), esto supone la implementación de *tirarDados* en la subclase *EnJuego*, así como la adición de la operación *getPosicion* a *Cliente*. Comenzando por ésta, el siguiente cuadro muestra su posible implementación:

```
public int getPosicion() throws RemoteException {
    return this.servidor.getPosicion(this);
}
```

Cuadro 102. *getPosicion* en la clase *Cliente*

De acuerdo con el cuadro anterior, es el *Cliente* quien pregunta al *Servidor* por su propia posición. La operación, entonces, debe declararse en la interfaz remota *IServidor* (Cuadro 103, izquierda, arriba) e implementarse en *Servidor* (izquierda, abajo). Como se observa, el *Servidor* localiza la partida en la que está jugando y le pregunta a ésta por la posición que, a su vez, delega la responsabilidad al *Momento*, que declara la operación como abstracta para implementarla en las dos especializaciones. El lado inferior derecho del Cuadro 103 muestra la implementación de la operación en *DecidiendoTurno*; pero para el caso de prueba nos interesa ahora implementar la operación en el estado *EnJuego*.

<pre>// En IServidor public int getPosicion(ICliente cliente) throws RemoteException;</pre>	<pre>// En Partida public int getPosicion(String login) { return this.momento.getPosicion(login); }</pre>
<pre>// En Servidor public int getPosicion(ICliente cliente) throws RemoteException { String login=cliente.getLogin(); Partida p=this. getPartidaEnQueEstaJugando(login); return p.getPosicion(login); }</pre>	<pre>// En Momento public abstract int getPosicion(String login);</pre> <pre>// En DecidiendoTurno public int getPosicion(String login) { return -1; }</pre>

Cuadro 103. La operación *getPosicion* en distintas clases de la aplicación servidora

Retomando la implementación de *tirarDados* en *EnJuego*, hay que destacar que, de acuerdo con los principios del Desarrollo dirigido por las pruebas, sólo iremos implementando aquello que sea estrictamente necesario para superar el caso de prueba. Por tanto, la implementación que, de momento, podemos dar a esta operación, es la que se muestra en el Cuadro 104.

```
public void tirarDados(int dado1, int dado2) {
    Jugador j=this.partida.getJugadorConElTurno();
    int posicionDestino=j.getPosicion()+dado1+dado2;
    posicionDestino=posicionDestino%40;
    j.setPosicion(posicionDestino);
}
```

Cuadro 104. Implementación de *tirarDados* (en *EnJuego*)

5. Compra directa de una calle

Desde la clase *TestCliente* hemos forzado a *fulano* para que saque un tres. Esto provocará que el jugador caiga en la casilla número 3, la Plaza de Merry del Val, que vale 70 euros. De acuerdo con las reglas del juego, *fulano* tendrá la opción de comprar la calle o de no comprarla; si no lo hace, saldrá a subasta entre todos los jugadores. Asumiremos, de momento, que el jugador la compra directamente.

Procediendo del mismo modo que en el caso anterior, añadimos al caso de prueba las líneas resaltadas: en primer lugar, *fulano* saca un 2 y un 1 en los dados; luego, compra la casilla; después, comprobamos que el dinero se le ha decrementado en el precio de la casilla. La última comprobación es ciertamente débil, en el sentido de que deberíamos comprobar no sólo que el dinero le ha disminuido, sino que realmente posee la Plaza de Merry del Val: sin embargo, dejaremos esta comprobación para más adelante.

```
assertTrue(fulano3.getDinero()==1500);
fulano.tirarDados(2, 1);
assertTrue(fulano.getPosicion()==3);
fulano.comprar();
assertTrue("Se esperaban 1430 euros", fulano.getDinero()==1430);
assertTrue(fulano.getPosesiones().size()==1);
```

Cuadro 105. Nuevas instrucciones en el caso de prueba

En la figura siguiente se muestra el escenario que deseamos seguir en la aplicación servidora: el *Cliente* le dice al *Servidor* que desea comprar la

casilla sobre la que se encuentra; éste pasa el mensaje a la *Partida* correspondiente, que le pasa el mensaje a su *momento*, que en este escenario está instanciado a *EnJuego* (en la clase *DecidiendoTurno*, la operación puede lanzar simplemente una excepción); *EnJuego* recupera el *Tablero* (al cual conoce, según la Figura 143 y al *Jugador* que ha lanzado, que es el que tiene el turno: entonces, el *Tablero* comprueba que la *Casilla* sea comprable (será preciso añadir la operación a *Casilla*, dándole por ejemplo una implementación por defecto para que devuelva *false*, pero redefiniéndola en calles, estaciones y compañías) y, si lo es, la añade a la colección de propiedades del *Jugador*.

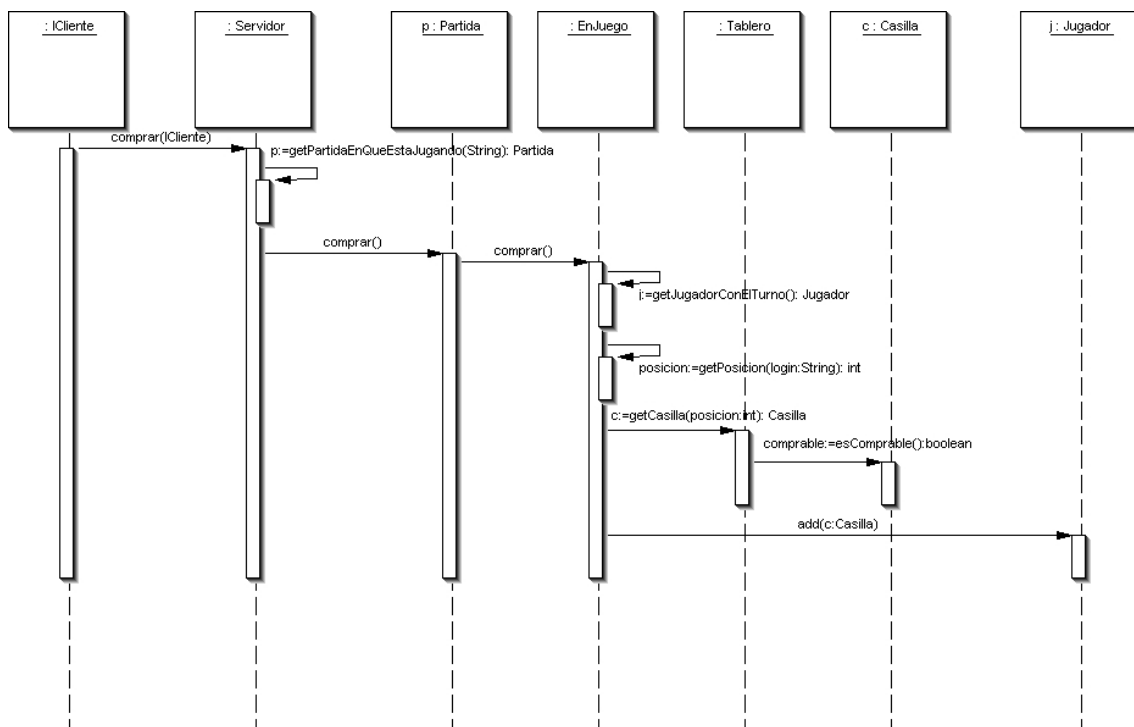


Figura 144. Escenario normal correspondiente a la compra de una calle

Así pues, la operación *comprar* en la clase *EnJuego* puede tener la implementación del Cuadro 106.

```

public void comprar() throws RemoteException, DineroInsuficiente {
    Jugador j=this.partida.getJugadorConElTurno();
    Casilla c=tablero.getCasilla(j.getPosicion());
    if (c.esComprable()) {
        j.add(c);
    }
}

```

Cuadro 106. comprar (en *EnJuego*)

La operación *add(Casilla)* en la clase *Jugador* añadirá la propiedad a la lista de propiedades del *Jugador*, le dirá a la propiedad quién es su dueño (Cuadro 107, izquierda) y disminuirá el dinero del *Jugador*. Como se ha dicho, *esComprable* tendrá una implementación por defecto en *Casilla* (lado derecho del cuadro), y una redefinición en los subtipos *Estacion*, *Calle* y *Compania* (centro).

<pre>public void add(Casilla c) { this.propiedades.add(c); c.setPropietario(this); decrementar(c.getPrecio()); }</pre>	<pre>// En Calle, Compania // y Estacion public boolean esComprable() { return (propietario==null); }</pre>	<pre>// En Casilla public boolean esComprable() { return false; }</pre>
--	---	---

Cuadro 107. Operación *add* en *Jugador*

Para superar el caso de prueba, añadimos la operación *comprar* a *Cliente* (Cuadro 108, izquierda) y a *ICliente* (derecha), recompilamos con *rmic* tanto el cliente como el servidor, y reconstruimos los ficheros JAR.

<pre>public void comprar() throws RemoteException { this.servidor.comprar(this); }</pre>	<pre>public void comprar() throws RemoteException;</pre>
--	--

Cuadro 108. Implementación de *comprar* en el cliente

Tras la modificación y adaptación del código, el caso de prueba se ejecuta y se supera satisfactoriamente.

6. Compra directa de una estación

Ahora haremos que *fulano3*, el jugador que tiene ahora el turno, caiga en la primera estación (quinta casilla) y la compre. Añadimos las dos instrucciones resaltadas al caso de prueba, ejecutamos y resulta que el caso de prueba falla porque no se decrementa el dinero de *fulano3*.

<pre>fulano.comprar(); assertTrue("Se esperaban 1430 euros", fulano.getDinero()==1430); fulano3.tirarDados(1, 4); fulano3.comprar(); assertTrue(fulano3.getDinero()==1300);</pre>

Cuadro 109. Adición de instrucciones para la compra de una estación

Tras una depuración, se aprecia que *fulano3* no dispone del turno, por lo que el método *tirarDados* de *Servidor* (Cuadro 92, página 236), rechaza la tirada sin lanzar excepción.

En el desarrollo del juego, cada jugador puede realizar varias operaciones mientras está en su turno; por tanto, dotaremos al cliente de una operación *pasarTurno*, que será ejecutada cuando el cliente haya terminado de realizar las operaciones que desee. Así, intercalamos una instrucción en el caso de prueba y realizamos los cambios pertinentes en las clases involucradas.

<pre>fulano.comprar(); fulano.pasarTurno(); assertTrue("Se esperaban 1430 euros", fulano.getDinero()==1430); fulano3.tirarDados(1, 4); fulano3.comprar(); assertTrue("Se esperaban 1300 euros", fulano3.getDinero()==1300);</pre>

Cuadro 110. Inserción de una instrucción en el caso de prueba

El caso de prueba, ahora, se supera satisfactoriamente.

7. Caída en una casilla poseída por otro jugador

A continuación, forzaremos a *fulano2* para que caiga en la casilla 3, la Plaza de Merry del Val, con lo que deberá pagar el importe correspondiente a *fulano*, su propietario. En el juego del Monopoly todo es negociable, por lo que quizá los dos jugadores involucrados mantengan ahora un diálogo para realizar algún tipo de trato. Así, implementaremos funcionalidades para:

1. Informar a *fulano* de que *fulano3* ha caído en una casilla que es suya.
2. Habilitar a *fulano3* para que pague su deuda directamente. En este caso, *fulano3* estará autorizado para pasar el turno. Si no paga, *fulano3* no podrá pasarlo.
3. No obstante lo dicho en el punto anterior, se debe permitir que *fulano* y que *fulano3* hablen y negocien. En cualquier momento, *fulano* (el propietario, que debe recibir el dinero), podrá dar por cerrado el escenario y pasar él mismo el turno.

7.1. Escenario 1: pago directo

En primer lugar, añadiremos nuevas instrucciones al caso de prueba para situar a *fulano2* en la casilla 3, hacer que este jugador pague su deuda, comprobar que su dinero se ha disminuido en 5 euros y que el de *fulano* ha aumentado en la misma cantidad:

```
fulano3.comprar();
assertTrue("Se esperaban 1300 euros", fulano3.getDinero()==1300);
fulano3.pasarTurno();
fulano2.tirarDados(1, 2);
fulano2.pagar(5.0, "fulano");
assertTrue("Se esperaban 1495 euros", fulano2.getDinero()==1495);
assertTrue("Se esperaban 1435 euros", fulano.getDinero()==1435);
```

Cuadro 111. Nuevas instrucciones para comprobar el pago directo

Para superar el caso, hay que implementar la operación *pagar* en

Cliente:

```
public void pagar(double importe, String destinatario)
    throws RemoteException, DineroInsuficiente {
    if (this.dinero<importe)
        throw new DineroInsuficiente();
    this.servidor.pagar(this, importe, destinatario);
}
```

Cuadro 112. Código de *pagar* (en *Cliente*)

...así como en *IServidor*, en *Servidor*:

```
public void pagar(ICliente cliente, double importe, String destinatario)
    throws RemoteException {
    String remitente=cliente.getLogin();
    Partida p=this.getPartidaEnQueEstaJugando(remitente);
    Partida pDestinatario=
        this.getPartidaEnQueEstaJugando(destinatario);
    if (!p.equals(pDestinatario))
        return;
    p.pagar(remitente, destinatario, importe);
}
```

Cuadro 113. pagar en la clase Servidor

...y en *Partida*:

```
public void pagar(String remitente, String destinatario, double importe)
    throws RemoteException {
    Jugador jRemitente=(Jugador) this.jugadores.get(remitente);
    Jugador jDestinatario=(Jugador) this.jugadores.get(destinatario);
    jRemitente.decrementar(importe);
    jDestinatario.incrementar(importe);
}
```

Cuadro 114. pagar en Partida

Y, como se observa, en la clase *Jugador* hay que crear la operación *incrementar(double)* que, igual que *decrementar* (que ya se utilizaba en el Cuadro 107, página 250), le comunica al cliente asociado que incremente su dinero:

```
public void decrementar(double importe) throws RemoteException {
    this.cliente.decrementar(importe);
}

public void incrementar(double importe) throws RemoteException {
    this.cliente.incrementar(importe);
}
```

Cuadro 115. incrementar y decrementar (en Jugador)

La implementación de las operaciones en el *Cliente* es la siguiente:

```
public void incrementar(double importe) throws RemoteException {
    dinero=dinero+importe;
}
public void decrementar(double importe) throws RemoteException {
    dinero=dinero-importe;
}
```

Cuadro 116. incrementar y decrementar (en Cliente)

El bonito paso de mensajes entre clientes y servidor se muestra en la figura siguiente. El caso de prueba se supera correctamente.

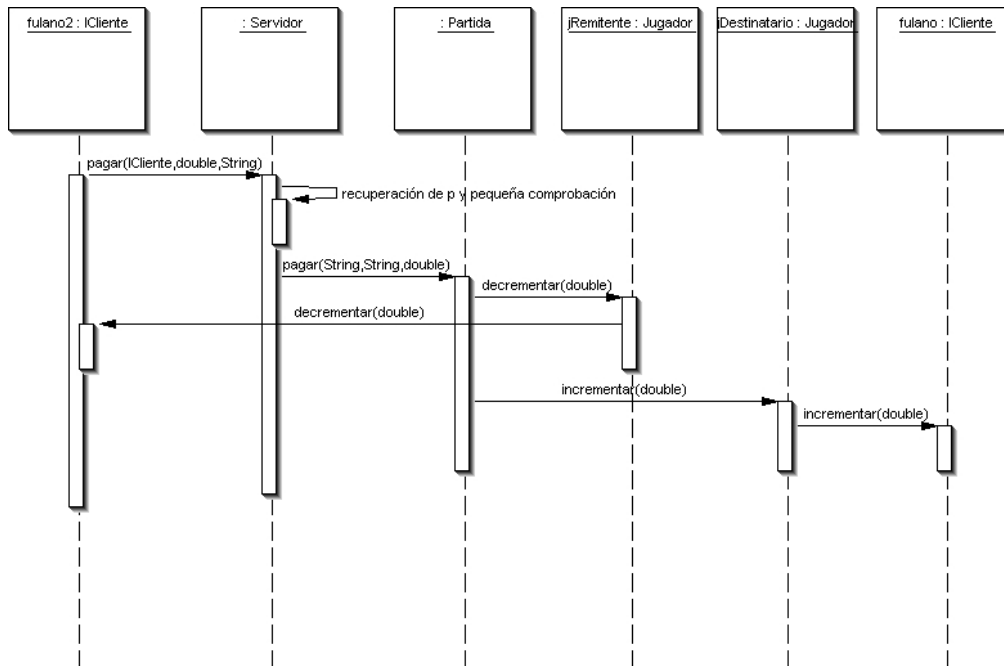


Figura 145. Paso de mensajes para realizar un pago directo

De acuerdo con la presentación del escenario realizada al principio de esta sección (página 251), *fulano2* debe poder pasar el turno tras haber realizado el pago. De acuerdo con el desarrollo dirigido por las pruebas, añadiremos tal comprobación al caso de prueba y escribiremos el código necesario para superarla.

En primer lugar, escribiremos código para comprobar que *fulano2* no puede pasar el turno mientras no pague. Cuando el *Servidor* detecte que *fulano2* quiere pasar el turno en esta situación, lanzará una *NoPuedePasarElTurnoException*. Esto se muestra en las líneas resaltadas en el cuadro siguiente: en el *try* se intenta pasar el turno; si no se lanza la excepción *NoPuedePasarElTurno*, entonces el caso de prueba encuentra un error; si sí se lanza, lo cual se corresponde con el comportamiento esperado, el control del programa salta al *catch*, que no da tratamiento. Después, *fulano2* paga la deuda, pasa el turno y se hacen otras comprobaciones.

```

fulano2.tirarDados(1, 2); // Cae en la Plaza de Merry del Val
try {
    fulano2.pasarTurno();
    fail("fulano2 ha pasado el turno");
}
catch (NoPuedePasarElTurno e) {}
fulano2.pagar(5.0, "fulano");
fulano2.pasarTurno();
assertTrue("Se esperaban 1495 euros", fulano2.getDinero()==1495);
assertTrue("Se esperaban 1435 euros", fulano.getDinero()==1435);
assertTrue("Fulano no tiene el turno", fulano.tieneElTurno());
  
```

Cuadro 117. Instrucciones añadidas al caso de prueba

Lo primero que haremos será habilitar algún mecanismo para que el *Servidor*, o la *Partida*, sean conscientes de que, después de la primera línea del cuadro anterior, *fulano2* ha adquirido una deuda con *fulano*. El paso de

mensajes que hasta el momento tiene lugar, desde que el jugador tira los dados hasta que su ficha se mueve, es el de la figura siguiente:

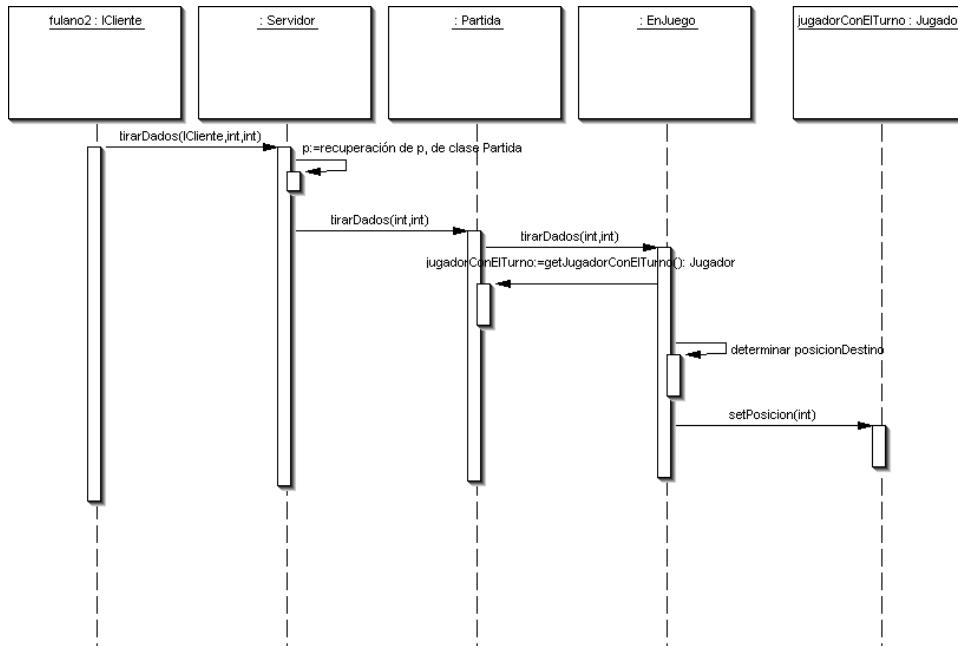


Figura 146. Paso de mensajes al tirar los dados durante el juego

Como se ve, el mensaje llega desde el jugador (a través de su correspondiente *ICiente*) hasta el *Servidor*, que lo pasa a la *Partida* en la que está jugando y ésta, a su vez, al *momento*, que está ahora instanciado al subtipo *EnJuego*. Es después del último mensaje de la figura en donde debemos implementar las operaciones para hacer la anotación de esa deuda, para lo que podemos crear una clase *Deuda*. Añadiremos un poco de código al método *tirarDados* (que teníamos en el Cuadro 104, página 248) de la clase *EnJuego* para que, si el propietario de la casilla es distinto del jugador que llega a ella, se cree un objeto de clase *Deuda* y se asigne a un nuevo campo, *deuda*:

```

public void tirarDados(int dado1, int dado2) {
    Jugador j=this.partida.getJugadorConElTurno();
    int posicionDestino=j.getPosicion()+dado1+dado2;
    posicionDestino=posicionDestino%40;
    j.setPosicion(posicionDestino);
    Casilla c=this.tablero.getCasilla(posicionDestino);
    if (c.getPropietario()!=null) {
        if (!c.getPropietario().equals(j)) {
            this.deuda=new Deuda(j, c);
        }
    }
}

```

Cuadro 118. Código de *tirarDados* en *EnJuego*. En negrita, el código añadido

El código de *Deuda* será de momento muy sencillito: poseerá dos campos, uno de tipo *Jugador* (que representará al deudor) y otro de tipo *Casilla*, mediante el que podemos acceder al acreedor (recuérdese del paso de mensajes de la Figura 144 y del Cuadro 107 que las casillas pueden tener un *propietario*) y al importe que se debe:

```

public class Deuda {
    private Jugador deudor;
    private Casilla casilla;

    public Deuda(Jugador deudor, Casilla casilla) {
        this.deudor=deudor;
        this.casilla=casilla;
    }

    public Jugador getAcreedor() {
        return casilla.getPropietario();
    }

    public Jugador getDeudor() {
        return deudor;
    }

    public double getImporte() {
        return casilla.getAlquiler();
    }
}

```

Cuadro 119. Código de la clase *Deuda*

También hay que implementar la operación *getAlquiler* en *Casilla*: tendrá una implementación por defecto en la superclase (que devuelva, por ejemplo, cero) y redefiniciones en las estaciones, calles y compañías, de manera que devuelva el valor del alquiler correspondiente.

Hasta ahora, la *Partida*, a través de su *momento*, queda enterada de la existencia de la *Deuda*. Ahora modificaremos el método *pasarTurno* para que lance una excepción si el jugador no está habilitado para ello (Cuadro 120).

```

public void pasarTurno() throws NoPuedePasarElTurno {
    if (this.deuda!=null) {
        String texto=deuda.getDeudor().getNombre() + " tiene una deuda de " +
            deuda.getImporte() + " euros con " + deuda.getAcreedor().getNombre();
        throw new NoPuedePasarElTurno(texto);
    }
    turno=(turno+1) % this.jugadores.size();
}

```

Cuadro 120. Código de *pasarTurno* (en la clase *EnJuego*)

Por último, nos queda implementar la operación *tieneElTurno* en la clase *Cliente*, que a su vez enviará el mensaje hasta el *Servidor*. Llevando la codificación hasta ésta, se hace necesario añadir la operación del cuadro siguiente. El caso de prueba ya se supera.

```

public boolean tieneElTurno(ICliente cliente) throws RemoteException {
    String login=cliente.getLogin();
    Partida p=this.getPartidaEnQueEstaJugando(login);
    return p.getJugadorConElTurno().getNombre().equals(login);
}

```

Cuadro 121. Operación añadida a la clase *Servidor*

7.2. Escenario 2: negociación entre jugadores

Para este escenario debemos habilitar la posibilidad de que los jugadores hable entre sí, de manera que puedan, por ejemplo, ponerse de acuerdo para condonar una deuda, retrasar su pago, etc. En esta ocasión, añadiremos código al caso de prueba para que *fulano*, que ya tiene el turno, saque 2 en los dados y avance hasta la estación, poseída por *fulano3*. En lugar de pagar la

deuda y continuar como hasta ahora, los dos jugadores negociarán y será *fulano3* el que autorice el paso del turno, incluso aunque no haya cobrado.

Se da además la circunstancia de que, para sacar un 2, *fulano* habrá tenido que sacar doble (un 1 en cada dado), por lo que continuará con el turno. De este asunto nos ocupamos más adelante.

En primer lugar, añadimos las siguientes instrucciones al caso de prueba para habilitar la posibilidad de que los jugadores puedan hablar.

```
assertTrue("Fulano no tiene el turno", fulano.tieneElTurno());
fulano.tirarDados(1, 1);
fulano.enviarMensaje("fulano3", "¿Te puedo pagar más tarde?");
fulano3.enviarMensaje("fulano", "De acuerdo");
```

Cuadro 122. Nuevas líneas añadidas al caso de prueba

Para que el caso se supere, implementamos el método *enviarMensaje(login:String, texto:String)* en *Cliente* (clase perteneciente al proyecto *clienteJugador*, véase Figura 129, página 226), cuyo código se muestra en el Cuadro 124. Esto supone la creación de *enviarMensaje(cliente: ICliente, destinatario:String, texto: String)* en *IServidor* y su implementación en *Servidor* (Cuadro 123).

```
public void enviarMensaje(ICliente cliente, String destinatario, String texto)
    throws RemoteException {
    Jugador jDestinatario=(Jugador) this.jugadores.get(destinatario);
    jDestinatario.recibirMensaje(cliente.getLogin(), texto);
}
```

Cuadro 123. Implementación en *Servidor*

El *Servidor* busca al *Jugador* en su lista (no es preciso acceder a la partida) y sobre él ejecuta *recibirMensaje(remitente:String, texto:String)*, operación que hay que añadir a la interfaz *ICliente* e implementar en *Cliente*.

Las dos operaciones añadidas a esta clase (una para enviar mensajes a un jugador a través del servidor, y otra para recibirlos) aparecen en el cuadro siguiente, al que ya se ha hecho referencia más arriba.

```
public void enviarMensaje(String destinatario, String texto) throws RemoteException {
    this.servidor.enviarMensaje(this, destinatario, texto);
}

public void recibirMensaje(String login, String texto) throws RemoteException {
    this.ventana.log(login + "> " + texto);
}
```

Cuadro 124. Operaciones añadidas a la clase *Cliente*

Tras ejecutar el caso de prueba, observamos que, por la consola, aparecen correctamente los dos mensajes:

```
fulano> ¿Te puedo pagar más tarde?
fulano3> De acuerdo
```

Figura 147. Mensajes enviados y recibidos por los clientes

A partir de este momento, *fulano3*, el acreedor, puede pasar el turno del juego. Sin tener en cuenta, de momento, el hecho que el jugador ha saca-

do dobles, debe superarse el caso de prueba con las dos siguientes nuevas instrucciones que le añadimos:

```
fulano.enviarMensaje("fulano3", "¿Te puedo pagar más tarde?");
fulano3.enviarMensaje("fulano", "De acuerdo");
fulano3.pasarTurno();
assertTrue("fulano3 no tiene el turno", fulano3.tieneElTurno());
```

Cuadro 125. Nuevas instrucciones añadidas al caso de prueba

El caso de prueba, de momento, no se supera: el *Servidor* recibe el mensaje *pasarTurno* desde el jugador y comprueba si el emisor se corresponde con el jugador que tiene el turno (único autorizado, de momento, para pasarlo). Si no lo tiene, ignora el mensaje, como se ve en el Cuadro 126, que muestra la implementación actual de la operación:

```
public void pasarTurno(ICliente cliente) throws RemoteException, NoPuedePasarElTurno {
    String login=cliente.getLogin();
    Partida p=this.getPartidaEnQueEstaJugando(login);
    if (!p.getJugadorConElTurno().getNombre().equals(login))
        return;
    p.pasarTurno();
}
```

Cuadro 126. Implementación actual de *pasarTurno* en *Servidor*, que hay que modificar

Una posibilidad pasa por comprobar si el jugador que pasa el turno es el acreedor, caso en el cual debe dejarle que lo pase:

```
public void pasarTurno(ICliente cliente) throws RemoteException, NoPuedePasarElTurno {
    String login=cliente.getLogin();
    Jugador j=(Jugador) this.jugadores.get(login);
    Partida p=this.getPartidaEnQueEstaJugando(login);
    if (p.esAcreedor(j)) {
        p.pasarTurno();
        return;
    }
    if (!p.getJugadorConElTurno().getNombre().equals(login))
        return;
    p.pasarTurno();
}
```

Cuadro 127. Modificación de *pasarTurno* en *Servidor*

La operación *esAcreedor(Jugador)* se implementa en la clase *Partida*, que a su vez le pregunta al *momento* que, en el caso del subtipo *EnJuego*, resuelve la cuestión mediante su campo *deuda*. A pesar de estos cambios, el caso de prueba sigue fallando, porque el método *pasarTurno* de *EnJuego* (Cuadro 120, página 255) no deja pasarlo si hay una deuda. Así, modificamos todavía un poco más el método *pasarTurno* de *Servidor* para que no deje pasarlo al jugador que, aun poseyéndolo, sea deudor:

```
public void pasarTurno(ICliente cliente) throws RemoteException, NoPuedePasarElTurno {
    String login=cliente.getLogin();
    Jugador j=(Jugador) this.jugadores.get(login);
    Partida p=this.getPartidaEnQueEstaJugando(login);
    if (p.esAcreeedor(j)) {
        p.pasarTurno();
        return;
    }
    String mensajeDeudor=p.mensajeDeudor(j);
    if (mensajeDeudor!=null) {
        throw new NoPuedePasarElTurno(mensajeDeudor);
    }
    if (!p.getJugadorConElTurno().getNombre().equals(login))
        return;
    p.pasarTurno();
}
```

Cuadro 128. Añadimos una comprobación adicional a *pasarTurno* de *Servidor*

El caso de prueba, finalmente, no encuentra fallos y JUnit muestra la barra verde:

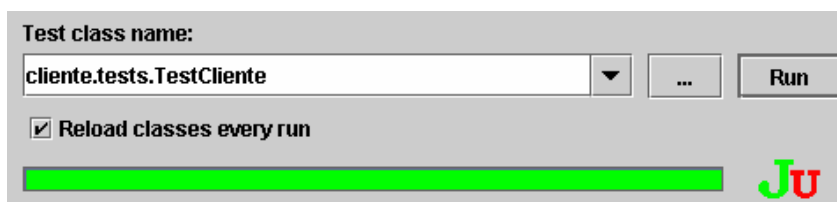


Figura 148. Superación de la última versión del caso de prueba

7.3. Control de la tirada de dobles

A pesar de lo implementado en la sección anterior, el turno no debe pasar a *fulano3* (última línea del Cuadro 125), sino que debe mantenerse en *fulano*, que ha sacado dobles. Así, hay que modificar la última línea del caso para hacer esta comprobación, dejándola de este modo:

```
fulano.enviarMensaje("fulano3", "¿Te puedo pagar más tarde?");
fulano3.enviarMensaje("fulano", "De acuerdo");
fulano3.pasarTurno();
assertTrue("fulano no tiene el turno", fulano.tieneElTurno());
```

Cuadro 129. Modificación del caso para controlar el turno en el caso de dobles

En la aplicación servidora modificaremos el código para controlar quién es el jugador que pasa el turno. Así, cambiamos la signatura de *pasarTurno* en *Partida*, *Momento* y en sus especializaciones, añadiendo un parámetro de tipo *Jugador*. Igualmente, hay que controlar el número de dobles con, por ejemplo, una variable *int* en la clase *EnJuego* (Cuadro 130).

```
public class EnJuego extends Momento {
    private Deuda deuda;
    private int dobles;
    ...
    public void tirarDados(int dado1, int dado2) {
        ...
        if (c.getPropietario()!=null) {
            ...
        }
        if (dado1==dado2)
            this.dobles++;
        else dobles=0;
    }

    public void pasarTurno(Jugador j) throws NoPuedePasarElTurno {
        if (dobles>0) {
            if (j.equals(this.getJugadorConElTurno()))
                throw new NoPuedePasarElTurno("Ha sacado dobles");
        } else
            turno=(turno+1) % this.jugadores.size();
    }
}
```

Cuadro 130. Código añadido a *EnJuego*

7.4. Tres dobles seguidos

Cuando un jugador saca tres dobles seguidos va a la cárcel. Una vez en ésta, puede: (1) permanecer tres turnos sin tirar para luego salir; (2) pagar 50 euros de multa y salir en su siguiente turno; (3) si dispone de una tarjeta de Suerte o de Caja de comunidad que lo libra de ella, sale en su siguiente turno.

Escribiremos código para satisfacer un escenario en el que *fulano* vuelve a sacar dos dobles y decide pasar el turno, que recae en *fulano3*. Éste tira y cae en una casilla de *fulano* que, aunque está en la cárcel, está habilitado para cobrar. Tras realizar el pago, *fulano3* pasa el turno a *fulano2*, que cae en la cárcel y pasa su turno, que debe recaer en *fulano3*, ya que *fulano* está encarcelado durante 3 turnos.

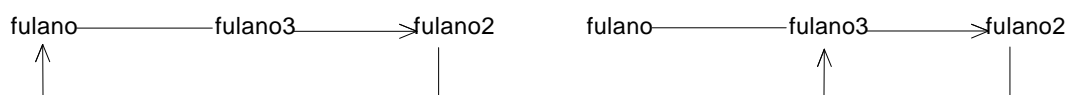


Figura 149. Turno normal (izquierda) y turno en este escenario (derecha)

Las adiciones necesarias al caso de prueba correspondiente al párrafo anterior son las del siguiente cuadro:

```

assertTrue("fulano no tiene el turno", fulano.tieneElTurno());
fulano.tirarDados(2, 2);
fulano.comprar();
fulano.tirarDados(3, 3);
fulano.pasarTurno();
assertTrue(fulano3.tieneElTurno());
fulano3.tirarDados(4, 2); // Cae en una casilla de fulano
assertTrue(fulano3.tieneElTurno());
fulano3.pagar(55, "fulano");
fulano3.pasarTurno();
assertTrue(fulano2.tieneElTurno());
fulano2.tirarDados(3, 2); // Cae en la cárcel, en modo "visita"
fulano2.pasarTurno();
assertTrue(fulano3.tieneElTurno());

```

Cuadro 131. Control del paso de turno tras ser encarcelado

El cambio en la aplicación afecta una vez más a la clase *EnJuego*, como se muestra en el cuadro siguiente. Cada vez que se sacan dobles se incrementa el campo *dobles*; si éste alcanza el valor de 3, se manda a la cárcel al jugador y se le asignan 3 turnos sin tirar. Además, adelantamos la comprobación del tercer doble con respecto a la implementación anterior (Cuadro 130) para que no se compruebe la casilla en la que el jugador con tres dobles hubiera caído, de modo que no se cree ninguna deuda.

```

public void tirarDados(int dado1, int dado2) {
    Jugador j=this.partida.getJugadorConElTurno();
    if (dado1==dado2) {
        this.dobles++;
        if (dobles==3) {
            j.setPosicion(tablero.getCarcel());
            j.setTurnosQueLeQuedan(3);
            dobles=0;
            return;
        }
    } else dobles=0;
    int posicionDestino=j.getPosicion()+dado1+dado2;
    posicionDestino=posicionDestino%40;
    j.setPosicion(posicionDestino);
    Casilla c=this.tablero.getCasilla(posicionDestino);
    if (c.getPropietario()!=null) {
        if (!c.getPropietario().equals(j)) {
            this.deuda=new Deuda(j, c);
        }
    }
}

```

Cuadro 132. Modificación de *tirarDados* para superar el caso de prueba (en *EnJuego*)

Las líneas que añadimos al caso de prueba para guiar el proceso de desarrollo son las siguientes:


```

fulano.tirarDados(3, 3); // Tercer doble: paso automático de turno
assertTrue(fulano3.tieneElTurno());
fulano3.tirarDados(4, 2);
assertTrue(fulano3.tieneElTurno());
fulano3.pagar(55, "fulano");
fulano3.pasarTurno();
assertTrue(fulano2.tieneElTurno());
fulano2.tirarDados(3, 2); // Cae en la cárcel, en modo "visita"
fulano2.pasarTurno();
// Como está en la cárcel, pasa a fulano3 (1 turno)
assertTrue(fulano3.tieneElTurno());
fulano3.tirarDados(1, 4);
fulano3.pasarTurno();
fulano2.tirarDados(2, 5);
fulano2.pasarTurno();
// Como está en la cárcel, pasa a fulano3 (2 turnos)
fulano3.tirarDados(2, 5);
fulano3.pasarTurno();
fulano2.tirarDados(1, 4);
fulano2.pasarTurno();
// Como está en la cárcel, pasa a fulano3 (3 turnos)
fulano3.tirarDados(1, 6);
fulano3.pasarTurno();
fulano2.tirarDados(6, 5);
fulano2.pasarTurno();
// Como lleva tres turnos, ya le toca
assertTrue(fulano.tieneElTurno());

```

Cuadro 133. Líneas para comprobar que el turno salta a *fulano*

Los cambios en la aplicación *Servidora* los realizamos, como última-mente, en la clase *EnJuego*, en donde modificamos el método *pasarTurno* para que compruebe que, si el jugador que pasa el turno ha sacado dobles, se lance una excepción; si el jugador que ha pasado el turno no es el que ha sacado dobles, pero es el acreedor de una deuda (comprobación que se hace previamente en *Servidor*, véase Cuadro 128), entonces se le deja pasar el turno. El jugador *siguiente* puede ser el siguiente en la lista o, si está sancionado por estar en la cárcel, ser otro.

```

public void pasarTurno(Jugador j) throws NoPuedePasarElTurno {
    if (!j.equals(this.getJugadorConElTurno())) {
        this.saldarDeuda();
    }
    if (dobles==0) {
        turno=(turno+1) % this.jugadores.size();
    }
    Jugador siguiente=(Jugador) this.jugadores.get(turno);
    if (siguiente.getTurnosQueLeQuedan()!=0) {
        siguiente.setTurnosQueLeQuedan(siguiente.getTurnosQueLeQuedan()-1);
        pasarTurno(siguiente);
    }
}

```

Cuadro 134. Modificación de *pasarTurno* (en *EnJuego*)

Como se observa, se han añadido dos operaciones (*getTurnosQueLeQuedan* y *setTurnosQueLeQuedan*) a la clase *Jugador*, que actúan sobre un nuevo campo (*turnosQueLeQuedan*) que debe ponerse a 3 cuando el jugador saca tres dobles, y que se va decrementando a medida que pasa turnos sin tirar.

El caso de prueba puede completarse para comprobar que *fulano* vuelve a recuperar el turno en el momento adecuado.

Capítulo 13. APLICACIÓN SERVIDORA (III)

En este capítulo se presenta la notación de máquinas de estados para describir el comportamiento de instancias de clases.

1. Máquinas de estados

Además de para describir las interacciones entre entidades (como los pasos en la ejecución de casos de uso que vimos en la sección 4 del Capítulo 6, página 173), otra de las utilidades principales de las máquinas de estados es la representación del comportamiento de instancias de clases. Además, y aunque menos frecuentemente, también se utilizan para describir el protocolo de uso de una parte de un sistema.

Para la primera forma de uso, los nodos representarán los distintos estados por los que puede pasar un objeto, y los arcos se corresponden con las transiciones entre dichos estados que, normalmente, serán disparadas por la ejecución de operaciones de la clase cuyo comportamiento están representando.

En la web del Object Management Group (<http://www.omg.org>) podemos encontrar toda la especificación de UML y, entre ésta, la correspondiente a las máquinas de estados. La figura siguiente muestra el metamodelo¹² de UML 2.0, que representan la sintaxis abstracta de las máquinas de estados.

Gráficamente, una máquina de estados es un grafo dirigido con diferentes tipos de arcos y de nodos. Algo de esto puede deducirse de la Figura 150, en donde se observa que una máquina de estados (clase *StateMachine*) está formada por un conjunto de regiones (agregación con *Region*); cada región posee un conjunto de vértices (*Vertex*) y un conjunto de transiciones (agregación con *Transition*). Los vértices (obsérvese que *Vertex* es una clase abstracta) pueden ser de diferentes tipos (clases *Pseudostate*, *State* y *ConnectionPointReference*). Los estados (clase *State*) pueden ser estados “a secas” (obsérvese que *State* es concreta) o bien estados finales (especialización *FinalState* de la clase *State*). De forma general, los vértices serán los nodos del grafo.

¹² Un metamodelo es un modelo que se utiliza para representar modelos.

Las transiciones de cada *Region* representan los arcos del grafo. Cada transición puede tener uno o más eventos disparadores (*Trigger*), que pueden provocar el cambio de estado.

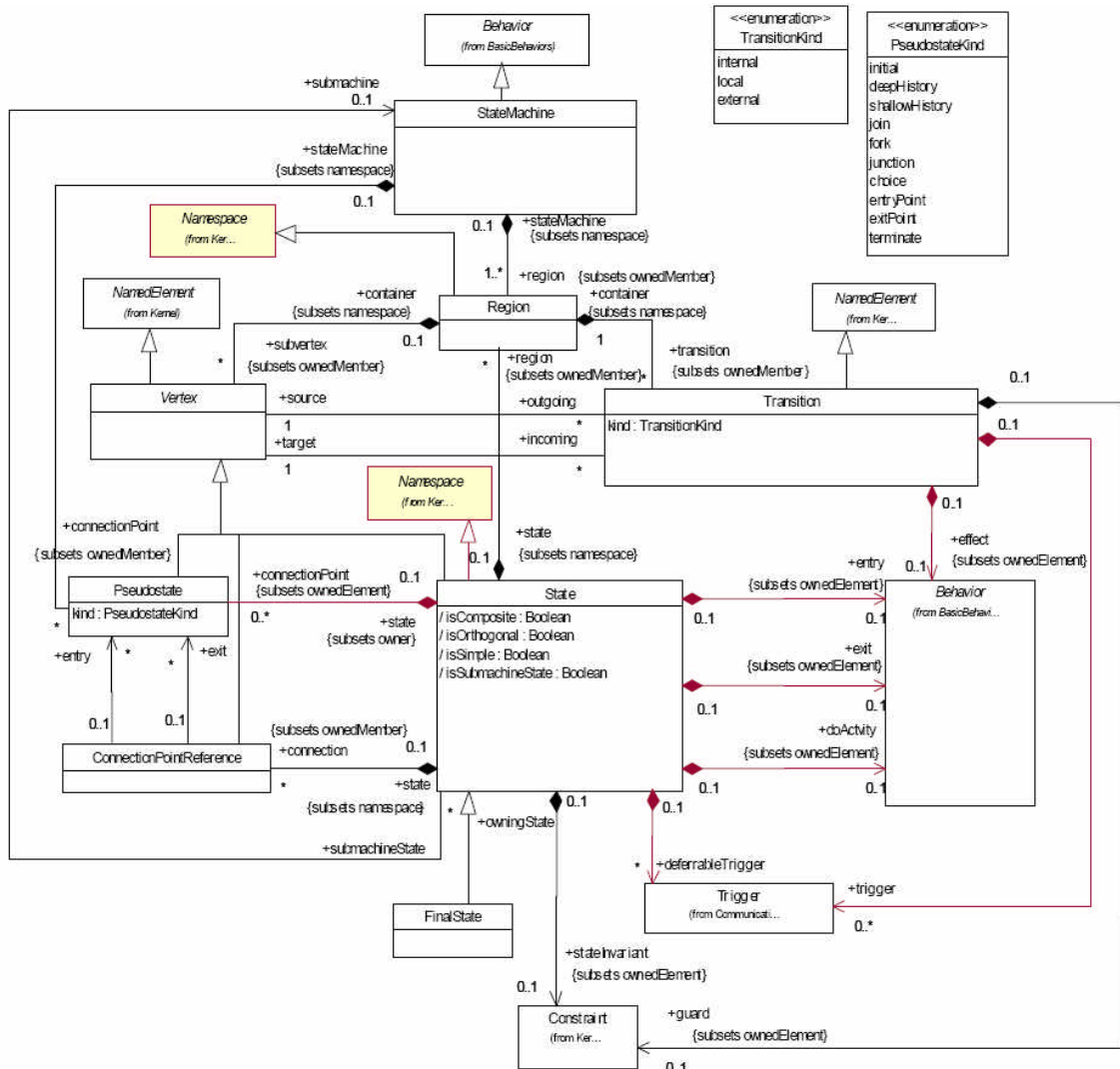


Figura 150. Sintaxis abstracta de las máquinas de estados. © OMG

Un estado describe una situación durante la cual se verifica alguna invariante. Un estado puede ser simple, compuesto o puede tratarse de un estado submáquina (Tabla 34). El tipo del estado se determina por los valores de sus atributos *isComposite*, *isSimple* e *isSubmachineState*. El atributo *isOrthogonal* determina si el estado es ortogonal: es decir, si tiene dos o más regiones.

Tipo de estado	Significado
Simple	El estado no tiene subestados (no tiene regiones ni submáquinas)
Compuesto	Contiene al menos una región o se descompone en dos o más regiones ortogonales. Cada región tiene un conjunto de vértices (todos disjuntos) y un conjunto de transiciones.
Estado submáquina	Semánticamente, un estado submáquina es equivalente a un estado compuesto. Sin embargo, se utiliza para factorizar comportamiento: es decir, cuando el mismo comportamiento debe aparecer en varios lugares, puede utilizarse un estado submáquina.

Tabla 34. Tipos de estados

Además de las transiciones que lo conectan con otros estados, un Estado puede tener acciones de entrada (agregación desde *State* hasta *Behavior* etiquetada *entry*), actividades (agregación desde *State* hasta *Behavior* etiquetada *doActivity*) y acciones de salida (agregación desde *State* hasta *Behavior* etiquetada *exit*).

Las acciones de entrada y las acciones de salida se ejecutan cada vez que se entra o se sale del estado; las actividades se ejecutan mientras se está en ese estado. Existen multitud de tipos de acciones. Las actividades responden a la misma descripción. Por ejemplo:

- De llamada a operación (*CallOperationAction*): representa una llamada a una operación de otro objeto.
- De envío de señal (*SendSignalAction*): representa una señal (mensaje asíncrono) que se envía desde un objeto a otro.
- De creación de objeto (*CreateObjectAction*).
- De destrucción de objeto (*DestroyObjectAction*).

Los eventos se corresponden con las operaciones que producen el cambio de estado: es decir, los eventos provocan el disparo de una transición. Algunos tipos de eventos utilizados frecuentemente son los siguientes:

- Evento de señal: representa la recepción de una señal, que es un estímulo asíncrono transmitido entre instancias (una excepción, por ejemplo).
- Llamada a una operación: representa la recepción de una solicitud de una operación específica. Dos casos especiales son el evento de creación de un objeto y el de su destrucción, que pueden estereotiparse respectivamente con «*create*» y «*destroy*».
- Evento de tiempo: representa que se ha llegado a un instante de tiempo predeterminado, que se expresa con el atributo *when*.
- Evento de cambio: 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.

1.1. Ejemplos

El estado de un objeto es función de los valores de sus atributos

Como sabemos, los usos habituales de las máquinas de estados son la descripción del comportamiento de instancias de clases, la definición de interacciones entre entidades y la descripción de protocolos de uso.

Para el primer caso (que quizá sea el más frecuente), es preciso tener en cuenta 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, y a esto dedicaremos el presente epígrafe. Por otro lado, es preciso hacer notar que es difícil encontrar herramientas de diseño que permitan utilizar toda la notación de las máquinas de estado.

La Figura 151, dibujada con el plugin de Eclipse que venimos utilizando, representa el comportamiento de una cuenta bancaria. Una vez que la cuenta ha sido creada, la única operación admitida es el ingreso de dinero, que provoca que se sume al atributo *mSaldo* el *importe* (acción *mSaldo+=importe*) y que la instancia pase al estado *Saldo positivo*. En este estado se admiten las operaciones *ingresar*, *retirar* y *cancelar*. Una consecuencia de la ejecución de *cancelar* es que se liquida la cuenta (véase la acción *liquidarCuenta*). Cuando se ejecuta la operación *ingresar* y la cuenta tiene *Saldo positivo*, no se produce cambio de estado de la instancia, aunque sí se suma el importe ingresado. Al finalizar la ejecución de *retirar* desde *Saldo positivo*, se evalúa el saldo de la cuenta con *getSaldo* (una operación de consulta que no puede tener efectos colaterales: es decir, que no puede alterar el estado del sistema) y se determina el siguiente estado. La barra vertical es un pseudoestado de tipo *choice*, que también determina el siguiente estado cuando la cuenta está en negativo y se produce un ingreso. *retirar* tiene una guarda en *Saldo negativo*, que determina que, en este estado, sólo puede retirarse dinero si el *titular* tiene un préstamo hipotecario. Para que la máquina de estados sea coherente con el diseño, la clase *Cuenta* debe tener un atributo *mSaldo* y otro *titular* que sea de algún tipo que admita la operación booleana *tienePrestamoHipotecario()*.

En la figura tenemos, por tanto, estados simples, un pseudoestado *choice*, eventos de tipo llamada, acciones y varias condiciones de guarda.

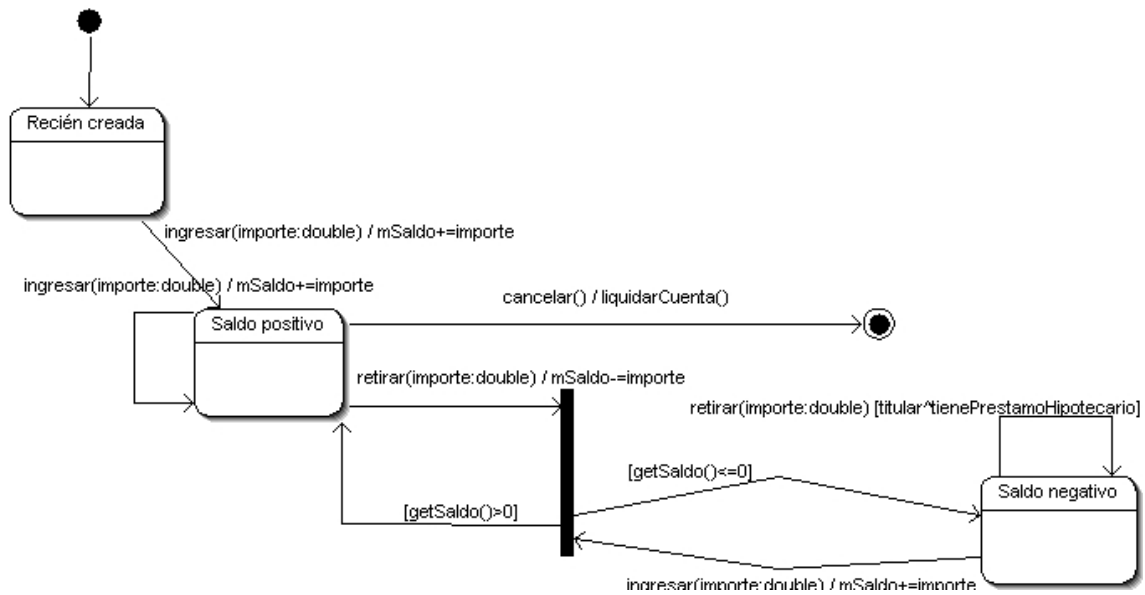


Figura 151. Máquina de estados para una *Cuenta* (ejemplo 1)

La siguiente máquina está dibujada con Poseidon for UML Community Edition 3.0. En ella se ha creado un estado compuesto *Cuenta activa*, en el que se encuentra la instancia cuando ofrece *Saldo positivo* o *Saldo negativo*. Se ha añadido una acción de entrada al estado *Recién creada* que asigna cero al atributo *mSaldo*. Cuando hay estados compuestos, las transiciones pueden tener lugar entre estados de niveles distintos (de *Recién creada* vamos a *Cuenta activa/Saldo positivo*) o entre estados del mismo nivel (de *Cuenta activa* vamos al estado *Calculando comisiones*). Cuando se desea cancelar una cuenta y se dan las condiciones para ello, se bloquean los productos asociados a la cuenta en *Cancelando cuenta*; a continuación, y de forma concurrente, se cancelan los productos y se imprimen varios documentos. La transición entre *Cancelando cuenta/Cancelación de tarjetas* y *Cancelando cuenta/Cancelación de préstamos* no está etiquetada por ningún evento ni condición, lo que indica que se pasa a cancelar los préstamos cuando se termina la cancelación de las tarjetas.

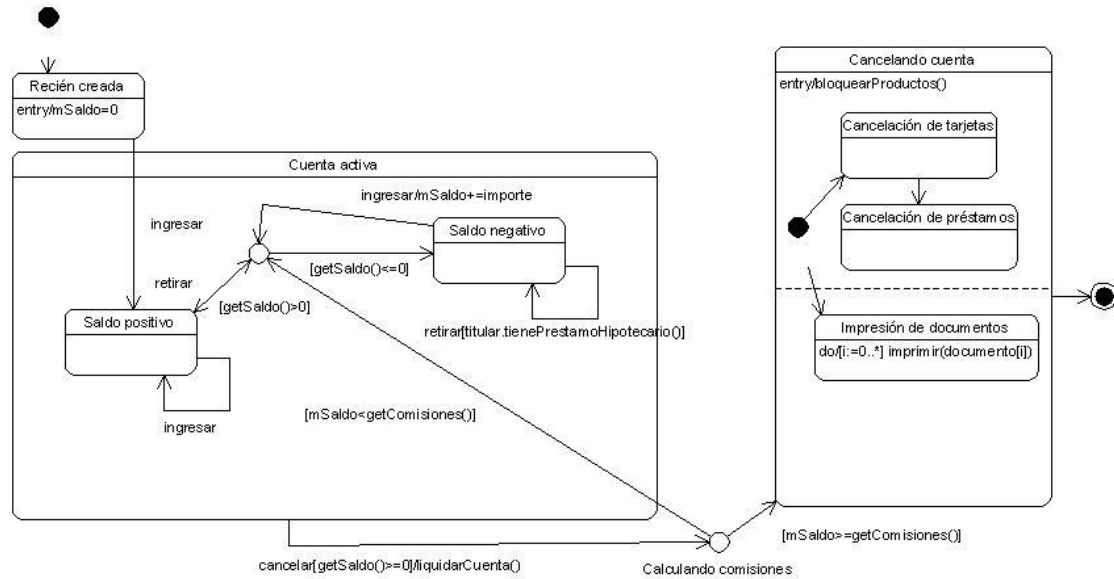


Figura 152. Máquina de estados para una Cuenta (ejemplo 2)

Puede haber transiciones sin eventos que las disparen pero que tengan sin embargo acciones. Con esto se significa que la transición se dispara automáticamente tras terminar las actividades indicadas en el estado, y que se ejecuta la acción indicada como consecuencia de la transición.

1.2. Derivación de especificaciones OCL a partir de máquinas de estados

Muchos elementos de las máquinas de estados pueden interpretarse como pre y postcondiciones de la clase que se está representando: de la Figura 152, deducimos que, por ejemplo, *ingresar* puede ejecutarse en todos los estados, aunque puede ir a estados distintos tras la ejecución. Una traducción fiel a OCL de la máquina puede expresarse como en el Cuadro 135.

```
Context Cuenta::retirar(importe : double) : void
pre:
    oclInState(#CuentaActiva.SaldoPositivo) or
    (oclInState(#CuentaActiva.SaldoNegativo) and titular.tienePrestamoHipotecario())
post:
    getSaldo()>0 implies oclInState(#SaldoPositivo) and
    getSaldo()<=0 implies oclInState(#SaldoNegativo)

Context Cuenta::ingresar(importe : double) : void
pre:
    oclInState(#ReciénCreada) or
    oclInState(#CuentaActiva)
post:
    if getSaldo()>0 then oclInState(#SaldoPositivo) else oclInState(#SaldoNegativo )
    and oclInState(#SaldoNegativo)@pre implies mSaldo=mSaldo+importe
```

Cuadro 135. Código OCL obtenido de la máquina de estados de la Figura 152

Como se observa, se utilizan la operación *oclInState* del tipo *OclAny* (supertipo de todos los tipos, la *Cuenta* incluida) y el símbolo almohadilla. Éste se utiliza en OCL para hacer referencia a los posibles valores de una

enumeración que, en este ejemplo, es el posible conjunto de estados de las instancias de la clase.

Como sabemos, el estado de una instancia es una función de los valores de sus campos. Así, podríamos expresar los estados *ReciénCreada*, *CuentaActiva.SaldoCero*, etc. como función de, por ejemplo, el campo *mSaldo*, y obtener una especificación OCL más próxima al código.

2. Descripción de clases de la aplicación servidora con máquinas de estados

Cuando se utilizan para representar el comportamiento de instancias de clases, las máquinas de estados se aplican a aquellas clases cuyas instancias tienen un comportamiento suficientemente significativo. En la aplicación servidora, por ejemplo, probablemente no se utilizarían máquinas de estado para describir el comportamiento de *Servidor*, que actúa como un mero intermediario entre los clientes y las clases *Partida*, *EnJuego*, etc., que incluyen la carga real de procesamiento del sistema.

No obstante, y dado que *Servidor* conoce una lista de objetos de clase *Partida*, puede considerarse que las diferentes partidas forman parte del *Servidor*, y describir entonces el comportamiento del sistema a partir de esta clase.

3. Lecturas recomendadas

El Object Management Group pone mucho material gratuito a disposición del público. Así, puede encontrarse en su web la especificación completa de UML 2.0. La semántica y notación de las máquinas de estados puede encontrarse en el capítulo 15 del documento titulado “Unified Modeling Language: Superstructure”.

Capítulo 14. APLICACIÓN SERVIDORA (IV): CONSTRUCCIÓN DE UN MONITOR DE SEGUIMIENTO

En este capítulo construiremos un monitor para que el administrador del sistema pueda seguir el desarrollo de cada partida. Se creará una ventana en la que se mostrará visualmente el desarrollo de cada partida. Cada vez que un jugador lance los dados, compre una casa, pague un dinero, etc., la operación aparecerá reflejada en la ventana.

1. Análisis del problema

En la Figura 153 se muestra una versión ampliada de la Figura 143, en la que ahora se han añadido las principales clases creadas durante los desarrollos de los capítulos anteriores.

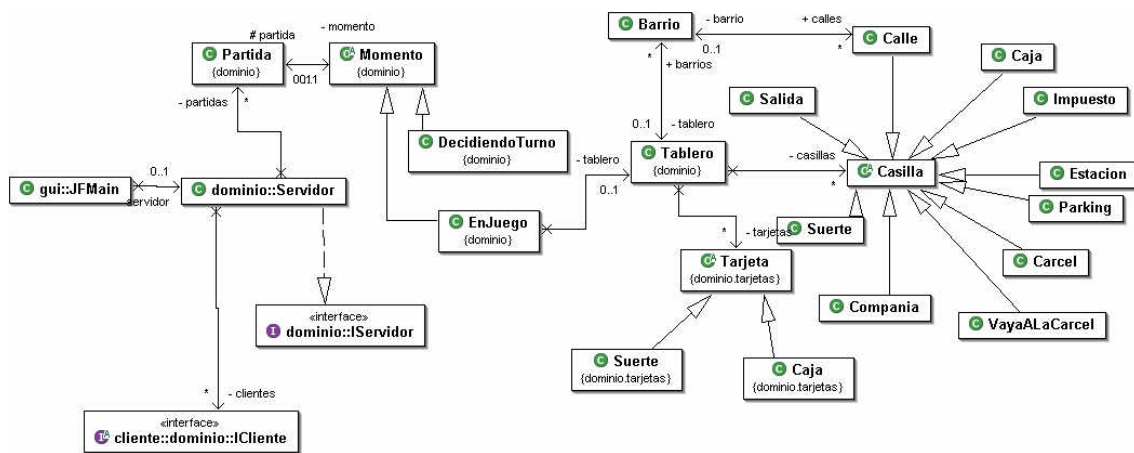


Figura 153. Estructura formada por las principales clases del sistema

Queremos añadir, a esta aplicación, una nueva ventana en la que se muestre la evolución de cada partida: así, cuando se cree una partida, se creará una ventana (u otro elemento visual, como un panel dentro de una ventana) que asociaremos a esa partida. Cada vez que un jugador realice una acción (que llega desde su *ICliente* al *Servidor*), la *Partida* notificará la operación a ese elemento visual que tiene asociado.

En el primer escenario que describimos (Figura 154), un cliente ejecuta la operación *crearPartida* sobre el *Servidor*, que crea efectivamente una

nueva partida, crea también un *JPanel* creado ex profeso y le dice a la partida que el elemento visual al que debe notificar es dicho panel.

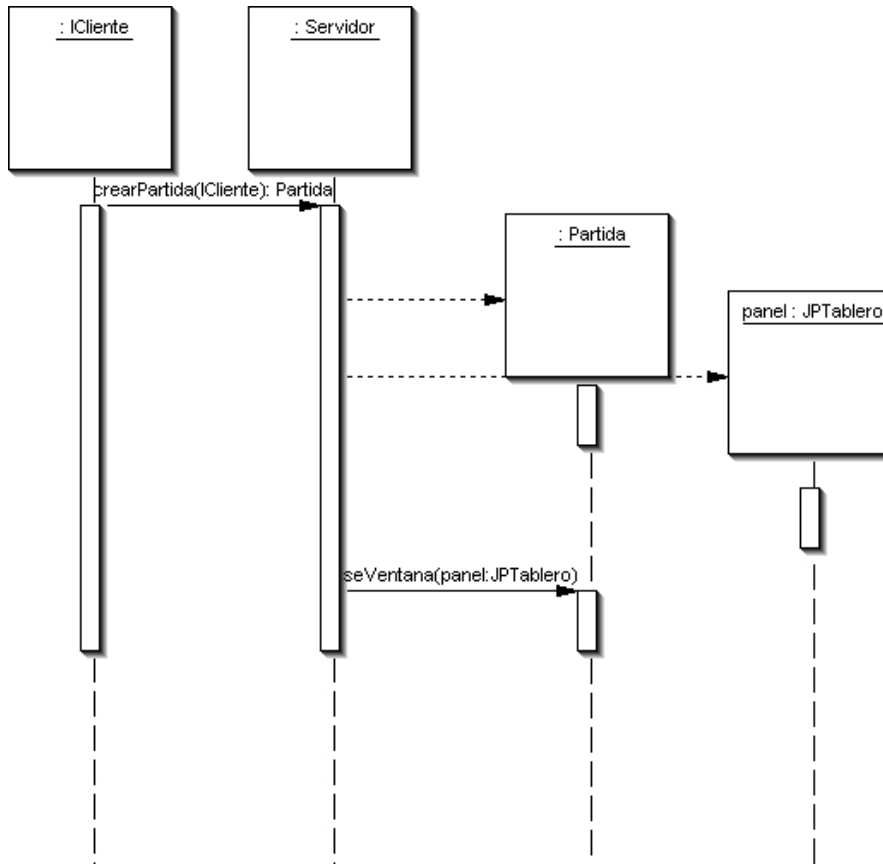


Figura 154. Escenario de la carga del panel de la partida

Con el fin de mantener el desacoplamiento entre el dominio y la presentación, el objeto *panel* que, en la figura anterior, se pasa desde *Servidor* a *Partida* no será un *JPTablero*, sino que lo pasaremos en la forma de una interfaz que represente elementos visuales capaces de mostrar la configuración de un tablero en un momento dado. Una forma posible de implementar este paso de mensajes consiste en añadir las instrucciones resaltadas en el Cuadro 136 al código de *crearPartida* en la clase *Servidor*: tras crear la partida, indicarle cuál es el *Jugador* que la ha creado, ponerla en la lista de partidas del *Servidor* y notificar a la ventana habitual (Figura 131) que se ha creado una partida, se construye una instancia de *JPTablero* y se le dice a la partida que es a esta instancia a la que debe notificar.

```

public Partida crearPartida(ICliente cliente)
    throws UsuarioNoRegistrado, RemoteException, YaEstaJugando {
    Jugador j=(Jugador) this.jugadores.get(cliente.getLogin());
    if (j==null)
        throw new UsuarioNoRegistrado(cliente.getLogin());
    if (estaJugando(j))
        throw new YaEstaJugando(cliente.getLogin());
    Partida p=Partida.nuevaPartida();
    p.setCreador(j);
    p.add(j);
    partidas.put(""+p.getId(), p);
    ventana.log(cliente.getLogin() + " ha creado la partida " + p.getId());
    JPTablero panel=new JPTablero();
    p.setVentana(panel);
    ventana.show(panel);
    return p;
}

```

Cuadro 136. Una posible forma de implementar el paso de mensajes

La solución mostrada evita el acoplamiento de *Partida* con *JPTablero*, ya que esta clase implementa una nueva interfaz, *IVentanaTablero*, y es a un objeto de este tipo al que conoce (Figura 155).

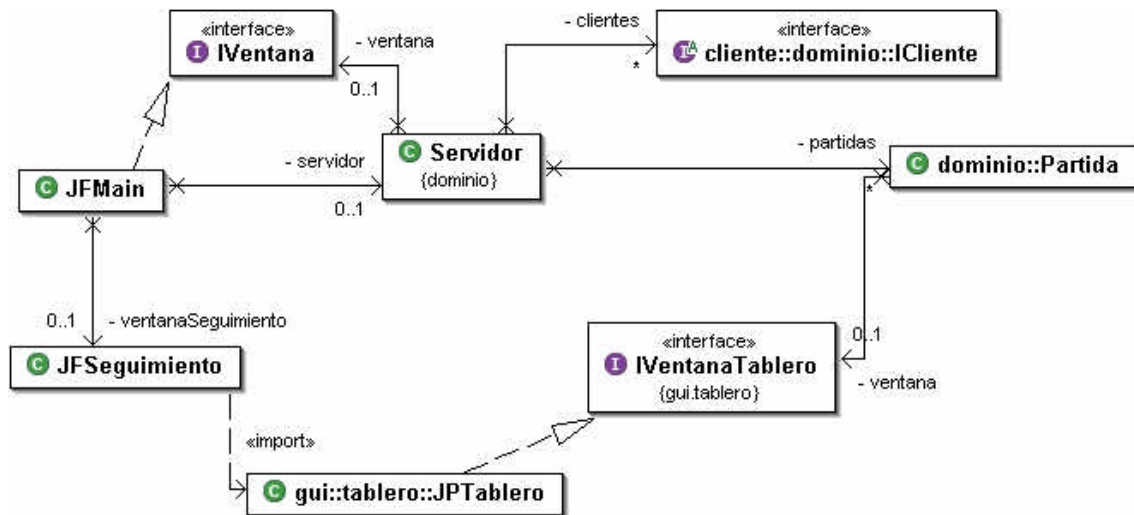


Figura 155. La *Partida* conoce a un *IVentanaTablero*, que puede instanciarse a un *JPTablero*

Sin embargo, el acoplamiento entre dominio y presentación se está dando en la clase *Servidor* ya que, como se veía en el Cuadro 136 (que muestra una operación de *Servidor*), se crea una instancia de tipo *JPTablero*. El acoplamiento entre dominio y presentación es indeseable, porque restringe la utilización de las clases de dominio (que son las que resuelven el problema y tienen la mayor complejidad) a un determinado tipo de ventanas, pudiendo así impedir su uso en otras plataformas o dificultando la evolución de la aplicación con la utilización de interfaces de usuario más modernas.

De este modo, evitaremos la solución mostrada en el Cuadro 136 y delegaremos a la propia ventana a la cual conoce el servidor (véase, en la figura anterior, que se trata del campo *ventana*, del tipo genérico *IVentana*) la creación de un tipo adecuado de *IVentanaTablero*.

Sustituiremos algunas líneas de código del Cuadro 136 para que garantizar la falta de acoplamiento. Se sugiere la implementación del cambio mostrado en el siguiente cuadro, en donde se han sustituido cuatro líneas por una nueva, en la que se realiza una llamada a una nueva operación *logCreacionDePartida(cliente: ICliente, p: Partida)*, que debe ser adecuadamente manejada por las clases que implementen la interfaz *IVentana*, tipo del campo *ventana*.

```
public Partida crearPartida(ICliente cliente)
    throws UsuarioNoRegistrado, RemoteException, YaEstaJugando {
    Jugador j=(Jugador) this.jugadores.get(cliente.getLogin());
    if (j==null)
        throw new UsuarioNoRegistrado(cliente.getLogin());
    if (estaJugando(j))
        throw new YaEstaJugando(cliente.getLogin());
    Partida p=Partida.nuevaPartida();
    ventana.logCreacionDePartida(cliente, p);
    p.setCreador(j);
    p.add(j);
    partidas.put(""+p.getId(), p);
    //ventana.log(cliente.getLogin() + " ha creado la partida " + p.getId());
    //JPTablero panel=new JPTablero();
    //p.setVentana(panel);
    //ventana.show(panel);
    return p;
}
```

Cuadro 137. Modificación del Cuadro 136 para desacoplar dominio de presentación

El código de *logCreacionDePartida* en *JFMain* se muestra en el Cuadro 138.

```
public void logCreacionDePartida(ICliente cliente, Partida p) throws RemoteException {
    this.log(cliente.getLogin() + " ha creado la partida " + p.getId());
    JPTablero panel=new JPTablero();
    p.setVentana(panel);
    if (this.ventanaSeguimiento==null) {
        this.ventanaSeguimiento=new JFSeguimiento();
    }
    this.ventanaSeguimiento.setVisible(true);
    ventanaSeguimiento.setTitle("ventanaSeguimiento");
    this.ventanaSeguimiento.carga(panel);
}
```

Cuadro 138. *logCreacionDePartida*, en *JFMain*

La siguiente figura muestra el resultado visual que se obtiene en el *Servidor* después de que llega el mensaje de creación de partida: cuando el mensaje le ha llegado al servidor, se ejecuta sobre el *JFMain* (que implementa la interfaz *IVentana*) la operación *logCreacionDePartida*, que pone visible (si no lo estaba ya) una instancia de una nueva clase llamada *JFSeguimiento* (véase Figura 155), sobre la cual se van cargando paneles de tipo *JPTablero*.

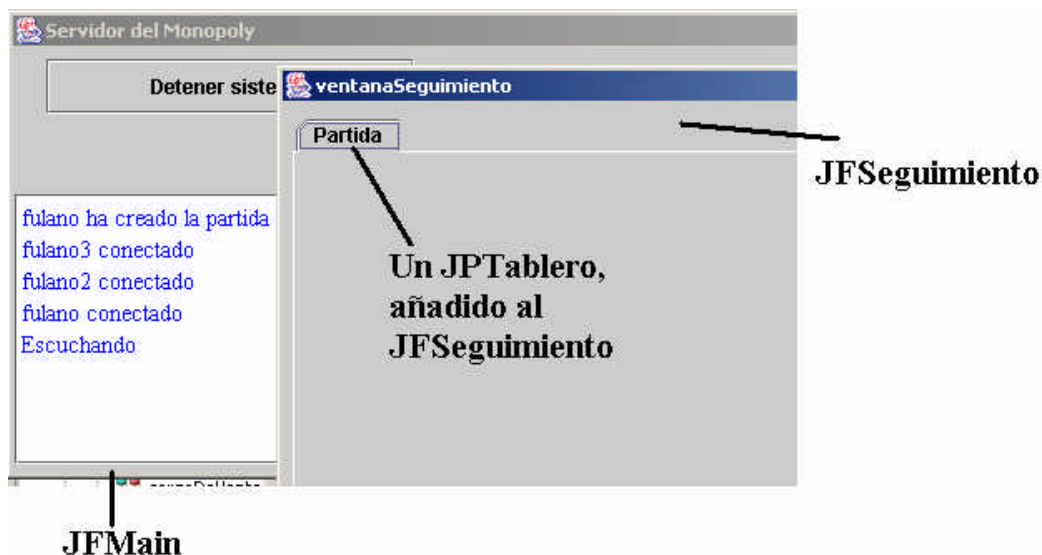


Figura 156. Resultado visual obtenido al crear una *Partida*

A partir de este momento, se desea que se notifique al *JPTablero* cualquier suceso que ocurra en la partida. Como se recordará, la partida se encuentra inicialmente en el estado *DecidiendoTurno*: pues bien, el primer mensaje que podemos considerar es *tirarDados* de la clase *Partida*, a cuyo código añadimos la instrucción resaltada en el Cuadro 139, que tiene implicaciones en la interfaz *IVentanaTablero* y en la clase *JPTablero* que mostrá- bamos en la Figura 155.

```
public void tirarDados(int dado1, int dado2) throws RemoteException {
    this.notifica(this.getJugadorConElTurno().getNombre() + " ha sacado " + dado1 +
        " y " + dado2);
    this.momento.tirarDados(dado1, dado2);
    this.ventana.tirarDados(this.getJugadorConElTurno().getNombre(), dado1, dado2);
    setTurno(momento.getJugadorConElTurno());
}
```

Cuadro 139. Adición de una instrucción a *tirarDados* (en *Partida*)

Procedemos exactamente de la misma manera con todas las operaciones que deban notificarse a la *IVentanaTablero* con un mensaje de texto. Tras decidir el turno y adquirir la *Partida* el *momento* de tipo *EnJuego*, debe notificarse la selección del tablero que, recuérdese (Cuadro 100, página 246), se hacía al azar. Añadiremos al constructor de *EnJuego* una llamada a una operación que notifica a la *IVentanaTablero* cuál ha sido el tablero seleccionado (Cuadro 140): en algún lugar habrá sido preciso indicar a *EnJuego* que debe notificar a la misma *IVentanaTablero* a la que hasta ahora ha estado notificando *Partida*.

```
public EnJuego() throws SQLException, RemoteException {
    jugadores=new Vector();
    tablero=Tablero.cargaTablero();
    ventana.notificaTablero(tablero);
}
```

Cuadro 140. Notificación del tablero seleccionado (en la clase *EnJuego*)

La operación *notificaTablero(t:Tablero)* se añade a la interfaz *IVentanaTablero* y se implementa adecuadamente en *JPTablero*.

1.1. Carga del tablero en *JPTablero* con una fábrica simple

Una fábrica es una clase cuyo principal objetivo es la construcción de instancias de determinados tipos. En el caso que nos ocupa, el método *notificaTablero* debe añadir al *JPTablero* algunos elementos que representen las casillas del tablero, que pueden ser de tipos muy diversos.

En la figura siguiente se ilustra cómo se dota a la fábrica de una funcionalidad encargada de crear las representaciones visuales adecuadas a cada tipo de casilla del tablero. En su operación *buildCasilla* se pregunta por el tipo de la *Casilla* que recibe como parámetro y se construye una instancia de la *JPCasilla* correspondiente.

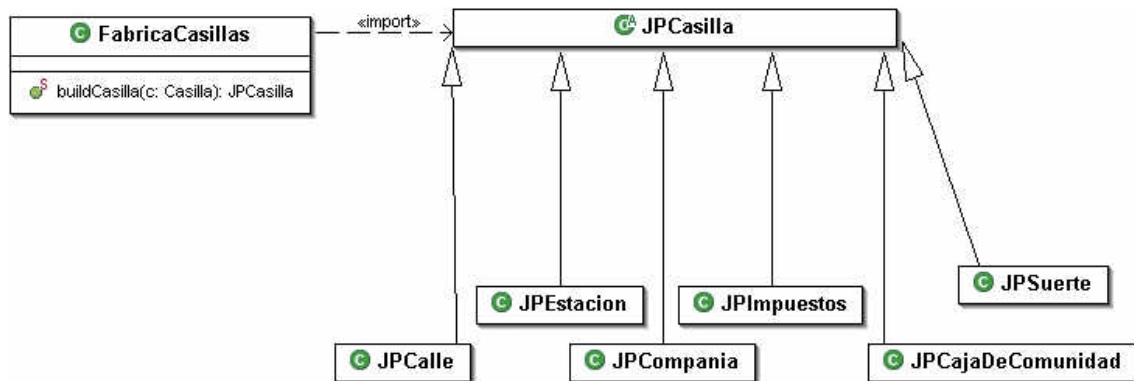


Figura 157. Jerarquía parcial de casillas “visuales” y una fábrica que se encarga de su construcción

JPCasilla puede ser un pequeño *JPanel* al que daremos el aspecto adecuado a la casilla que corresponda. El cuadro siguiente muestra un fragmento del código de la operación *buildCasilla* de la fábrica:

```

public static JPCasilla buildCasilla(Casilla c) {
    JPCasilla result=null;
    if (c instanceof Calle) {
        result=new JPCalle();
        result.setCasilla(c);
    } else if (c instanceof Compania) {
        result=new JPCompania();
        result.setCasilla(c);
    } else if (c instanceof Caja) {
        ...
    }
    return result;
}

```

Cuadro 141. Posible implementación de *buildCasilla* (en *FabricaCasillas*)

La operación del cuadro anterior se llama desde la operación *notificaTablero(tablero:Tablero)*, implementada en *JPTablero* (Cuadro 142): se recorre la lista de casillas y, merced a la fábrica, se obtiene una representación visual adecuada, que se sitúa sobre el panel.

```
public void notificaTablero(Tablero tablero) {
    for (int i=0; i<tablero.casillasSize(); i++) {
        Casilla c=tablero.getCasilla(i);
        JPCasilla casilla=FabricaCasillas.buildCasilla(c);
        if (casilla!=null) {
            casilla.setSize(85, 59);
            int x=getX(i), y=getY(i);
            casilla.setLocation(x, y);
            casilla.setVisible(true);
            this.add(casilla, null);
        }
    }
}
```

Cuadro 142. notificaTablero (en JPTablero)

Un problema importante del diseño y la implementación dadas a la fábrica es la elevada complejidad ciclomática de *buildCasilla*, con tantas sentencias *if* como subtipos de *Casilla*, y la excesiva dependencia de otras clases del sistema: piénsese que, si en un futuro se añadieran nuevos tipos de casillas al juego, sería preciso modificar el código fuente de esta operación.

1.2. Carga del tablero en *JPTablero* con un *Builder* (Constructor)

El *Builder* es uno de los patrones de creación de Gamma, Helm, Johnson y Vlissides¹³. Se utiliza cuando existe un conjunto de varias jerarquías de herencia y debe crearse un “lote” de objetos, formado por un conjunto de especializaciones de estas clases de la jeraquía.

Tal conjunto de jerarquías existe en nuestro caso, ya que disponemos de dos clases abstractas (*JPCasilla* y *Casilla*) con varias especializaciones cada una. En función del subtipo de, por ejemplo, *Casilla*, se debe crear una instancia de la otra clase (*JPCasilla*), también con el subtipo adecuado. Con este patrón, se asigna a uno de los objetos del lote la responsabilidad de crear el resto de objetos, que serán instancias de alguno de los subtipos del resto de jerarquías de herencia.

En nuestro ejemplo concreto hay dos jerarquías en paralelo, pero con total dependencia una de otra (Figura 158).

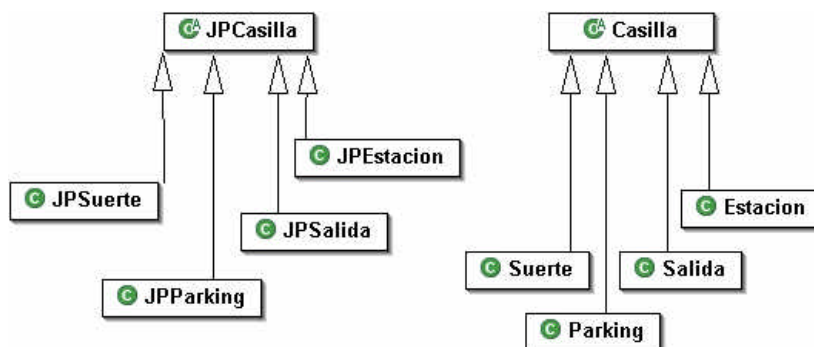


Figura 158. Dos árboles de herencia en paralelo

¹³ Descritos en su libro “Patrones de Diseño”, con edición en castellano de 2004 (editorial Addison Wesley).

En una de las superclases del conjunto de jerarquías (en nuestro caso, solamente dos) crearemos una operación que se encargue de instanciar el resto de objetos que componen el lote de objetos necesario. Normalmente, la operación será abstracta en la superclase, pero se encontrará redefinida en todas las subclases. En la figura siguiente se ha añadido a *JPCasilla* la operación *construirLote*, que crea un objeto de clase *Lote*, que conoce a tantas instancias como sea necesario para construir el lote de productos: en este caso, también sólo dos (las instancias de *JPCasilla* y de *Casilla* de los subtipos que correspondan). En el lado derecho de la figura se muestra una posible implementación de *construirLote* en *JPSalida*, una de las especializaciones de la clase a la que hemos decidido otorgar la responsabilidad de crear el lote completo de productos.

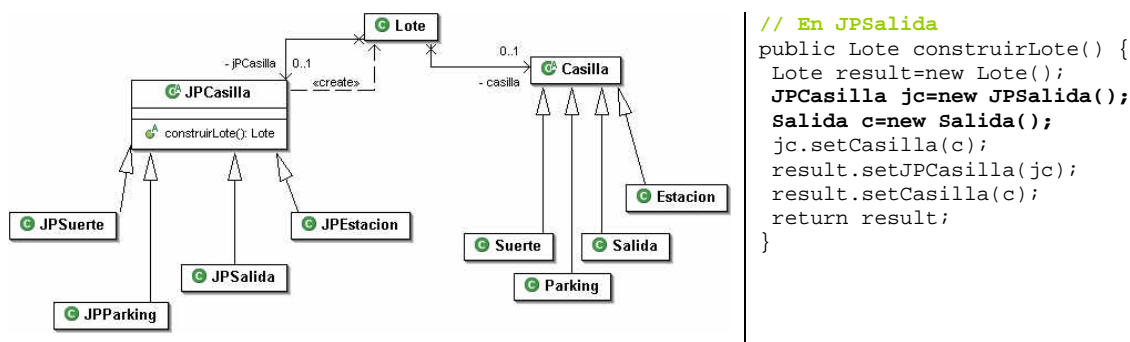


Figura 159. Con el *Builder*, se asigna a uno de los objetos la responsabilidad de crear el resto del lote (izquierda). A la derecha, posible implementación de la operación en una de las especializaciones

Al utilizar este patrón se incluye habitualmente una clase (a la que se conoce con el nombre de *Director*) que se encarga de instanciar el *builder* al subtipo concreto, y que llama también a la operación de creación. En la Figura 160 se ha añadido el *Director* al diagrama de clases de la figura anterior. Se ha modificado la operación *construirLote* de *JPCasilla* y sus especializaciones añadiendo un parámetro, correspondiente al tipo de la casilla.

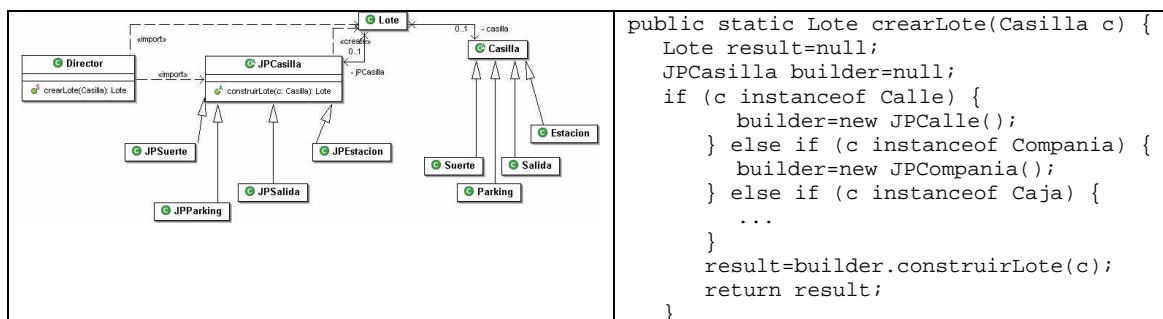


Figura 160. Adición del *Director* al esquema de la Figura 159. A la derecha, *construirLote* en *Director*

Si bien hemos dicho hace dos párrafos que la operación *construirLote* será *normalmente* abstracta en la superclase, puede hacerse concreta y apro-

**Patrón
Template-
method**

vechar la herencia para especificar el comportamiento de las subclases en la superclase. Con esta alternativa, que responde al patrón “Plantilla de métodos” (*Template-method*) implementamos en *JPCasilla* la operación con el código mostrado en el lado izquierdo del cuadro siguiente, en el que se llama a una operación abstracta (*construirJPCasilla*), que se encuentra implementada en cada especialización (en el lado derecho, redefinición de esta operación en *JPSalida*).

<pre>// En JPCasilla public Lote construirLote(Casilla c) { Lote result=new Lote(); JPCasilla jc=construirJPCasilla(); jc.setCasilla(c); result.setJPCasilla(jc); result.setCasilla(c); return result; } protected abstract JPCasilla construirJPCasilla();</pre>	<pre>// En JPSalida protected JPCasilla construirJPCasilla() { return new JPSalida(); }</pre>
--	---

Cuadro 143. Declaración de *construirLote* como una operación concreta en la superclase, pero que llama a una abstracta redefinida en las subclases

1.3. Carga del tablero en *JPTablero* con una *Abstract factory* (Fábrica abstracta)

Con el *Builder*, asignamos a una de las clases la responsabilidad de crear el resto de instancias del lote de productos. La solución, aunque buena, eficiente y fácilmente reutilizable, asigna a una clase del dominio del problema responsabilidades de creación de objetos, que no forman parte de su resolución. Una alternativa pasar por la utilización del patrón *Abstract factory*, mediante el cual se crea una clase abstracta y varias especializaciones cuya única responsabilidad es la creación de las instancias del lote.

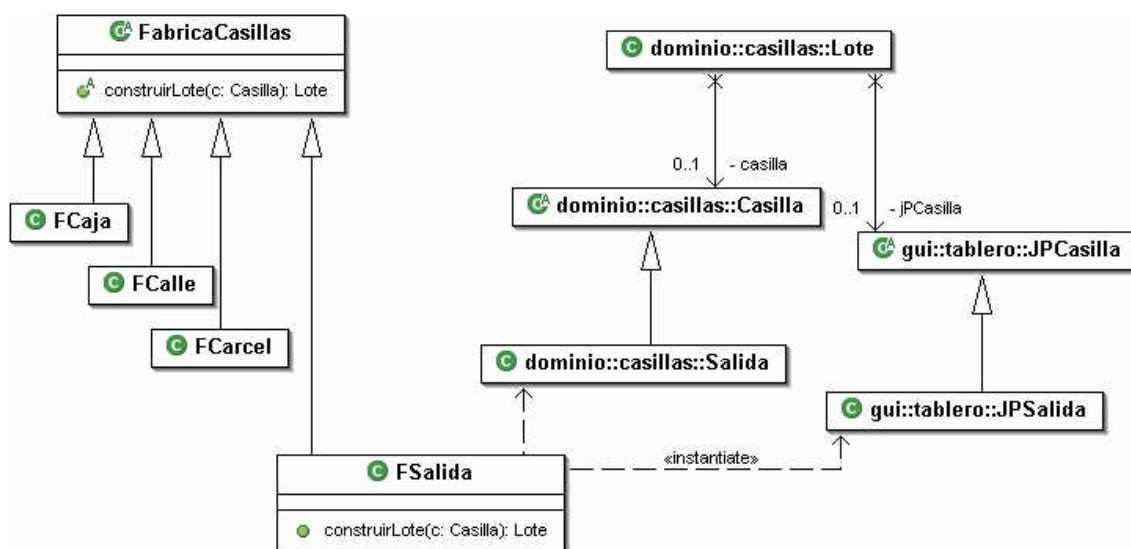


Figura 161. Esquema de la solución con la Fábrica abstracta

Con esta alternativa, *construirLote* también suele ser abstracta en la superclase, si bien puede aplicarse una vez más el patrón Plantilla de métodos y hacerla concreta.

Por otro lado, también es necesario disponer de una clase *Director*, que instancie la fábrica al subtipo correspondiente.

1.4. Carga del tablero en *JPTablero* con una fábrica introspectiva

La introspección o reflexión es una capacidad que ofrecen algunos lenguajes mediante la cual, por ejemplo, se puede en tiempo de ejecución cargar clases que eran desconocidas en tiempo de compilación; los objetos pueden interrogarse a sí mismos sobre aspectos como la clase a la que pertenecen; un objeto puede inspeccionar su estructura, su colección de operaciones, las clases de las que hereda, etc.

El lenguaje Java y los lenguajes de .NET permiten esta posibilidad. En Java, por ejemplo, se dispone de un conjunto de clases (agrupadas en el paquete *java.lang*) que habilitan gran parte de estas posibilidades. La Figura 162 muestra parte de algunas de las clases que permiten utilizar introspección en lenguaje Java: *Class* es el tipo utilizado para representar clases; a partir de una instancia de *Class*, podemos obtener su lista de constructores (objetos de clase *Constructor*), métodos (*Method*) y campos (*Field*), que pueden ser inspeccionados en tiempo de ejecución para conocer, por ejemplo, su tipo (en el caso de los campos), su tipo devuelto (para los métodos), los tipos de los parámetros (en los constructores y métodos) o, incluso, llamarlos (operaciones *newInstance* en *Class*, que llama al constructor sin parámetros de la clase, si es que lo tiene; *newInstance* de *Constructor*, a la que se pueden pasar los parámetros si es que el constructor los tiene; *invoke* en *Method*).

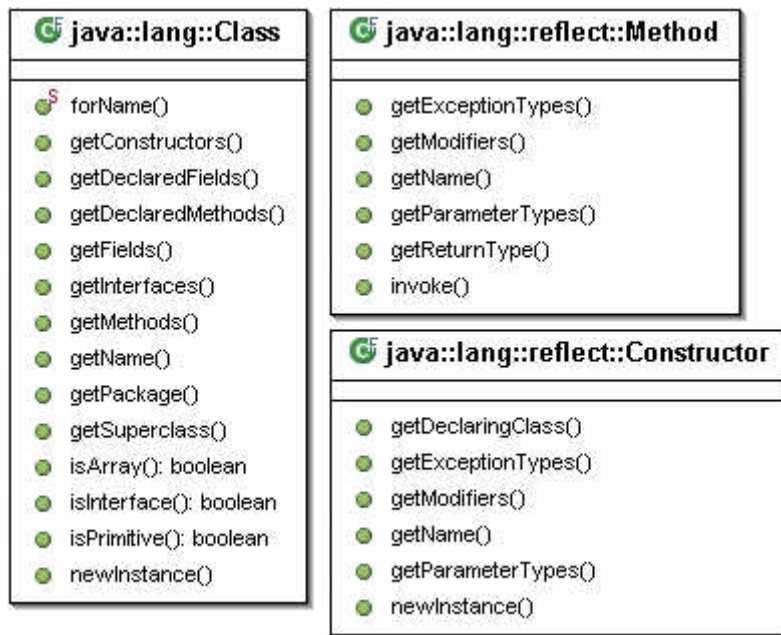


Figura 162. Algunas de las clases Java que permiten la introspección

Para ilustrar el uso de la introspección en este contexto, actuaremos sobre la operación *construirLote* de *JPCasilla* (Cuadro 143). Lo que haremos será instanciar la clase (de tipo *Class*) cuyo nombre sea *JP* seguido del nombre del tipo de la casilla que se pasa como parámetro. Yendo poco a poco, observemos en primer lugar que el resultado de ejecutar la función *main* que se ofrece en el lado izquierdo del siguiente cuadro es el que se muestra en el lado derecho:

<pre>public static void main(String[] args) { Salida casilla=new Salida(); System.out.println(casilla.getClass().getName()); }</pre>	<p>dominio.casillas.Salida</p>
--	--------------------------------

Cuadro 144. Construimos una instancia de un tipo cualquiera y mostramos el nombre del tipo

Como se observa, se obtiene el nombre completo del tipo del objeto, incluyendo el paquete. En el siguiente cuadro separamos en primer lugar el nombre del tipo del nombre del paquete; a continuación, creamos “reflexivamente” una *JPCasilla* correspondiente al tipo de la casilla y terminamos la construcción:

<pre>public static void main(String[] args) throws ClassNotFoundException, InstantiationException, IllegalAccessException { Salida casilla=new Salida(); String nombreDelTipo=casilla.getClass().getName(); String paquete=nombreDelTipo.substring(0, nombreDelTipo.lastIndexOf(".")); nombreDelTipo=nombreDelTipo.substring(nombreDelTipo.lastIndexOf(".")+1); System.out.println(paquete + "\n\r" + nombreDelTipo); // Carga de la clase JPCasilla Class c=Class.forName("gui.tablero.JP"+nombreDelTipo); Object o=c.newInstance();// Llamada al constructor JPCasilla jp=(JPCasilla) o; // Transformamos la instancia a la clase deseada jp.setCasilla(casilla); }</pre>

Cuadro 145. Ejemplo de creación de una instancia con introspección

La instrucción `Class.forName(String nombreDeClase)` del cuadro anterior carga la clase cuyo nombre se pasa como parámetro, lo que equivaldría a una instrucción `import nombreDeClase`. Si la clase no se encuentra, se lanza una `ClassNotFoundException`. La siguiente instrucción crea una instancia de `nombreDeClase` llamando a su constructor sin parámetros, si es que lo tiene: en este ejemplo, sí; si no lo tuviera, deberíamos recuperar el constructor deseado de su lista de constructores e invocarlo pasándole un array con los objetos que deseemos pasar como parámetros. El objeto construido se obtiene en forma de un `Object` genérico, que debe transformarse, para nuestro dominio, a una `JPCasilla`, y eso es lo que hacemos en la penúltima línea.

Para evitar que se nos lance la `ClassNotFoundException` debemos (es decir, que la clase deseada no pueda cargarse) mantener cierta uniformidad en cuanto a los nombres de tipos de las casillas de *dominio* y sus respectivas representaciones en *gui.tablero*. En la Figura 163 puede apreciarse que es preciso ajustar los nombres de varios tipos de casillas: *Caja* y *JPCajaDeComunidad*, *Impuesto* y *JPImpuestos*, *VayaALaCarcel* y *JPVaya*.

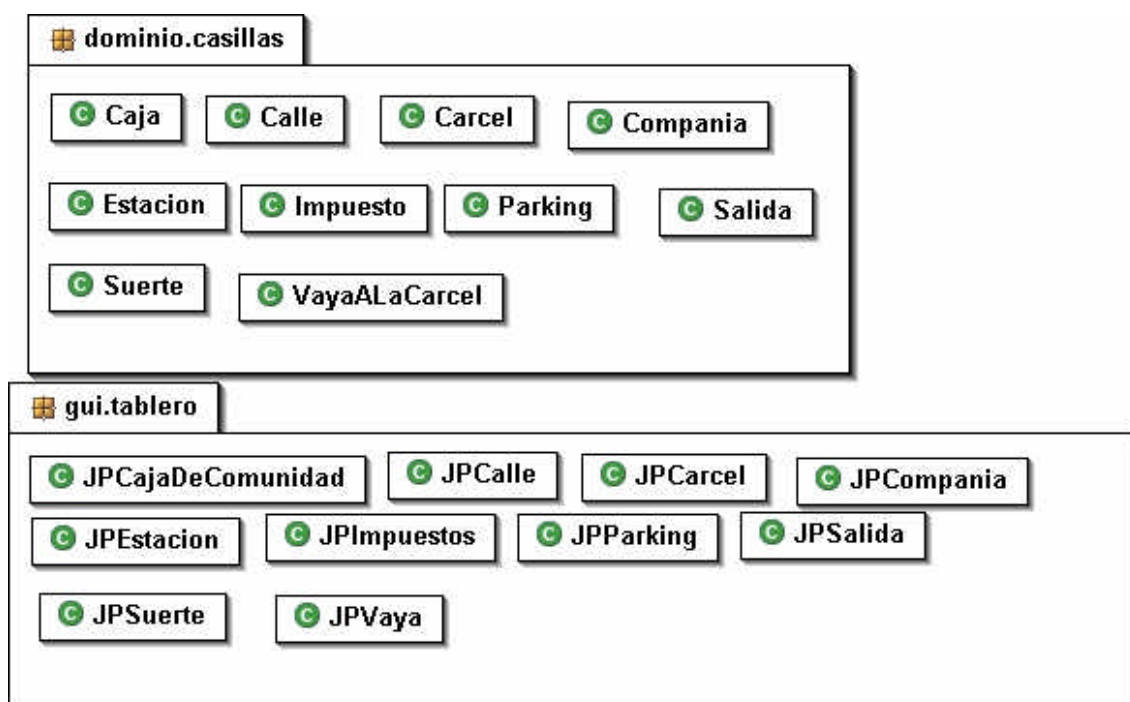


Figura 163. Casillas en el dominio y en la presentación

Daremos por buenos los nombres de *dominio.casillas* y modificaremos los nombres de los tipos correspondientes de *gui.tablero* con el asistente de refactorización incluido en el entorno de desarrollo, que actualiza las declaraciones y usos de todos los objetos.

Con estas consideraciones, la operación `construirLote` de *JPCasilla* ya no llamará a ninguna operación abstracta que deba redefinirse en las subclases.

```

public static Lote construirLote(Casilla casilla)
    throws ClassNotFoundException, InstantiationException, IllegalAccessException {
    String nombreDelTipo=casilla.getClass().getName();
    String paquete=nombreDelTipo.substring(0, nombreDelTipo.lastIndexOf("."));
    nombreDelTipo=nombreDelTipo.substring(nombreDelTipo.lastIndexOf(".") +1);
    Class c=Class.forName("gui.tablero.JP"+nombreDelTipo);
    JPCasilla jp=(JPCasilla) c.newInstance();
    jp.setCasilla(casilla);
    Lote result=new Lote();
    result.setJPCasilla(jp);
    result.setCasilla(casilla);
    return result;
}

```

Cuadro 146. Implementación “reflexiva” de *construirLote* (en *JPCasilla*)

1.5. Resultado visual

Cualquiera de las soluciones discutidas para realizar la carga del tablero puede funcionar, cada una con sus ventajas y sus inconvenientes. Si bien hemos optado por la última solución, el resultado visual debe ser en cualquiera de los casos una ventana del estilo de la mostrada en la Figura 164.

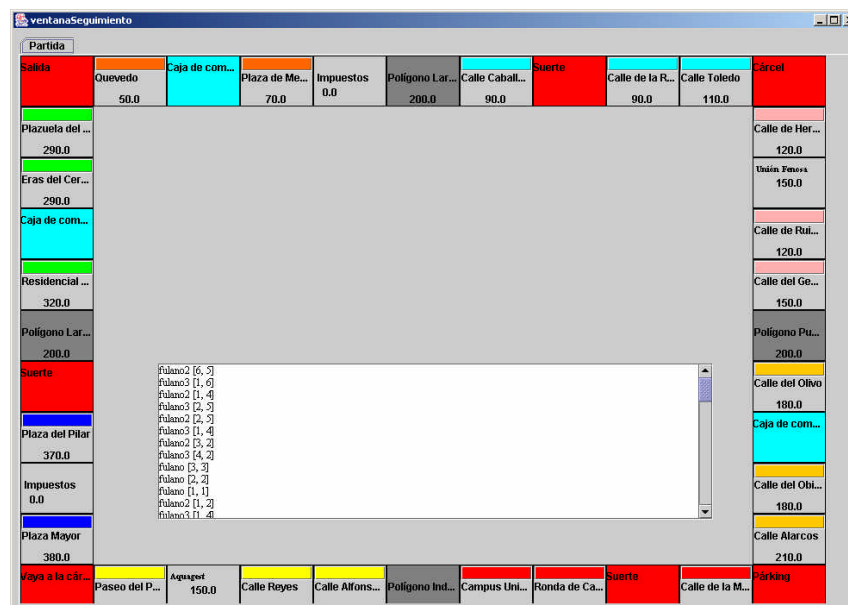


Figura 164. Aspecto de *JFSeguimiento* con el *JPTablero* correspondiente a una partida

2. Seguimiento del juego

La zona de texto del centro de la Figura 164 va mostrando las operaciones que los jugadores van realizando, como las tiradas de dados. Para facilitar el seguimiento del juego, nos gustaría dotar al servidor de la posibilidad de ir mostrando las casillas que va visitando cada jugador. Para esto, actualizaremos el *JPTablero* cuando la partida esté en el momento *EnJuego* y el jugador lance los dados, operación que teníamos implementada en el Cuadro 132 (página 260), y de la que reproducimos un fragmento en el cuadro siguiente.

```

public void tirarDados(int dado1, int dado2) {
    Jugador j=this.partida.getJugadorConElTurno();
    ... // Control de dobles
    int posicionDestino=j.getPosicion()+dado1+dado2;
    posicionDestino=posicionDestino%40;
    j.setPosicion(posicionDestino);
    Casilla c=this.tablero.getCasilla(posicionDestino);
    ... // Control de deudas
}

```

Cuadro 147. Fragmento de la implementación de *tirarDados* (en *EnJuego*)

Tras controlar si la tirada ha sido de dobles (y pasar el turno si es el tercero), la instancia de *EnJuego* comprueba la posición de destino, sitúa al jugador en la casilla que corresponda y luego controla la posible deuda que deba adquirir el jugador que ha movido. En algún momento del fragmento del código anterior debemos notificar al *JPTablero* que el jugador ha caído en la casilla correspondiente, representada por el objeto *c* indicado en la línea resaltada en negrita.

Podemos optar por varias soluciones:

- 1) Hacer que *c* le diga al *JPCasilla* que le apunta que el jugador ha caído en esa casilla, con lo que la *JPCasilla* mostrará de alguna manera la ficha correspondiente al jugador.
- 2) Hacer que, al ejecutar la operación *setPosicion* sobre *j* (de clase Jugador), se recupere la casilla sobre la que ha caído el jugador, y entonces se le comunica el hecho a la *JPCasilla* que le apunta.
- 3) Hacer que *tablero* le diga a su correspondiente *JPTablero* que el jugador ha caído en la casilla que corresponda, de manera que la responsabilidad quede delegada a estas clases.

En los dos primeros casos, la notificación al *JPCasilla* se realiza a través de un objeto de tipo *Casilla*; pero no hay conexión entre *Casilla* y *JPCasilla*, aunque sí al revés. En algún momento es necesario establecer la doble conexión, haciendo que *Casilla* conozca a su *JPCasilla* correspondiente. Como siempre, deseamos mantener la separación modelo-vista (entre dominio y presentación), por lo que haremos una interfaz *IBotonCasilla* que será a quien notifique *Casilla* y que estará implementada por *JPCasilla*.

El establecimiento de la doble conexión lo haremos en el método *construirLote* de *JPCasilla*, que mostrábamos en el Cuadro 146, añadiendo la línea resaltada:

```

public static Lote construirLote(Casilla casilla)
    throws ClassNotFoundException, InstantiationException, IllegalAccessException {
    String nombreDelTipo=casilla.getClass().getName();
    ...
    JPCasilla jp=(JPCasilla) c.newInstance();
    jp.setCasilla(casilla);
    casilla.setBoton(jp);
    ...
}

```

```
    }
    return result;
}
```

Cuadro 148. Adición de una instrucción a *Casilla* para que conozca a su “botón”

La implementación de *setBoton* en *Casilla* es la siguiente:

```
public void setBoton(IBotonCasilla boton) {
    this.boton=boton;
}
```

Cuadro 149. *setBoton* (en *Casilla*)

Con estos cambios, el diseño de este fragmento del sistema queda como sigue:

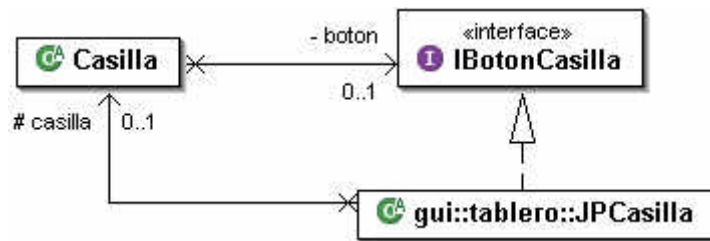


Figura 165. Relación entre *Casilla* y su representación visual

Cuarta parte

Capítulo 15. APLICACIÓN CLIENTE

XXX

1. Introducción

En la tercera parte de este libro habíamos comenzado la construcción de un cliente visual (Figura 130, página 227) para facilitar el desarrollo de la aplicación servidora, que posteriormente habíamos sustituido por una clase de test que nos fue guiando el desarrollo del servidor.

En esta cuarta parte completaremos el desarrollo del cliente para que permita, a los jugadores, el desarrollo cómodo de la partida.

Capítulo 16. PARÁMETROS DE CALIDAD EN SOFTWARE ORIENTADO A OBJETOS

// TODO

1. Auditoría

En el desarrollo de un sistema software participa un grupo de desarrolladores que comparte, entre otros artefactos, su código fuente. Por esta razón es frecuente establecer en las organizaciones estándares de codificación que normalicen la indentación del código, el nombrado de variables, clases, campos, métodos, etc.

Algunos entornos de desarrollo poseen plugins que auditan el código fuente, indicando las violaciones a reglas más o menos aceptadas por la comunidad. Así, por ejemplo, el entorno JDeveloper incluye validación de reglas de acceso (visibilidad pública de campos, por ejemplo), de orden de uso de modificadores (una clase puede declararse como *abstract public*, por ejemplo, pero la *Java Language Specification* recomienda seguir el orden *public abstract*), de manejo de excepciones inadecuado (poner un *catch* vacío, por ejemplo), de documentación inadecuada del código (según los estándares de *Javadoc*), de objetos declarados pero nunca usados, etc. La Figura 166 muestra el resultado de ejecutar este plugin sobre la *Aplicación para investigadores*, así como la explicación ofrecida respecto de una regla violada de importancia *alta*.

Muchas de las violaciones indicadas por los auditores de código pueden ser rápidamente resueltas con refactorizaciones para las que, en muchos casos, el entorno de desarrollo puede que nos ofrezca asistencia automática.

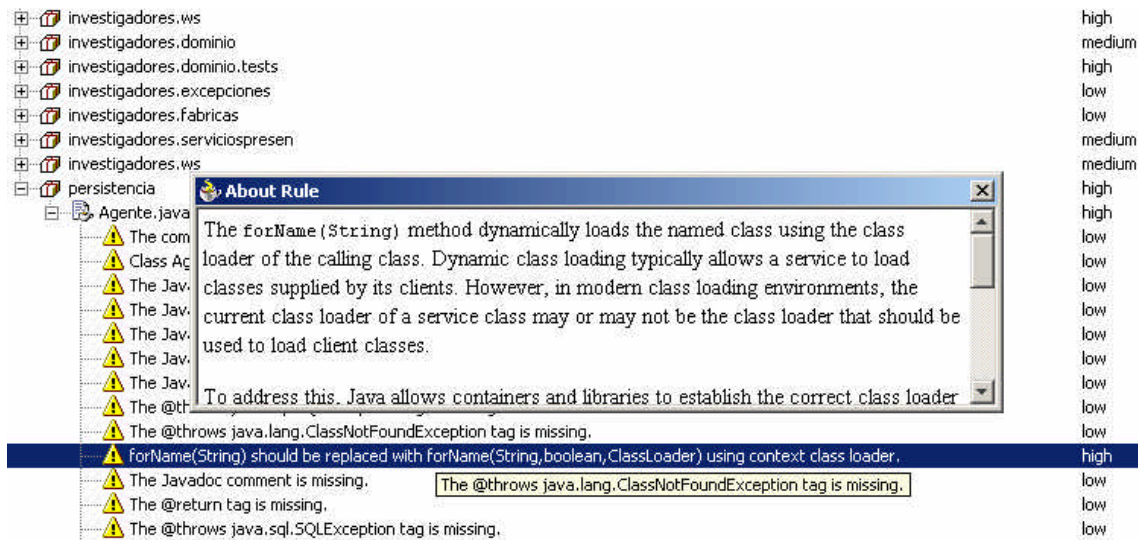


Figura 166. Resultado de la auditoría de la *Aplicación para investigadores*

2. Medición de código

Además de las reglas de estilo de codificación, el código puede ser medido con el fin de estimar de forma más objetiva su nivel de calidad. Existen miles de métricas para evaluar la calidad de cualquier artefacto software imaginable (diagramas de todos los tipos, código orientado a objetos o no, casos de prueba, bases de datos, documentos de requisitos, etc.). En el contexto del software orientado a objetos, las métricas más utilizadas son las propuestas en 1994 por Chidamber y Kemerer¹⁴, que no obstante ya habían publicado antes en un congreso. En 1993, Li y Henry¹⁵ analizaron las métricas de aquéllos.XXX

2.1. Métricas para sistemas orientados a objeto

2.1.1 DIT: *Depth of inheritance tree* (profundidad en el árbol de herencia)

Esta métrica mide la profundidad en el árbol de herencia de una clase, considerándose que la clase que se encuentra en la raíz de una jerarquía tiene DIT=0.

Puede suponerse que, a mayor profundidad de una clase en el árbol de herencia, más dificultad habrá para reutilizarla y más dependencias tendrá

¹⁴ *A metrics suite for object oriented design*, IEEE Transactions on Software Engineering, 20(6), 476-493.

¹⁵ *Object-oriented metrics that predict maintainability*, Journal of Systems and Software, (23), 111-122.

de otras clases. El valor de DIT, por tanto, no debería ser muy alto para no dificultar el mantenimiento y la reutilización del código.

2.1.2 NOC: *Number of children* (número de hijos)

Esta métrica mide el número de especializaciones directas de una clase, de manera que una clase sin subclases tendrá $NOC=0$.

Cuanto mayor sea el valor de NOC, más afectará su mantenimiento al resto del sistema.

2.1.3 RFC: *Response for a class* (respuesta para una clase)

Esta métrica es la suma del número de métodos locales y del número de métodos llamados por métodos locales. El valor permite estimar la complejidad de las llamadas realizadas en la clase: cuanto más alto sea, más dificultad habrá para realizar, por ejemplo, trazas de su ejecución, por lo que el valor no debería ser muy alto.

2.1.4 LCOM: *Lack of cohesion of methods* (carencia de cohesión de los métodos)

En la página 40 decíamos que la cohesión viene a ser el grado de relación que tienen entre sí los elementos de un conjunto. En una clase, parece lógico esperar que sus operaciones utilicen principalmente los campos definidos en la propia clase; lo contrario dificultaría, probablemente, la facilidad de comprensión del código de la clase.

Esta métrica cuenta el número de conjuntos disjuntos de métodos locales. Dos métodos se colocarán en distintos conjuntos si no acceden a ningún campo común.

El valor de esta métrica va de cero hasta el número de métodos: valdrá cero cuando todas las operaciones tengan al menos un campo en común, y valdrá el número de métodos cuando cada método acceda a un campo diferente.

A mayor valor, menos cohesiva será la clase (es decir, se le estarán asignando responsabilidades muy dispares) y más difícil será su mantenimiento y su reutilización.

2.1.5 CBO: *Coupling between objects* (acoplamiento entre clases de objetos)

XXX

2.1.6 **WMC: *Weighted method per class* (complejidad total de los métodos de una clase)**

Esta métrica mide la complejidad ciclomática total de los métodos de una clase. La complejidad ciclomática es el número de caminos linealmente independientes que pueden seguirse para ir desde el inicio de un método hasta cualquiera de sus posibles salidas. Si se representa la lógica de control del método con un grafo de flujo, el valor coincide con el número de áreas distintas que pueden encontrarse.

Cuanto mayor sea el valor de esta métrica, más difícil será realizar trazas de la ejecución de la clase, y más difícil será conseguir casos de prueba para probar completamente todos los posibles caminos de su lógica de ejecución. Cualquier instrucción condicional (un *if*, un *for*, etc.) introducen ramas nuevas en la posible ejecución, incrementando estas instrucciones en uno la complejidad ciclomática del método en el que se encuentran.

2.1.7 **MPC: *Message-passing coupling* (acoplamiento por paso de mensajes)**

Esta métrica cuenta el número de métodos de otras clases que utiliza la clase considerada.

A mayor valor, mayor dependencia de esta clase respecto de otras, por lo que su valor debería ser relativamente bajo.

2.1.8 **NOM: *Number of methods* (número de métodos)**

Esta sencilla métrica cuenta el número de métodos de la clase considerada. En ocasiones se separa la cuenta de métodos según sus modificadores (por ejemplo, métodos públicos o privados, métodos estáticos, etc.).

Cuanto mayor sea su valor, es muy probable que la clase tenga una mayor dificultad de mantenimiento.

2.2. **Clasificación y utilidad de las métricas anteriores**

Como sabemos, el bajo acoplamiento y la alta cohesión son dos parámetros de calidad que todo ingeniero de software debe tener presente a la hora de diseñar sistemas. Con el primer término, queremos referirnos a que es bueno que una clase dependa poco de otras clases: lo contrario puede provocar efectos indeseados cuando se modifica una de las clases de la que se depende; con el segundo, significamos que es bueno que las responsabilidades de una clase estén relacionadas entre sí.

De las métricas anteriores, son de acoplamiento todas aquellas que, de algún modo, consideren en su evaluación las relaciones de una clase con otras (Tabla 35). Es importante tener siempre presente que en multitud de

estudios empíricos se ha demostrado que las clases con más fallos son aquellas que presentan un acoplamiento más elevado¹⁶.

Métrica	Por qué es de acoplamiento
DIT	Porque considera las clases de las cuales hereda (acoplamiento mediante herencia)
NOC	Porque considera las relaciones de una clase con sus especializaciones directas (acoplamiento mediante herencia)
MPC	Porque considera el número de mensajes que la clase envía a otras clases (acoplamiento por paso de mensajes)
CBO	Porque considera directamente el acoplamiento entre clases de objetos

Tabla 35. Métricas de acoplamiento para objetos

Por otro lado, sólo la métrica LCOM es realmente una métrica de cohesión.

2.2.1 Definición de umbrales

Ninguna de las métricas anteriores sería útil si no dispusiéramos de valores que nos permitieran determinar el grado de bondad de las mediciones obtenidas. En la literatura científica existen multitud de estudios que demuestran la mayor o menor utilidad de algunas métricas, pero es difícil encontrar umbrales que nos permitan conocer con relativa exactitud si un cierto valor es bueno o malo.

La Agencia Espacial de Estados Unidos¹⁷ dispone de una página web en la que tiene publicados algunos umbrales de varias métricas bien conocidas para sistemas orientados a objeto.

Métrica	Umbral recomendado	Máximo admisible
NOM	<20	<40
WMC	≤100 (<10 para cada método)	XXX
CBO	≤5	

Tabla 36. Umbrales de algunas métricas recomendados por la NASA

Cuando una clase supere los umbrales de alguna de las métricas será necesario refactorizar para, por ejemplo, dividir las responsabilidades de la clase original en dos nuevas clases.

¹⁶ Esto también nos sirve para prestar más atención a estas clases durante sus pruebas

¹⁷ <http://satc.gsfc.nasa.gov/metrics/codemetrics/>

3. Modificación del código y nuevos patrones

En el código final de la Aplicación para investigadores, incluimos en la clase *Solicitud* multitud de operaciones para construir listados en formato HTML con todas las solicitudes de un proyecto, las pendientes de autorizar, las pendientes de validar, las rechazadas, las pagadas, etc. Éstos son todos métodos estáticos que acceden directamente a la base de datos y construyen Strings sin manipular realmente instancias de la clase (Figura 167).

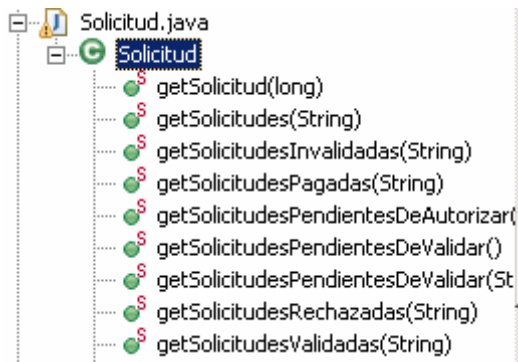


Figura 167. Operaciones estáticas, algunas de las cuales pueden ser llevadas a clases auxiliares

Su *carencia de cohesión* (métrica LCOM) es por tanto evidente y, de hecho, así se demuestra al medir el proyecto con el plugin Eclipse Metrics para el entorno Eclipse: en la Tabla 37, los números en cursiva representan los valores de algunas métricas que exceden el valor recomendado por este plugin, entre ellos LCOM.

CC (max)	LOCm (max)	NOL (max)	NOP (max)	NOS (max)	Ce	LCOM	NOF	WMC	Type
4	39	4	1	30	18	285	6	56	Proyecto

Tabla 37. Fragmento de la tabla con algunas métricas de la Aplicación para investigadores

Bien mirada, las operaciones mencionadas no forman realmente parte del dominio de la clase *Proyecto*, sino que ejecutan operaciones auxiliares, que prestan servicio a esta clase. La situación es propicia para utilizar el patrón Fabricación Pura¹⁸, que nos dice que deleguemos a una clase auxiliar las operaciones que no formen parte del dominio de la clase. Ya en la lejana Figura 7 (página 37) delegábamos las operaciones de persistencia de una clase *Empleado* a otra asociada.

Otros umbrales que también se superan, no sólo en *Proyecto* sino en otras, es el número de sentencias por método, una métrica recogida por este

¹⁸ El patrón Fabricación Pura (*Pure Fabrication*) es un patrón de Larman.

plugin. En particular, todos los métodos de persistencia que hemos ido construyendo poseen una gran parte de su estructura repetida. Al lector, sin duda, ya le suenan las siguientes líneas como parte de los constructores materializadores de las clases persistentes:

```
public Clase(valores de la clave)
    throws Exception {
    String SQL=...;
    Connection bd=null;
    PreparedStatement p=null;
    try {
        bd=Agente.getAgente().getBD();
        p=bd.prepareStatement(SQL);
        p.set...
        ResultSet r=p.executeQuery();
        if (r.next()) {
            ...
        } else {
            throw new Exception(...);
        }
    }
    catch (Exception ex) {
        throw new Exception(ex.getMessage());
    }
    finally {
        bd.close();
    }
}
```

Cuadro 150. Código común a muchas clases persistentes

Para estas situaciones podemos utilizar el patrón Plantilla de Métodos (*Template method*).

3.1. Patrón Fabricación Pura

Mediante el patrón Fabricación Pura, nos llevamos a una clase auxiliar las operaciones que no estén directamente relacionadas con el propósito de una clase. De este modo, cuando alguien (incluida la clase de dominio) necesite ejecutar uno de los servicios que se han delegado, se lo pide a la fabricación pura.

En nuestro ejemplo, podemos utilizar el asistente para refactorización de Eclipse para llevarnos a una nueva clase (*investigadores.dominio.fps.FPPProyecto*) las operaciones de *Proyecto* que se encargan de generar listados en HTML (Figura 168).

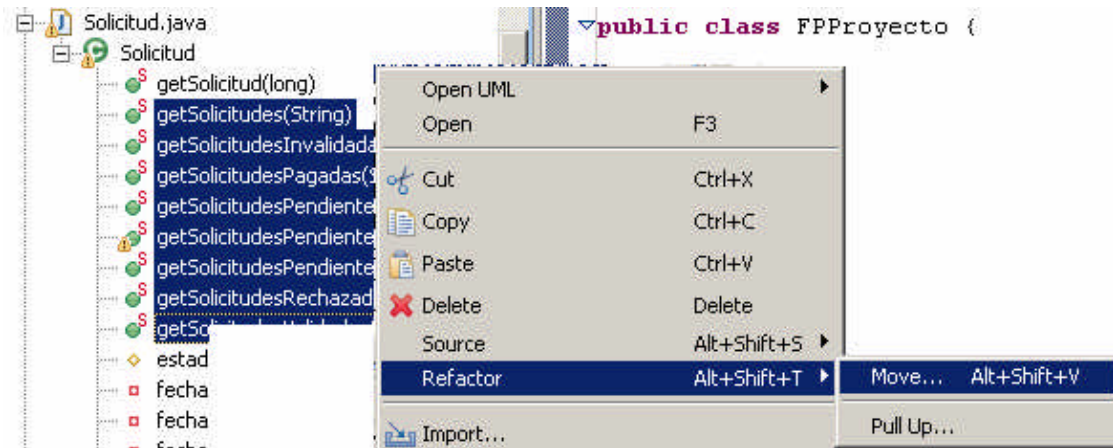


Figura 168. Refactorización para crear una fabricación pura para *Proyecto*

3.2. Patrón Plantilla de métodos

Una plantilla de métodos es un método concreto que se define en una clase abstracta y que, en su implementación, llama a operaciones concretas (cuya implementación es común para todas las subclases) y abstractas (cuya implementación es distinta en cada subclase); éstas estarán redefinidas en las subclases correspondientes.

Supongamos que el método *insert* de *Solicitud* tiene el código que se muestra en el Cuadro 151.

```
public void insert() throws Exception {
    Partida partidaSolicitada=new Partida(getProyecto(), getPartida());
    if (partidaSolicitada.getDisponible()<getImporte())
        throw new DisponibleInsuficienteException();
    if (!getProyecto().pertenece(getInvestigador()))
        throw new Exception("El investigador no pertenece al proyecto");

    String SQL="Insert into Solicitud (Proyecto, Partida, Investigador, Importe, " +
        "Descripcion, FechaDePresentacion) values (?, ?, ?, ?, ?, ?)";
    Connection bd=null;
    PreparedStatement p=null;
    try {
        bd=Agente.getAgente().getBD();
        p=bd.prepareStatement(SQL);
        p.setString(1, this.getProyecto().getCodigo());
        p.setString(2, this.getPartida());
        p.setString(3, this.getInvestigador().getLogin());
        p.setDouble(4, this.getImporte());
        p.setString(5, this.getDescripcion());
        long hoy=(new java.util.Date()).getTime();
        p.setDate(6, new java.sql.Date(hoy));
        p.executeUpdate();
        this.setFechaDePresentacion(new java.sql.Date(hoy));
        this.setNumeroDeSolicitud(this.ultimoNumeroDeSolicitud());
    }
    catch (Exception ex) {
        throw new Exception(ex.getMessage());
    }
    finally {
        bd.close();
    }
}
```

Cuadro 151. Código original de *insert* en *Solicitud*

Observando con detenimiento el código, vemos que se realizan en primer lugar una serie de “operaciones previas” (comprobación de precondi-

ciones), que luego se “construye la instrucción SQL”, que luego se establece la conexión y se “asignan los valores a la *PreparedStatement*”, que se ejecuta la sentencia y que se realizan unas “operaciones posteriores”. Al final, siempre añadimos los bloques *catch* y *finally*.

Todos los métodos *insert* de nuestras clases persistentes siguen este estilo de codificación. Entonces, podemos crear una superclase *Persistente* de la que hereden, entre otras, la clase *Solicitud* que ahora nos ocupa y crear una plantilla de métodos de este estilo:

```
public abstract class Persistente {
    public Persistente() {
        super();
    }

    protected void operacionesPrevias() throws Exception {
    }

    protected abstract String construirSQL();
    protected abstract void asignacionDeValoresParaInsert(PreparedStatement p)
        throws SQLException;

    protected void operacionesPosteriores() throws Exception {
    }

    public void insert() throws Exception {
        operacionesPrevias();
        String SQL=construirSQL();
        Connection bd=null;
        PreparedStatement p=null;
        try {
            bd=Agente.getAgente().getBD();
            p=bd.prepareStatement(SQL);
            asignacionDeValoresParaInsert(p);
            p.executeUpdate();
            operacionesPosteriores();
        }
        catch (Exception ex) {
            throw new Exception(ex.getMessage());
        }
        finally {
            bd.close();
        }
    }
}
```

Cuadro 152. Plantilla para la ejecución del método *insert*

Como vemos, dentro del método concreto *insert* de la clase abstracta *Persistente* se ejecuta en primer lugar una operación concreta llamada *operacionesPrevias* (pero que puede estar redefinida en la superclase); luego, se llama a la operación abstracta *construirSQL*, se establece la conexión, se ejecuta la operación abstracta *asignacionDeValoresParaInsert* y se llama a la concreta *operacionesPosteriores*. Si convertimos *Solicitud* en una especialización de *Persistente*, no será preciso que aquella implemente la operación *insert* (se ejecutará la versión heredada del Cuadro 152), aunque sí las dos abstractas que está heredando y, en este caso, también las dos concretas para redefinirlas:

```
protected String construirSQL() {
    String SQL="Insert into Solicitud (Proyecto, Partida, Investigador, Importe, " +
        "Descripcion, FechaDePresentacion) " +
        "values (?, ?, ?, ?, ?, ?)";
    return SQL;
}

protected void asignacionDeValoresParaInsert(PreparedStatement p) throws SQLException {
    p.setString(1, this.getProyecto().getCodigo());
    p.setString(2, this.getPartida());
    p.setString(3, this.getInvestigador().getLogin());
    p.setDouble(4, this.getImporte());
    p.setString(5, this.getDescripcion());
    long hoy=(new java.util.Date()).getTime();
    p.setDate(6, new java.sql.Date(hoy));
}

protected void operacionesPrevias() throws Exception {
    Partida partidaSolicitada=new Partida(getProyecto(), getPartida());
    if (partidaSolicitada.getDisponible()<getImporte())
        throw new DisponibleInsuficienteException();
    if (!getProyecto().pertenece(getInvestigador()))
        throw new Exception("El investigador no pertenece al proyecto");
}
```

Cuadro 153. Redefinición de operaciones en la subclase

3.3. Ventajas y desventajas de la utilización de patrones

De forma general, la utilización de patrones como los dos vistos (aunque ocurre también con muchos otros) aumenta la cohesión de las clases del sistema pero le aumenta el tamaño (en cuanto a número de clases) y el acoplamiento, ya sea de paso de mensajes (como en la Fabricación Pura) o de herencia (como en la Plantilla de métodos). Si a todas las clases persistentes las hacemos especializaciones de una superclase *Persistente* como la mencionada en el Cuadro 152, es evidente que creamos fuertes dependencias en un número grande de subclases. Sin embargo, es muy probable que se aumente la facilidad de mantenimiento del sistema.

Al final, y desde el punto de vista exclusivo de su medición, un buen diseño orientado a objetos es aquél en el que se encuentran niveles de compromiso entre varias de sus métricas: acoplamiento bajísimo conlleva probablemente una cohesión muy baja, muchos métodos en cada clase con muchas líneas de código y con complejidad ciclomática muy alta.

XXX Discusión tesis de Garzás

4. Lecturas recomendadas

Artículo de Sneed sobre calidad técnica

Artículo de Chidamber y Kemerer sobre métricas OO.

Artículo de Briand sobre acoplamiento y propensión a fallos.

Apéndice I. Notación de UML

Índice de términos

- .NET, 216
- acoplamiento, 40, 292
- Agente de base de datos, 98, 130
- Agregación, 21
- arquitectura multicapa, 31
- artefacto, 56
- artefactos*, 49
- Asociación, 22
- AssertionError*, 172
- Auditoría de código, 289
- Bag*, 160
- bibliotecas de operaciones, 19
- bind, 215
- caja blanca, 132
- caja negra, 131
- campo, 17
 - valores, 18
- campo estático, 18
- Capa de datos, 31
- Capa de dominio, 31
- Capa de negocio, 31
- Capa de persistencia, 31
- Capa de presentación, 31
- caso de prueba, 105
- caso de uso, 51
 - Descripción textual, 81
- Casos de uso
 - priorización, 79
- catch*, 35
- ciclo (en el Proceso Unificado de Desarrollo), 52
- clase, 17
 - campo, 17
 - comando (ejemplo), 18
 - comportamiento, 17
 - consulta (ejemplo), 18
 - estructura, 17
 - miembro, 19
 - operaciones, 17
 - clase abstracta, 20
 - clases persistentes*, 31
 - cohesión, 40, 292
 - Collection*, 160
 - Comando, 17
 - comando (ejemplo), 18
 - Comienzo, 53
 - complexType*, 216
 - componente*, 57
 - Composición, 21
 - Connection*, 34
 - Construcción, 53
 - constructores, 30
 - Consulta, 17
 - consulta (ejemplo), 18
 - contexto, 164
 - contrato, 173
 - Create**, 31
 - createRegistry, 215
 - criterio de cobertura, 132
 - criterios de cobertura, 139
 - datos, capa de, 31
 - delegación de las responsabilidades de persistencia, 36
 - Delete*, 32
 - Dependencia, 22
 - Desarrollo dirigido por las pruebas, 51
 - Descripción textual de los casos de uso, 81
 - Diagrama de casos de uso, 57
 - diagrama de colaboración, 93, 101
 - Diagrama de componentes, 56, 57
 - Diagrama de despliegue, 56, 57
 - diagrama de secuencia, 93
 - Diagramas de casos de uso, 56
 - diagramas de flujos de datos, 50
 - diagramas de interacción, 101
 - diagramas de secuencia
 - verificación y validación, 104
 - diseños orientados a interfaces, 41

- dominio
 - modelo de, 54
- dominio, capa de, 31
- Elaboración, 53
- espacio de nombres, 47
- especialización, 19
- estado, 17
- estado del sistema, 82
- estático, 18
- estereotipo, 64
- excepciones
 - tratamiento según el tipo, 172
- extensión (relaciones de, en casos de uso), 59
- Fabricación Pura, 36, 295
- fail*, 136
- finally*, 35
- flujo de eventos, 82
- flujo de eventos alternativo, 82
- flujo de trabajo (en el PUD), 53
- flujo normal de eventos, 82
- foco de control, 95
- fork*, 173
- generalización, 19
- herencia, 19
- inclusión (relaciones de, en casos de uso), 59
- Insert*, 31
- instancia, 17
- interfaces, 41
- interfaz, 41, 214
- interfaz remota, 214
- Invariantes, 158
- iteraciones, 51
- iterate*, 163
- jar*, 222
- java.sql*, 34
- junit*, 133
- Lenguaje Unificado de Modelado, 23
- línea de vida, 95
- lista de comprobación
 - para casos de uso, 91
 - para diagramas de secuencia, 104
- LocateRegistry.createRegistry, 215
- máquinas de estados, 263
 - y OCL, 268
- materializar*, 31
- mecanismos para la gestión de la persistencia, 33
- mensaje, 95
- metamodelo, 263
- método, 19
- Método
 - comando, 17
 - consulta, 17
- metodologías, 49
- métricas, 290
 - umbrales, 293
- middleware, 213
- miembro, 19
- Modelo de análisis, 56, 57
- modelo de diseño, 56, 65
- Modelo de diseño, 57
- modelo de dominio, 54
- Modelo de prueba, 56, 57
- namespace, 47
- Naming.bind, 215
- Naming.lookup*, 215
- negocio, capa de, 31
- nombres de rol, 27
- Object Constraint Language, 153
- Object Management Group, 23, 263
- Object Modelling Technique, 49
- objeto, 17
- OCL, 158
 - Bag, 160
 - Collection, 160
 - iterate, 163
 - OclAny, 159
 - OclExpression, 160
 - OclType, 159
 - Sequence, 160

- Set, 160
- oclInState*, 268
- OMG, 23
- OMT, 49
- operación abstracta, 20
- operación estática, 18
- operaciones, 17
- operaciones estáticas
 - uso para la creación de bibliotecas de operaciones, 19
- package, 47
- paquete, 47, 187
- paso de mensajes, 22
- patrones
 - Agente de base de datos, 98
 - Fabricación Pura, 36, 295
 - Proxy, 218
 - Singleton, 130
 - un árbol de herencia, una tabla, 113
 - un camino de herencia, una tabla, 115
 - una clase, una tabla, 110
- persistencia, capa de, 31
- plan de iteraciones, 79
- polimorfismo, 20
- Postcondiciones, 158
- precondición, 82
- precondiciones, 82
- Precondiciones, 158
- PreparedStatement*, 38
- presentación, 41
- presentación, capa de, 31, 41
- Priorización de casos de uso, 79
- privado, 23
- Proceso Unificado de Desarrollo, 51
- producto software*, 17, 49
- protegido, 23
- proxy*, 218
- pruebas
 - criterios de cobertura, 132
 - de caja blanca, 132
 - de caja negra, 131
 - pruebas de regresión, 131
 - pruebas funcionales, 131
 - público, 22
- Read**, 31
- refactorización, 289
- relaciones de extensión, 59
- relaciones de inclusión, 59
- Relaciones entre clases, 21, 104
- RemoteException*, 215
- requisito funcional, 51
- ResultSet*, 35
- riesgos, 54
- RMI
 - compilación de clases remotas, 222
- rmic*, 222
- Select*, 32
- Sequence, 160
- servicios, 23
- Servicios web, 216
- Set, 160
- Simple Object Access Protocol*, 216
- Singleton*, 130
- sistema orientado a objetos, 17
- SOAP, 216
- socket*, 214
- software*, 17, 49
- subclase, 19
- subsistema, 47
- superclase, 19
- Test-Driven Development, 51
- Transición, 53
- try*, 34
- UML, 23
 - multiplicidad, 24
 - Representación de campos, 24
 - Representación de operaciones, 24
 - visibilidad, 24
- un árbol de herencia, una tabla, 113
- un camino de herencia, una tabla, 115
- una clase, una tabla, 110
- UnicastRemoteObject*, 214

Unified Modeling Language, 23

Update, 32

Verificación y validación

 de diagramas de secuencia, 104

versión, 52

visibilidad, 22

vista funcional, 54

Vista funcional, 56

web services, 216

Web Services Description Language, 216

WSDL, 216

XML, 216